AI Challenge: Ants

Foivos Antoulinakis, Nick Babcock, Jamie Danielson,
Ryan Landay, Brij Patel, Kumar Saketh

December 22, 2011

1 Introduction

The supercomputer Deep Blue was able to beat Garry Kasparov at chess in 1997 largely through massive computational power and exhaustive search. It was obvious very early that such an approach would be extremely infeasible for the Ants game. Each side in chess starts with 16 pieces; even if we could choose any piece and move it to any of the 64 locations on the board, there would only be 1,024 possible moves at each turn. By comparison, with the Ants game, we have to decide each turn, for each ant, whether to not move it, or move it in one of the four possible directions. So if the ants are spread apart on a map, away from obstacles, we would surpass that number with only 5 ants: $5^5 = 3,125$. With 20 ants, there would be $5^{20} = 95,367,431,640,625$ possible moves. It's not even feasible to evaluate all of our own bot's possible moves, let alone evaluate a search tree like a chess program. This wouldn't even really be feasible to evaluate separately for small groups of ants. Additionally, the bot has limited information about the world, so even if this were computationally feasible, it isn't clear how we would evaluate positions or take into account enemy moves and random food spawns. The goal of the competition seems to have been to get contestants to design bots that approach the problem with some degree of strategy, as a human would.

The game is a turn-based strategy game designed to favor aggressive bots. The only way to score points is to raze an enemy hill. Bots gain two points for each enemy hill they destroy. However, each bot loses one point if its hill is destroyed. In order to get more ants, they need to gather food. For every piece of food a bot gathers, it gets one new ant. Lastly, fighting is determined by an algorithm such that whoever has more ants in an immediate area wins.

We ended up structuring our bot into four main parts. The highest priority is sending ants to capture nearby enemy ants hills we can see, as this is how we score and eliminate enemy players, so we do this first. Early versions of our bot did this without regard to how well the hill is being defended. The next-highest priority is collecting visible pieces of food, so we take our remaining ants and move them towards nearby pieces of food. We then call a function that uses a complex strategy to engage enemy ants in battle. This appears to be a relatively

critical part of the code, looking at battles between highly-ranked bots; before this was finished, the function was empty, and our bot did not respond to enemy ants. After this, we call a function that uses the remaining ants to explore unseen parts of the map; this actually requires some creativity. An earlier version of our explore function was computationally intensive and tended to exhibit strange oscillating behavior. It wasn't immediately obvious that such a separation of responsibilities would work well (as opposed to a more integrated approach), but the final version of the code still has basically the same structure, although the individual functions are more involved now. The final version of our code has a three explore-related functions called, with the second one trying to move ants in the general direction of enemy hills, and the third one a last resort effort to find something interesting to do. We also added a function at the end to move ants with nothing to do that would be obstructing our other ants.

In order to easily have multiple people working on the code at the same time, it quickly became apparent we would need to utilize some sort of version control. Having individuals editing the same file and then trying to share that file resulted in much confusion. Often their changes were in vain because others would make changes that broke the new features. To help rectify this, we chose to use Git. Git is a non-linear, distributed version control system. Our git server is hosted on James' server, alpha.gekinzuku.com. This helped alleviate the difficulties of collaboratively working on the project.

2 Architecture

Our code's architecture can be thought of as a three layered onion. The top layer is our Bot struct, which houses the power to decide which ants have what jobs, such as food gatherer, explorer, "hill razer", or fighter. To get the information to make these decisions we must travel down into the State layer, which has several important functions. Inside State reside the variables about the world and world algorithms. We are able to set such variables by reading the files that the game engine sends us, and then we have the ability to construct the world. This is abstracted through the last layer, the core of our bot, which is a stacked array with each element correlated to a specific row and column. The elements in the array are Squares. The Square class encapsulates an unsigned char, which is able to hold eight flags. From these flags we can describe any spot on the map. With the addition of keeping track of who owns the Square, we are able to describe any Square in fourteen different ways. Therefore, just by analyzing this array we are able to determine whatever we want, such as where food is or where enemy ants are threatening us.

Bit Position	Value
1	isThreatened
2	isFocused
3	isExplored
4	isFood
5	isAnt
6	isHill
7	isWater
8	isVisible

Additional Flags
isFriendlyHill
isEnemyHill
isFriendlyAnt
isEmpty
isMovingAnt

One last aspect that a Square keeps track of is how many ants have died in that square and who they belonged to. Nowhere in our code do we use this; however, it also was not a strain on any of our resources, and so we decided to keep it around for sake that it might one day become useful. Overall though, we believe that the Square class is extremely efficient with space and offers us anything we would hope to know about the map. To keep the the Square array up-to-date is the State's responsibility. As soon as a turn starts, we read in the information that the game engine sends us, putting the locations of enemy ants, enemy hills, and food into vectors. We chose to use vectors, as they represent a contiguous block of memory, which is efficient for random access, insertion and deletion at the end, and iteration. In our code we use the locations of enemy ants, enemy hills, and food only for iteration, and so a vector is a perfect fit. On the other hand, we store our ants inside a list, so it has a $O(1)$ insertion and deletion anywhere inside of it. We chose a list because if an algorithm wanted an ant only for itself, it could remove it efficiently and continue on without a slowdown. Besides those variables, State keeps track of a few distance algorithms, so when we want an estimate from one square to the next, we can call one of them. Then when Bot has decided that it wants to send an ant to a certain position, it calls State, which then sends the information to the game engine.

Not applicable to the onion analogy but nonetheless integral to our code are a few other classes and structs. One of them is the Location struct, which keeps track of the row and col, which is used throughout, and so we ended up writing an overloaded equality operator. Next is the Ant struct and is used to track just our ants. Basically it is a wrapper struct around a boolean flag saying if it's busy and its current position. The last of our structs is the Route struct, which holds an ant, where it wants to go, and the distance between them. There were a couple structs from the starter package that we didn't even need to touch or look at. They were the Timer struct which returned how long it has been since the start of our turn and the Bug struct which we used as a logging tool.

The latest addition to our code and by far one of the most influential is the Map class, which was born out of the necessity to cache distances between Squares. Before this class we used a four dimensional array to track every distance from each square to all the other squares. This soon became a problem

as illustrated below.

$$\begin{aligned}
&\text{Max row size} = 200 \\
&\text{Max column size} = 200 \\
&\text{Array size} = (200 \cdot 200)^2 \\
&\text{Array of short integers} = 2 \cdot (200 \cdot 200)^2 \approx 3.2 \text{ GB}
\end{aligned}$$

Since 3.2 GB exceeds the 1 GB we are given, this method simply doesn't work. The idea with the Map class is to only keep track of distances from what we deemed important Squares (i.e., Squares that had food or was an enemy hill). When we read in the information we would see if we detected new food, and if we had, create a new Map, else if the food disappeared, delete that specific Map. When a map was created it would start from goal square and then work outward in a breadth first fashion establishing distances to all the explored squares with respect to obstacles. It also kept track of something we called "degrees of freedom" for each Square. Basically it was how much wiggle room that square had and still be able to reach the goal. The freedom of a square is calculated as follows:

$$\begin{aligned}
&n = \text{number of squares available to go to} \\
&m = \text{minimum freedom of adjacent squares} \\
&\text{Freedom} = n + m - 1
\end{aligned}$$

Once these arrays were filled, we were able to call upon Map to see what the best move would be to advance ourselves to the goal. The best move is the neighbor that resulted in the lowest distance but also the highest freedom. If in the following turns we see a previously unexplored square, we pass that location into all of our Maps, which then update distances and freedoms accordingly. The Map class is used extensively in our bot as it gives us the potential to query our ants against it to see what ant is the closest and then it will determine the best move for us. This class is in large part why have succeeded today.

3 Distance algorithms

The bot's main parts rely on several algorithms to perform their computations. One important building block is how to estimate and compute distances. There are three traits about how the AI challenge is structured that makes the estimation of distances non-trivial. First the edges are wrapped. This means that ants can travel to the edge of the map and appear on the other side. This in effect forces a calculation of two distances, as it could be faster for an ant to wrap rows and/or columns. Next comes the fact that ants can't travel in a straight line to and from locations diagonally apart from each other. They are forced to either go North, East, South, West each turn. But the biggest factor that can cause a misjudgement is water. An ant could be two squares away from food, but in between them could be a wall of water that will make any computation

that takes into account only the starting and ending location give a gross underestimate. Therefore we have used three formulas throughout development: the Euclidean distance, the taxicab or Manhattan distance, and a custom one based on the Bresenham line-drawing algorithm. Each of the listed formulas have strengths and weaknesses, which makes each useful for different purposes.

The Euclidean distance in a two dimensional plane is the straight line distance between two points, which makes it not useful at estimating distances for all but pairs of points that lay on the same axis. However, it comes into use when dealing with updating map information, such as what an ant can see or what squares are threatened by enemy ants as they are certain radius away. Using the Euclidean distance allows us to set these flags correctly. The taxicab distance will return how many squares two locations are from each other if one cannot traverse a diagonal. This metric is attractive to use as it outputs how many turns it will take to reach the goal. The two formulas are illustrated below and take into account row and column wrapping.

$$
\begin{aligned}
&\text{Distance between points } P \text{ and } Q \\
&D_1 = |P_{\text{row}} - Q_{\text{row}}|, D_2 = |P_{\text{col}} - Q_{\text{col}}| \\
&\Delta\text{Rows} = \min(D_1, \text{Rows} - D_1) \\
&\Delta\text{Cols} = \min(D_2, \text{Cols} - D_2) \\
&\text{Euclidean distance} = \sqrt{(\Delta\text{Rows})^2 + (\Delta\text{Cols})^2} \\
&\text{Taxicab distance} = \Delta\text{Rows} + \Delta\text{Cols}
\end{aligned}
$$

To give a concrete example about how these two formulas differ, image two points $(0, 0)$ and $(3, 4)$. The Euclidean distance will return five and taxicab will return seven. The question might be posed, why would we ever use Euclidean over taxicab when calculating the distance between an ant and its goal? Imagine ants at $(0, 0)$ and $(3, 11)$ with the goal still being at $(3, 4)$. Both ants will need to take at least seven turns to reach the goal. The key difference is when there is an obstacle in the way. The ant going south from $(3, 11)$ will need to add at least two more moves to reach the goal as it has circumnavigate the obstacle. On the other hand, the ant from $(0, 0)$ has the option of going North or East, so the obstacle will be of minimal hindrance. In this instance we would prefer using the Euclidean distance. Yet if we moved the starting positions of one of the ants from $(3, 11)$ to $(3, 10)$ and removed the obstacle, it will only require six turns to reach the goal, whereas the other ant still needs seven. The Euclidean distance will still say the ant at $(0, 0)$ is closer to the goal, which is wrong. Therefore the accuracy of the Euclidean and taxicab distance functions vary depending on the situation, and there are situations where we could plausibly consider using the Euclidean distance for distance estimation.

The third distance algorithm, which is quite different from the Euclidean or taxicab, is the Bresenham. The original algorithm described by the author, J. E. Bresenham, is "given for computer control of a digital plotter." While it was created for plotting the best fit line between two points, it can easily be manipulated to return the distance between two points with respect to obstacles.

This is achieved through Bresenham's most distinguishing feature: traversal of each square along the desired path. The start and end points can be thought of lying on a line created by a linear equation of the form

$$y = mx + b$$

With algebraic manipulation the linear equation becomes

$$0 = (\Delta y)x - (\Delta x)y + (\Delta x)b$$

When a square's coordinates are plugged into the equation and outputs a zero, then that square lies along the best fit line. Since this algorithm allows us to see what squares lie along the path, we can account for obstacles. If the current square that we are evaluating is water, get the number of adjacent water locations and then add two, and that is the estimate to maneuver around that particular square. Of course, if there are no obstacles in the way, the Bresenham will return the same value as the taxicab.

It may then be surprising to hear that in the current state of our code we do not use the Bresenham anymore. This is largely due to performance issues we did not foresee, and also because the algorithm's occasional distance overestimation limits its usefulness for pathfinding. We did not anticipate performance issues, as Bresenham wrote, "the algorithm may be programmed without multiplication or division instructions and is efficient with respect to speed or execution and memory utilization." These issues are probably a result of the sheer number of times the function was getting called from other algorithms coupled with a less-than-efficient implementation. The checking of how many adjacent water locations there are and getting the next location probably cause some of the slowdown; the algorithm is not inherently slow, however, as with some slight modifications, "it is capable of drawing lines at a rate of 2.2 million pixels per second on a 486/33," technology from the early 1990's.

Even if we could fix it to avoid performance problems, it's still flawed so as to be unusable. Experiments showed that it overestimates the actual distance in about 1% of cases. This situation would only arise when one of our ants wants to take a diagonal path to its goal and along the line of best fit is one square of water. The Bresenham will add two onto the total distance, whereas realistically the ant can still get to its goal in the same number of turns as if the obstacle was not there because on a diagonal there are generally two moves that brings an ant closer to the goal. With this possibility of an overestimation, it cannot be used as an admissible heuristic in search algorithms. In the end we settled on using Euclidean for updating map information dealing with radii, taxicab for estimation, and we learned that Bresenham should be left to what it does best: drawing pixels.

4 A* search

While many of the computations we'd ideally like to do for the Ants game are computationally infeasible, optimal pathfinding is, fortunately, very practical.

The simplest way to do this is to start at the start square and begin searching each neighboring square, essentially keeping a list of the shortest known paths to each square, and the previous square so the path can be reconstructed. We can set this search up to be breadth-first and expand nodes in the search tree with the shortest paths first, so that all paths are found in order of length, and if a path to a new square is found, it must be optimal.

Breadth-first search works well if one wants to calculate the distance between one point and every other square on the graph, e.g. to find the closest ant to a piece of food. If we only want to find one path though, it requires searching more nodes than is necessary. A* search improves on breadth-first search by using a heuristic to estimate the remaining cost from each node to the destination node. Similarly to how breadth-first search expands nodes in order of total path distance so far, A* expands them in order of the sum of total distance so far and an estimate of the remaining path distance to the destination. If we have a heuristic that never overestimates the total remaining distance (an "admissible" heuristic), A* is still searching the shortest possible paths first and the result is still guaranteed to be optimal. Our A* algorithm uses the taxicab distance as an admissible heuristic. This essentially causes the search algorithm to search paths that move directly towards the goal before trying to find paths that maneuver around obstacles. The taxicab distance is a very good estimate when there are few obstacles on the path; on very twisty, maze-like paths, it underestimates quite a bit and the algorithm has to search more nodes.

While A* works very well for finding the optimal path between the ant and the food, it still wasn't quite fast enough to call it as many times as we wanted. Although it's guaranteed to return an optimal path when given an admissible heuristic like the taxicab distance, we experimented with using the non-admissible Bresenham function because it appeared to fix some situations where an ant would get stuck trying to decide between multiple pieces of food because the next step of the found path takes it closer to a different piece of food according to the estimation metric. However, as mentioned above, the Bresenham function turned out to not be fast enough to be used as a heuristic in A* search. As the number of ants increased, this became a very large problem. So we decided to use taxicab distance as a heuristic instead.

Once we had fixed that problem, we had some computational time to spare and tried using A* to find the path between each food and every ant. This appeared to work on small maps but sometimes has problems on large maps or with large numbers of ants, in which case A* would try to explore more nodes than we had time for. The search tree is deeper for longer distances, making it computationally costly. We tried to optimize it by only letting ants within a certain distance (estimated by using taxicab distance) run A*. In spite of this, the code still sometimes timed out in larger maps, although profiling suggested that at least part of the problem might have been due to the explore function. We ended up developing alternative approaches instead of investigating the problems further.

The solution was to use the Map class. We isolated where we were calling A* search, and it was when we were going to food or going to hills. We decided

to create Maps to these important squares as a Map has all the distances from all the squares already calculated using breadth-first search. This alleviated the need for an algorithm to calculate distances, and when we did find an ant closest to the goal square, we simply called Maps give us the best move.

5 Priority queue

Knowing what the next best move can hold substantial rewards if it can be implemented correctly. Our thought here was to create a data structure that allowed us to hold all possible combinations of the distances between ants and possible goals. The best move would be the ant that is closest to a particular goal. Since we know how many combinations there are (number of ants · number of goals), the straightforward solution would be to create an array to hold all the distances, populate the array, and then use a sorting algorithm. We could then iterate through it finding all the best moves. In hindsight this may have been the most efficient method; however, we decided to go with `std::priority_queue`, mainly because of its ease of use. All we needed to do was instantiate it with a comparison class for Routes, push all the combinations, and pop them back out. Extremely easy, but, again, may have not been the best decision. The default underlying container in a priority queue is the vector, which is known for its random access but every time a Route is pushed onto the queue, rearrangement of elements may be necessary. Even with a heap implementation, the aforementioned method of using an array is more efficient in the aspect that we know how many elements the array will hold and we know when we are done adding elements so we can sort it just once. Ultimately, the priority queue did not seem to be causing a bottleneck, so its usability outweighed the performance cost.

6 Battle resolution

Battle strategies are one of the most important aspects of the game. Our objective is to raze hills, and there is no way to do so unless we first kill the ants that defend it. Also, while growing our army, we should be killing enemy ants with the minimum casualties possible. How do we do so though?

First we have to understand how battles work. Each ant that is attacked is assigned a number equal to the enemy ants within its attack radius. Then each ant survives only if all the enemy ants attacking it have larger numbers assigned to them. This makes predicting results of battles without knowing the placement of every ant very complicated, because one enemy ant having the same number as one of ours, while we "thought" it to be higher, can result in our ant dying.

The first problem we have in making out battle strategies efficient is finding which ants are fighting together, so that we don't have to account for ants that are insignificant to our calculations. The best way we were able to think of is to find the first enemy ant that has the ability to attack one ant of our own, assume

both of them move towards each other, and add it to a vector, enemySquad. Then find all ants of our own that can be attacked by that ant and add them to mySquad. Then find the ants that can be attacked by the ants in mySquad and add them to enemySquad, and so forth, until there are no new additions to be made. This ensures that all ants that should be accounted for are, while no extra ants are included in our calculations slowing us down.

Then how do we figure out our moves? Should we try to calculate every move that every ant in the fight can make? Should we do so for multiple turns for best results? Some basic calculations immediately rule out calculating every possible move, as this would time out extremely easily. We therefore have to find some way to ignore some of the moves that can possibly be made, to speed it up. To do so we came up with the following rules. First of all, for every ant in our squad, if it has a number n assigned to it, it means that n enemy ants will have their numbers increased by 1 because of it. Therefore, the sum of all the assigned numbers to our ants must equal the number of the enemy ants. That is of course assuming that there is only one opponent attacking our ants at a point of time. Fighting two or more opponents simultaneously happens extremely rarely and is something that we can overlook. So we should try to have more ants than our opponent does. Still, the only way to do so that initially came to mind was to calculate every move we could make, and see what the result could be, which is not an improvement at all.

Then we realized that there was a way to ignore most moves: our opponent's ants can only move towards us, stay still or retreat. First of all, if they retreat there is either not a battle, or, if only some of them retreat, we will have more ants engaging, which is our goal. So the only two cases for which we have to account for are the enemy ants staying still, or moving towards us. If they stay still, we do not have to calculate anything for them, because we know their exact positions, but if they move it seems that we still have to. Assuming though that the enemy ants are forming a squad that is only on one side of our ants, that is, our ants are in a somewhat close formation, the enemy ants can make one or two moves to approach us. These moves though lead to very similar results. If there are two moves for each ant to approach us, it means that our ants are diagonally to them. In such a case, they will have probably formed a diagonal wall. Let's assume that we are to their northwest. For every one of the ants moving north, there will be one ant that can't move west because of it, and vice versa. So if they all want to approach us they all have to move north, or west. The difference between each radius is virtually only one square to us, which is a very minor difference. So we can treat all squares the enemy ants can go to as the same for our calculations and call them threatened squares.

Now that we have found which ants are engaging, and we have found a way to ignore the number of the enemy moves for our calculations, we have to determine which exact squares are going to be considered threatened. The first idea was to treat as threatened all squares that are directly outside of the enemy's attack radius. But if there is water blocking one or more of the enemy ant's moves then it's actually threatened squares are less, and we should account for this fact. So we have to check and see every move the enemy ant can make,

and assign threatened squares according to those.

Afterwards we find and store in an array of vectors how many and which ants are attacking each of our ants for every move our ant can make, and how many can attack it next round for every move it can make. This is done so that we don't have to calculate these every time while trying to identify our moves, effectively speeding up our program.

Next we'll present the basic algorithm that is used to find out the best moves for each ant. For each and every ant we either assume it will retreat, not finding its move, or we will find all moves that lead to a "desired square," and then go to the next ant until we've determined what all ants will do. For each ant, if it moves to a desired square we increment a counter by one, and for every new enemy that it includes in the battle, we decrease it by one. After we've found all moves the counter is greater than the maximum, these moves are copied to a vector, bestMoves.

This above algorithm is called multiple times, as we have to be sure that we can outnumber our enemies both if they stay still and if they move towards us. So first we call it for desired squares being the ones that are being attacked right now, and then if can lead to a maximum that is greater than zero we also call it for the desired squares being those that can be attacked next turn. If the maximum is still greater than zero we make these moves. If we didn't find any moves that are valid, then we call this algorithm again for desired squares being those that are threatened.

What the above effectively does is, we can be very aggressive, moving our ants directly attacking the enemy ants, we do so, else we try at least making sure that if the enemy ants proceed, they will lose. If we cannot even do that, then we have to retreat.

Now let us make small specifications about the above. First of all, the algorithm above works recursively, calling itself for every next ant. Also it prioritizes solutions that have the least amount of our ants engaging for the same value of the maximum. More specifically, if we can chose between attacking four ants with five, or attacking one ant with two, then we will chose the second one, as the ratio of ants is greater.

This algorithm seems to work very well in the amount of time we have, only crashing for immense battles with more than 20 ants on each side, which happens extremely rarely. It is also timed so that if that is going to happen we just make the best moves without trying to improve on them. So far the battle strategies seem to be working very well, not having problems against bots that are ranked even better than ten, which is impressive.

7 Exploration

For the first version of the explore function, we thought that iterative deepening depth-first search would be an efficient method to find the closest not visible Square. Thus, this would allow us to more effectively spread around our ants all across the entire map, which has many advantages for us, including knowledge

of the map, food locations, hills, and enemies. The reason that we first chose iterative deepening was because it performs a depth first search while traversing through the nodes on the same depth in breath-first method. If the solution was not found at the depth, the depth is increased and it starts over. Therefore, iterative deepening is complete in the fact that it allows us to find a solution if a solution exists. If we consider the branching factor, or number of possible paths from each square, as b, and we know that the closest solution will be a depth of d away, we can formulate iterative deepening a bit better:

$$\text{Time complexity} = O(b^d)$$
$$\text{Space complexity} = O(bd)$$

With the AI competition, neither the branching factor nor the depth is constant. This is because as iterative deepening expands out, it remembers the nodes it visited. The first branch has four unvisited nodes and then followed by several three branches until the majority of nodes have only two branches that haven't been visited. The most volatile variable is the depth of the solution. If an ant is in the middle of nowhere we know that it would have to travel one square outside its view radius, so the depth would it be approximately one plus the square root of 77. Needless to say, as more ants populate the map, iterative deepening will have to expand increasingly more squares to find that one that is not visible, and this became a problem. We would have many of our ants wanting to travel all the way across the map to find that one square that is not visible. Then once we found the spot, we called the A* search function. Having two expensive algorithms called for each ant ended up being a recipe for disaster. In the end we ended up scrapping this algorithm, but we could have improved it drastically by remembering our initial direction when we started iterative deepening.

The current explore function is more complex, but is simple if one looks at it in a series of steps. We start by looking at all our exploring ants and seeing if they are threatened by an enemy ant. If so, find a neighboring square that is not threatened and can be moved to. If a neighbor square was found, move to the next exploring ant. At this point we know that either the ant isn't threatened or it can't move out the threatened zone. So we create a queue that will house Locations, and then we do a breadth first search until we either find a good battle location, an unexplored square, or we're more than 18 squares away from the ant. If that ant still can't find something to do, it sees which one of its neighbor has the most unseen squares in their radius. If all the neighbors have no squares that are unseen we do a breadth first search up to 50 squares away looking for a battle position or an unexplored square. The next explore function that comes later essentially has the algorithm just mentioned except that it goes out to a distance of 250 squares away, in a last-ditch effort to find something productive to do. If this function fails, we make sure said ant is out of the way of other ants.

8 Debugging

Frustratingly, programmers usually can't completely understand what their code is going to do, and as a result, all but the very simplest programs contain bugs. As such, a lot of time was spent debugging. We utilized a variety of methods for this purpose. We used profiling to diagnose timeout issues, and spent a lot of time digging through our debug log to find logic errors and bugs that cause the program to crash.

The profiler is a tool that repeatedly records which functions are running (we used Mac OS X's Instruments, which has a default of once per millisecond) and calculates how much CPU time each of them is using. One of the unique aspects profiling allowed us to see was that it revealed bottlenecks where we wouldn't expect them. A prime example of this `std::endl`, which is a function of `std::ostream` that inserts a new-line character and then flushes the buffer. This means that these two statements are equivalent:

```
std::cout << std::endl;
std::cout << '\n' << std::flush;
```

The difference between inserting a new-line character and `std::endl` is small but critical. The time to flush the buffer when writing a few lines is relatively costless. However, when it is possible for 600,000 lines of debug information to be written in under a few seconds, it becomes inherently slow. Flushing the buffer numerous times soon became a timeout catalyst, as the profiler found that std::flush was taking up 36% of the CPU time. The fix is simple: replace all occurrences of `std::endl` with '\n'. This solved our flushing the buffer timeout, but another problem took its place. When the buffer is flushed, it "synchronizes the buffer associated with the stream to its controlled output sequence. This effectively means that all unwritten characters in the buffer are written to its controlled output sequence as soon as possible." Thus, flushing the buffer guarantees that if the bot crashes, the last line in the debug file is also the last insertion, making it somewhat easy to find where and why our bot crashed. With only using the new-line character, we could only guess why our bot crashed, as the last line in the debug file could be from the turn before. Therefore to resolve this issue, it is key to find a balance between knowing when to flush the buffer and when to use the new-line character.

9 Possible improvements

There are known structural weak points in our code that could be improved. The two biggest offenders are State and Bot. They are large, have too many responsibilities, and the distinction between them is blurred. Enclosed by these two classes are 1731 lines of code, comprising nearly 60% of all our code. We suspect that through simply deleting functions not used anymore and extracting common code into separate functions we could cut down the lines of code down to around 1000. The problem still remains that State and Bot have too many

responsibilities. Having too many responsibilities will make a class more susceptible to change, and thus prone to errors. It would be worthwhile for us split Bot and State into more maintainable pieces of code. Having these pieces in different files would allow to work more concurrently without the fear of having problems when we tried to merge our code. The structure we put together for this competition is acceptable for the six week period that we had to compete, but if we were to work on this code any longer, a major rewrite would be in order. If we were to do a rewrite, we probably would set up a coding standard. For example, when initializing and filling an array you will find in our code `malloc`, `memset`, `new`, and `std::fill`. This intermixing of C code in C++ code can oftentimes be confusing, and so we should decide what standard to use.

Architecturally, the computer scientists Brian Foote and Joseph Yoder would describe our code as a Big Ball of Mud, which "is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle." From the beginning there was never a discussion on how our code should be structured, and we instead slapped code onto the starter bot package and figured out what this affected. This lack of discussion most likely stemmed from the fact that none of us knew what our final code should look like or what types of algorithms we would use. The result was trial and error. Someone would stumble across a new algorithm or have an idea and implement where ever they saw fit. Whether the code change did improve our bot or not, needless code was not deleted, as old algorithms were kept around or the new code changes weren't removed when they didn't work. As can be imagined, when this happened to multiple people and code was shared, no one knew exactly what functions were needed, and the result can still be seen today. For instance, scattered throughout our code are predicate classes no longer used, the Bresenham distance algorithm, A* search, and exploration implemented as an iterative deepening search. (One reason why we left these in the code is that we presented on some of these ideas to the regular ENGR151 class, and if our code was going to be shared with them, we wanted them to see how such algorithms worked). The most beneficial action would be for us to rewrite our bot from scratch, which would promote better understanding of our existing code, but more importantly we would be able to construct a structure that was more intuitive and easier to maintain.

10 Conclusion

Overall, we are extremely grateful for the opportunity we were given, as we were able to forgo regular assignments as well as tests to pursue perfection. With this time, we were able to code and debug a bot that can compete with the best in the world. Our collegiate rivals, the Spartans, are soundly beaten, as we, a team of freshmen, were able to defeat a team of their post-doctorates. To thank for this time and for giving us advice along the way is Professor Atkins. Also to thank is Jessica Horowitz, who would put in the long hours, oftentimes without sleep, to make sure that she had something new to teach us in the morning. Without them we wouldn't have been exposed to knowledge or the resources

that gave us the ability to be in the top 25 in the world. If such an opportunity arose again, we would be more than happy to answer the call.

11 References

"Ostream::flush - C Reference." Cplusplus.com - The C Resources Network. Web. 16 Dec. 2011. <http://www.cplusplus.com/reference/iostream/ostream/flush/>.

"Algorithm for Computer Control of a Digital Plotter." J. E. Bresenham. Web. <http://www.cse.iitb.ac.in/~paragc/teaching/2011/cs475/papers/bresenham_line.pdf>.

"Michael Abrash's Graphics Programming Black Book Special Edition: Dead Cats and Lightning Lines." Phatcode.net / Main. Web. 16 Dec. 2011. <http://www.phatcode.net/res/224/files/html/ch37/37-02.html\#Heading5>.

"Priority Queue." SGI - The Trusted Leader in Technical Computing: HPC, Servers, Storage, Data Center Solutions, Cloud Computing. Web. 16 Dec. 2011. <http://www.sgi.com/tech/stl/priority_queue.html>.

"Informed Search Algorithms." Stuart Russell and Peter Norvig. Web. 16 Dec. 2011. <http://aima.eecs.berkeley.edu/slides-pdf/chapter04a.pdf>.

"Uniformed Search." M. Wellman and E. Olson. Web. 16 Dec. 2011. <http://april.eecs.umich.edu/courses/eecs492_w11/wiki/images/f/f2/L3.pdf>.

"Big Ball of Mud." Brian Foote and Joseph Yoder. Web. 16 Dec. 2011. <http://www.laputan.org/mud/>.