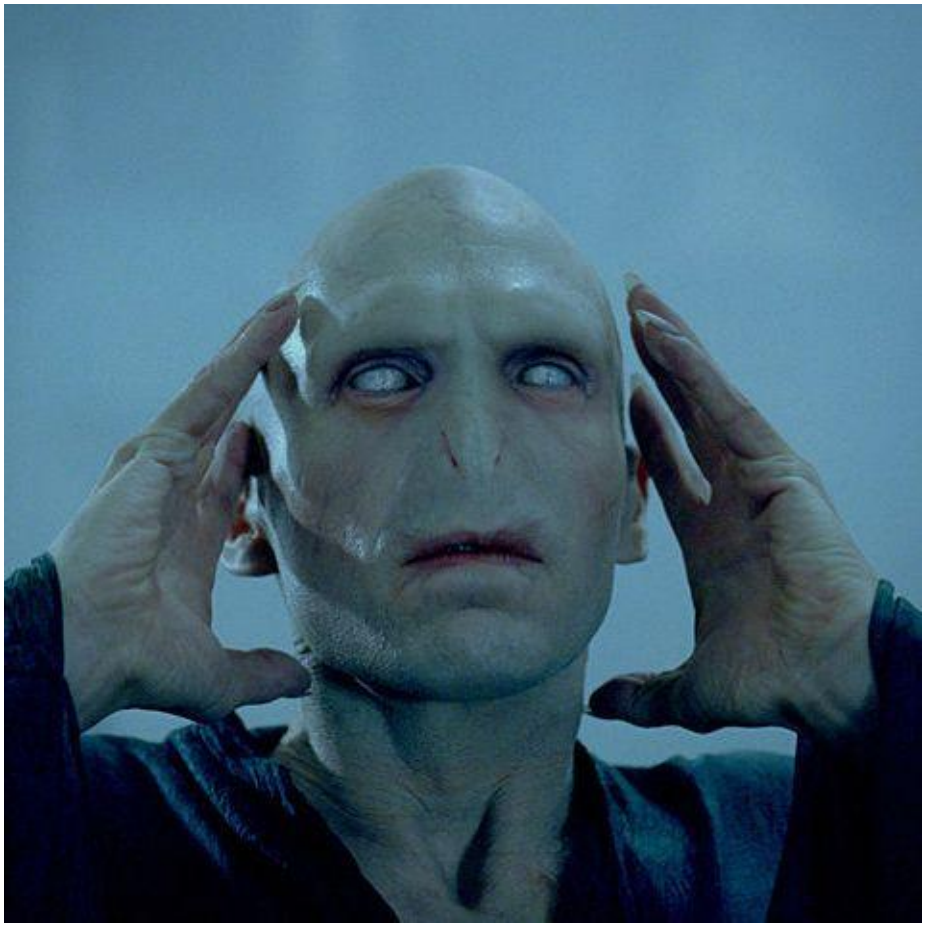
How to Train your VOLDEMORT

Strategy report team #260

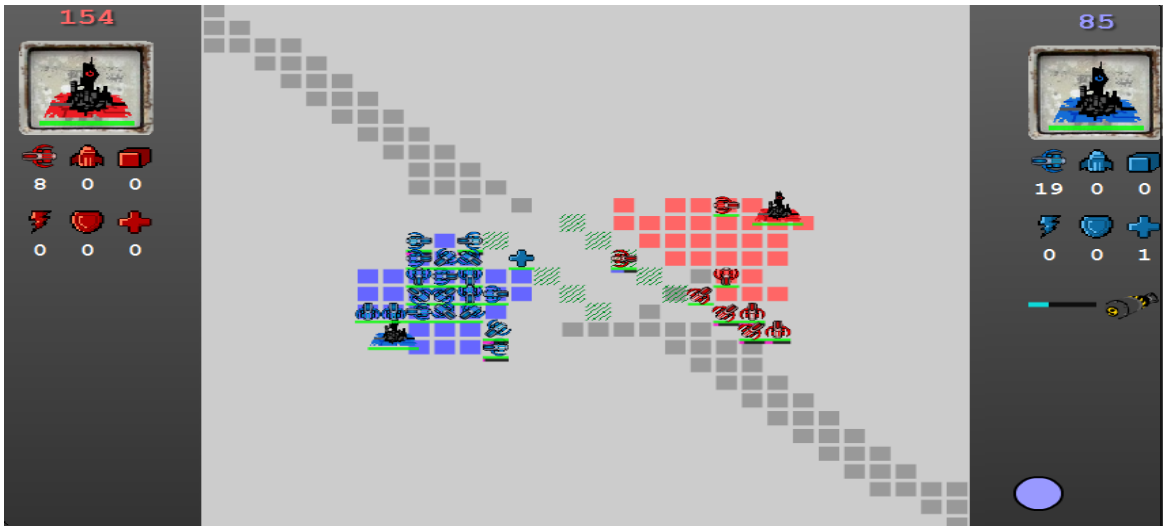


By Pedro Cattori, Brando Miranda, Kelly Zhang and Kaosisochukwu Uzokwe.

Contents

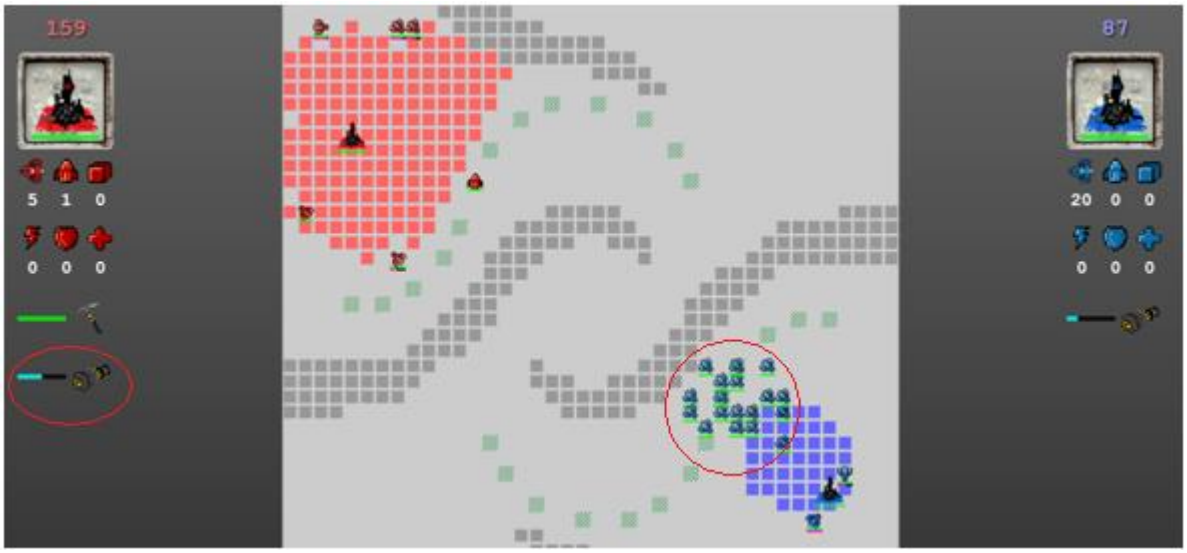
How to Succeed as a Newbie

Our first priority was to try to keep a quick development cycle when making bots to keep us grounded from straying off into theory-only approaches and observe the effectiveness of different strategies. Make sure that you have more than one bot and test it continuously against your other bots and against other people in scrimmages. This way you discover what is actually good in practice and what isn't. Playing scrimmages not only tests your code, but you gain insight on what other people are doing and how they are making your strategy suck. Also, listen to the advice past and more experienced people say. For example, a very big thing that made our bot very strong was that it had good micro. If your micro code is good, you can win engagements even with fewer bots. To be honest, our team would have NOT even implemented micro code if we wouldn't have been told in the lecture that Cory (winner 2012) gave. We weren't even aware really that it existed. Thus, this also points out, if you are a newbie and have no experience, consider strongly going to every lecture. You will find some good ideas that can be useful, or inspiring lectures by more experienced battlecoders. However, whatever they say, make sure you test it and see if it works. Being new to BattleCode and the bytecode limitations, we were initially hesitant when writing code, taking too much time considering bytecode costs. As it was slowing down our development cycle, we decided to do easy bytecode optimizations as we went, but don't worry too much about it unless we were writing an expensive algorithm. Basically, get started immediately and have bots. DON'T BE SCARED OF BYTECODES UNTIL THEY REALLY ARE A PROBLEM. The more bots you have, the more options you will have and the better you can mix your bots to make a strong bot. Also, if you ran out of ideas, or don't know how to approach specific problems, sometimes spending some time watching your scrimmages and watching the sprint and seeding tournament, could be a good idea for inspiration. For example, some of our micro code was effective because it used techniques that had been observed from the top teams for long periods of time, like 3 to 5 days. Thus, you could base your code by watching strategies and techniques that already work. Also, discuss them with your team mates, they can give you good feedback or point out weaknesses you had not considered.



The Optimized Nuke Strategy

In the end, we decided to go for a pretty bare Turtle-Nuke strategy. In doing this, we noticed that we were sometimes letting the other team beat us to start researching nuke and then we wouldn't react. To fix this, we had our HQ sense if the enemy nuke was half done every turn. If so, it then checked our own progress on our nuke. If we were past the halfway mark, we ignored the other teams nuke and kept working on ours since we had the edge. If not, we sent our defensive swarm all at once to try to destroy the enemy HQ before their nuke was done, or maybe at least delay their nuke by forcing the other team to build more soldiers for their own defense.

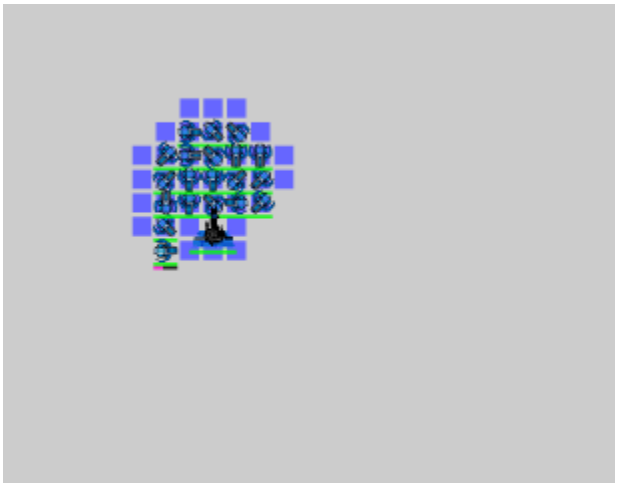


Note how we sensed half-nuke, and we went straight to attack the enemy HQ

We tried to choose our uninterrupted nuke start time such that an earlier nuke would have had to sacrifice it defenses, thus allowing us to sense their earlier nuke was half done and charge in for the HQ kill, or that other Turtle-Nukes would be slower, putting the pressure on them to react. Obviously, there is no such threshold, but through scrimmages and testing we settled on a timing we found to be successful.

To determine if we wanted to build the nuke at all, we checked our power generation rate. If we were losing power (negative power gen. rate), it meant we had a substantial enough defense in our opinion so we should start researching nuke immediately. We also didn't want to spawn if we had no power to do so, so we coded in this condition, and if we didn't spawn we researched nuke. To calculate the power generation rate, we did a scan for all of our generators. For each generator we added 10 to our power gen. rate counter. We new the HQ provides us with 40 power per turn, so we initialized our power gen rate counter to 40 to account for this. We then subtracted 2 from the counter for every robot we had. This is lower than the actual power gen rate as we never reach the 10,000 bytecode limit, so our robots are not consuming 2 power per turn. This approximation worked well enough in testing and scrimmages, so we kept it.

Basically, if you are going to nuke with an aggressive swarm, make sure you have very good micro that takes into account that you have support from you swarm! Also, if you are going to nuke, make sure that your nuke is as efficient as possible and its always ahead of other attempt-nuke players. Make sure you HQ is efficient and that it spawns when you need it to and it researches nuke when if best for you. Protect your nuke with mines and artillery if you can!



Note the positioning of the mines to defend the HQ and slow down attacks and give advantages to our defences.

The importance of effective Micro, Swarming and Static Evaluators

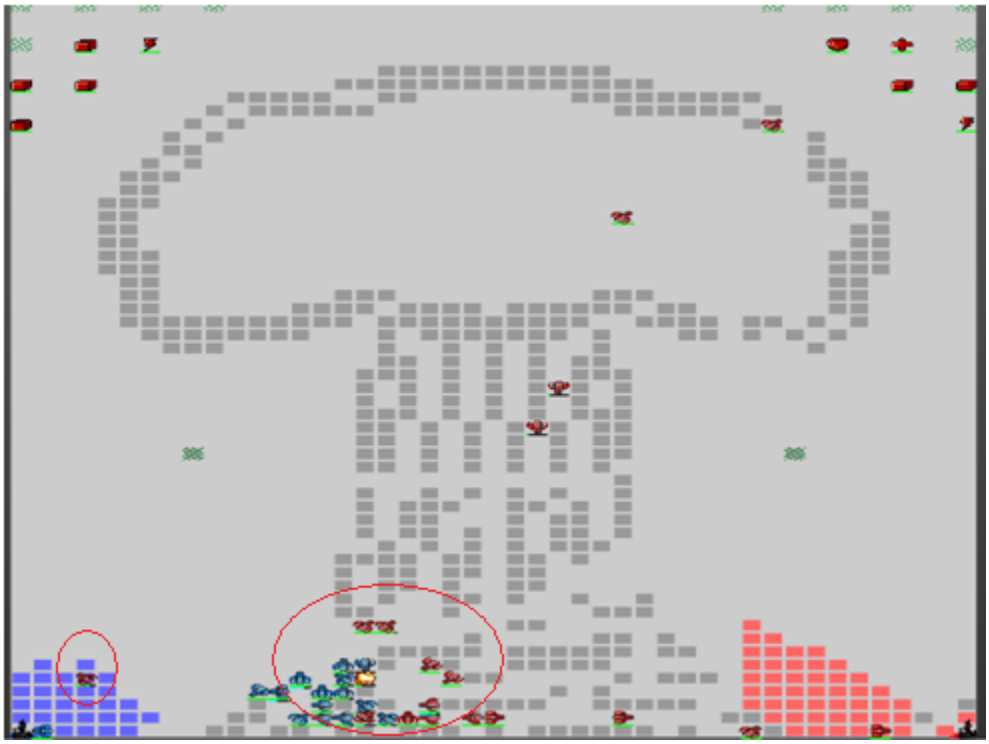
When your main strategy is to have an efficient and fast nuke, you need to make sure that you have a good way to protect it. This is where having a good micro is very important. The way you win battles is by winning engagements, so the importance of micro is enormous. However, coding for micro can be a bit challenging because there are many different cases that one can take into account and so many options one can consider. However, there is a way to maximize your micro depending on your strategy. For example, our strategy was based with an aggressive swarm that would lay mines to protect the HQ. Thus, the micro algorithms and static evaluators have to take into account that the micro is based on an aggressive swarm. For example, other micro codes that I observed took into they were a rush player. An obvious and very effective example of this is OLD.SCHOOL that had a semi-aggressive micro. His strategy would send bots to the enemy HQ and they would retreat if they sensed an enemy within a certain range. This had positive effect, since he would retreat towards his HQ were bots were being spawned quickly and he would take advantage of “accidental” swarm (of course it wasn’t accidental; they knew they would for swarms if they retreated, that’s why they did it). So micro has to take into account your overall strategy to be most effective.

Our strategy for micro, however, did not base that our robots were rushing, but rather that they had a very high probability of being part of an aggressive swarm defending the HQ. Thus, what our robots did was, if they sensed anyone in the shared vision close to the HQ or the swarm, they engage in battle and surround the enemy. This would force our bots to be in battle and engagements and when they went to an engagement, that's when the micro code triggered. If one robot was in battle, it would statically evaluate all the possible spaces that it could be and it would count how many enemy bots would be if he moved to that position and choose to move to the position with least enemy robots. This would maximize the damage the enemy robot took and minimize the damage the robot took. Further, if there were positions that statically evaluated the same, then what the robot would do is make a further evaluation and move to the spot where the allied robots were most giving damage to the enemy. i.e. it favoured positions that surrounded enemies. Further, if even like that, there were ties, the robot would chose to NOT stay in same position because we know that our strategy is to be an aggressive swarm. Thus, the chances that a allied bot is behind us trying to help in the battle is high, thus, move and make space for them because overall, it will increase our numbers and the damage the enemy take. All of these computations and evaluations did require for loops and expensive calls, however, they were so critical during battle that its definitively worth using. Furthermore, it never got close to using 10,000 bytecodes, so just do it!

Also, an important part of our micro was the med-bay. Every time we would try to build encampments we would make sure we had a med-bay close to our swarm and HQ so that instead of wasting time re-spawning, we used that to research nuke and give health to the bots that were alive. Also, our bots would stay in battle if in micro mode and not recover energy if it would damage the size of the swarm. They would only need it in critical situations.

The swarming behaviour was simple; it chooses a randvouz and the bots piled up there and started building mines to protect the HQ. The mines were vital in protecting the swarm and giving us time to re-swarm whenever our swarm decreased in size. Furthermore, it is quintessential to not be mining, defusing mines or building encampments while in battle, because you can't auto-attack! So our micro took care of that too.

A big problem that we had for micro was that having so many options paralyzed us from coding. Don't let that happen. If you run out of ideas or you don't whats good, try watching battles of people that have been successful in implementing micro. Watch, learn and adapt your algorithms to your own strategy!



Note that the mines defend us from backdoor attacks. Note the battle field and the micro-ing position of our robots.

Navigation and Pathing Algorithms

We started by considering pathing algorithms. We felt that finding the optimal best path was hard (in regards to bytecode cost/power consumption), so we wrote an algorithm that (we thought) guaranteed a non-increasing distance towards the goal destination by only considering moving to square with a lower distance to the goal. If there was a mine on one of these squares, we initially skipped it. After the initial sweep, if we had not chosen to move, we then considered defusing mines in the 3 adjacent squares toward the goal. This worked alright, but we had issues with soldiers getting stuck, either in a cycle, or behind our own robots. We tried keeping track of the previous square we were on to avoid getting stuck in a 2-square cycle. However, this did not resolve other cycles and our pile-up of robots. This should

We then implemented a probabilistic algorithm that kept track of how many times the soldier had been on each square considered for moving and favoured squares that it had not moved to with some probability. This resolved a lot of cyclic issues, but not all. And the pile-up was improved but not removed. This was not an issue for swarming, so it worked well for our defensive swarm, but so not for our encampment capturing soldiers.

For them, we settled on an algorithm that considered moving away from the goal. Each of the

adjacent squares was scored. This time we used movesAway() as our metric of distance, instead of the distanceSquareTo() provided. We don't care how far away something is in euclidean space if we can move diagonally each turn. movesAway() took two MapLocation arguments an return the maximum of the differences between the two arguements' x coordinates and between the two arguments' y coordinates. This tells us how many moves it would take to reach one MapLocation from another without considering mines and enemies. The score was also dependent on whether or not there was a mine on that square, and if so, whether or not our team had researched Defusion. We also weighted squares based on the number of times the soldier had visited that square. We took all of these metrics and added them up to get the score for the square. We were careful to weight moving backwards as higher than defusing a mine in a forward square, but less than moving to a previously visited square. For about 5% of our encampment capturing soldiers, we saw that they took a clearly suboptimal route, but still managed to reach their goal. However, for the other 95% encampment capturing soldiers, there was a marked improvement in their pathing decisions.

Defend yourself from Backdoor Attacks

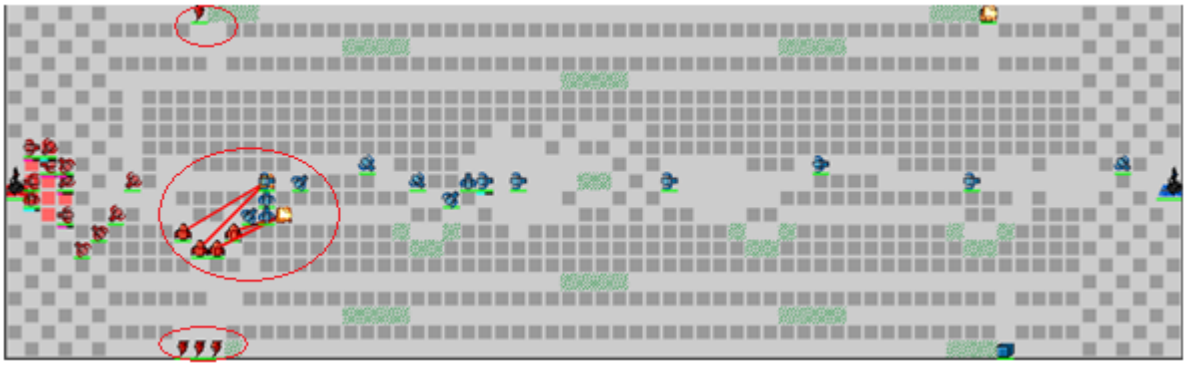
We had problems early on with our swarm just hanging around while a lone soldier from the enemy team backdoored us and killed our HQ. We changed our code around to make the swarm attack the visible enemy closest to the HQ, instead of the one closest to the rendezvous for the swarm. We also added a broadcast from the HQ that alerted our army if an enemy was within the HQ's filed of vision. If so, all soldiers headed back to the HQ to block/stop the backdoor attack.

Note that the mines defend us against backdoor attacks and that the soldiers protect the HQ when the attack is detected.

Balancing Macro and Econ Strategies

Once we had code to get our encampment capturers to get to their target encampments, we needed them to decide which encampment to build. In our early development we were experimenting with a macro based player, so we used our power gen. rate function to tell us if we needed more power or not. If so we prioritized building generators. Unfortunately, if multiple encampment capturers were trying to capture and our power gen rate was low, they all decided to build generators. We decided to make a predicted power gen rate function that added the number of generators being built to the number of generators already built. We did this by broadcasting to a channel that kept track of the count of generators in progress. When a generator was built we decremented the count in the channel by 1.

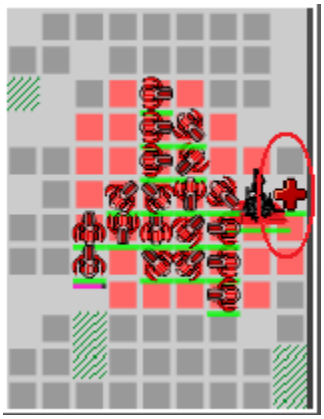
We also needed a way to keep artillery in areas most used by the enemy. An artillery off in the corner won't do much good. We made a function that determined if the encampment square about to be captured was in our specified "Artillery Area". To select this area, we chose squares that had a direction towards our HQ that was opposite to the direction to the enemy HQ. Because of the way direction was determined, this actually traced our a nice oval-ish shape with the two HQ's on the perimeter. Anything inside the oval was a viable artillery position.



Note the positioning of the artillery, straight in the middle where they are efficient at defending attacks. Also, note how generators are placed in locations out of the battlefield so that they don't get destroyed.

We noticed that when our swarm was attacked, we lost some soldiers and our HQ spent valuable turns spawning replacements instead of researching nuke. We went with a Medbay if an encampment was sufficiently close to our HQ/swarm rendezvous. In the long run, we would heal soldiers instead of replacing them, allowing our nuke to progress more rapidly.

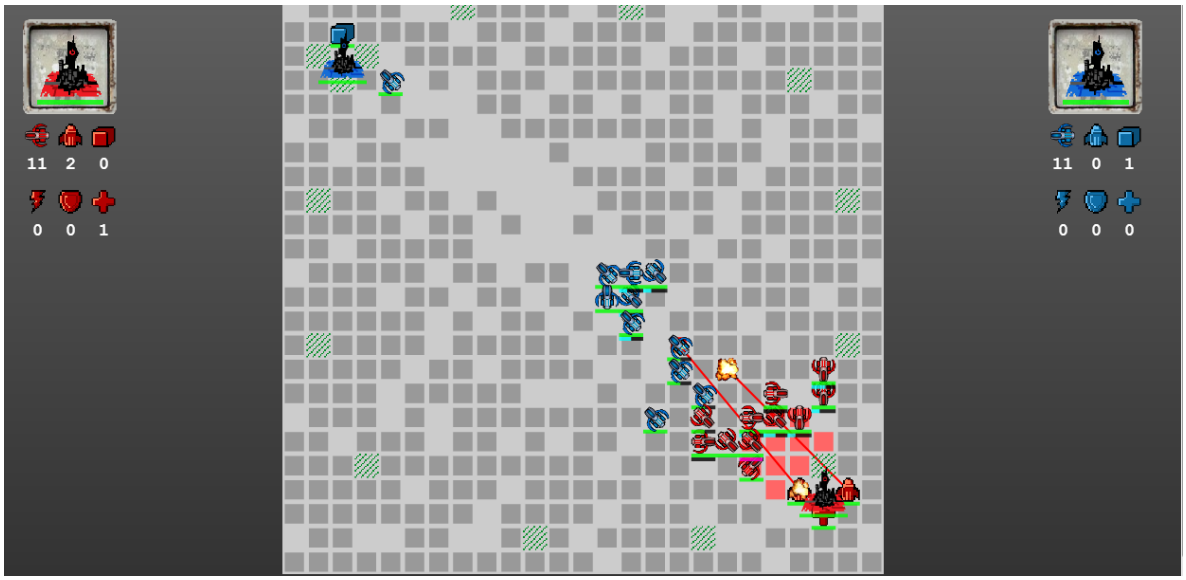
With the decision of a defensive swarm came the decision to avoid building suppliers. We needed a mass of defensive units and we didn't care if they accumulated slowly. Hopefully our mines, artillery, and medbay would hold off any initial rushes and allow our swarm to build up.



Note the medbay is in a great location, protected and easy for our robots to use.

Generators were only considered on maps with a high number (>40) encampment squares, as we felt we might have to hold off a macro based player and a small swarm wasn't going to cut it.

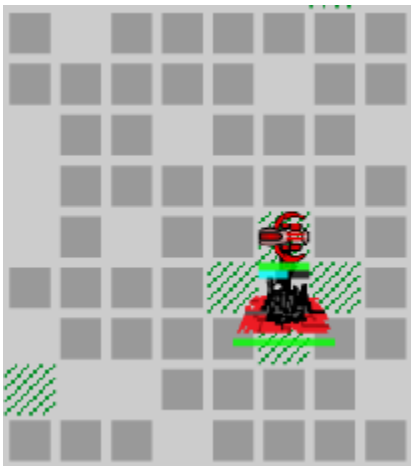
With artillery built and the success we had in pathing with static evaluators, we had our artillery use the same idea. We scanned for enemy robots inside the artillery range, and then sorted the enemies based on distance from our artillery. If an attack on that square would cause hits to 1.5x more enemies than allied units, we would fire. We knew that our medbay would heal some damage done to allies and we didn't want our artillery to stop giving us an advantage on soldier battles.



HQ code can make the difference

Some maps started out with mines and encampments completely surrounding the HQ. Our HQ looked for such scenarios. If surrounded by only encampments, it decided on a "spawn square" from these squares designated NOT to be captured. If surrounded by a combination of mines and encampments, it chose a mine square as the "spawn square", but spawned the first soldier on a non-mine square and broadcast the need for a defusal, so that the first soldier would first defuse the mine on the spawn square and then proceed to whatever task it was assigned.

We were worried of trapping ourselves with our own encampments, especially after witnessing the embarrassment of the rapunzel map. At first we made a "clear path" function which drew a straight line from the HQ and if an encampment square was on the line, the soldier capturing it would stop, or the encampment on the line would suicide. Crude, but guaranteed we wouldn't get stuck. Unfortunately, choosing a good path to clear was not easy, especially since a lot of maps have the majority of the encampments in some sort of line passing through the HQ. Our second idea was to use a scout, which is explained in detail under the section "If we had more time…".



This is an example of a nasty map were, without good HQ code, it could end badly.

Implementing personalized data structures and algorithms

Another thing that is important is using personalized data structures instead of the built in data structures in Java. Implementing hash functions and hash tables with array are easy to implement and use less bytecodes. Also, if you plan to use sorting algorithms, make sure you implement in-place sorting algorithms rather than using sorting algorithms that make copies and copies of arrays. We implemented quicksort, only using pointers and it's not too expensive, nor difficult to implement, considering that CLRS explains very clearly how to implement them.

If we had more time…

Distributed sorting with MinHeap:

Well to be honest we actually did implement this one! But not in the robot we submitted. In the final moments before the deadline, we decided to leave our quicksort as it was since we ended up only sensing for encampments relatively close to the HQ, so we thought our quicksort would do the job and we wouldn't have to worry about switching it for our minheap sorting algorithm on the last day, focusing instead on micro and encampment decisions. When we were still using our macro bot, which considered taking much more encampments, we wanted to sort the encampments with respect to minimum distant to HQ. We did notice that on some encampment-heavy maps, our HQ was reaching its bytecode limit trying to sort all the encampments on the first round. We also realized that we did not need them all to be sorted right from the get go. This lead to the idea of using a minheap, which would take O(n) to build, and O(lg n) to pop from, instead of using a quicksort with worstcase O(n^2). Every time we spawned a soldier, we simply popped another element of our minheap (held by the HQ) and broadcast it to a series of channels

designed to act as an array of MapLocations. Because the array was in sorted order, the soldier simply scanned the list for the first encampment not already assigned or the first encampment that had been destroyed as this would correspond to the nearest available encampment. By distributing the work over many turns, our HQ was now well below the bytecode limit, and saving us power compared to our quicksort implementation.

Messaging attacks:

While we did depend on broadcasts, we made an effort throughout the competition to remove our dependency on them anywhere that was unnecessary. We did this anticipating that other teams would message attack us, especially one the cost to write a broadcast was reduced from 1 bytecode to .03 bytecodes. This inspired us to think up our own message attacking protocols. We were considering having restrictions on the channels we used (such as only using channels in a certain range, or having fixed channels). This would allow us to know which channels or range of channels not to attack. We basically wanted all robots consuming few bytecodes (<500) to use some bytecodes to message attack channels we did not use. However, iterating through channels using a loop takes more than a bytecode per iteration, which quickly adds up to not much message attacking. Instead of looping, we considered writing metacode that would simply write ~10000 lines of code with each line being a broadcast with some garbage data to a unique channel, skipping over the channels/range of channels we were using. We could then put this code into the run method of our generators, suppliers, and medbays (and shields if we had any), who usually spend around 20 bytecodes per turn.

Scout:

After trying the "clearPath" solution to avoid trapping ourselves with our own encampments, we came up with a concept for a scout that would help us avoid trapping ourselves without the shortcomings of "clearPath". The basic concept is to have a scout attempt to reach the enemy HQ while avoiding going over encampment squares, but preferring encampment squares to moving backwards or revisiting a square. Any encampment square it walked over would be marked (via broadcast) as a disallowed encampment square. This would limit the amount of encampments we would choose not to capture. As we ended up with a Turtle-Nuke that captured a max of 7-10 encampments, this became much less of a risk, so we decided not to implement anything further.

Acknowledgements

Our team name was taken from two sources: the Right Now Lectures we all attended last semester for 6.034 and Voldemort, because we like shouting Voldemort at each other.

Special thanks to Cory Li. That lecture is probably the reason we've made it this far.

Finally, THANK YOU DEVS! BattleCode was an awesome experience we hope to repeat in the years to come.

Sincerely,

The members of team rightNowLecture.VOLDEMORT()