Cambridge Battlecode Postmortem

Team Pantheon

Derek Wang
University of Oxford

Yufan Wang
Georgia Institute of Technology

Alex Thummalapalli
Georgia Institute of Technology

Ioannis Pantidis
Delft University of Technology

Overview

We competed in the inaugural Cambridge Battlecode as Team Pantheon, consisting of Derek Wang, Yufan Wang, Alex Thummalapalli, and Ioannis Pantidis.

We entered the competition with mixed levels of experience. For three of us, this was our first year doing Battlecode. Ultimately, we placed first in the Grand Finals with our bot, Khaos. The final bot, along with our tooling, can be found in our GitHub repository.

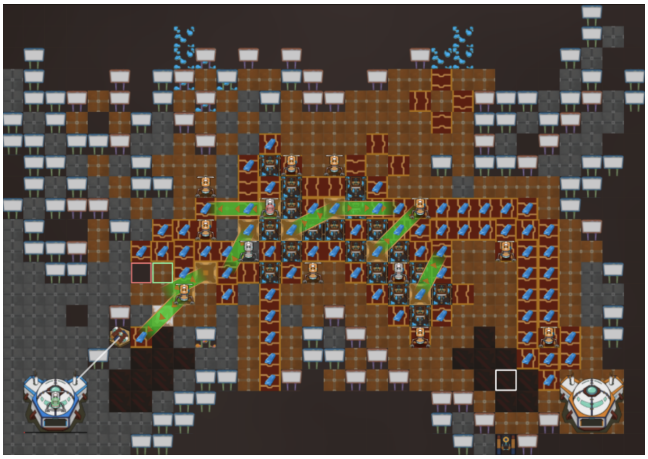


Figure 1: A Grand Finals match between Pantheon (gold) and something else (silver).

Game Overview

Cambridge Battlecode is a six-week competition in which teams design and develop algorithms to autonomously play a turn-based strategy game. This year's game involved balancing resource management, map control, and combat by dynamically routing conveyor lines across a 2D map.

Though it was inspired by MIT Battlecode, this competition featured some key differences; notably, a switch from Java to Python, new time limit constraints in place of bytecode counting, and a large jump in complexity.

Each match is played on a rectangular map, ranging in size from 20×20 to 50×50 and guaranteed to be horizontally, vertically, or rotationally symmetric. Each team starts with a single core unit that can spawn builder bots, each running an independent copy of the same program.

Though the map is initially composed of four basic tile types (empty, titanium ore, axionite ore, walls), builder bots can construct new infrastructure throughout the game such as conveyors, turrets, and roads.

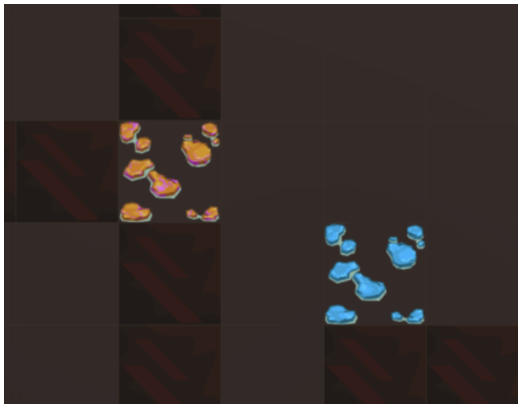


Figure 2: Map features: titanium ore in blue, axionite ore in orange, and walls in black.

Resources

Titanium is the primary resource and is used to build buildings, heal, and as ammunition for most turrets. Raw axionite is what is mined from axionite ore. Raw axionite and titanium can be combined to create refined axionite (typically referred to as simply "axionite"), which is a premium resource with more limited uses, although it can be converted back into titanium or used for tiebreaks.

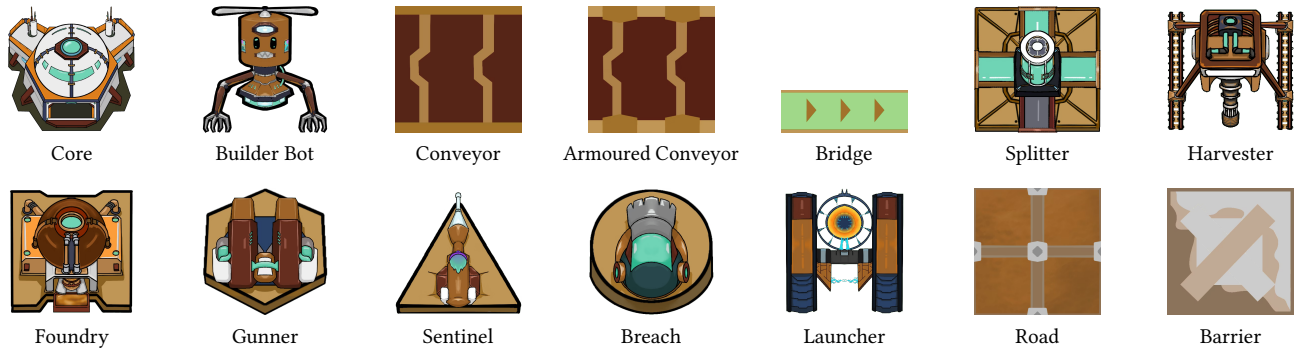


Figure 3: Game entities and buildings.

Core

Each team starts with a core, placed symmetrically on opposite sides of the map. The core is an immobile 3×3 building.

It can spawn builder bot units on one of its tiles once per turn, and it is walkable only by allied builder bots.

Resources must be delivered to the core before they become globally accessible to all units. The core also has the ability to convert axionite to titanium in a 1:4 ratio.

Builder Bots

The primary unit in the game is the builder bot. It is the only unit that can move or construct buildings.

Builder bots can move once per turn to an adjacent tile with a walkable building on it. They can also perform an action once per turn to heal an adjacent entity, construct a building on an adjacent tile, or attack a building on the tile they occupy. Builder bots can also freely destroy adjacent ally buildings without using up their action.

Builder bot vision extends to squared Euclidean distance 20.

Conveyors

- **Conveyors** send resources to the tile in front of them. They accept resources from the three cardinal non-facing sides.
- **Armoured conveyors** act like regular conveyors but have more health and cannot be damaged by builder bots. They cost both titanium and axionite to build.
- **Bridges** are more expensive than conveyors but have more output options: any tile within squared distance ≤ 9. They accept resources from any side.
- **Splitters** only accept resources from behind, but they send outputs equally to the three remaining cardinal sides.

Economic Buildings

- **Harvesters** can be built on titanium or axionite ore to harvest resources. They output resources every four turns to a cardinally adjacent building, like a conveyor or turret.
- **Foundries** convert titanium and raw axionite into refined axionite. They accept input from any side and will output refined axionite after consuming both types of input resources.

Turrets

Turrets are the primary way of attacking. All turret types except launchers require ammo to fire, meaning resources must be fed into them from a non-facing side. There are four types of turrets:

- **Gunners** are high-DPS, low-range turrets that can only shoot the first target in range. They can turn by spending titanium.
- **Sentinels** are lower-DPS turrets with a long and wide range. Unlike gunners, they can shoot any tile within their range.
- **Breaches** are very high-DPS turrets that deal splash damage. However, they can only use refined axionite as ammo, so they are seldom used.
- **Launchers** are utility turrets that can pick up and launch adjacent builder bots on either team to any walkable tile in their vision.

Miscellaneous Buildings

- **Roads** are cheap, walkable buildings. Since builder bots cannot walk on empty tiles, a road must be placed before that tile is passable.
- **Barriers** are cheap, high-health buildings that block builder movement.

Winning the Game

There are two win conditions: either destroying the enemy core or winning by tiebreak after 2000 rounds have passed. Tiebreakers are determined in order:

- Axionite collected
- Titanium collected
- Harvesters alive
- Axionite stored
- Titanium stored

Bot Summary

We went through several substantial iterations and even entire rewrites over the course of the competition.

Starter

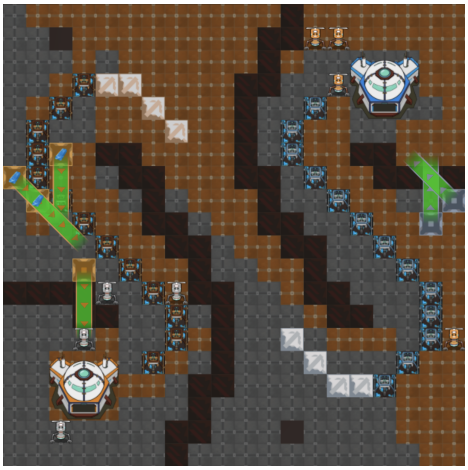


Figure 4: Starter (Gold) vs Starter (Silver)

We started with a basic economy bot with A* pathfinding, conveyor routing, and minimal attacking.

Artemis

Our initial baseline evolved into a launcher-centric bot named Artemis.

A dedicated builder bot would rush the enemy core using launchers to launch itself forward in hopes of disrupting their early economy very quickly. Then it would find a conveyor to attempt to attack and replace with a turret to destroy the enemy core. It would place additional launchers to fling away any enemy builder bot attempting to heal the tile the rusher was attacking.

Figure 5: Artemis (Gold) vs Starter (Silver)

Artemis also used a defensive launcher network both to protect our conveyors and harvesters and to help ally bots quickly move across the map.

Demeter

Figure 6: Demeter (Gold) vs Artemis (Silver)

After merging with Team Food, we bug-fixed Alex's bot and ported over our map info, pathfinding, and gunner logic from Artemis. This defensive bot solely focused on developing infrastructure, but also had dynamic turret placement logic that would repurpose our own conveyor lines if they were needed more for defense. However, we soon hit a plateau due to overcomplicated code and strategies and had to begin a rewrite.

Figure 7: Lethe (Gold) vs Demeter (Silver)

Lethe

Keeping only our map info and pathfinding modules, we fully rewrote our bot with a memoryless architecture. The idea behind the memoryless system was to have our bot recompute information every turn based on global map knowledge, rather than retaining stale strategies or states. This allowed our bot to be highly dynamic and make much more optimal decisions each turn, and we finally started winning games against top teams like Blue Dragon.

Hades

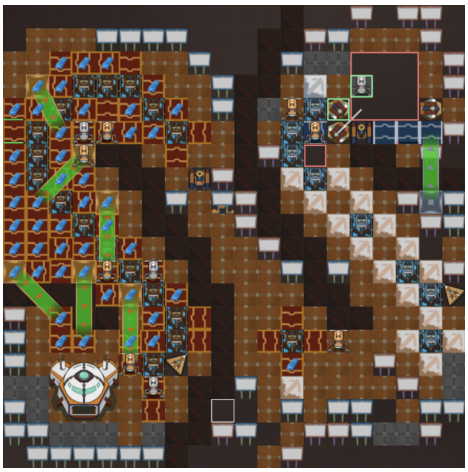


Figure 8: Hades (Gold) vs Lethe (Silver)

After the US Qualifiers, we added many small improvements and partial rewrites to Lethe. We also revamped our coordination system, which previously used markers for bot coordination, since it was being disrupted by other teams placing roads everywhere and blocking marker placement spots. The new system used factors like closest builder to a target tile to decide whether to pursue a task.

Khaos

Figure 9: Khaos (Gold) vs Hades (Silver)

A day before the final submission deadline, we merged in a brand new chokepoint detection system and attack route strategy that greatly improved our win rate. We named this bot Khaos due to its unpredictable and dynamic nature, which made it difficult for opponents to counter.

Strategic Overview

The rest of this postmortem documents how Khaos works. We'll start with general strategies, then cover technical implementation from the map info layer through pathfinding, states, turret logic, and runtime optimization.

Memoryless Architecture

Our bitmasked map info layer is fast enough for us to recompute key decision-making and planning logic from scratch each turn. Rather than defining state transitions based on previous states, we score seven candidate states each turn based on current map knowledge and execute the highest scoring state.

The intent is for long-term behavior to emerge from every builder acting optimally on its current local knowledge. This makes each bot's actions much more dynamic, since they don't persist stale goals and can easily adapt to the actions of other bots, newly discovered information, etc.

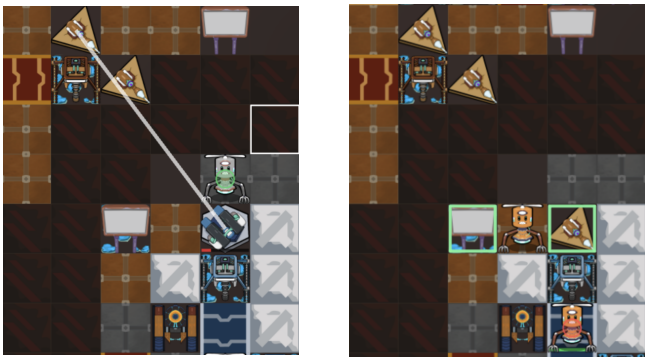
Adversarial Strategies

Although these territorial strategies are not hard-coded, they emerge from our state management and scoring system and are worth highlighting.

Ore hopping (Figure 10) is our strategy of using ally-controlled harvesters as staging points for attack turrets. By placing turrets along the frontier of our controlled territory, we can shoot the defenses around nearby opponent harvesters. Our builder bots can then take over the enemy harvester by building more turrets around it.

Ore hopping is a consistent way of pushing our front line forward, since slowly taking over enemy harvesters bolsters our economy and increases map control. This often will create turret combat along the frontier.

The only downside of ore hopping is that it is highly dependent on ore spread across the map. If ore clumps aren't spread out enough, a turret placed at an ally ore cannot reach an enemy ore, and it is harder to hop and meaningfully expand our territory.



Before After

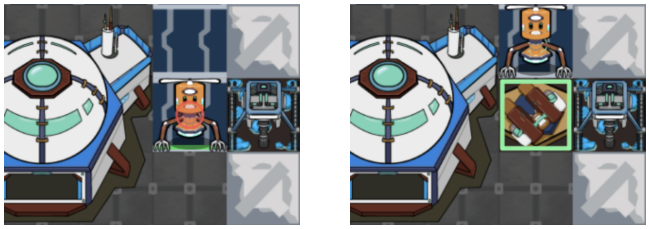
Figure 10: Ore hopping.

Seeding (Figure 11) is a more aggressive way of gaining territory. This involves builder bots intercepting an enemy conveyor line by attacking it until an allied "seed" turret can be placed in the gap. The analogy is that much like seeding a plant, our infrastructure can grow from a single turret investment as it destroys enemy targets around it and frees up space for more of our turrets and conveyors to be placed.

Seeding is easily countered by enemy builder bots simply out-healing the attack if nearby, though there are special cases described later where this can be prevented.

Map Info

The map is at most $W \times H \leq 50 \times 50$ tiles, so we can fit binary information about tiles within a Python big integer of at most 2500 bits. We encode information about tile (x, y) in bit $n = x + yW$. By using big integers for various properties (whether a tile is a wall, if we've visited a tile, etc.), we can quickly query and compose spatial information about the map with C-level bitwise operations.



Before After

Figure 11: Seeding.

We frequently use geometric translations of a set of tiles by using the integer shifts $\ll 1, \gg 1, \ll W, \gg W$ to shift a tile set east, west, south, north respectively. These geometric translations are useful for shifting bitmasked frontiers, computing neighborhoods/zones, etc.

Overall, `map_info` maintains around thirty derived masks to store information about:

- Environment/Entity type
- Facing direction
- Current/previous visibility
- Resource flow
- Danger zones
- Placement legality

We can combine these masks to query various relevant subsets of the board. For example, an avoid mask can be quickly computed by ORing walls, enemy turret threat masks, and some buildings.

Values that cannot be a single bit are stored in Python lists indexed by n: building id, entity type, hit points, facing, conveyor output tile, and a reverse-adjacency bitmask of every conveyor-type building that outputs onto n. That reverse adjacency of the conveyor graph allows for very fast conveyor flow related calculations.

Updating Vision

`update_at(pos)` calls the cambc controller to update our knowledge of a specific tile. It is called for every tile in our vision. We also call it on relevant tiles after we build something or when new tiles enter our vision after moving.

Since this method is called so frequently, we only update relevant info to avoid wasting runtime on unneeded controller calls:

(1) Core-area tiles are handled by separate core logic; only core health is refreshed each turn and the function returns early.

(2) Symmetry tracking runs once per newly-seen tile. Observing a tile that contradicts a symmetrical counterpart clears that symmetry flag.

(3) Walls can never hold a building and never change, so if the observed tile is a wall, the function returns before the building lookup.

(4) Otherwise, we check the building id. If it is unchanged from last turn, we only refresh health and stored resources, as well as facing direction for gunners (the one unit that turns).

(5) If the building id changed or was cleared, we update all of our bitmasks with the new entity info and bump the structural version counter so any cached calculations that rely on map info are marked stale. For markers, we add their decoded message into the message queue.

Predicting Symmetry

Since maps are guaranteed to be either horizontally, vertically, or rotationally symmetric, we look for tiles that contradict each possible symmetry until there is only one remaining candidate. We then use the determined symmetry to set the enemy core location and update our knowledge about unseen tiles:

```
if (hor + ver + rot) == 1 and not solved_sym:
    solved_sym = True
    # update enemy core location
    their_core = flip(my_core)
    # mirror seen tile envs to unseen half
    for n in seen_tiles:
        f = flipped_index(n)
        if not (seen & bit(f)):
            copy_env(n → f); seen |= bit(f)
    struct_version += 1
```

If there are two or more possible symmetries, we still commit to a prediction on where the enemy core is. We default to assuming rotational symmetry unless it's already been eliminated, in which case we predict the closer of the horizontally or vertically symmetrical core locations, with the tiebreak remembered so the prediction doesn't oscillate between turns. This predicted location is used to predict ally vs enemy territory and for attack routing.

Memoization

We memoize complex bitmasks based on a struct version counter that is incremented on structural changes to the known map (wall discovered or building discovered / placed / destroyed).

For some masks, we also add our observed resource bitmasks to the cache key so that the masks are recomputed as resources flow:

```
cache_key = (_struct_version,
             _bm_conv_ti,
             _bm_conv_raw_ax,
             _bm_conv_refined)
```

Derived Masks

We use our basic map info masks to derive more complex masks for decision making.

Enemy turret threat: We compute both a soft mask (enemy sentinel attack tiles) and a hard mask (enemy gunner / breach attack tiles) to guide builder decisions.

We purposely never step within hard mask tiles due to the higher DPS of gunners and breaches. We allow stepping into soft mask tiles in our pathfinding, just at a higher cost.

To construct these masks, we precompute the attack patterns of sentinels and breaches for each facing direction, then apply these patterns for turrets facing each direction:

```
for d in 8 directions:
    turrets_d = enemy_turrets & dir_mask[d]
    for (dx,dy) in OFFSETS[turret_type][d]:
        acc |= shift(turrets_d & col_mask, dx + dy*W)
```

Since gunners can only shoot the first tile in front of them, we check their facing direction one step at a time, ignoring any tiles past a blocking wall.

The hard and soft masks are memoized on `_struct_version` to minimize recomputation.

Turret feed propagation: We construct a bitmask to mark which tiles are properly fed. We propagate whether a tile is fed from harvesters, foundries, and conveyors with visible resources on them up to 4 steps downstream (based on output tiles). This bitmask is used for turret placement logic to only consider tiles with a source of titanium or refined axionite feeding them.

Route targets and dead ends: One pass walks the conveyor graph backward from the core area via the reverse adjacency, in layers, recording both the reached set and a topological-ish visit order. A second pass then overlays loaded and visible state:

```
reaches_core, order = route_reaches_core()
# walk in core-ward order
for n in order:
    rl = (run_len[target(n)] + 1
          if loaded(n) else 0)
    run_len[n] = rl
    if rl == 4:  # 4 in a row: line jammed
        mark 4 tiles from n unroutable
# a *visible* 4-run jams the whole chain:
# flood both up- and downstream through
# my conveyors
route_targets = core_area | (reaches_core
                & ~unroutable
                & ~guard_conveyor)
```

If we observe 4 loaded conveyors in a row, we consider that conveyor line unroutable, and ignore it in our Dijkstra's for conveyor route planning. This approximates max flow since we don't want to connect more harvesters to a route that is already full.

As a side effect, the same pass sets a dead-end mask (targets of loaded conveyors whose output is not an accepting building, plus conveyors pointing into enemy hard buildings) and a feeding-enemy mask (loaded conveyors whose target is an enemy turret). The route state attempts to route dead-ends and feeding-enemy conveyors to routable targets.

We also have a visible-graph check for whether a turret could possibly still be fed. This check is used by turrets to decide self-destruction validity: any unseen adjacent tile or off-screen upstream feeder keeps the answer `True`, so a turret is only allowed to self-destruct when the visible local structure *proves* it is not currently supplied.

Carrying resources: We also maintain a bitmask that expands three steps up and downstream from a resource-carrying conveyor. We maintain separate bitmasks for each resource type so we can check if a tile could carry a particular resource.

Avoid pathing: We also maintain a mask of tiles we should avoid when pathfinding. Our universal avoid mask includes walls, non-walkable buildings, tiles around enemy launchers, and tiles covered by enemy turrets. We can also compute avoid masks that include ally conveyors (so we don't change existing infrastructure when planning a new conveyor route), enemy roads adjacent to enemy bots (since they are easily healable), chokepoint barriers, and non-landlocked ore in the harvest zone.

We also have an enemy POV mode to calculate avoid masks with inverted perspective (shows how the enemy would perceive tiles to avoid).

Communication

Since robots are sandboxed, all communication must be relayed via markers. A unit may place at most one marker per turn carrying a `u32` value.

Payload Format

Earlier iterations also used markers to communicate task claims between builders and to use launchers to speed up builder bot movement, but those were retired in Hades. The symmetry-broadcast payload repurposed the lower bits as a 7-bit *corresponding-position* code, a 21-bit environment sample, and 3 symmetry bits.

All field widths and shifts are module constants and the encode/decode pair is a straight bit-pack:

```
def encode(target, type, sym, sender):
    return ((target & POS_MASK)
            | (sym    << SYM_SHIFT)
            | (sender << SENDER_SHIFT)
            | (type   << TYPE_SHIFT)) ^ key
```

The final XOR with `key` encrypts our markers. We initially added this as a joke. However, we kept it in to make it harder for teams to decode markers in replays to figure out how our comms system works.

Symmetry Broadcasts

Although the primary purpose of this marker type is to share map symmetry, we also use it to share map knowledge about distant parts of the map.

We first bucket the map into 5×5 cells. Since map dimensions are at most 50×50, there are up to 100 buckets, and so we can communicate which bucket we are sharing info about with 7 bits. We then use 21 bits to communicate binary information about 21 tiles within the bucket (all tiles within squared radius ≤ 5 from the center of the bucket). If the marker is placed on an even x coordinate, we communicate whether or not the tile environment is a wall. Otherwise, we communicate whether or not it is a titanium ore. This results in each type being communicated by roughly half of our markers.

Any ally bot that reads the marker then uses the learned info to update the corresponding `_bm_env` entry and `_bm_seen`:

```
def apply_symmetry_broadcast_map(marker_pos, bucket, bits):
    env = ORE_TI if marker_pos.x & 1 else WALL
    for i in set_bits(bits):
        dx,dy = LEARN_MAP_OFFSETS[i]
        note_env(bucket.x+dx, bucket.y+dy, env)
```

Walls are the more valuable of the two payload types, since knowledge of impassable tiles helps our pathfinding and

chokepoint detection logic. Titanium ore observations don't affect these since ore is passable, but it does help for more efficient harvesting since we know where harvesters should be placed before exploring the map.

Overall, our communication system is supplementary only. Bots don't depend on communicated info, but they can benefit from using markers to learn more of the map.

Pathfinding

We run a bitmasked version of Dial's algorithm to find the optimal path to a given target. By changing the allowed steps and cost, we can use this both for builder bot movement and conveyor route planning.

The search is run in reverse, from target tiles to start, so that path reconstruction is not necessary, as it can simply find the lowest distance-to-target tile it can move onto.

Dijkstra's for Movement

For movement, we define four different edge types: a normal step (cost 1), a step destroying an ally barrier (cost 15), a step into a soft threat (cost 20), and both (cost 35).

We consider any walls or non-walkable buildings to be impassable. We do not mark builder bots as impassable to prevent oscillations in our route path as bots move around. Instead, we only check occupancy at the end of our Dijkstra's, when considering the move we would make this turn.

We then use a circular array of frontier bitmasks as a bucket queue, starting from a mask of our target tiles and iterating until we reach a tile we could move onto this turn:

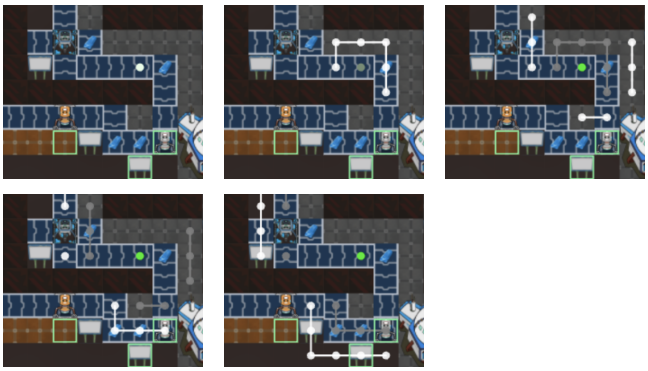


Figure 12: Dijkstra frontier expansion over five iterations.

```
# seed the four cost classes into the bucket
frontier[0]           = target & no_barr & no_thr
frontier[bcost]      |= target & barr    & no_thr
frontier[tcost]      |= target & no_barr & thr
frontier[bcost+tcost]|= target & barr    & thr
i = 0
while True:
    cur = frontier[i % cyc] & ~visited
    visited |= cur
    # reached a valid neighbor
    if cur & can_move_to: break
    nbr = expand_chebyshev(cur) & ~visited & ~avoid
    frontier[(i+1)    % cyc] |= nbr & nb_nt
    frontier[(i+1+bc) % cyc] |= nbr & b_nt
    ... (3 more cost classes)
    i += 1
```

When a frontier bit lands on a valid neighbor of the bot (that is, a neighbor we can step onto right now), that direction is chosen, after a four-stage tiebreak among reachable neighbors:

(1) **We prefer walkable tiles** so we don't waste our action cooldown on building a road.

(2) **We skip tiles next to borders and friendly bots** up to Chebyshev 4 if possible. Avoiding edges is good for maximizing vision. Avoiding builder bots helps prevent clumping or situations where we accidentally step onto another builder bot's target tile.

(3) **We prefer walking in zigzags.** If we walk along the same diagonal for two steps, we prefer the opposite diagonal next. The intuition is that zigzagging gives us additional vision coverage.

(4) **We prefer diagonals over cardinals**, again to maximize vision coverage.

We also break ally barriers and place roads as needed when moving onto a tile. We will try to replace any barriers we broke at the start of each turn.

Dijkstra's for Routing

For routing, we define two different edge types: a conveyor step (cost 1) and a bridge step (cost 6).

When building our target tile mask, we allow for ending on existing conveyors, rather than the core itself. However, we add a cost of 4 to loaded conveyor tiles, to bias our route planner to form new conveyor lines unless the existing line is significantly better.

Since we cannot store parent pointers when running Dijkstra's, we still need to calculate the desired output of the first conveyor or bridge we should build. We do this by testing

each possible output to see if it exists in the matching prior cost layer to our current tile.

We use different target tile sets when attack routing to the ring around the enemy core or routing to a foundry.

Conveyor Cost

We calculate the titanium cost of the optimal conveyor route to decide whether we should start building it now. This is to prevent builders from starting routes from very far away if we are low on titanium.

We calculate the cost using this formula:

$$\text{cost}(d) = \left(3\,d\,s + 0.015\,d(d-1)\right) \cdot 0.65,$$

for path length d at current scale s. This is an arithmetic series formula that lets us calculate the base $3ds$ cost of building the conveyors, with an additional $0.015d(d-1)$ term to account for the increased cost of each conveyor as scaling increases.

The 0.65 constant is an arbitrary discount we apply to the total cost. We don't want builders to be too conservative about starting routes back to the core, since we gain titanium over time (from passive income every four turns, or other existing routes).

BFS for Closest Target

We used Dijkstra's for navigation and routing since we wanted to account for multiple edge weights. However, we use a much faster unweighted BFS when we want to find the closest target for action planning.

This frontier BFS mimics builder bot movement by only expanding passable tiles, minus tiles we want to avoid (such as tiles adjacent to enemy launchers). This is used when finding the closest ally building we could heal, the closest enemy we could chase, etc.

The `closest` function also exposes a `side` flag that lets us optionally run the BFS from the enemy's point of view. This is used to determine how far the enemy is from a target, which helps us reason about whether or not we could reach that target first.

Floodfill for Voronoi Partition

Each builder scores its states independently, so an explicit mechanism is needed to stop builders from competing over tasks. We use a claim-partition pass that performs competing simultaneous Chebyshev flood-fills from the bot versus all other friendly bots and returns only the candidates the bot reaches first:

```
my_front, other_front = my, others
while remaining_claims and (my_front or other_front):
    my_front    = expand(my_front)    & passable & ~claimed
    claimed |= my_front; remaining &= ~my_front
    other_front = expand(other_front) & passable & ~claimed
    claimed |= other_front; remaining &= ~other_front
return my_claimed & candidates
```

Stuck Detection

If more than 2 + (id mod 8) turns pass without movement progress, builder bots will move to a random adjacent tile to get unstuck. The (id mod 8) term adds some variance based on builder bot id so that this behavior doesn't inadvertently sync up across builder bots.

The Core

The core is mainly responsible for spawning builder bots. It will also convert axionite to titanium as needed.

Spawning Bots

The spawn policy has multiple checks:

```
if not has_ally and closest_enemy:
    spawn_on_core_tile_nearest(closest_enemy)
else:
    baseline = 400 if allies ≥ 12 else 200
    if titanium > baseline + scaling:
        spawn_toward_plan() or spawn_toward_center()
```

The first four builders are spawned in different directions for better exploration coverage at the start of the game. After that, bots are spawned toward a visible enemy if there is no ally nearby as a defensive mechanism. Otherwise, we wait until we have enough titanium and then spawn allies toward the center.

Initial Spawn Plan

The core orchestrates initial exploration by spawning the first four builders on the tile facing the direction they should explore.

To find the best directions, we first check which of the eight possible directions should be considered. For each direction, we check the ray pointing in that direction. If the ray's endpoint is out of vision, or there is titanium along the ray path, we consider the direction valid, since there is value in sending a builder bot in that direction.

From the set of valid directions, we pick the four that maximize the product of pairwise angular separations, to spread the directions out:

```
best = argmax over 4-subsets S of
       prod(angular_dist(a,b) for a,b in pairs(S))
```

We tiebreak by preferring diagonals if there are multiple sets of directions with the same pairwise angular separation, since diagonals give us better vision coverage.

Once we have the four initial directions, we spawn bots in order of their direction's proximity to the map center, since that is more likely to be a contested region and we want to get there before the enemy does.

Converting Axionite

We use the following logic for axionite conversion:

```
if round < 1500 and titanium < 4 * harvester_cost:
    convert(clamp((3*harvester_cost - titanium)//4))
```

This allows us to convert axionite to titanium when we are running low, up until turn 1500, since accumulating axionite for the tiebreaker is important in long-running games that might reach round 2000.

Builder Bots

Builder bots do the following each turn:

(1) **Update map info** using the cambc controller and any learned map info from ally markers

(2) **Rebuild broken barriers** if we destroyed any previously for movement

(3) **Score states** and run the best state

(4) **Place a launcher or barrier** if we detect a nearby chokepoint

(5) **Heal an adjacent building** and fall back to healing ourselves otherwise

(6) **Destroy enemy feeders** if adjacent

(7) **Try spamming roads** if empty tile adjacent

(8) **Place marker** to broadcast symmetry and map info

An enemy feeder is an ally conveyor, bridge, or harvester that outputs onto an enemy turret. For harvesters, we also check that there are no ally turrets before destroying it.

When spamming roads, we prioritize placing roads adjacent to ally conveyors. We also place roads around enemy bots or buildings if an enemy is nearby. Road spam helps us with map control since enemies have to destroy the roads first before they place their own buildings. We do not spam roads in front of our gunners to keep their firing path unblocked.

Persistent Builder State

There are a few exceptions to our memoryless architecture. We maintain TTL caches for each state so that builders will not reconsider target tiles that failed recently. We also change the behavior of the first few builders:

The very first builder spawned will restrict its targets to squared distance ≤ 100 of the core, so it stays near the core as a guard.

The first four builders will check the tile they spawn on to decide their initial explore direction (see Initial Exploration). They will clear this target upon arrival or after a 30-round timeout.

States

Builder bots have seven possible states they can run. Every state exposes a `score()` function that uses global map info to calculate state priority and return a value strictly below a fixed ceiling (shown in Table 1). We then run the logic for the highest scoring state.

Table 1: Builder states by score ceiling.

State	Max	Role
attack	9	build sentinel / gunner
heal	8	repair / chase invaders
route	7.75	route resources back
secure	7.5	surround an ore tile
harvest	4	build a harvester
disrupt	2	barrier enemy-side ore
explore	1	explore unseen tiles

We calculate state scores in descending order of the max possible score for that state. This avoids having to compute `score()` on lower-ceiling states that cannot win once we've found a provably better score:

```
for state in states_desc:        # sorted by MAX_SCORE
    if best_score ≥ state.MAX_SCORE:
        break
    s = state.score()
    if s > best_score:
        best_score, best = s, state
```

State Pattern

Each state follows the same pattern. The `score()` function derives candidate tiles for the state (such as buildings we can heal for heal state). We intersect the candidate tile set with a Voronoi claim partition to avoid multiple builders contesting the same target tile. We then cache the result for `run()` to use if the state ends up being chosen.

Attack (ceiling 9)

Attack is our highest priority state where we place turrets to target enemy buildings. We prefer placing turrets on empty tiles or less important infrastructure (roads, guard conveyors, etc). We will fall back to placing on non-guard conveyors/bridges or enemy infrastructure we could damage, but at a higher threshold.

Figure 13: Attack.

Bit-Sliced Score Planes

For each facing and turret type, we maintain a score plane stack: a list of 9 bitmasks where bit i of the integer score at tile n is `(planes[i] » n) & 1`. This lets us do bulk score updates and queries over the whole board with bitwise parallelism, rather than iterating over an array.

To add a constant c to the score of every tile in a mask M, for each set bit of c we XOR the corresponding plane with M (toggling that digit at every tile in M simultaneously) and ripple any AND-carries up through higher planes:

The key property is that the cost of calculating scores is independent of the number of tiles scored. The per-digit loop is bounded by the fixed plane count, and each iteration is one big-integer AND/XOR over the whole map. This is what makes exhaustive placement evaluation fit within our per-turn budget.

Scoring Tiles

We group enemy building types by equal score so one masked add handles a whole score class. For each of the eight facings, the sentinel pass shifts each enemy-type mask by the reverse of every sentinel attack offset under that facing and ANDs with the valid-placement mask. Wherever a bit survives is a tile from which a sentinel facing d would hit that enemy, so we add the type's score into the planes there.

The advantage of this approach is that we never iterate over individual placement tiles. The total time scales with the number of hit offsets, facings, building types, and score bits, rather than with the board size.

Gunner scoring follows the same idea along a directional ray of up to three steps. The score is discounted by 0.9^k at step k, the ray is blocked by walls and our own non-road buildings, and the core is single-counted at its closest hit. This step discount prioritizes placing gunners directly adjacent to targets, since they can't shoot over tiles like sentinels can. We also add a rotation bonus equal to the sum over all eight facings shifted right by 3, to still favor targets we might rotate toward later.

We apply a penalty to placement positions under threat by enemy turrets. We also apply a penalty to placements that would destroy an ally non-guard conveyor or bridge. All of this is memoized per round keyed by the structural state, enemy presence, and placement masks.

Filtering Tiles

The candidate pass returns the legal placement mask per facing direction. We seed from tiles fed by titanium or axionite, plus the Manhattan dilation of titanium harvesters and foundries not already used by one of our sentinels. Sentinels consume more ammo than a harvester can provide, so we avoid building multiple around the same harvester.

We then intersect with seen and buildable tiles, and remove:

- Tiles within a two-step enemy-bot BFS through their passable graph. We treat our launcher 3x3 zones as impassable for enemies, since an enemy entering one is flung back.
- Our own turret feed chains, so a new placement never cuts off an existing turret.
- Per-direction loader blockers, since feeders cannot load a turret pointing into them.

Sentinels additionally exclude tiles within enemy gunner or breach range, since they shoot slowly and are unlikely to win the trade. Gunners are permitted on those tiles to enable harvester combat.

We only consider placement tiles whose scores pass a threshold. Sentinels and gunners have different thresholds, since they have slightly different scoring systems. The threshold is also higher for enemy infrastructure we would have to destroy first before placing, or ally infrastructure we don't want to destroy (non-guard conveyors and bridges).

Placing Turrets

If we can place a turret on a preferred tile this turn, either from our current tile or an adjacent tile, we pick the highest scoring such placement. This is to favor turrets we can place this round over potentially better placements we would only reach in a later round.

```
# Add the constant c (c = sum of 2^b for b in `bits`) to the stored score of EVERY tile in `mask`,
# in O(NUM_PLANES) whole-board big-integer ops -- cost independent of how many tiles are in `mask`.
def add_bits_to_planes(planes, bits, mask):
    for i in bits:                          # for each set binary digit of the constant c
        carry = planes[i] & mask            # tiles where digit i overflows
        planes[i] ^= mask                   # add c's i-th digit to all those tiles at once
        j = i + 1
        while carry and j < NUM_PLANES:     # ripple the carry up through higher planes
            new_carry = planes[j] & carry
            planes[j] ^= carry
            carry = new_carry; j += 1

# Bitmask of tiles in `cands` whose stored score ≥ thr, computed in one MSB-first pass.
def ge_threshold_mask(planes, thr, cands):
    eq, gt = cands, 0
    for i in range(NUM_PLANES - 1, -1, -1):
        p = planes[i]
        if (thr >> i) & 1: eq &= p          # thr digit set: tile must match to stay "equal so far"
        else: gt |= eq & p; eq &= ~p        # thr digit clear: a set score digit => strictly greater
    return gt | eq
```

Figure 14: The two main bit-sliced primitives. `add_bits_to_planes` adds a score constant to an arbitrary set of tiles in O(planes) big-integer operations regardless of how many tiles are in the set. `ge_threshold_mask` thresholds the whole board against a scalar in one pass from the most significant bit down.

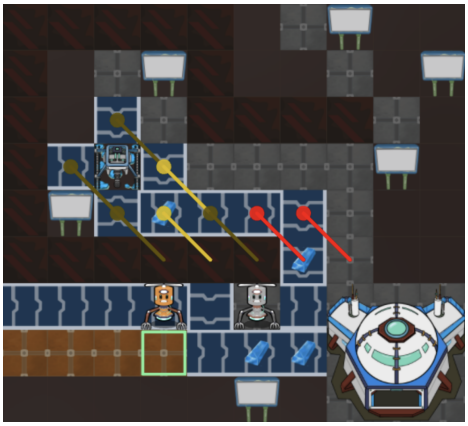


Figure 15: Valid turret locations. Gunner in red and sentinel in blue.

Otherwise, we pick the closest preferred or fallback tile. Among candidates within two BFS steps, we maximize

$$\text{enemyBFS}_{\text{adj}}(\text{cand}) - \text{myBFS}(\text{cand}),$$

treating our launcher zones and barriers as enemy-impassable.

If the target is an enemy conveyor that a visible enemy could heal, we scan adjacent buildable tiles for a launcher or barrier placement that strictly increases the enemy's BFS distance to the conveyor without worsening our own approach. Barriers require a delta of at least 2; launchers require at least 5, reflecting their relative cost.

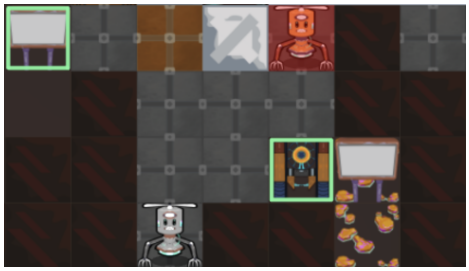


Figure 16: This launcher prevents the enemy from advancing, as the launcher's 3x3 and the barrier form a wall

We normally avoid damaging enemy conveyor targets if a nearby enemy could heal them. However, we have three exceptions:

- We locally outnumber the threat (at least 2 ally builder bots within squared distance 25 versus a single enemy within two BFS steps).
- An allied sentinel is in sight, so we damage more than they can heal anyway.
- The target's 3x3 is fully sealed by walls, non-walkable buildings, or our launcher 3x3s, so enemies cannot reach the target tile.

Heal (ceiling 8)

The heal state defends against opponent attacks in two ways. First, we heal damaged ally buildings adjacent to us. Second, we chase enemy bots that wander into our conveyor zone before they can attack our infrastructure.

Score returns 7 if any friendly building has more than 2 damage, or if a friendly sentinel and enemy sentinel are within each other's attack ranges. We treat these sentinels as very damaged so we stay near it as they trade damage with the enemy sentinel. Score also returns 7 if a chase target exists within the conveyor zone, defined as within 8 pathing steps of a conveyor. We score 2.5 if a chase target exists but is farther.

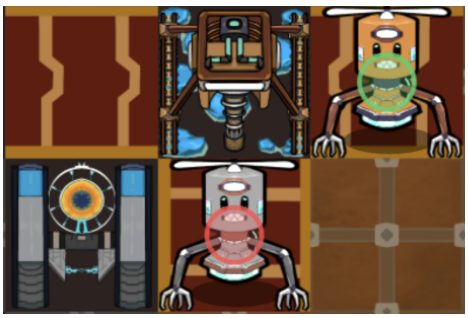


Figure 17: Heal.

At the end of each turn, we scan adjacent tiles and heal the one maximizing damage times a fixed type priority. We consider all very damaged buildings before less damaged ones, and consider six priority tiers:

(1) Core

(2) Turrets

(3) Armored conveyors, harvesters, and foundries

(4) Regular conveyors

(5) Barriers, bridges, and splitters

(6) Roads

For chase target selection, we remove enemies under attack by a turret and enemies that another friendly bot is strictly closer to. We then prefer the closest surviving target.

When chase fails on pathing, the launcher fallback fires as a counter-launcher response:

```
for enemy on a damaged friendly conv (unclaimed):
  # T1: placeable cell adjacent to enemy
  T1 = adj(enemy) of those four types
  # T0: a stand-tile within 8 BFS steps of T1
  T0 = adj(T1) reachable within 8
  pick (enemy, T1) minimizing my dist to T0
```

This most commonly happens when an enemy places a launcher to keep our bot away from its attack target. In this case, we place a launcher in response to keep their bot away. We will place a gunner instead if the target tile is one of our guard conveyors.

Route (ceiling 7.75)

The route state builds conveyor lines from harvesters back to the core. It tries to route from dead-end conveyors, orphan harvesters, and orphan foundries. A structure is an orphan if it hasn't been routed back yet and has at least one cardinal neighbor we can build on.

Score returns 5 normally, but 7.75 (above heal's 7) when any claim is important:

```
important = expand_chebyshev(enemy_bots, 5)
          & ~(my_harvesters | my_foundries)
          | feeding_enemy
return 7.75 if important & claims else (5 if claims else 0)
```

We will place armoured conveyors instead of normal conveyors if we have axionite available and at least 5× the cost in titanium.

If we don't have enough to continue routing, but there is an enemy nearby, we place a barrier on the dead-end's output, or the dead-end conveyor itself if needed, to prevent enemies from taking advantage of it.

When placing conveyors, we upgrade to armored conveyors when we have at least 5x their cost in titanium and we have axionite available.

Attack Route (override)

We added attack routing in Khaos for the final tournament. Instead of routing resources back to our core, the builder extends the conveyor network toward the opponent to disrupt their economy and infrastructure. The override triggers when our BFS distance to the enemy core is at most 1.5× our BFS distance to our own core.

In testing, we found that aggressive attack routing was usually a net win. Even when the route never reached the enemy core, the damage it caused to enemy infrastructure and area control was usually enough to justify the loss of resources.

If we realize we can extend the route into a gunner, we do that instead. We only place a sentinel when a gunner would be significantly worse, such as when we need to shoot enemy buildings from behind a wall.

In Figure 19, rather than simply placing a sentinel to shoot the core, in two out of the three routes, the bot realized that the route could be extended slightly to support a gunner.

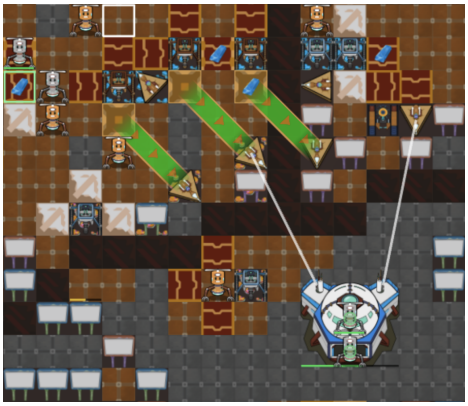


Figure 18: Attack Route.

Figure 19: Attack Route with Gunner.

In the third route, enemy roads blocked the extension, so it opted for a sentinel instead.

Secure (ceiling 7.5)

Secure and harvest generally execute in sequence. Secure first places guard conveyors on the four cardinal neighbors of an ore tile. Once the tile is secured, harvest places a harvester on it. This sequence prevents enemies from placing turrets to take advantage of harvesters with exposed sides.

Figure 20: Secure.

The secure state scores 7.5 if a claim sits on an already-built harvester or foundry, since we don't want to leave these buildings exposed to enemy turret placements. The state scores 3 if there is an ore, but no harvester yet, since we still want to secure it before we harvest.

We generally secure an ore using guard conveyors. These inward-facing conveyors are not valid outputs of the ore tile, so they never actually carry resources. However, they prevent the enemy from placing a turret adjacent to the ore, and they have significantly more health than a road. We place these instead of barriers to improve the traversability of our infrastructure for builder bots.

While securing, we also compute the optimal route and place the first conveyor facing outward, instead of guarding by facing inward, to slightly speed up our economy.

The route-target pass detects this pattern via a guard-conveyor mask so that conveyors whose target is open ore are excluded from the routing graph.

To prevent oscillations that would otherwise occur as the builder bot moves to place guard conveyors, when it is close enough to several unsecured ores, it will pick the most secured ore (the most sides covered) within Chebyshev distance 2, rather than just the closest one.

Harvest (ceiling 4)

Harvest places a harvester on an ore tile. We pick from possible ore tiles that don't already have a harvester and are secured.

The possible-ore mask includes titanium ore, plus axionite ore after round 750 if we've already harvested titanium. It excludes landlocked ore (ore whose four cardinal neighbors are all ore or unseen, since these are difficult to reach by conveyor). We also avoid existing ally infrastructure (conveyors, bridges, etc.), enemy blocking buildings, and tiles under enemy turret threat.

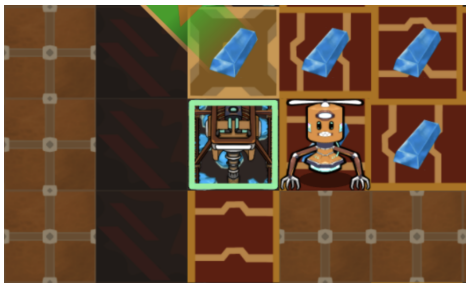


Figure 21: Harvest.

We compute `secured()` for the entire map using bitmask operations. A tile is secured if every cardinal side is off the map, a wall, or one of our non-road non-marker buildings.

If building the conveyor line to the target is too expensive, we add the tile to a temporary failure cache so we don't consider it for 100 rounds.

Disrupt (ceiling 2)

Disrupt places barriers on ore tiles outside our harvest zone, to block the enemy from harvesting them. We skip ores under enemy turret or launcher coverage. This state only runs when we have significant resources (at least five harvesters worth of titanium), and acts as a cheap way of denying the enemy's economy.

If the target tile already has an enemy road, we fire on it first. If it's a friendly road or marker, we destroy it and then place a barrier.

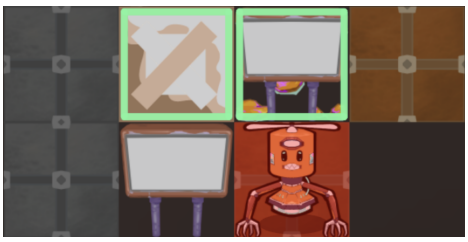


Figure 22: Disrupt.

Explore (ceiling 1)

If we have nothing better to do, our builder bots wander toward unexplored territory.

Our first few bots have their own initial explore logic. Otherwise, we wander toward a random tile near the frontier of our explored region.

Initial Exploration

For the first four builders, the core uses the tile it spawns each builder on to communicate which direction the builder should explore. This lets the core orchestrate initial exploration for better coverage.

On the new builder's first turn, it reads its own position relative to the core, recovers the intended facing by comparing its tile with the core, and reproduces the same ray-endpoint computation the core runs, capped at 12 Chebyshev steps to prevent the bot from wandering off too far.

The bot follows that target until it is within squared distance 18 or 30 rounds have elapsed, after which the slot is cleared and default explore takes over.

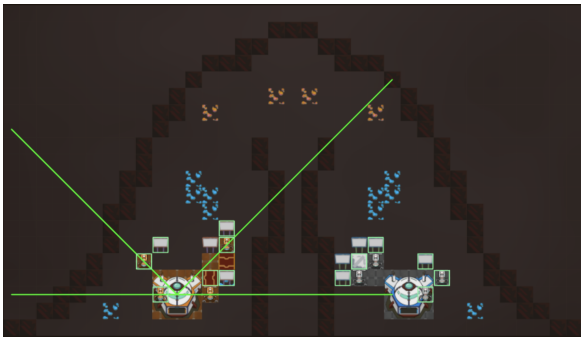


Figure 23: Initial Exploration Targets.

Frontier Sampling. If not following the initial explore plan, the explore-target generator floodfills from the bot's current position and other known friendly bot locations. It then samples a random tile from a frontier ring near the edge of the explored area once the flood terminates:

```
seeds = me | friendly_bots
frontier = seeds; ring_buffer = deque(maxlen=6)
while frontier and c < 100:
    frontier = expand_chebyshev(frontier) & passable
               & ~visited
    visited |= frontier; ring_buffer.append(frontier); c+=1
target = random_bit(ring_buffer[0])     # the ring at c-5
```

Seeding from teammates biases each bot's frontier away from where others already are, allowing for better area coverage.

Turrets

Sentinels and gunners share the same priority system for scoring tiles. The goal is to prioritize disrupting the enemy infrastructure as much as possible, by considering in order:

(1) Enemy foundries that feed at least one enemy turret and none of mine.

(2) Enemy harvesters that feed more than one enemy turret and none of mine.

(3) Enemy turrets that can hit one of my turrets, plus the enemy conveyors that feed them and the harvesters whose chain reaches them.

(4) As (3) but for any enemy turret, dropping the "threatens mine" guard.

(5) Enemy builder bots

(6) Enemy roads, conveyors, splitters, bridges, or barriers cardinally adjacent to any harvester on either team.

(7) Any enemy entity, other than roads or markers

If any target is adjacent to two or more enemy builder bots, we consider it after all of the priorities above, regardless of its normal priority.

Within each priority, we tiebreak by considering:

(1) The highest health entity we could one-shot

(2) The entity furthest from enemy builder bots (so it takes them longer to move toward and heal it)

(3) The highest weight entity (varies between gunner and sentinel, but generally prefers shooting enemy turrets, then infrastructure)

(4) The lowest health entity

We ignore conveyors, harvesters, or foundries that feed our own turrets, so we don't shoot enemy buildings that we are benefiting from. We also ignore buildings if an ally builder bot is on the same tile, since the builder bot would be hit instead.

Sentinel

Our sentinels coordinate shots by waiting to shoot if they notice an ally sentinel recently shot a target in vision. This applies if we see a target's health drop and see another ally sentinel with that target also in vision. In this case, we wait until the health drops again so we can shoot on the same turn and destroy it before enemy builder bots can heal.

```
if locked and can_one_shot(locked):
    fire(locked)
elif locked:
    wait    # don't waste 18 dmg
elif select_best gives a one-shot:
    fire it # one-shots ignore lock
else:
    fire select_best pick
```

We only coordinate one-taps this way for sentinels with a higher id than the other ally sentinel in vision. This is because entities act in increasing id order, so the higher-id sentinel will always be able to react to the lower-id sentinel's shot within the same turn, but not vice versa.

Our sentinels will consider self-destructing if they are not adjacent to a harvester and there is no enemy builder bot nearby. They will self-destruct after 16 turns if they have no ammo, or after only 4 turns if they also have no possible upstream feeder.

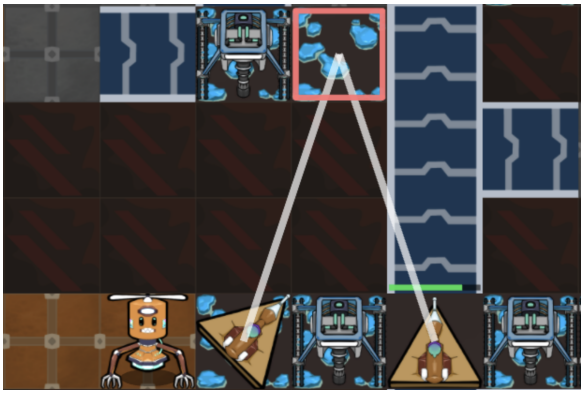


Figure 24: A sentinel one-shot on an enemy harvester.

Gunner

Gunners can only shoot in their facing direction, so we fire if there is an enemy target in front of us. Otherwise, we consider rotating if we see an enemy building, or an enemy bot standing on a damaged ally conveyor that doesn't have a nearby ally builder bot to heal it. This is to prevent enemies from continuing to freely attack the conveyor.

Rotation requires at least 60 global titanium, so a gunner doesn't burn our economy by rotating repeatedly.

Launcher

Launchers throw enemy bots. A launcher classifies itself once on placement. A chokepoint launcher (one with no enemy conveyor within its 3×3) first tries to fling the enemy back to a tile they recently occupied, using the tracked per-bot position history. This forces them to retrace their path.

If that fails, or for offensive and defensive launchers, we use the region-minimizing throw heuristic. For every legal destination, we Chebyshev-flood the region the bot could navigate from there, using enemy-POV passability and treating unseen tiles as impassable. We then choose the destination that yields the smallest reachable region. We tiebreak by greatest wall-only BFS distance from the seed set consisting of every conveyor plus the launcher itself:

```
for dest in legal_launch_tiles:
    region = flood_chebyshev(dest, enemy_passable
                                & seen)
    size[dest] = popcount(region)
throw to argmin (size[dest], -wall_bfs_dist(dest))
```

The effect is to either keep the enemy out of our territory, or keep them away from a tile we are trying to disrupt.

Chokepoint Detection

We also have a significant amount of logic geared around chokepoint detection and subsequent launcher or barrier placement, and so will dedicate the next few pages to that. Chokepoint detection starts running once we have learned enough of the map, and then continues running in the background through an incremental save-state for the rest of the game.

The goal is to produce a small set of tactically useful tiles where a single barrier or launcher can change the connectivity of the map. In practice, these are narrow corridors between larger regions, especially corridors that separate our controlled territory from the opponent's side. Due to features such as barrier walkthrough and a launcher effectively blocking a 3×3 area, these regions become chokepoints that only our bots can pass.

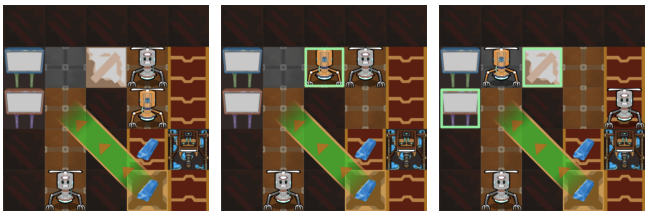


Figure 25: Chokepoint barrier walkthrough.

Simplified Geometry Core

The standalone chokepoint logic needed to run without SciPy, Shapely, NumPy, or compiled geometry libraries since they were banned for this competition. Hence, a custom geometry engine was needed. The base library was vendored from the Foronoi Python library, but completely restructured and optimized for the purposes of this competition.

The core idea was to replace heavyweight polygon operations with a small raster-backed geometry layer. The map is first lifted into a higher-resolution boolean mask, where each game tile becomes a small `scale` × `scale` block. The scale is derived from the requested sample spacing, giving enough sub-tile precision for geometry while keeping the whole map tiny:

```
scale = raster_scale_from_spacing(sample_spacing)

analysis_mask = build_analysis_mask(rows, cols, scale,
                                    analysis_polygon)
obstacle_mask = build_obstacle_mask(walls, rows, cols,
                                    scale, analysis_mask)

kept_obstacles, discarded = split_obstacle_mask_by_area(
    obstacle_mask, min_area * scale * scale)

free_mask = build_free_mask(analysis_mask, kept_obstacles)
```

The old version relied on polygon union, difference, intersection, and boundary extraction from external libraries. The runtime version instead does each step directly on the mask: scanline polygon fill for the analysis region, block filling for wall tiles, flood-fill component filtering for small discarded obstacles, and boolean subtraction for free space. Since the maps are at most 50×50, even a moderately upsampled mask is small enough for pure Python.

Once free space is known, the boundary is extracted by scanning for free/non-free transitions. Consecutive collinear boundary pixels are merged into longer axis-aligned segments, which keeps the later sampling step much smaller than treating every raster edge independently:

```
segments = boundary_segments_from_mask(free_mask, scale)

# Internally:
# - scan top/bottom exposed edges row-wise
# - scan left/right exposed edges column-wise
# - merge adjacent horizontal and vertical runs
```

Those merged boundary segments become the input sites for a sweep-line Voronoi implementation. In float mode, the runtime chokepointer sets the numeric backend to ordinary Python floats rather than `Decimal`. This loses arbitrary precision, but the inputs are already quantized to map/sub-tile coordinates, and the speedup is large enough to matter for all-map iteration.

```
set_foronoi_numeric_mode("float")

points = collect_boundary_samples(sample_spacing)
vor = Voronoi(Polygon(analysis_polygon))
vor.create_diagram(points=points)

for edge in vor.edges:
    p1 = edge.origin()
    p2 = edge.twin.origin()
    if segment_is_inside_free_space(p1, p2):
        add_raw_edge(p1, p2)
```

The runtime also replaces geometric containment queries with mask containment. A candidate Voronoi edge is accepted only if several sampled points along the segment remain inside the free-space mask:

```
def segment_is_inside_free_space(a, b):
    steps = max(5, ceil(length(a,b) * scale * 2))
    for k in range(1, steps):
        p = lerp(a, b, k / steps)
        if not point_is_inside_free_space(p):
            return False
    return True
```

Finally, each Voronoi vertex needs a clearance radius, which is the distance to the nearest wall/free-space boundary. Instead of asking a geometry library for distance-to-boundary, the runtime stores the merged boundary segments in two flat lists: vertical segments and horizontal segments. Radius then becomes a direct minimum over squared point-to-segment distances:

```
best = INF

for x0, y1, y2 in vertical_boundaries:
    dx = x - x0
    dy = clamp_distance_to_interval(y, y1, y2)
    best = min(best, dx*dx + dy*dy)

for y0, x1, x2 in horizontal_boundaries:
    dy = y - y0
    dx = clamp_distance_to_interval(x, x1, x2)
    best = min(best, dx*dx + dy*dy)

radius = sqrt(best)
```

This float runtime is intentionally less general than a full geometry kernel. It only needs axis-aligned tile maps, clipped analysis polygons, boundary sampling, free-space containment, and clearance radii. By specializing to exactly those operations, the chokepointer stays fully pure Python and remains fast enough to run.

Online Save-Stating

This separation was important. The geometry code is relatively complex, whereas the in-game usage has to be simple and robust. The builder should not care whether a chokepoint came from a Voronoi edge, a pruned graph chain, or a merged region boundary. It only needs to know whether standing near a tile and spending an action there is likely to improve our position.

At a high level, the pipeline is:

```
known_walls = map_info.wall_mask_from_seen_and_symmetry()

state = chokepoint_state.resume_or_init(known_walls)
state.step_until_budget()

if state.finished:
    for choke in state.blocker_candidates:
        register_chokepoint_tile(choke.tile,
                                 choke.kind,
                                 choke.radius)
```

The actual computation is too expensive to run from scratch inside a single builder turn, so the analyzer is incremental. Each call performs a bounded amount of work and stores the intermediate state globally. Once it finishes, the resulting chokepoint tiles are cached until the known wall map changes enough to invalidate them. This is why our builder bots time out every turn.

Map Reduction

The first step is to decide which part of the map we actually need to analyze. Because every map is symmetric, once symmetry is solved we only analyze our half of the map and mirror the result. This cuts the work by roughly a factor of two and also prevents duplicate chokepoints from being reported on both sides of the same symmetric corridor.

```
if symmetry == VERTICAL:
    analysis_poly = half_map_containing(my_core)
elif symmetry == HORIZONTAL:
    analysis_poly = half_map_containing(my_core)
elif symmetry == ROTATIONAL:
    analysis_poly = clipped_by_perpendicular_bisector(
        my_core, enemy_core)
else:
    analysis_poly = full_map
```

The input to the geometric pipeline is the set of known walls intersected with this analysis polygon. Unknown tiles are treated conservatively: we do not want to plan a permanent blocker around a corridor that may later turn out not to exist. Symmetry broadcasts help here because they let us infer far-away walls before physically exploring them.

Voronoi Skeleton

The main version of the detector uses a Broodwar Terrain Analyzer (BWTA)-adjacent medial-axis approximation. We sample points along the boundary of the free space: outer map boundary, wall perimeters, and any obstacle holes. Then we run a Voronoi diagram over those samples. Voronoi edges that lie inside free space approximate the medial axis of the traversable map.

The intuition is that a point on this graph is equally far from two or more obstacle boundaries. Wide open rooms produce high-radius graph regions, while narrow corridors produce low-radius graph chains between those regions.

```
samples = []
for segment in free_space_boundary:
    samples.extend(sample_segment(segment, spacing=1.0))

diagram = voronoi(samples)

for edge in diagram.edges:
    if edge.is_finite() and edge_inside_free_space(edge):
        graph.add(edge)
```

For each graph vertex, we compute a clearance radius: the distance from that vertex to the nearest free-space boundary.

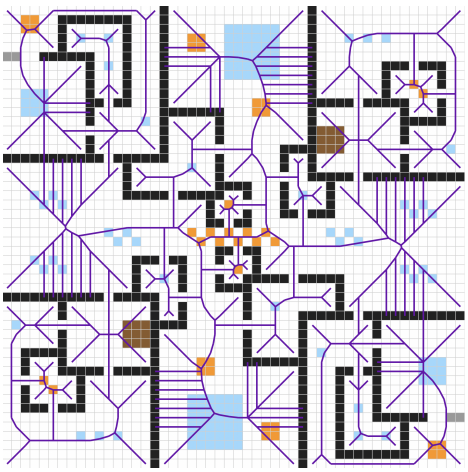


Figure 26: Voronoi on "Cubes"

This radius is the main scalar used by the later pruning and chokepoint extraction passes.

```
for v in graph.vertices:
    radius[v] = distance(v, free_space.boundary)
```

A small radius means the vertex lies in a tight corridor. A large radius means the vertex lies in an open area. Chokepoints are not simply the smallest-radius vertices, because wall corners and dead-end branches also have tiny radius. They are the smallest-radius vertices on meaningful paths between larger regions.

Graph Pruning

The raw Voronoi graph contains many branches that point into wall corners or small pockets. These are not useful chokepoints, because blocking them does not separate two important regions. We prune these branches with a simple radius monotonicity rule.

Leaves are removed if they are narrower than their parent. Repeating this peels off branches that run from open space into a dead end or wall corner, while preserving chains that connect two larger areas.

```
active = all_vertices
queue = leaves(graph)

while queue:
    leaf = queue.pop()
    parent = only_neighbor(leaf)

    if radius[leaf] < radius[parent]:
        remove leaf from graph
        if degree(parent) == 1:
            queue.push(parent)
```

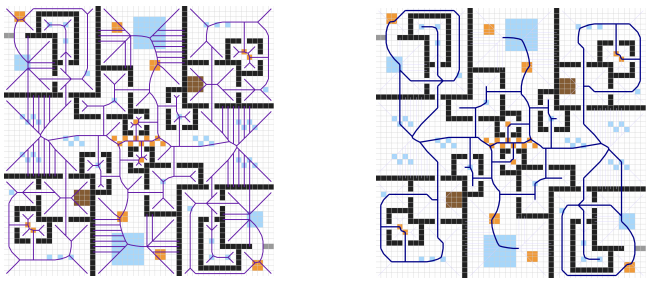


Figure 27: Graph Pruning

Region Nodes

After pruning, we identify region nodes. A region node is either a topological junction or endpoint, or a locally maximal radius point. The locally maximal points are important because a long corridor between two rooms may have no graph junction at the room center. The maximum-clearance point marks the room-like part of that chain.

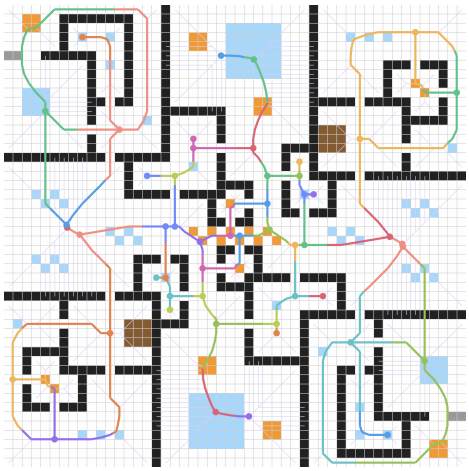


Figure 28: Region Nodes

```
for v in graph.vertices:
    if degree(v) ≠ 2:
        regions.add(v)
    elif radius[v] ≥ REGION_MIN_RADIUS:
        if no nearby vertex has radius ≥ radius[v]:
            regions.add(v)
```

The "nearby" test uses Chebyshev distance bounded by the candidate's own radius. This makes the region test scale with local feature size: in a large room, nearby maxima suppress each other, while in a narrow corridor, small local bumps do not become separate regions.

Choke Extraction

Once region nodes are known, chokepoints are extracted by walking every chain between two adjacent regions. Along each chain, the vertex with minimum radius is the bottleneck.

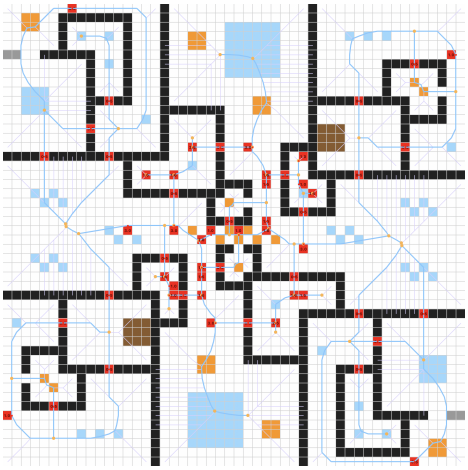


Figure 29: Chokepoints... ?

Uh oh. That's a lot of chokepoints, including some we cannot feasibly block. We can get around this issue by merging nearby regions together to remove some of the duplicate chokepoints we see.

The merge pass treats the region graph as a set of region nodes connected by candidate chokepoints. A chokepoint is only useful if it is substantially narrower than the regions it separates; otherwise it is more like a harmless local wiggle in the medial graph than a place where placing a blocker changes movement.

We process candidate chokepoints from widest to narrowest and maintain a union-find over region nodes. If a choke is too wide relative to either of the two adjacent regions, we merge those regions and delete the choke:

```
parent = {region: region for region in region_nodes}

def find(v):
    while parent[v] ≠ v:
        parent[v] = parent[parent[v]]
        v = parent[v]
    return v

def union(a, b):
    ra, rb = find(a), find(b)
    if ra == rb:
        return
    if radius[ra] ≥ radius[rb]:
        parent[rb] = ra
    else:
        parent[ra] = rb
```

The main rule compares the choke radius against the radii of the two regions it connects. If the bottleneck is close in size to the surrounding regions, then it is not really a bottleneck, so the two regions should be considered one larger region:

```
for start, choke, end in chokes_by_decreasing_radius:
    r_start = find(start)
    r_end   = find(end)

    if r_start == r_end:
        remove(choke)
        continue

    choke_r = radius[choke]
    small_r = min(radius[r_start], radius[r_end])
    large_r = max(radius[r_start], radius[r_end])

    if (choke_r > ratio_small * small_r
            or choke_r > ratio_large * large_r):
        union(r_start, r_end)
        remove(choke)
```

This removes many duplicate chokepoints in open areas. For example, if two large chambers are connected by several nearby shallow dips in radius, the widest dips are merged away first, leaving only the narrowest meaningful separator.

We also added a second rule for the common case where a small intermediate region is created between two chokepoints. If a region has exactly two active chokepoints and the larger of them is still too wide relative to the region itself, we merge through it. This prevents tiny artificial regions from producing pairs of redundant blockers:

```
# after ignoring already-removed chokes
for region in (r_start, r_end):
    active = active_chokes_touching(region)

    if len(active) == 2:
        if max(radius[c] for c in active) >
     ratio_two_choke * radius[region]:
            union(r_start, r_end)
            remove(choke)
            break
```

After merging, any removed choke vertices are deleted from the choke set, and rounded blocker placements are recomputed from the remaining chokes.

```
for region in regions:
    for edge in outgoing_edges(region):
        path = walk_until_next_region(edge)

        if path reaches another region:
            choke = argmin(path, key=radius)
            if radius[choke] ≤ MAX_CHOKE_RADIUS:
                chokes.add((region, choke, other_region))
```

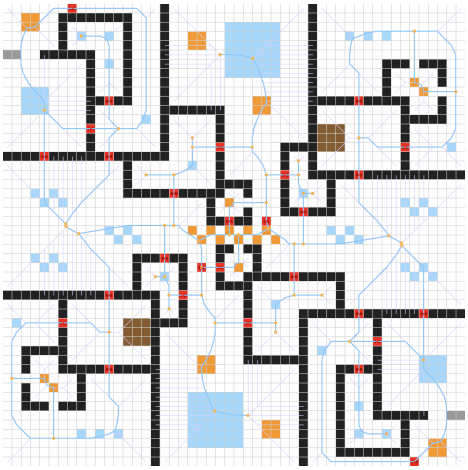


Figure 30: After Merging

This gives us a graph-level interpretation of a chokepoint: it is the narrowest point on a route between two larger spaces. That distinction is what prevents every wall-adjacent pixel from being treated as important.

We then convert the geometric choke into an in-game blocker. Very narrow chokes can be blocked by a single barrier. Wider chokes use a launcher, because a launcher controls a larger effective area by throwing enemy builders back out of the corridor.

```
def blocker_kind(radius):
    if radius ≤ WALL_CHOKE_RADIUS:
        return BARRIER
    if radius ≤ LAUNCHER_CHOKE_RADIUS:
        return LAUNCHER
    return None
```

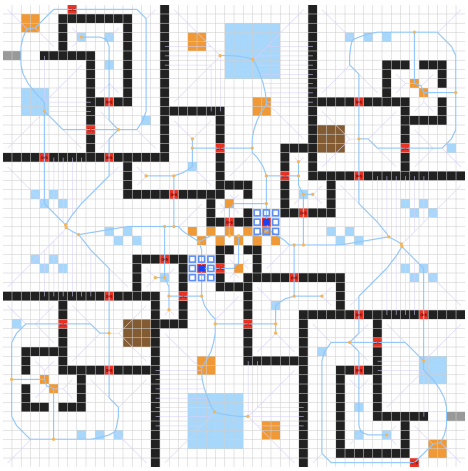


Figure 31: Classify Launchers

The geometric point is rounded back to a buildable tile. Candidates whose rounded footprints overlap are clustered, and we keep the tightest choke in each cluster.

```
for choke in chokes:
    tile = round_to_tile(position[choke])
    kind = blocker_kind(radius[choke])
    if kind:
        candidates.append((tile, kind, radius[choke]))

clusters = overlap_clusters(candidates)
for cluster in clusters:
    keep min cluster by radius
```

Using Chokepoints In-Game

The chokepoint output is consumed passively by builders. We do not create a dedicated "chokepoint state" with a high score ceiling, because blocking terrain is useful opportunistically but should not override obvious attacks, repairs, or routing work. Instead, after the main state runs, the builder checks whether it can cheaply place a chokepoint blocker nearby.

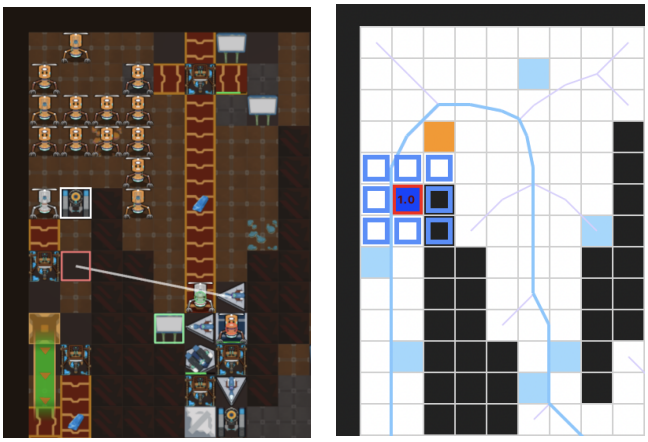
```
def try_passive_chokepoint_action_only():
    if action_on_cooldown:
        return False

    for choke in nearby_chokepoints():
        if already_blocked(choke):
            continue
        if can_build(choke.kind, choke.tile):
            build(choke.kind, choke.tile)
            return True

    return False
```

Launchers built at chokepoints use a specialized throw policy. Instead of maximizing range, we attempt to get the opponent builder stuck walking back into the chokepoint infinitely. We hence launch to the last nth position they were seen.

```
if launcher_is_chokepoint_launcher:
    enemy = best_enemy_builder_in_range()
    dest = position_from_enemy_history(enemy)
    if can_launch(enemy, dest):
        launch(enemy, dest)
```

This is especially effective when the enemy needs a builder to heal an attack route or repair a broken conveyor. Throwing the builder backward turns one launcher action into several turns of lost walking time.

For instance, the figure above depicts a chokepoint launcher blocking a horde of opponent bots from healing or replacing a defensive building that was just broken; without the chokepoint launcher, they could easily have prevented this. However, our builder is able to move through freely and, a few turns later, place a turret.



In Game Tooling

Figure 32: Chokepoint Blocks Opponent

Incremental Execution

The full detector is too expensive to recompute every turn. The in-game version therefore runs as a resumable job. Each phase stores enough state to continue later: obstacle components, boundary samples, Voronoi edges, pruned graph state, and final blocker candidates.

```
while budget_left() and job.phase ≠ DONE:
    if job.phase == BUILD_GEOMETRY:
        job.build_some_components()
    elif job.phase == SAMPLE_BOUNDARY:
        job.sample_some_edges()
    elif job.phase == BUILD_GRAPH:
        job.process_some_voronoi_edges()
    elif job.phase == PRUNE:
        job.prune_some_leaves()
    elif job.phase == EXTRACT:
        job.extract_some_chokes()
```

This makes the computation compatible with the 2000 μs unit budget. A builder can spend a small slice of time advancing the global chokepoint job without risking timeout. Once the job finishes, all builders share the same result.

Medial-Axis Wall Reduction

One optimization we added later was reducing certain wall polygons to their medial axis before feeding them into the BWTA-style runtime. The original boundary-sampling approach samples every exposed wall perimeter. This is accurate, but large solid wall blobs can contribute many perimeter segments whose fine detail has little strategic effect.

For sufficiently thick or compact wall components, we can replace the component's full perimeter with a much smaller medial-axis representative. Conceptually, instead of describing every pixel of the wall outline, we describe the wall mass by its skeleton and local radius. The surrounding free-space medial axis changes only slightly in these cases, but the number of generated sites drops by a constant factor.

```
for wall_component in components:
    if is_large_compact_component(wall_component):

     sites.extend(sample_wall_medial_axis(wall_component))
    else:
        sites.extend(sample_wall_perimeter(wall_component))
```

We only apply this reduction in cases where it is unlikely to remove a meaningful feature. The optimization is therefore conservative: it is mainly a way to avoid oversampling large blocks of impassable terrain.

Performance Optimization

Runtime optimization of our bot was crucial not just to fit within the 2000 μs turn budget, but also to free up time for our chokepoint calculations to run in the background.

Table 2 shows that all of our top ten runtime callers deal with map info or pathfinding.

Bitmasks

One of our main optimizations was using bitmasks to store map info, since using bit operations on Python big integers was significantly faster than iterating over or indexing into arrays.

For example, we can iterate over all set bits in a mask using the canonical two's-complement LSB idiom:

```
while mask:
    lsb = mask & -mask        # isolate lowest set bit
    n = lsb.bit_length() - 1  # its position
    # ... use Position(n % W, n // W) ...
    mask ^= lsb               # clear and continue
```

Thus, if we build a mask containing possible target tiles (for explore targets, turret placements, etc.), the time complexity is proportional to the number of set bits rather than the number of tiles.

Local Variable Hoisting

Python attribute and global lookups are not free, and some of our inner loops run hundreds of thousands of times per turn. Hence, every hot function binds the module and controller members it needs to locals once before the loop:

Table 2: The ten largest runtime consumers (ignoring chokepoint consumers), ranked by total self time. Self times are additive and sum to total CPU, so % is each function's share of the total self time (≈ 6,483 ms over 6,070 profiled turns of one map); the listed ten account for ≈ 35%. μs/call is mean self time per invocation.

Function	Calls	Self (ms)	μs/call	%
map_info:update_at	470,546	883.9	1.9	13.6
map_info:update	6,070	336.7	55.5	5.2
pathing:_claim_zone_on_passable	3,412	222.4	65.2	3.4
map_info:_compute_route_targets	6,716	218.0	32.5	3.4
pathing:bfs_move	3,655	123.1	33.7	1.9
map_info:_carrying_expand	5,321	117.8	22.1	1.8
Controller.get_tile_building_id	414,358	113.2	0.27	1.7
map_info:expand_chebyshev	71,745	111.2	1.55	1.7
pathing:_closest_impl	15,170	84.3	5.56	1.3
map_info:update_move	2,752	76.6	27.8	1.2

```
# bfs_move, before the frontier loop iterates:
bm_et = map_info._bm_et;
nlc = map_info._not_left_col;
board = map_info._board_mask;
bcost = barrier_cost;
# update_at, binding hot Controller methods to locals:
get_tile_env         = rc.get_tile_env
get_tile_building_id = rc.get_tile_building_id
get_hp               = rc.get_hp
```

Integer-Indexed Tables

cambc enums are not integer-valued, so using them as dictionary keys or in frozenset membership tests costs a hash per access. map_info converts every enum to a sequential index once at import and thereafter uses list indexing and boolean lookup tables instead of dictionary lookups.

Caching and Memoization

Many of our derived bitmasks, calculated targets, etc., are memoized on the map info structures they depend on. We also memoize some calculations irrespective of local map changes. The harvest zone, which is a Voronoi floodfill from our core against the predicted enemy core used to approximate safe harvest locations, only runs when symmetry predictions update. The column and row guard masks and the eight per-facing direction templates for sentinels are likewise built at init and not updated again.

Incremental Updates

We prioritize updating our map info structures rather than fully rebuilding them. For example, update_at maintains conveyor output and reverse conveyor output bitmasks as buildings are updated, rather than rescanning the entire map.

Furthermore, after a move, update_move diffs the old and new vision masks and calls update_at only on tiles that became visible:

```
for tile in nearby:
    bit = 1 << (tile.x + tile.y * W)
    new_visible |= bit
    if not (old_visible & bit):   # only the delta
        update_at(tile)
```

This reduces the number of tiles we scan after moving from all tiles within vision to only the newly discovered tiles.

TTL Caches

States that depend on expensive calculations to find target tiles remember the tiles that failed. We use Time to Live (TTL) caches that store target tiles we want to skip for 100 rounds, so we don't waste compute trying to calculate if a recently failed target tile should be acted on again.

Debugging

log.py binds log at module load to one of two definitions depending on a single boolean:

```
DEBUG_LOGGING = False
if DEBUG_LOGGING:
    def log(*args, **kwargs): print(*args, **kwargs)
else:
    def log(*args, **kwargs): pass
```

Every log(...) call site thus becomes a no-op function call when we turn the flag off before submitting our bot.

Results

Our tournament results are summarized in Table 3. Sprints were the weekly tournaments held throughout the competition, where teams could evaluate their performance and gain a better understanding of the meta. The International

Qualifiers was a tournament for the top eight teams outside the UK, offering a chance to advance to the Grand Finals.

The Grand Finals was the culminating in-person tournament for the top eight international and top eight UK teams to watch the matches play out.

The Meme and Pong tournaments were for-fun tournaments held after the Grand Finals. The Meme tournament featured several edge-case and troll maps. The Pong tournament involved min-maxing axionite collection by hardcoding strategies specifically for the Pong map. This map was unique in that the game would end in a forced draw if both sides avoided placing roads near the barrier in the middle, since then neither side could reach the opponent's core.

Table 3: Tournament Results.

Tournament	Date	Result
Sprint One	3/22	Top 128
Sprint Two	3/28	Top 8
Sprint Three	4/5	Top 4
Sprint Four	4/11	Top 8
Sprint Five	4/19	Top 4
International Qualifiers	4/20	3rd
Grand Finals	**5/4**	**1st**
*Meme Tournament**	5/9	Top 8
*Pong Tournament**	5/9	2nd

*For-fun tournaments played with different rule-sets.

Figure 33 shows our Elo progression over time. We went into the Grand Finals as the first seed (Figure 34 shows the top eight seeded teams before the Grand Finals).

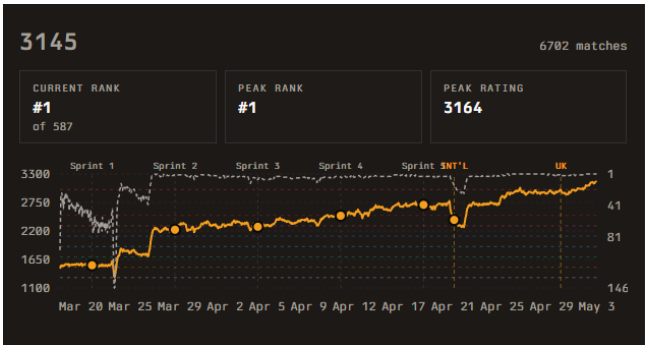


Figure 33: Elo rating over the course of the competition.

Conclusion

We learned a lot over the course of the competition through trial and error. We had many more bot versions outside of the ones listed at the beginning (such as Hermes and Hephaestus, along with sub-versions of all of our bots) and were still adding in new strategies like attack route just hours before the final submission deadline.

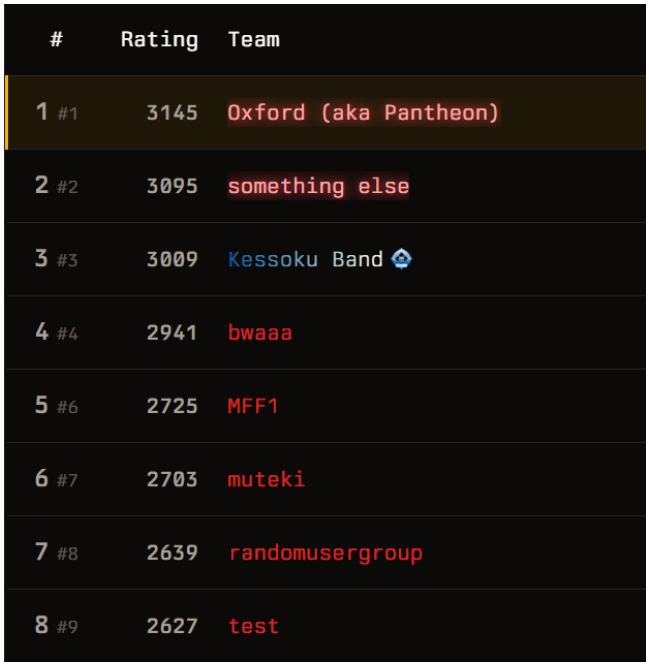


Figure 34: Top eight seeded teams for the Grand Finals.

Overall, our final solution is different from most Battlecode solutions, which tend to have persistent states and defined state transitions. However, we believe our memoryless approach was one of the main factors in our bot being very dynamic and adaptable. It also allowed for a lot of complex behavior to emerge without hardcoding strategies, which made our bot less prone to overfitting.

In the end, our team had a lot of fun competing, and we're already looking forward to the next competition.