# Money Is All You Need *†

Battlecode 2025 Postmortem

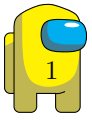
Cyril Sharma, Egor Gagushin from team **Om Nom**

February 2025

## Contents



*Ok fine, we needed some paint too.
†Generalizes to all problems.



# 1 Introduction

We (Cyril Sharma, Egor Gagushin, both Purdue '25) competed in Battlecode 2025 as team Om Nom. This is our third year competing together, having placed 5th and 4th in 2023 and 2024 respectively. We came in 3rd in the final tournament this year, winning the US Qualifiers along the way. You can find our code here – look under the templates folder if you don't want to read unrolled java (the code there is not in java, but more on that later). Here are some of our thoughts about this year's competition as well as Battlecode in general.

# 2 Game Overview

The objective of this year's game was to paint as much of the map as possible – the first team to paint 70% of the paintable squares won the game. Another way to win was to destroy all enemy units.

There were two types of resources this year: **Money** and **Paint**.

**Money** was needed to produce units, buy towers and activate economy boost patterns (called SRPs).

**Paint** was needed to produce units, for the win condition, to resupply units with paint and to paint special patterns, which were prerequisites for acquiring SRPs and towers.

## 2.1 Units

### 2.1.1 Soldiers

Soldiers are one of two units (see Splashers) that can attack towers, but they have almost 2.5 times the DPS of Splashers, so they are much better attacking units. They are the only unit which can place individual squares of paint at a time, making them the only unit capable of making SRPs and building towers.

### 2.1.2 Moppers

Moppers are the only unit that can attack other units. Their attacks do not cost paint, which makes them the only unit capable of surviving indefinitely without a tower. They can also clean up enemy paint, making them essential for cleaning up enemy paint off of ally patterns.

### 2.1.3 Splashers

Splashers are the only unit that can paint over enemy paint with ally paint. They're also the only unit which can paint several squares at once. For these reasons, splashers are the best unit for capturing enemy territory.

### 2.2 Towers

Towers are immobile units that can spawn units. Money and Paint Towers will passively generate the corresponding resources. Crucially, all towers will spawn with 500 paint, and by default a single Money Tower produces 4x of its resource as a single Paint Tower. This meant that, in general, you were paint-bound, not money-bound. There are also Defense towers, which we did not use (but maybe should have).

## 3 Timeline

### 3.1 Sprints 1 & 2

For the first week, we floundered around. We didn't upload our bot for several days, and had no idea what the meta was. After uploading, we realized we had kind of been focusing on the wrong areas. We began to prioritize packing SRPs super efficiently and using splashers. Until sprint 2, almost all of our efforts were focused on improving economy and expansion as much as possible.

### 3.2 Quals

At this point, we were a little concerned. Our bot wasn't doing too well in the rankings, and we were having a hard time finding improvements. We spent a long time studying replays and then realized some pretty obvious things.

1. **Soldiers should not blindly prioritize going home**

   In our case, the soldiers went home unconditionally at around 50 paint to refuel. This would lead to disastrous scenarios where we would barely get to a tower and then retreat without using roughly 10 attacks! Changing the priority meant that we killed towers with much higher odds. The same logic was applied to finishing towers and SRPs before going home, if possible.

2. **Units should be more paint efficient**

   Previously, we ignored paint penalties (which are imposed if you do not stay on your own paint, or if you are near ally units), because the penalties seemed pretty minor. However, after seeing the huge win from making the soldiers not go home prematurely, we realized that this was important. Saving even a little paint means you can land a few more attacks, which can easily win you the game.

   We adjusted our soldier micro to ensure that soldiers would almost always stay on paint while attacking enemy towers.

   We added a greedy tiebreaker in our pathing to prioritize moving on painted squares. Fixing this made our bot much more likely to reuse paint trails, which saved some soldiers a lot of paint.

3. **We need to be robust to losing Paint Towers**

   Losing a Paint Tower at the beginning of the game was devastating, because if you didn't rebuild it, you would never make paint again and so all your units would die and you were basically guaranteed to lose. This was one of the reasons why rush was so effective.

To combat this, we found a trick for **computing the number of Paint Towers**. Whenever our estimated number of Paint Towers hit 0, we would just build another one.

4. **Soldiers should live useful lives**

   There were so many bugs where we would get stuck switching between targets, or get stuck painting different colors for the same square, etc.

5. **Aggression**

   We realized that it did not matter how good we were if we could not build towers or if we lost towers very early on. So, we figured, at least we could be more aggressive ourselves. We added an early rush (though building towers along the way), started adding a blob of paint to all ruins we see in the hopes of preventing enemy towers from going up, and sending splashers towards enemy territory using symmetry.

6. **Mopper defense**

   In an effort to defend against rush, we would have towers spawn moppers towards the enemy soldiers to reduce their paint, and hence the amount of damage they deal to the tower.

After uploading, our bot seemed to do pretty well against most teams. Winning the US Qualifiers was a complete surprise nonetheless. We won with much larger margins then we had been winning in scrims (almost every match was won with a 4-1 ratio). Our theory was that the maps had been really favorable (complex pathing, large rush distance). Indeed, we did not re-upload a bot until the next Thursday, and our bot's elo was relatively static. However, when they uploaded the tournament maps our bot jumped about 100 ELO reaching the 2000s for the first time.

## 3.3 Finals

1. **Don't go over bytecode on the first turn**

   Basically every unit we spawned would allocate some massive array and then go over bytecode on the first turn. This is terrible because snowballing matters the most in the early game. Fixing this gave us a substantial win rate against our old bot.

2. **Nuke Towers**

   We saw this strategy from **Gone Whalin**. We had noticed that we would frequently lose games despite having loads and loads of money. Nuking towers essentially let us trade money for paint (and at a better ratio than what was possible without SRPs, too!) and this let us balance our resource production, giving us a huge boost in performance.

3. **Rush**

   For the longest time **confused** was the highest elo team on the leaderboard. His strategy used rush and seemed really effective. We spent a long time reverse engineering his rush, only to discard it last minute – when it worked, it was extremely effective, but when it didn't, we were almost guaranteed to lose.

4. **Bug Fixes**

   We made our pathing not walk the edge of the map and bug towards the target. We got rid of a bug where we would keep switching the square around a ruin we wanted to paint, which caused our bug target to keep resetting and prevented progress in certain scenarios.

The morning of the final submission day we were consistently losing either 0-10 or 1-9 to **confused** and **Just Woke Up** (albeit with an outdated version of our bot). But, to our surprise and elation, after submitting later that day, we were pretty much even against both.

# 4 Exploration

Our exploration was painfully simple. We tried a few things, like making units go to the edge, only sampling locations we hadn't been to before, exploring methodically by exploring in some radius of the last tower, etc. but none of this really seemed to help.

```
 public static void explore(MapInfo[] near) throws GameActionException {
    if (exploreTarget == null || rc.getLocation().distanceSquaredTo(exploreTarget) <= 5) {
        exploreTarget = new MapLocation(
            Globals.rng.nextInt(rc.getMapWidth()),
            Globals.rng.nextInt(rc.getMapHeight())
        );
    }
    Pathing.pathTo(exploreTarget);
}
```

In general, our questionable exploration code was mitigated by our units having good local heuristics to decide what to do. Soldiers would try to build SRPs or Towers, splashers would rush towards enemy towers using symmetry, and moppers would try to mob enemy units.

# 5 A Fast Symmetry Checker

One cool thing we worked on was a really fast symmetry checker. For our pathing we already load the entire map into bitmasks, why not save those bitmasks and use them for cracking the symmetry?

Specifically, what we ended up doing was storing every feature of the map as 60 long longs, with a 1 indicating presence. We had masks for ruins and walls. We would also store the "reversed" mask, where instead of storing an item at its position (x, y) we would store it at (MapWidth - 1 - x, y). Then, for testing symmetry we would project each row visible in our bitmask to its symmetries, and check that the portions of the masks we've seen are equivalent. If they weren't equivalent, the symmetry didn't hold. For example, here is how we checked vertical symmetry (see **Jinja** for syntax).

```
public static void checkVertical() throws GameActionException {
    switch (rc.getLocation().x) {
        {% for posX in range(0, 60) -%}
        case {{ posX }} -> {
            {% for x in range(posX - 4, posX + 5) -%}
            {%- if x >= 0 and x < 60 -%}
            seen_fused = TileLoader.seen{{ x }} & TileLoader.seen_reversed{{ x }};
            if ((TileLoader.wall{{ x }} & seen_fused) !=
                (TileLoader.wall_reversed{{ x }} & seen_fused)) {
                VSYM = 0;
            }
            if ((TileLoader.ruin{{ x }} & seen_fused) !=
                (TileLoader.ruin_reversed{{ x }} & seen_fused)) {
                VSYM = 0;
            }
            {% endif -%}
            {% endfor %}
        }
        {% endfor %}
        default -> {}
    }
}
```

Since we check an entire row of vision at a time for symmetry, this runs about 9 times faster then the naive implementation. This was pretty interesting because, typically, symmetry checking is done at the end of the turn if you have spare bytecode. It is not guaranteed to check every square and runs really slowly. With our implementation, it was 1k bytecode, checked every square, and was guaranteed to crack the symmetry if it was possible to do so from the information this bot had seen.

# 6 Computing the Number of Paint Towers

There were two pieces of global information available to us: the amount of money our team had, and the number of towers our team had. Using money, you could almost exactly estimate the amount of money income per turn, by just tracking the change in money over 10 turns and taking the maximum. Since it is unlikely that you spent money on each of those turns, at least one of those turns is usually the actual income.

Unfortunately, money income is not enough to determine the number of Paint Towers, because it is a function of both the number of SRPs and the number of Money Towers. Specifically, income looks something like income $= (20 + 3 * \#\text{SRPS}) * \#\text{Money Towers})$. However, because SRPs increase income by multiples of 3, and towers increase income by multiples of 20, income almost always uniquely determines the number of SRPS, Money Towers, and Paint Towers (not building Defense Towers helped). In situations where the solution was not unique, we favored solutions where the number of Money Towers was larger than the number of SRPs (since this is the case in the early game, and for late game we almost certainly have a Paint Tower anyway). Our code looked something like this,

```
switchLabel: switch (highestIncome) {
    {# computeIncomeMap spits out all solutions (number of moneyTowers and SRPs)
    {# For each income. #}
    {% for income, pairs in computeIncomeMap(maxSrps=25, maxMoneyTowers=25).items() %}
    case {{income}} -> {
        {% for (moneyTowers, srps) in pairs %}
        if ({{ moneyTowers }} <= numTowers) {
            numSrps = {{ srps }};
            numMoneyTowers = {{ moneyTowers }};
            break switchLabel;
        }
        {% endfor %}
    }
    {% endfor %}
    default -> {
        return;
    }
}
numPaintTowers = numTowers - numMoneyTowers;
```

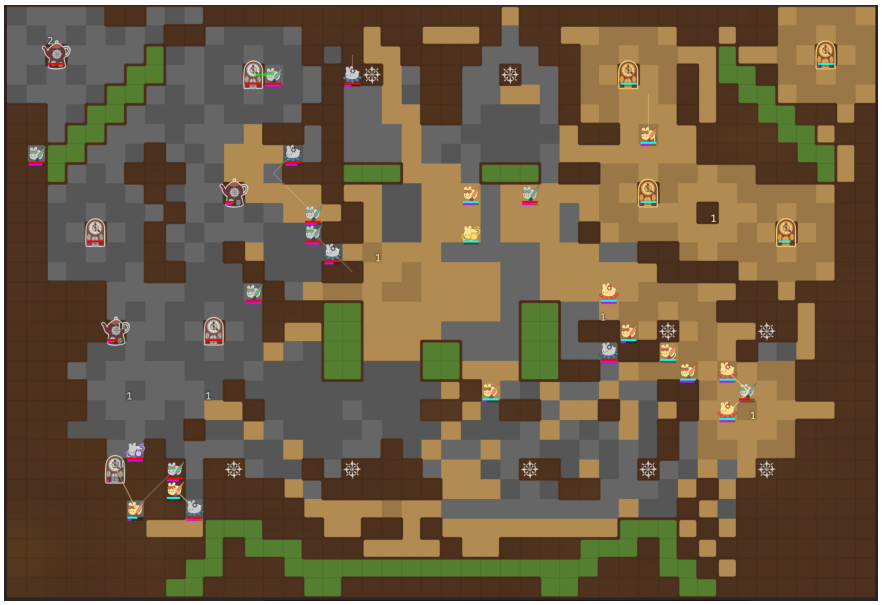


Figure 1: Oh no, we lost our only Paint Tower (yellow)!

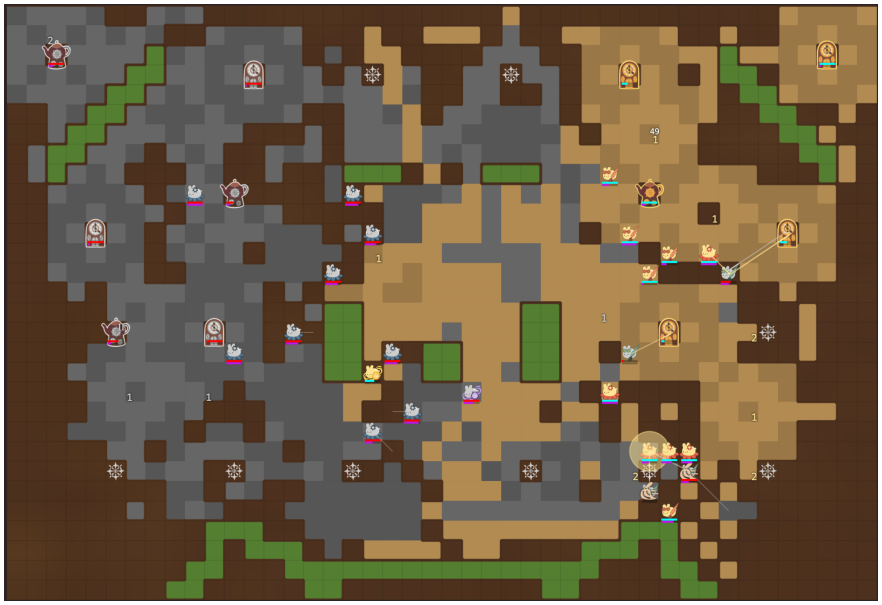


Figure 2: Now we have one again!

# 7 Pathing

Before the competition had started, we brainstormed ideas to make our pathing robust. We had tried a few solutions over the previous years, and they all left something to be desired.

**Greedy Pathing** := This is what most teams start with. It scores the eight adjacent squares (based on distance, cost, etc.) and picks the square with the highest score that it can move too. It has no guarantees to reach the destination, but, on the other hand, it is pretty robust to units clogging the path.
**Optimal Pathing** := This is an upgraded version of **Greedy Pathing**. This is usually a BFS or some toned-down version of Djikstra's algorithm. It can make more optimal decisions than **Greedy Pathing** (it is less likely to get stuck behind walls), can effectively account for unit clogging, but because you're limited by vision radius, it still doesn't have any guarantee to reach the destination. The one downside compared

to **Greedy Pathing** is that because you usually have to compute some algorithm over all of vision radius, your scoring criteria might have to be relatively simple.

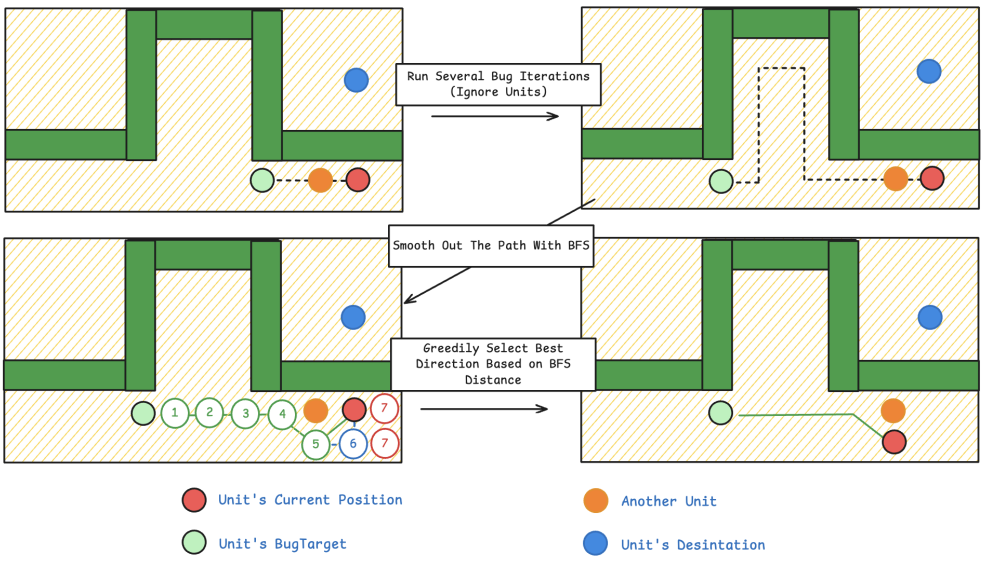
**Bug Pathing** := Essentially, use Greedy or Optimal Pathing until we get stuck. Then switch to a "move around obstacles" mode. Unlike either of the previous pathing techniques, this will actually guarantee that you will reach the destination (at least in the absence of other units). This is really nice, and is why essentially every team has a bug implementation. However, **Bug Pathing** has some pretty big problems. First, it has no good way to account for unit clogging. Consider what happens when a unit blocks your way while bugging around an obstacle. Because bug only tells you a single direction to move in, your choices are don't move at all, or treat the unit like a wall. Treating the unit like a wall breaks all the invariants of **Bug Pathing** (it is not designed for dynamically changing obstacles), and standing still can keep your units stuck forever. Second, it takes pretty inefficient paths, as it essentially tells you to follow the edge of walls. For example, if you are doing something like a right turn, it will not cut across that corner.

**Our Pathing** := Our goal was to obtain all the guarantees of **Bug Pathing**, while being robust to unit clogging and still taking efficient routes. We came up with the following approach.

**Algorithm 1** Optimal Bug

**Step 1: Advance BugTarget**
Initialize BugTarget as the previous BugTarget if it exists
Otherwise, initialize it to the current position
**for** several iterations **do**
    Advance BugTarget using the bug algorithm (treat squares occupied by units as passable).
**end for**
**Step 2: Compute BFS Distances**
Run a BFS backwards from BugTarget
Compute distances to neighboring squares
**Step 3: Tie-breaking with a Heuristic**
bestSquare := The best square amongst squares which decrease the BFS distance.
"Best" can be defined arbitrarily, for us it was a combination of distance to the BugTarget and paint penalty.
**Return** bestSquare

Or more visually,



This is essentially an optimized version of bug, so we still have a guarantee to reach the destination. The BFS smooths out the bug path and gives us robustness to unit clogging, as instead of treating units like walls and messing up all the bug invariants, we can just choose whichever direction gets us closest to the bug target. A third, unforeseen advantage is that this is also really cheap. Once the BugTarget reaches the edge of vision radius, we don't advance it anymore. Hence, a typical iteration of our pathing uses one iteration of bug, followed by a BFS. We use bitmasks to compute the BFS, which makes the whole procedure somewhere on the order of 1.5k bytecode.

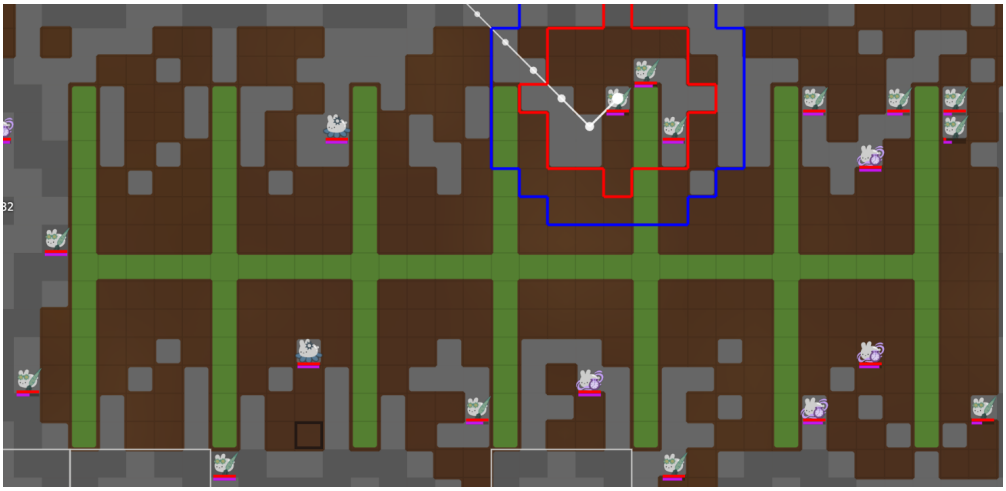


Figure 3: This map highlights one of the advantages of our pathing. Note that there is no paint in the parking spots, which is because our units never go there – as soon as they see the walls and that the bug goes around, they cut straight to the other side.

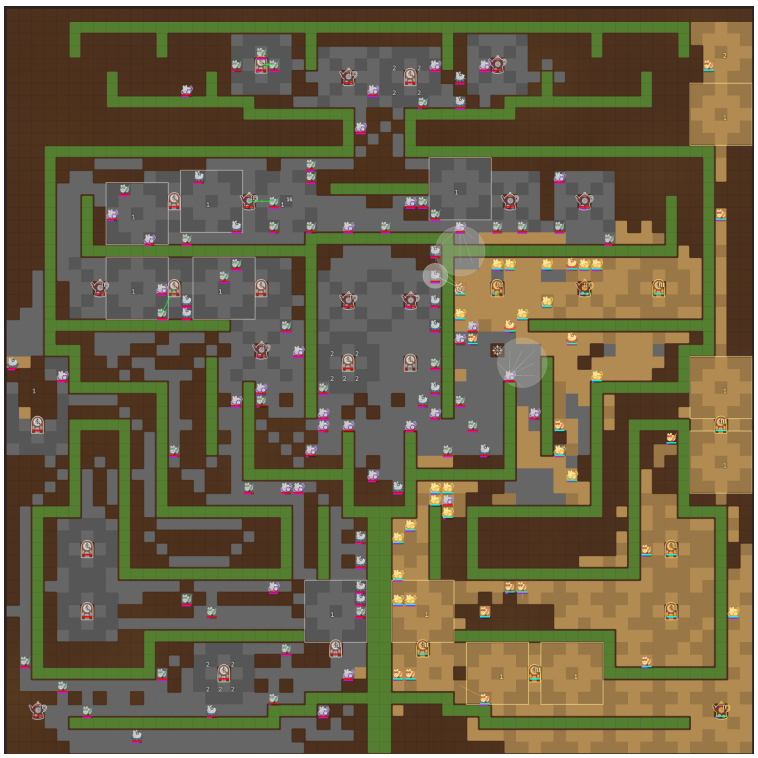


Figure 4: On this maze map, our pathing allows us to expand faster and have our units spread out across the map.

# 8 Jinja

There are so many opportunities for unrolling in Battlecode that it is impractical to write code generation scripts for all of them. To overcome this, I (Cyril) wrote a compiler. However, as the competition approached, the thought of having to add a laundry list of features (object inlining, array inlining, etc.) and make sure the compiler was 100 percent bug-free seemed really daunting. Then I remembered Jinja. Jinja is a templating engine. It is essentially a superset of most programming languages designed to ease the burden of code generation. You can do stuff like

```
{% for i in range(10} %}
System.out.println({{i}});
{% endfor }
```

To create 10 `System.out.println()` calls without a loop.

You can also do loads of other stuff, for example

```
{% set VISION_RADIUS = 9 }
{% set CENTER = intdiv(VISION_RADIUS, 2) }
int center_x = {{ CENTER }};
int center_y = {{ CENTER }};
int adjacency_mask = {{ some_complicated_function(VISION_RADIUS) }};
```

This made code generation a breeze. Whenever we had write a feature, we could just write it in the optimized way immediately, because there is barely any overhead to writing `{{ for i in range(10) }}` vs `for (int i = 0; i < 10; i++)`. Take splasher micro. Almost everybody used some sort of "score each square based on the amount of enemy paint I splash". The easiest implementation is a quadruple for loop. With Jinja, writing this the unrolled way looks like this:

```
{% for i in range(-3, 4, 1) %}
{% for j in range(-3, 4, 1) %}
MapLocation mloc{{i+3}}{{j+3}} = new MapLocation(x + {{ i }}, y + {{ j }});
PaintType mpaint{{i+3}}{{j+3}} = PaintType.ALLY_SECONDARY;
if (rc.canSenseLocation(mloc{{i+3}}{{j+3}})) {
    MapInfo minfo{{i+3}}{{j+3}} = rc.senseMapInfo(mloc{{i+3}}{{j+3}});
    if (!minfo{{i+3}}{{j+3}}.hasRuin() && !minfo{{i+3}}{{j+3}}.isWall()) {
        mpaint{{i+3}}{{j+3}} = minfo{{i+3}}{{j+3}}.getPaint();
    }
}
{% endfor %}
{% endfor %}

{% for i in range(-2, 3, 1) %}
{% for j in range(-2, 3, 1) %}
if (rc.canAttack(mloc{{i+3}}{{j+3}})) {
    count = 0;
    {% for k in range(-1, 2, 1) %}
    {% for l in range(-1, 2, 1) %}
    switch (mpaint{{i+k+3}}{{j+l+3}}) {
        case ENEMY_PRIMARY, ENEMY_SECONDARY -> count += 1;
        default -> {}
    }
    {% endfor %}
    {% endfor %}
    if (count > bestCount) {
        bestLoc = mloc{{i+3}}{{j+3}};
```

```
            bestCount = count;
        }
    }
    {% endfor %}
    {% endfor %}
```

Almost the same amount of space as the straight-forward quadruple for loop implementation, but without any of the hassle of a specialized script. There is also syntax highlighting, so it is much easier to write syntactically valid code.

You can also do QOL things

**Debug macros**

```
{% debug('var1, 'var2', 'var3') } => System.out.println("var1 " + var1 + " , " + ...);
```

**Seeding** (where seed is passed to the make file as an argument).

```
rng.setSeed((long) rc.getID() + {{ seed }});
```

# 9 Tower Nuking

One problem that we were thinking about throughout most of the competition but never really had a good solution for, was which towers to build. It is very important to build Money Towers, because you need money to build towers – so whenever we tried to go more paint-heavy, we would do worse. But without paint you cannot build units, so you need to balance the two resources somehow. For the longest time, our solution was to build Money Towers until we had a certain amount of money, which worked to some extent, but often wouldn't try to correct the balance until it was too late.

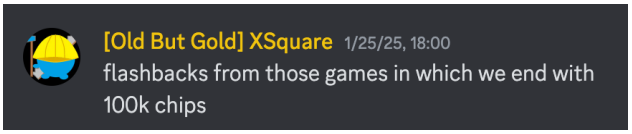



Figure 5: A common struggle.

While watching some **Gone Whalin** replays, we often wondered how they did so well after losing their Paint Tower – for most teams, that meant a sure loss. "Wow, they must be really good at keeping units alive and useful" we thought. As we looked closer, we realized they were destroying their own towers and rebuilding them! At that point, their earlier comment about SRPs being incompatible with their strategy made a lot more sense. We realized that it didn't matter if we built too many Money Towers, as long as we had a way to convert it to paint which this approach provided: each tower spawns with 500 paint, allowing you to trade 1000 money for 500 paint. In fact, if you have few SRPs it is even better than building Paint Towers, since Money Towers have 4x the production of Paint Towers. But even with a lot of SRPs, it was a strict improvement for us, as it would stop us from accumulating insane amounts of money.

Thus, we decided to add it. We didn't want to sacrifice SRPs though, so we designated about a third of our Money Towers to do this when certain conditions are met – when we are past some money threshold, the pattern around it is correct, and there is a unit nearby. We marked the corners around the towers so that they themselves know they are "farms" and the units around know to repair the pattern if it is broken. The marks also helped units not build SRPs near "farms", and considering there were some nice overlaps, this didn't seem to hurt our SRP building too much.

It took a bit of work to coordinate – destroying the towers and not rebuilding them is really not the move. We briefly considered using communications for a two-way handshake, but some heuristics ended up being good enough. It worked quite well – we were building up money a lot less, and even when we did, once we could farm one tower, we could spend everything almost instantly. Here is an example:

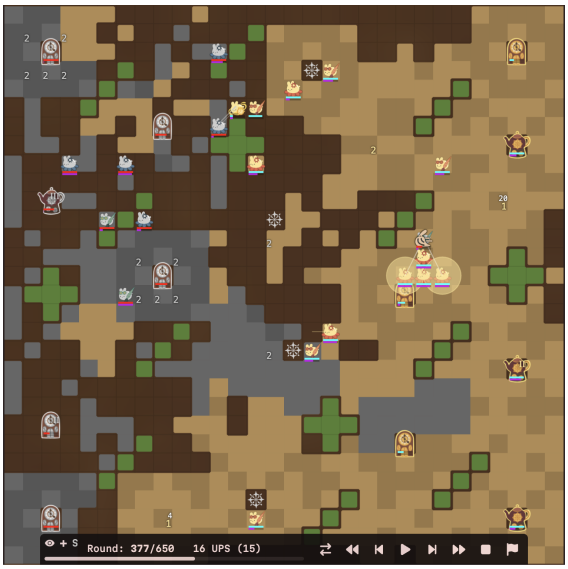



Figure 6: This looks pretty bad for us (gray), but we saved up over 13,000 in money.


Figure 7: In less than 60 rounds we spent all of our money and pumped out a lot of splashers, going from 8 units to 23.

Indeed, as the title suggests, money is all you need!

## 10 SRPs

The original pattern for the SRPs allowed them to overlap quite a bit. Considering that every single SRP boosted the production of each tower by 3, there was a lot of snowballing. This resulted in **Teh Devs** changing the pattern to one that did not allow as much overlapping, as well as an added cost (200 money) and 50 round activation period. We hard coded the initial pattern, which did not do wonders for our rating after the changes.

Along with adjusting the pattern, we moved away from a fixed tiling – though it worked well most of the time, it could perform quite poorly on adversarial maps. Our algorithm was pretty simple: any soldier could start building an SRP centered on their location as long as certain conditions were satisfied. Namely, if it didn't mess with building a tower (or "farms") or other SRPs, and if we did not see any enemy units or paint. We also had the soldiers that completed an SRP add the squares 4 away in the cardinal directions to the list of locations to try (as long as we didn't immediately see an issue), which let us get some advantages of a fixed tiling while accounting for the map.

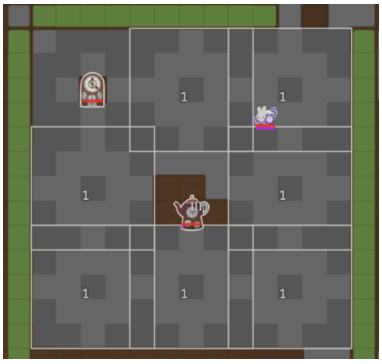


Figure 8: Neat packing of SRPs.

## 11 Communications

We didn't use the actual communications much. The only thing we used them for is symmetry checking, just 3 bits. However, we also used marks to coordinate things:


Figure 9: Paint tower.


Figure 10: Money tower.


Figure 11: A "farm".


Figure 12: SRP center.

## 12 Micro

Our micro for soldiers was pretty straightforward, just back in and out of tower action range, so that you can land two hits for every hit you receive. Prioritize maximizing net damage, and if all else is equal, try to move on squares with low paint penalty.

For moppers, we do the exact same thing, except that if there are no enemy moppers, we will not back out of range (this makes it easier to chase enemy soldiers / splashers). The other difference is that we will not try to chase enemies off of our own paint (since moppers die extremely fast when they're not on ally paint).

Our framework for this was somewhat interesting; it is essentially **XSquare** micro (where we compute statistics for each square and choose the best square through a series of tie breakers), but it is unrolled. There are no objects and no function overhead. This is nice because accessing instance variables from a non-static object requires an extra load of the "this" pointer, roughly doubling the overall cost of the micro. Inlining this made the micro fast enough to not need bytecode checks. See **Jinja** for more information on how this works.

```
/* Instead of making an object, use this macro */
/* To create a bunch of local variables corresponding to fields of the object. */
{%- macro initTargetWithSuffix(i, suffix) -%}
{% set dir = directions[i] %}
{%- set val = 'true' if dir == 'Direction.CENTER' else ('rc.canMove(' ~ dir ~ ')' ) -%}
MapLocation targetLoc_{{suffix}} = SquareManager.m{{dirX[i]+2}}{{dirY[i]+2}};
boolean canMove_{{suffix}} = ({{val}});
int penalty_{{suffix}} = SquareManager.penalty{{dirX[i]+2}}{{dirY[i]+2}};
double paintDmgAttackRange_{{suffix}} = 0;
double myPaintDmg_{{suffix}} = 0;
int minDistToEnemy_{{suffix}} = 100000;
boolean inTowerRadius_{{suffix}} = false;
{% endmacro -%}

/* Use a Jinja macro to avoid any function call overhead. */
/*------------------ ADD ENEMY --------------------*/
{# TODO: Factor into seperate pieces for each unit. #}
{%- macro addEnemy(i) -%}
{%- set robotVar = 'robot' -%}
```

```
{%- set dir = shortDirections[i] -%}
{%- set label = 'addEnemy' ~ _ ~ dir %}
{{label}}: {
    if (!canMove_{{dir}}) break {{label}};
    int d = targetLoc_{{dir}}.distanceSquaredTo({{robotVar}}.location);
    switch (robot.type) {
        case SPLASHER: {
            if (robot.paintAmount == 0 ) break {{label}};
            if (actionReady && d <= mopperDesiredActionRadiusSquared) {
                myPaintDmg_{{dir}} = GameConstants.MOPPER_SWING_PAINT_DEPLETION;
            }
            if (d < minDistToEnemy_{{dir}}) minDistToEnemy_{{dir}} = d;
            break {{label}};
        }
    ...
}
```

# 13 Unit Spawning

Initially, we produced certain units only on specific sets of rounds (ex. only make splashers on rounds at most 20 modulo 50). We did this because we were concerned that if towers tried to build different things at the same time, it would be hard to save money, and so only soldiers or moppers would be produced (relatively cheap units in terms of money). This was a pretty bad system, as we would often produce a ton of one unit and then have no resources left for anything else.

We soon realized that we were basically never money-bound (except at the beginning of the game, when we are trying to expand) and switched to a much more controllable system where towers would select a unit, save up for it, and then produce it. This allowed us to produce our units in a fixed ratio throughout the game.

Funnily enough, we found a bug right before finals where mid to late game our Money Towers would only produce splashers. Fixing it somehow made everything worse, so we ended up keeping it.

# 14 Final Thoughts

We really enjoyed this year's game. The increased focus on macro meant there were more cool strategies and variation between different teams' bots. We kept finding ways to improve our bot right until the end, and it felt like we could continue doing so if the competition went on for longer.

The finals were lots of fun as always – thanks to **Teh Devs** for organizing everything. Going in, we knew it was going to be essentially a coin flip against **confused** and **Just Woke Up** (congrats!) – unfortunately it didn't go our way, but it is still the closest we have come to winning. It was great meeting the other teams (shoutout to the guys who played games with us on Saturday night), and we hope to be back next year.

Special thanks to Harrison Grill (their Chicken Katsu in particular), where we spent a lot of time coming up with strategies and ideas. And thanks to **XSquare** for being awesome!