Silicon Iodine Axionite Yttrium Nitrogen

Inaugural Cambridge Battle(code?) Postmortem

Team **something else**

Members: **osteo**, **Jython**, **Coderz75**

2nd Place

What happens if you have your bot overthink everything?

Repository

"we are something else"

Spring 2026

Contents

1 Introduction

We placed 2nd worldwide. Yay! :)))

1.1 Team overview

We entered the competition as four solo teams, eventually merging into one. **amcsz** was busy with school so decided to bow out, leaving us three: **osteo**, **Jython**, **Coderz75** aka **pulsar not blackhole**. We chose the name **something else** after [**an annoying organiser**] told us to rename[1] to **something else** after people got confused after we renamed to **MFF-1** after being named **Lorem Ipsum**[2] after being named **awu's apprentice**.[3]

Whenever someone saw something wrong, we would try and fix it. This led to all of us working on multiple parts of the bot and many merge conflicts.[4] However, to vaguely specialize, it was like this: **Jython** worked on combat micro: turret micro, harvester "shielding", roading, **osteo** on fast core algorithms: pathfinding, max flow, turret tracking, and **Coderz75** on foundries, axionite, rush, and BREACH[5]!

1.2 Overview

We assume you are familiar with the rules of this year's game and will not introduce them again. There is not much special about our bot other than that it is generally good at everything without many noticeable weaknesses, except for the notably simple attack routing. You may find interest in our pathfinding or marker strategies. The majority of our strategy is described in the following couple sections (with pictures!)

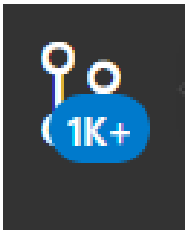
[0] "Silver Is All You Need" (see footnote 20)

[1] Despite us being vocal about the devs forcing us to change our name to **something else**, we have great appreciation towards them for their instrumentation of an amazing game this year. Note that they had to endure endless spamming from **Jython** and **Coderz75**. Please note **osteo** definitely did not spam at all.

[2] This is different than team **Lorem Ipsum** on ladder and MIT Battlecode, which are **Coderz75**'s old team.

[3] "after" is right-associative in this sentence.

[4] *many* could be an understatement.



help.

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

2 Architecture Overview

Our bot's per-turn control flow is divided into three sequential layers:

1. **Precompute** (`start_turn`) — build all shared data structures that states and the decision layer will query.
2. **Decision** (`determine_state`) — priority-ordered logic that selects a single state and its arguments.
3. **Execution** (`run_turn`) — dispatch to the chosen state class.

After execution, `end_turn` handles housekeeping: heal attempts, marker placement, and road-spam.

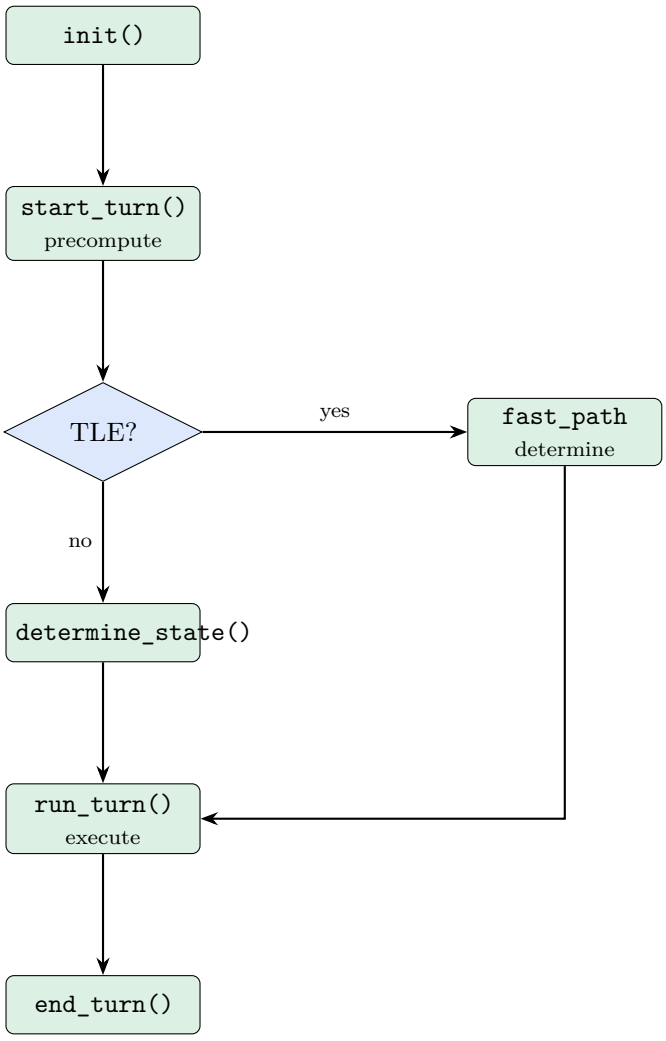


Figure 1: Per-turn control flow. The fast path skips expensive precompute checks when `TLEManager` detects budget pressure.

3 Information Infrastructure (Precompute)

Each turn, before any decisions are made, we populate a set of shared data structures. Systems are listed roughly in dependency order.

> In this section you might see `ti` used a lot. Here, `ti` means *tile*, NOT titanium.
> Most of the names and attributes in this section are understandable. When we don't think it is, we will put a comment.

3.1 Map & Symmetry

Every turn, we scan all nearby tiles and store mapdata in a list. This mapdata includes quite a lot of precompute:

Tile Info computed for each nearby tile each turn[6]

```
class TileInfo:
    __slots__ = (
        'env', 'round', 'easily_passable', 'harvester_adjacent',
        'has_building', 'building_hp', 'building_id',
    ↪ 'is_building_ally',
        'entity_type', 'target', # target refers to transporter
    ↪ target
        'has_bot', 'bot_hp', 'bot_id', 'is_bot_ally',
        'allied_bots_adjacent',
        'has_turret', 'turret_direction',
        'turrets_adjacent', 'ally_turrets_adjacent',
    ↪ 'enemy_turrets_adjacent',
        'ally_transporters_adjacent', 'enemy_transporters_adjacent',
        'resource_id', 'resource_type',
        'ally_outward_transporters_adjacent', # non-shield
    ↪ transporters (see later sections)
        'is_pointed_to', 'ally_non_road_buildings_adjacent',
        'ally_core_adj',
        'is_shield', # shield explained later
        'ally_fed_foundries_adjacent',  # fed by a (presumably)
    ↪ axionite line
    )
```

At each turn, we also check symmetry, guaranteed by the devs to be either horizontal, vertical, or rotational. Once symmetry is deduced, we all are happy :)

3.2 BfsBureau

See 7.1 for more details about how BFS works.

Each turn, we calculate:

- `bfs20_dist` - A mini weighted BFS for all tiles within vision range. Used throughout our bot for determining locally reachable locations.
- `bfs20_dist_adj` - Like the previous, but the BFS is computed with respect to moving next to the target tile. This is good for cases like healing and building, where we do not need to be on the target tile.

[6] According to Jython, Claude[7] said using ___slots___ would be faster. Supposedly it was.
[7] And Google.

- `enemy_bot_dist` - Scores all tiles within vision range regarding its distance to an enemy bot
- `enemy_bot_dist_adj` - Like previous, but instead scans adjacent positions
- `ti_ore_adj` - Flags all regions cardinally adjacent to a titanium ore (used for axionite routing, where we avoid routing next to titanium ores to ensure purity in routes).

3.3 DarkForest

`DarkForest` is the heart of the flow algorithms. It simply takes all it knows from the map, and directly computes flow at every single relevant point.

Darkforest Information

```
class DarkForest:
    nodes: list[TreeNode | None] = [None] * {{ PADDED_GRID_SIZE }}
    kind: list[int] = [0] * {{ PADDED_GRID_SIZE }}   # type of route (ax,titanium, etc._
    flow: list[int]         # flow in subtree (bottom-up)
    pressure: list[int]     # pressure at node (top-down, reset at sink boundaries)
    node_kind: list[int]    # propagated kind per node (top-down)
    core_sink_set: set[int] = set()  # titanium-only ALLY_CORE reachable nodes below pressure threshold
    sink_set: set[int]           # alias for core_sink_set (backward compat)
    leaf_set: set[int]           # titanium leaf nodes (no bridge + has flow), used for finding foundry positions
    ax_tagged: list[bool]        # True if node has any raw axionite flow in subtree
    sight_last_id: list[int]     # last resource-ID seen at this position; -1 = never
    sight_last_round: list[int] # game-round when that ID was last observed
    sight_flowing: list[bool]    # sight-based flow: True if fresh valid resource seen recently
    foundry_positions: set[int] = set()  # persistent: encoded positions of built foundries
    refined_ax_line: set[int] = set()    # recomputed each tick: ALLY_CORE nodes on refined-axionite output paths
```

3.4 Resource Tracking

- **`OreExecutive.fill`** - Scans nearby tiles for possible harvester locations (unused ores). Sorts them based on a metric: `int(5 * DIST_CORE + DIST_ME)`.
- **`HarvesterAdjacent.fill`** - Scans through tiles cardinally adjacent to harvesters, compiling information about each adjacent tile:

Harvester Adjacent location information

```
class AdjacentInfo:
    __slots__ = (
        'position', 'bfs_dist', 'bfs_dist_adj', 'is_harvester_ally', 'ti', 'hpos',
        #hpos = pos, ti means tile
        'consider_route', 'dist_to_ally_core', 'has_ally_transporter',
```

```
        'easily_buildable', 'is_canonical_ally_harvester',
    ↪ 'is_working_shield',
        'harvester_ally_turrets_adjacent',
    ↪ 'harvester_enemy_turrets_adjacent',
        'enemy_turrets_adjacent', 'ally_turrets_adjacent',
        'sentinel_dir_info',
        'gunner_dir_info',
        'h_outward_adj', # ally_outward_transporters_adjacent
        'h_has_fed_foundry' # ally_fed_foundries_adjacent > 0
    )
```

3.5 Combat Tracking

- **`Attacker.update`** - Calculates the number of nearby allies and enemies, to be used later.
- **`VisionTracker.fill`** - Computes all nearby transporters (allies and enemies), enemy roads (taking into factor those next to harvesters), disconnected transporters and mis-routed transporters, and ally harvesters.
- **`HealTargeter.fill`** - Computes all possible nearby heal targets:

Heal Target Information

```
class HealTargetInfo:
    __slots__ = (
        'position', 'building_heal', 'building_hp', 'bot_heal',
    ↪ 'bot_hp',
        'harvester_adjacent', 'is_transporter', 'is_turret',
        'has_enemy_bot', 'bfs_dist_adj', 'entity_type',
    ↪ 'core_info', 'is_launcher'
    )
```

- **`SitterTakedown.fill`** - Computes all possible locations for a **sitter takedown** (see figure 5).

Sitter Takedown Target Information

```
class SitterTargetInfo:
    __slots__ = (
        'position', 'dist_enemy_core', 'has_ally_transporter',
        'enemy_bots_nearby', 'harvester_nearby',
    ↪ 'launchers_adjacent',
        'bfs_dist_adj', 'flowing_enemy_transporters_adjacent',
    ↪ 'enemy_buildings_adjacent',
        'weight' # weight for pathfinding. See the pathfinding
    ↪ section
    )
```

3.6 TLEManager

> The per-turn time limit is **2 ms**. `TLEManager` tracks remaining budget and sets `daylight_savings_time = True` when the bot is at risk of timing out, triggering the fast path in fig. 1 and skipping the most expensive precompute steps.

4 Mode System

Modes are a coarse behavioral layer that gates entire categories of state transitions. A bot's mode is set once per turn and checked throughout `determine_state`.

Mode	Constant	Description
None	`NONE`	Initial state; transitions out on round 1.
Econ	`ECON`	Default mode. Prioritises harvester and conveyor builds; no rush targets.
Healer	`HEALER`	Assigned to the third (out of 4) bot spawned; focuses on staying near the core for healing until `Constants.HEAL_OVER`. These will also spam roads near the core
Early Rush	`EARLY_RUSH`	Suppresses econ builds; pursues aggressive rush targets. Triggered when every third bot can afford a gunner and estimated income ≥ 10.

Table 1: Bot modes and their transition conditions.

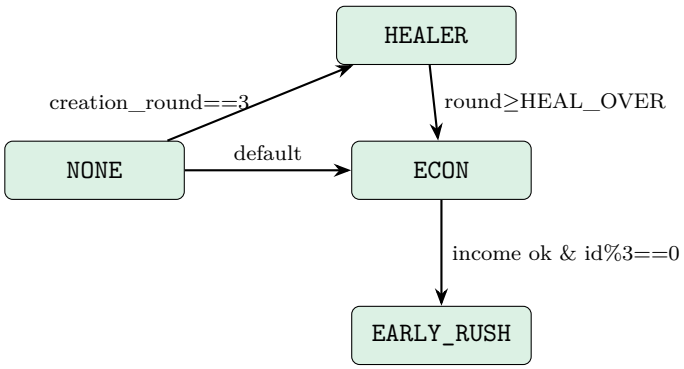


Figure 2: Mode transitions. Modes are monotone — bots do not revert except `HEALER` → `ECON`.

5 State Selection (`determine_state`)

When we are determining states the logic is a very convoluted ordered priority cascade: the first matching condition wins. We group the conditions into 11 tiers.

Priority	Tier	States triggered
1	Emergency overrides	`Destroy`, last-healed-turret continuation
2	Turret takedown	`BuildGunner`
3	Healing	`MoveTo` (heal)
4	Mis-route correction	`Reroute`
5	Early Defenses	`BuildAdvancedShield`, `BuildTurretWrapper`, `BuildShield`
6	Pre-Routing Combat	`Attack` (Preroute)
7	Active Routing	`RouteBreach`, `RouteFoundry`, `Route` (core), `MoveTo` (InitRoute)
8	Structures & Mid-Game Defenses	`BuildTurret`, `BuildLauncher`, `BreachBuild`, `FoundryBuild`
9	Main Combat	`Attack` (ShouldFire, Primary, Secondary near route)
10	Economy & Targeting	`MoveTo` (Stalk), `BuildHarvesterAx`, `BuildHarvester`
11	Exploration & Fallbacks	`Attack` (Secondary fallback), `Rush`, `MoveTo` (Patrol), `MoveTo` (Explore)

Table 2: Priority tiers in `determine_state` (top = highest priority).

5.1 Tier 1 — Emergency Overrides

Destroy harvesters. We move and destroy any known ally harvesters within vision range directly feeding enemy turrets (as in the turrets are cardinally adjacent to it). We prioritize more accessible harvesters, specifically ones with a smaller `BfsBureau.bfs20_dist_adj` distance.

Last-healed turret continuation. If the bot healed a turret within the last 5 rounds and the turret still needs healing, we re-commit to the heal position rather than bouncing to a different state. Without this, bots would abandon a half-healed turret every time a lower-priority target changed.

5.2 Tier 2 — Turret Takedown

If:

1. We can afford a gunner (`BuildManager.can_afford_gunner()`)
2. There is a valid space[8] adjacent to a harvester
3. The space has an open line of sight to an enemy turret,
4. No other ally turrets nearby
5. `VisionTracker.me_is_canonical_ally`

We move to build a gunner in that space to takedown an enemy turret.

[8]Valid space means the space either is empty, has a shield piece (conveyors or roads surrounding harvesters to prevent easy takedowns), or is a launcher (launchers are already defensive), and if the space is accessible.

All functions in the form of `BuildManager.can_afford_*` are custom functions that algorithmically decides whether we can afford an object or not. Most of the time, they respect a **reserve** of titanium, but not always. More information at 7.3.

`VisionTracker.me_is_canonical_ally` is our function allowing us to prevent "clumping" of many builder bots trying to do the same action. It organizes the visible tiles into "zones" of influence from ally builder bots, where ally builder bots closer to that tile will get priority for certain actions. Tiebreakers are decided by ID.

Figure 3: Case of some very cool turret combat against team **Blue Dragon** with us as silver.

5.3 Tier 3 — Healing

The heal target selector (`HealTargeter`) compares all nearby allies and chooses between them. Comparisons are made in the following order:

1. If one is near-inaccessible (`BfsBureau.bfs20_dist_adj >= 100`)
2. Avoid healing launchers over others
3. If one needs to be healed
4. Prioritize Turrets
5. Prioritize Transporters
    - Prioritize transporters adjacent to harvesters
6. Prioritize lower HP
7. Prioritize tiles with an enemy bot on it (in case they fire)
8. Closer tile
9. If we can also heal a bot on that tile
10. The lower HP bot.

If we would overheal, we ignore it, unless it is next to a harvester.
If the core has relatively high HP but is relatively far away, we avoid healing the core

- We do not heal if `CORE_HP < 500 - bfs_dist_adj * (BOTS_NEARBY_CORE) * 4`

5.4 Tier 4 — Mis-route Correction

Very simply, if:

- There is a mis-routed transporter nearby
    - Mis-routed transporters are ally transporters that feed into enemy buildings.
- It is easily reachable
- We are the canonical ally

- We are *not backtracking* while routing[9]

we break the route and try to route back to our core.

5.5 Tier 5 — Early Defenses

Three building decisions are checked in this tier, in order:

1. **Advanced shield** — a barrier is preferred when ≥ 2 allied turrets are adjacent and no enemy turrets threaten the tile.
2. **Turret wrapper** — delegates to `BuildTurretWrapper`, which picks gunner vs. sentinel based on the precomputed `GunnerDirInfo`.
3. **Basic shield** — fallback conveyor/barrier adjacent to a harvester.

> The first conveyor build (`buildingFirstConveyor`) temporarily *suppresses* Tier 5 turret and shield decisions, so a new route can be established before we commit tiles around the harvester.
>
> Shields around harvesters were very important, preventing enemy bots from building turrets there and taking down our supply lines. When roads health was decreased, we began using conveyors and barriers instead since they had more health for a relatively similar cost. Huge shout out to teams **Blue Dragon** and **Kessoku Band** for first pioneering this.

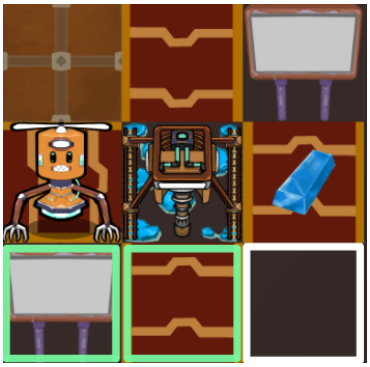


Figure 4: Shielding in action.

5.6 Tier 6 — Pre-Routing Combat

A pre-route special case fires an `Attack` before entering routing if the route origin is under threat.

5.7 Tier 7 — Active Routing

Checked in priority order: `RouteBreach`, `RouteFoundry`, `RouteToCore`.

If we find a valuable place to route from, for example an unconnected transporter with flow, or a location adjacent to an already built harvester not connected to our core yet, we will also route from there.

> There is a case where if we are nearby an enemy core, we would rather build a turret and attack the enemy core than route back to our base.

[9] Routing states have their own mis-route code. When mis-route was implemented, it clashed with the routing states and caused issues, so the older code for backtracking during routing was better.

5.8 Tier 8 — Structures & Mid-Game Defenses

1. **Primary Turret / Shield Placement** — If a valid `turretpos` is found, builds an `AdvancedShield` if there are ≥ 2 adjacent ally harvesters and 0 adjacent enemy harvesters; otherwise, defaults to a `TurretWrapper`.
2. **"Sitter" Takedown** — Checks for an optimal placement of a launcher.
   - **Sitter Takedowns** (see figure 5) are simply when we build a launcher to launch an enemy farther away. These happen under the following conditions:
     - `BuildManager.can_afford_launcher()`
     - If near enemy transporters and at least one enemy, OR near 3 or more enemies and near an enemy harvester
     - No nearby launchers
     - Location accessible
     - If not already adjacent: then check if we are the canonical ally
3. If we are supposed to build a *breach* here.
4. If we are supposed to build a *foundry* here.

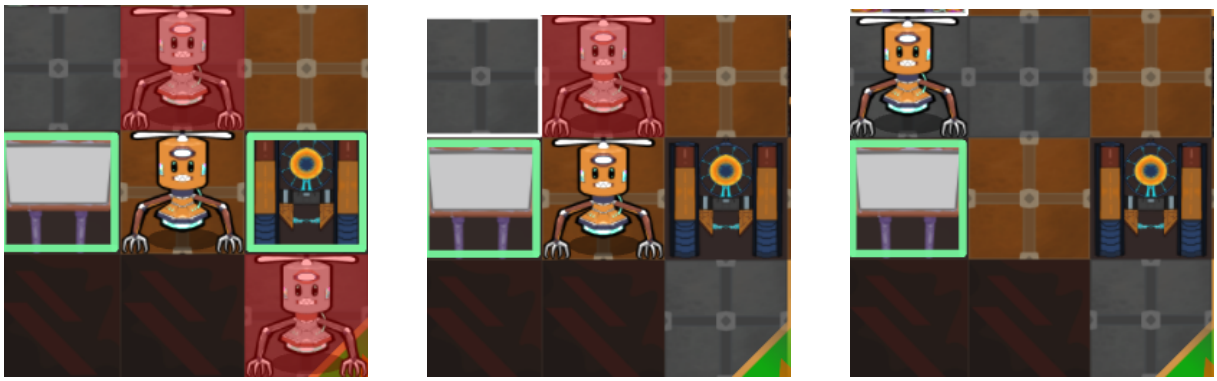


Figure 5: Case of **sitter takedown**. We place a launcher to clear the enemy silver bots (launched off the picture).

5.9 Tier 9 — Main Combat

1. **Local Attack** — Triggers an `Attack` command if `should_fire` is active and either `enemy_bot_dist_adj` $\geq$ `2`
   - If the tile we are on is an attackable enemy tile and has confirmed flow (by either sight or connection)
   - If low HP
   - Twice as many allies as enemies nearby
   - Enemy is far away (roughly 2 turns away) OR the target is a road and and there is no adjacent enemy (so it can be destroyed in a single turn).
2. If there are any other valuable **primary attack positions** nearby
   - Primary attack positions are enemy roads near a harvester, or nearby accessible transporters that have a confirmed flow, with no bots adjacent. Targets are filtered in this order:
     (a) Isn't flowing into an ally
     (b) Easily reachable
     (c) Disregard raw-axionite lines
     (d) Prioritize ones that are flowing
     (e) It doesn't have a bot nearby

(f) Avoid ones that are significantly farther away
(g) Transporter Health
(h) Prioritize ones near harvesters
(i) Computed Flow

3. If there are any **secondary attack positions** near us while routing (to free up space)
    - Secondary targets are any enemy roads given that the round is larger than 100 and we are roughly nearby a core (enemy and our own). These are filtered by whether a transporter is pointing towards or if one is closer.

> Rush bots will NOT attack UNTIL they have seen the enemy core to prioritize core destroys.

5.10 Tier 10 — Economy & Targeting

1. If there is a known **axionite** or **titanium** ore that hasn't been routed
    - Priority is given to axionite if we currently have no axionite, else we prefer titanium over axionite. See 3.4 for heuristics.
2. If there is a **stalk target** nearby
    - **Stalk targets** are known nearby enemy bots, prioritizing ones that pose immediate threats, for example near ally transporters and are relatively close. When in the stalk state, the bot just moves towards the stalk target.

> Bots in the `EARLY_RUSH` mode skips over tier 8 entirely, unless it is routing for an **attack route**.
> Priority is given to stalk over ores if the stalk target is 2 moves away from us or if the stalk target is closer than the target ore.

5.11 Tier 11 — Exploration & Fallbacks

1. **Rush**
    - Only rush bots enter the rush state.
    - Until symmetry is known, we move to what we consider to be the next probable core location.
    - If symmetry is known, and we have seen the enemy core (confirmation), we scan a `20 x 20` region around the enemy core for available ores and foundries. If found, we pathfind to the source, build a harvester, and route back to the enemy core to place a turret
        - In the case of a nearby foundry, we will build a **breach**.
2. **Patrol**
    - Roughly one third of our bots move towards our existing known supply chains in order to "patrol" them
    - If we know there are only a few bots (`total unit count < 10`) all bots will patrol.
    - We will only patrol if we know we can afford a harvester (`BuildManager.can_afford_harvester()`) so economy becomes an earlier priority.
3. **Explore**
    - Healer bots will stay near the core.
    - Regular bots will pick a random location anywhere on the map and pathfind to it until the position is reached or if it is confirmed to be inaccessible.

> In hindsight, the `20 x 20` region for sources for attack routes was probably too small. We unfortunately had only seen a breach once during the grand finals, but unfortunately the

accompanying foundry was not routed properly so it did not fire. See figure 6.

However, we were still the only team to place a breach at all, which can be considered a success.

Figure 6: Breach in finals. Notice that it would have worked if the foundry outputted refined axionite.

Figure 7: Here, we went to an available ore, build a harvester, and routed to the enemy core for a core destroy.

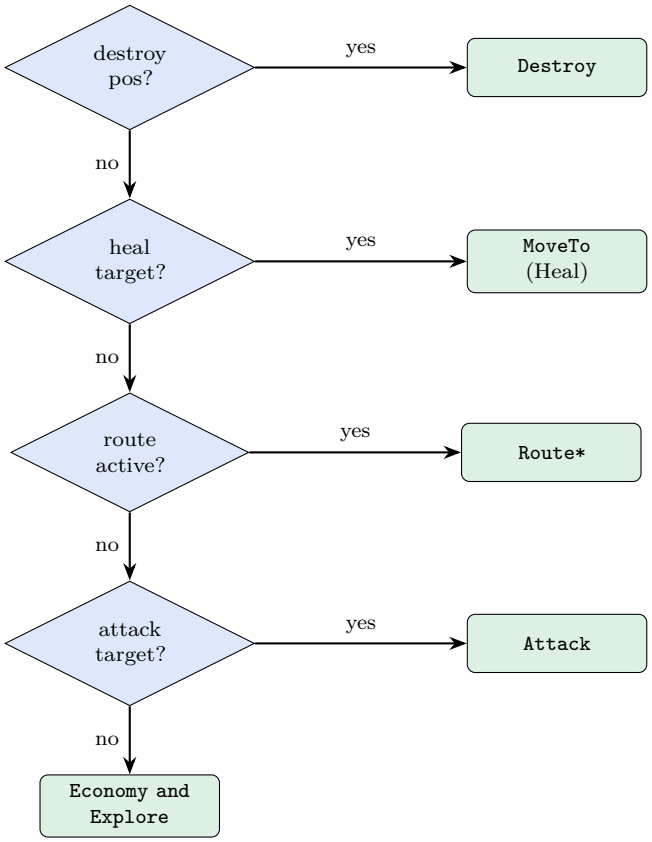


Figure 8: Simplified `determine_state` flowchart (expand with full tier detail — see table 2).

6 States

We have quite a few states overall in our bot. Many of which are redundant. Next time hopefully we will spaghetti code less.

6.1 Movement States

1. `StateMoveTo` - General movement state for stalking, patrol, healing, and exploration.

6.2 Combat States

1. `StateAttack` - This state focuses on moving towards a target. If we are on top of the target, we fire.
2. `StateDestroy` - Used to destroy ally harvester feeding enemy turrets (see 5.1).
3. `StateRush` - Used for moving towards sources nearby enemy core in order to start an **attack route** (see figure 7).

6.3 Conveyor Routing States

1. `StateBuildHarvester, StateRoute` - General routing. `StateBuildHarvester` moves towards target titanium ore (see 5.10), and once harvester placed it triggers `StateRoute`.
2. `StateBuildHarvesterAx, StateRouteFoundry, StateFoundryBuild` - Like previous but for axionite ores. `StateRouteFoundry` routes to a possible foundry location, **prioritizing being closer to the titanium source ore**. Once routed it enables `StateFoundryBuild` which waits in place to build the foundry.
3. `StateRouteBreach, StateBreachBuild` - `StateRouteBreach` is enabled when an attack route source has been determined (see 5.11), which then moves to a target location prioritizing being nearby the core. `StateBreachBuild` will first try to build a breach, but if not will settle for a gunner. `StateBreachBuild` will remain active until a breach is built.
4. `StateReroute` - When a mis-routed ally transport has been detected. Once reached and destroyed, it will trigger `StateRoute`.

> Our team was unique in choosing to build foundries farther away from the core and instead focusing on building them near the titanium ores. We did this to allow for branched out networks and to maximize efficiency (1 titanium ore v. 1 axionite ore).
> Surprisingly, we found little to no detriment with this decision. Enemy teams generally did not exploit our branched out structures.
> We tried enabling it so we can reuse foundries by routing more titanium and axionite, but the implementation was buggy and was called off.

6.4 General Building States

1. `StateBuildTurret` - General turret building state. Depending on nearby targets, will build a sentinel or a gunner.
2. `StateBuildTurretWrapper` - A wrapper for `StateBuildTurret`. Essentially it used pre-computed cases to make it faster.
3. `StateBuildGunner` - Specifically used to build a gunner
4. `StateBuildLauncher` - Specifically used to build a launcher
5. `StateBuildAdvancedShield` - Build barriers near harvesters where turrets are placed to stop enemies from building counter turrets.
6. `StateBuildShield` - Builds shields made out of inward facing (useless) conveyors around harvesters (see figure 4).

7 Cool Stuff

7.1 Pathfinder

We separate pathfinding into two categories: general movement and resource routing. In this section, "bitmask BFS" refers to BFS together with an algorithmic speedup from representing the BFS wavefront as bitmask, allowing fast expansion with bitshifts.

For general movement, we use a weighted search. The weights used were 1 for passable, 2 for passable but requiring placement of a road, as well as additional weight for being in sentinel and gunner range.

The main idea is this: tiles around the source should have higher resolution. Farther away nodes matter less, so we can assume they are uniform cost, setting their weight to 1 if passable and infinity otherwise (uniform costs allow the use of fast techniques like such as bitmasks[10]).

The algorithm works as follows. Firstly, a Dijkstra's is run from the source, early stopping on weight 10. This gives a connected subgraph of all nodes reachable within weight 10 from the source, call this collection of nodes the "frontier". Then, separate each node in the frontier by its first hop - i.e. the first move that the bot needs to take to reach this node in the least weight. This gives 8 classes. Then run 8 forward BFS's using bitmasks (seeding each with the frontier of a specific class, and stepping each BFS in parallel) until the target is hit. The first hop is returned along with the distance.[11] On an $n \times n$ map, this gives an asymptotically $O(k + n^2)$ algorithm, where k is some annoying-to-compute function of Djikstra stopping weight. On maps with large open spaces, which is common in practice, bitmasks yield a theoretical improvement of up to 64x (or whatever world size is), as 64 tiles can be expanded at once.

Resource routing uses the same fine-to-coarse idea. Starting at the source and tracking first hops, the search is expanded using only cardinal conveyor steps until depth 6. In addition, a fixed set of 16 *seed bridge offsets* - spanning up to distance 3 and including knight jumps such as (±2,±1) - are fired directly from the source to help the search reach over walls. All reached nodes after this process are collected. This local neighbourhood is then inspected for sinks (early return) before being enqueued as the seed frontier of a bitmask BFS. The main loop then expands each class using a pruned set of 8 *main bridge offsets* (±3,0),(0,±3),(±2,±2). The search is expanded with bitmasks until an accepting sink is reached, at which point the first hop is returned, along with the distance (in number of hops).

Resource routing pathfinding showing the seed and main bridge offsets. We use Jinja templates to generate 4 variations of the function. One of the options avoids adjacent titanium ores. This is useful for routing axionite lines such that they don't get clogged or mogged.

```
# generate 4 different find_bridge_route variants
{% for avoid_ti_adj, check_sinks in (0,0),(0,1),(1,0),(1,1) %}
{% set suffix -%}
    {{- '_avoid_ti_adj' if avoid_ti_adj -}}
    {{- '_check_sinks' if check_sinks -}}
{%- endset %}
# ---===
@classmethod
def find_bridge_route{{ suffix }}(
    cls, start: Position, sink_set: set[int], max_iter: int = 500,
    avoid_pos: set[int] = set(),
```

[10] Bitmasks can be used even with non-uniform costs, at a price increase. See Dial's algorithm.

[11] After implementing, I realised that this could be sped up by about 8x by having the bitmask BFS run in reverse from the destination until it hits a frontier node. However, it was working well enough and I got lazy and didn't want to mess with things.

```
):
    {% set main_bridge_offsets = [
        (3,0),(-3,0),(0,3),(0,-3),
        (2,2),(2,-2),(-2,-2),(-2,2),
    ] %}

    {% set seed_bridge_offsets = [
        (3,0),(-3,0),(0,3),(0,-3),
        (2,2),(2,-2),(-2,-2),(-2,2),
        (1,2),(2,1),(2,-1),(1,-2),
        (-1,-2),(-2,-1),(-2,1),(-1,2),
    ] %}
```

7.2 Markers

Markers were **integrah**[12] to our strategy and we used them to communicate important information.[13]

> "Each Builder's `Player` instance is a **completely isolated Python environment** - global/static variables are not shared. Markers are the only mechanism for inter-unit coordination."

...is what the docs say. As you might expect, it is wrong. There exist other methods of inter-unit communication, as pointed out by some integrah member of the Discord community: for example, one is able to use `os.nice` to increase the monotonic niceness counter, which is visible to all bots. There were many other exploits of questionable legality which were present in the game. Because both bots (yours and your opponent's) run on the same process, there is a way to use a CPython `bytearray` vulnerability to get access to raw memory and spawn nasal demons that dance on the battlefield and breach everywhere. We can attest that at least one team, **test**, tested this briefly, summoning demonic shapes.

For legal reasons we would like to clarify we *definitely* did not use any of these exploits in our final bot.

7.3 BuildManager & Cost Scaling

Resource costs scale multiplicatively with the number of buildings already constructed. The scaling contributions are:

[12](sic.) Integrah is an alternative spelling of "integral", well-known within the cambc (Cambridge Battlecode) community. We use it here not to confuse the reader but to remain faithful to cambc convention.

[13]We only use markers to communicate symmetry. This is important because our enemy core detection depends on it so if removed, it would completely break our bot.

Building	Base cost	Scaling contrib.
Road	1 Ti	+0.5%
Conveyor	3 Ti	+1%
Splitter	6 Ti	+1%
Bridge	20 Ti	+10%
Armored Conveyor	5 Ti + 5 Ax	+1%
Barrier	3 Ti	+1%
Gunner	10 Ti	+10%
Sentinel	30 Ti	+20%
Harvester	20 Ti	+5%
Launcher	20 Ti	+10%
Foundry	40 Ti	+50%
Breach	15 Ti + 10 Ax	+10%

Table 3: Building costs and scaling contributions.

The system protects a specific amount of reserve funds before allowing any new construction, acting much like a savings cushion to ensure it never completely runs out of money. The size of this cushion shifts dynamically based on the current stage of the game, generally requiring a larger safety net as time goes on to prepare for more demanding phases.

However, this rule is flexible and adapts to the financial health of the situation. If regular income drops too low, the system drops this cushion requirement to zero for basic supply structures such as Conveyors, Armored Conveyors, Bridges, and Splitters—so expansion doesn't completely stall out.
For these same structures, the cushion rule is completely ignored early in the game to allow for rapid initial growth, and the required savings threshold is automatically scaled down the closer the construction site is to a main base.

7.4 MarketMaker

Income Estimation

The system keeps a short memory of its recent financial history spanning the last several moments. It tracks the exact change in wealth from one moment to the next, taking the maximum value observed to determine its current capability.

- **Track History:** Maintain a rolling window of the last 20 rounds for resources.
- **Calculate Delta:** For every simulation tick, compute the instantaneous change:

$$\Delta\text{Wealth} = \text{Resource}_{\text{current}} - \text{Resource}_{\text{previous}}$$

- **Maximize Peak:** Scan the rolling window history array and extract the absolute peak delta:

$$\text{Income}_{\text{estimated}} = \max(\Delta\text{Wealth}_0, \ldots, \Delta\text{Wealth}_{19})$$

This rolling evaluation stabilizes the economy against sudden dips or temporary spikes.

8 Other units

8.1 Core

8.1.1 Burning

Our final[14] strategy for burning axionite was: Burn all[15] axionite for titanium before round 1000. It was a gamble, but the choice proved fruitful. The 1 to 4 conversion from axionite to titanium often gave a competitive edge with a surplus of titanium.

8.1.2 Spawning

All builder bots are spawned from the core. We spawn 4 bots at the start (3 economy, one healer). Whenever we can afford a builder bot along with what is known as a "required leftover," then we build a builder bot. It is scaled by the games scale ratio (not pictured).

Number Spawned	Required Leftover
<8	90
<10	100
<20	110
else	110+(10 × `UNIT_COUNT`)

Table 4: Required leftover titanium when spawning. Scales along with the game scale.

We will spawn in an emergency as well. If we notice 6 enemies near our core, or 2+ with us spawning over 9 round ago, then we will try to spawn a bot. If we have below 450 HP and lost HP in the last 1 or 10 turns, then we will also try to spawn a bot.

8.2 Launchers

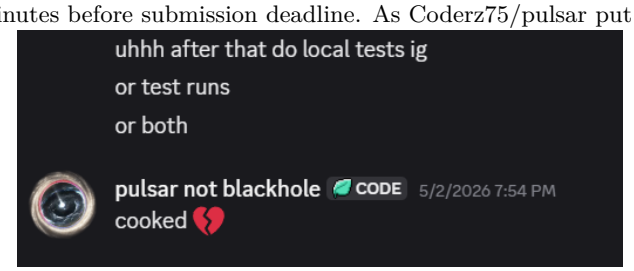
When throwing an enemy, the launcher scores all possible tiles so that:

1. Farthest away
2. Doesn't throw onto our own transporters and core
3. Penalties for launching next to an enemy launcher, and next to our own transporting building.

When throwing an ally:

1. Only will throw an ally if there is an enemy bot nearby it
2. Doesn't throw an ally off a tile that's already damaged
3. Scores for landing next to an enemy transporter
4. Penalties for launching next to an enemy launcher

[14]Decided roughly 10 minutes before submission deadline. As Coderz75/pulsar put it: cooked



cooked

[15]All but 17 axionite

5. Penalties for launching nearby an enemy bot.

See figure 5.

8.3 Sentinels

Sentinels will attack enemy buildings in the following order:

1. Avoid attacking harvesters that feed an ally
2. Prioritize enemy turrets
   (a) Prioritize ones that can shoot us
   (b) Prioritize gunners
3. Prefer non-roads
4. Prefer buildings over bots
5. If between bots then focus on the one with lower HP
6. Prefer buildings that are critical (can be one-shot)
7. Prefer targets where **multiple sentinels can attack**[16]
8. Prefer lower HP buildings.

8.4 Gunners

Gunner prioritizes in the following order:

1. Do not attack ally bots
2. Do not attack harvesters feeding ally turrets (such as itself)
3. Prioritize turrets
   (a) Prioritize those that can shoot us.
   (b) Prioritize gunners
4. Prioritize targets right next directly reachable[17]
5. Prioritize targets next to harvesters
6. Prefer current direction.
7. Prefer non-roads
8. Prefer non-bot tiles

See figure 3.

8.5 Breach

No comment here[18].

[16]This allows multiple sentinels to "chain" attacks and target one specific tile.

[17]Would not need to to turn nor destroy another target before that damaging the target in question

[18]To our chagrin, the devs did not buff breach despite multiple weeks of pestering. In fact, the devs had preemptively *nerfed* breach before anyone could use it, as opposed to Supercell[19]. Due to this, our priorities on this were minimal. Breach has a very measly algorithm that is "shoot as far as you can." This was factored into attack routes (see 5.11). See figure 6.

[19]Or the unit in XSquare's favorite song.

9 What Worked / What Didn't

9.1 What worked well

- **Precompute.** Computing so much about the map each turn allowed us to have our bot make very educated decision with the most up to date mapdata as possible
- We were the top team with the most second-place finishes in the entirety of the competition![20]

9.2 What didn't work

- **Precompute.** Computing so much every turn obviously had some issues, where especially on larger maps our bots started to TLE. We had solved it with the `TleManager` (see 3.6) but obviously this was simply inefficient in the long run.
- **Priority cascade brittleness.** Adding one new thing or changing one thing in the priority cascade led to the entire bot acting differently, which was a pain to work with.
- Pestering the devs to buff breach.[21]

10 Conclusions

The three of us all agree that this was the most fun programming competition we've all been a part of. In fact, we only met each other during this competition itself. Despite being the first year this competition has been ran, it was remarkably well thought out by the devs.[22]
Similarly we woud like to thank our goated competitors. Honorable mentions go to:

1. **Blue Dragon** - Despite not being eligible for the final tournament[23], the contributions this team had made to this game was invaluable for us
2. **Kessoku Band** for being such a powerful competitor for so long, we had fun with our games against y'all
3. **Pantheon/Oxford** for being the grand champions. Your bot was insanely powerful and we look forward to reading your postmortem
4. **Test** for all the exploits they found and made the tournament interesting
5. The **rest of the finalists!** and those teams who barely didn't qual. The competition was fun because of the players.

We're excited to see how this plays out in the future!

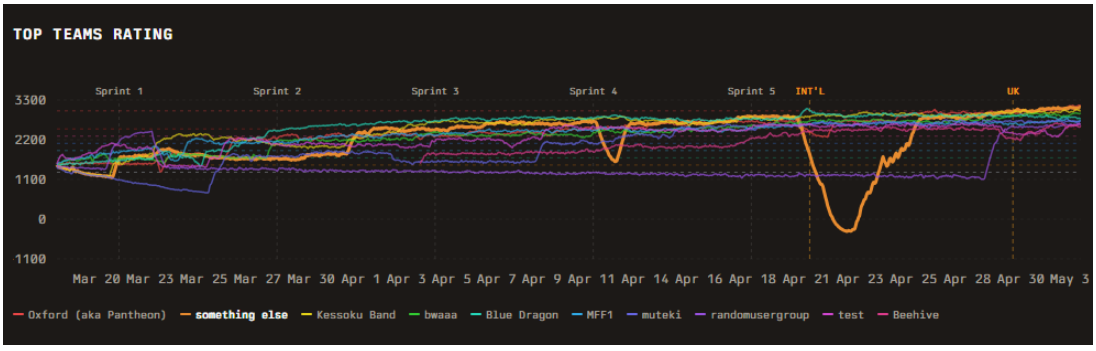


Figure 9: Us dominating both the top (and bottom) of the leaderboard

[20] Second place **sprint 3**, **International Qualifiers**, and **Grand finals**, with mention to **Lorem Ipsum**'s second place finish in **sprint 2**
[21] Believe me we tried.
[22] Despite our repeated defamation against them with regards to breach.
[23] Final tournament for students only