Analysis for Assignment One: Maze Runner

Ilana Zane, William Bidle, Abinaya Sivakumar

February 16, 2020

1) Find a map size (**dim**) that is large enough to produce maps that require some work to solve, but small enough that you can run each algorithm multiple times for a range of possible p values. *How did you pick a **dim**?*

We experimented with different maze sizes and we found that the most ideal size was a dim size equal to 75. Looking ahead in the project we knew we had to run our code often (i.e. when plotting Solvability vs. Density), so if our dimension size was too large it would take too much time to produce mazes and run the algorithms. This was especially noticeable in A*-Euclidean, as for larger mazes, it would run slower due to the amount of comparisons made with long float values. The maze had to also be detailed enough to give adequate, yet varying results for each algorithm. We were able to run much bigger mazes (such as a dim of 500), however these mazes took a little too long to process later and were hard to analyze when plotted. A dimension size of 75 met each of the above requirements and was not too big that it would be tedious to show our results graphically.

2) For $p \approx 0.2$, generate a solvable map, and show the paths returned for each algorithm. *Do the results make sense? ASCII printouts are fine, but good visualizations are a bonus.*

Below are the generated paths for the 5 algorithms over the same 75 by 75 maze (see Figure 1 through Figure 5). Each of these results match up exactly with what we expected each algorithm to return. Since we are dealing with a finite length puzzle and the position of the goal node is known, each of the algorithms besides DFS will return the shortest path length. Even the algorithms without a heuristic (BFS and Bi-Directional BFS) will return the shortest length, as it will be the first solution they find. For example, if we watch the progression of the BFS algorithm (see BFS.mov), we can see that BFS will expand almost all of the nodes in the maze before finding the goal node and will definitely find the shortest path.

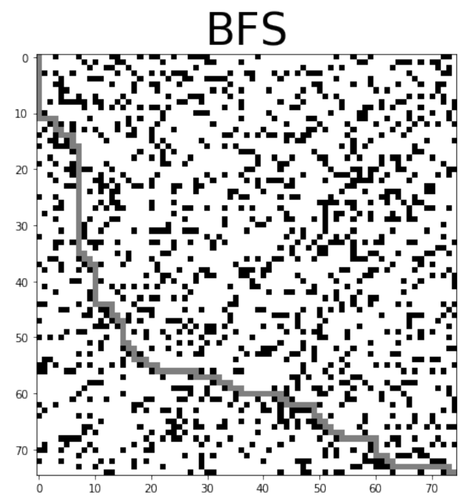

Figure 1

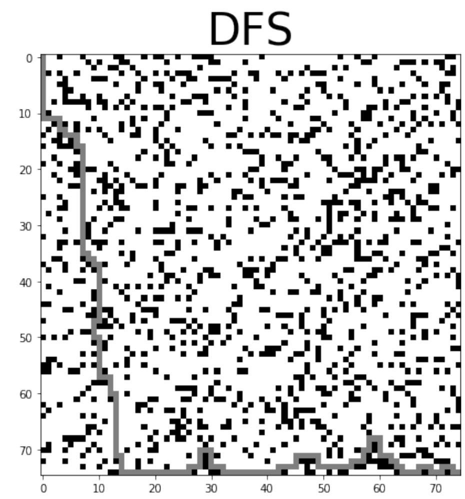

Figure 2

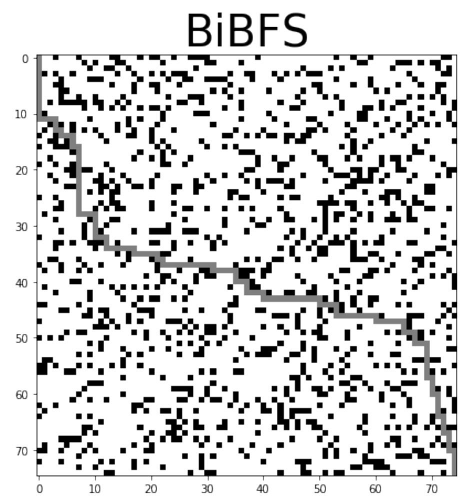

Figure 3

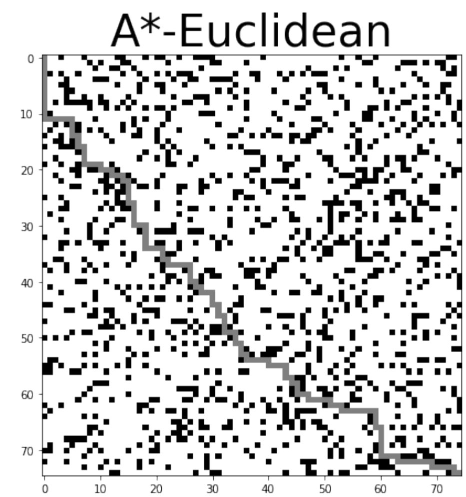

Figure 4

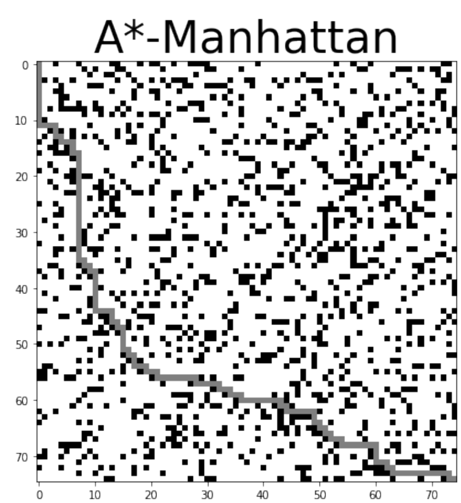

Figure 5

3) Given **dim**, how does maze-solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple mazes and checking them for solvability. What is the best algorithm to use here? Plot *density vs solvability*, and try to identify as accurately as you can the threshold p_0 where for $p < p_0$, most mazes are solvable, but $p > p_0$, most mazes are not solvable.

As seen in Figure 6, for a fixed dimension size of 75, each algorithm tends to behave similarly as we increase the probability of a cell being occupied. This is fairly expected, as we are dealing with a finite dimension size, and therefore will have all of our algorithms find a solution to the goal if one is available. On average, if one cannot find a solution, then the others will not be able to as well, as seen when p is approximately 0.4 (the graph is cut off because everything beyond 0.4 is zero). The data in Figure 6 was generated by running each algorithm over 75 mazes for probabilities in the range [0, 1], averaging the values together, and making a cut at p = 0.4. We can see from the data that mazes become mostly solvable at and below a probability of approximately 0.3, and are mostly unsolvable for any p greater than 0.3.

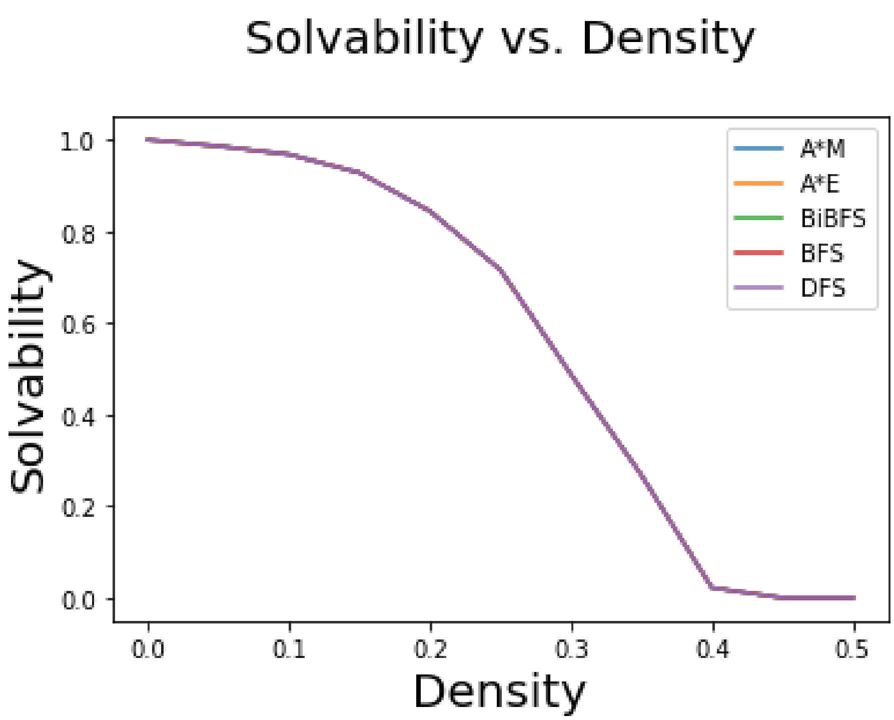

Figure 6

4) For p in $[0, p_0]$ as above, estimate the average or expected length of the shortest path from start to goal. You may discard unsolvable maps. Plot *density vs expected shortest path length*. What algorithm is most useful here?

Based on Figure 7, we see that all of the algorithms have the same outcome, except for DFS. At around a density of 0.4, the path length rapidly decreases towards 0 because from the previous graph (Figure 6) we see that at the same density leads to unsolvable mazes, and an unsolvable maze will have a path length of 0. Based solely off of shortest path length returned, it would be most useful to use any algorithm besides DFS.

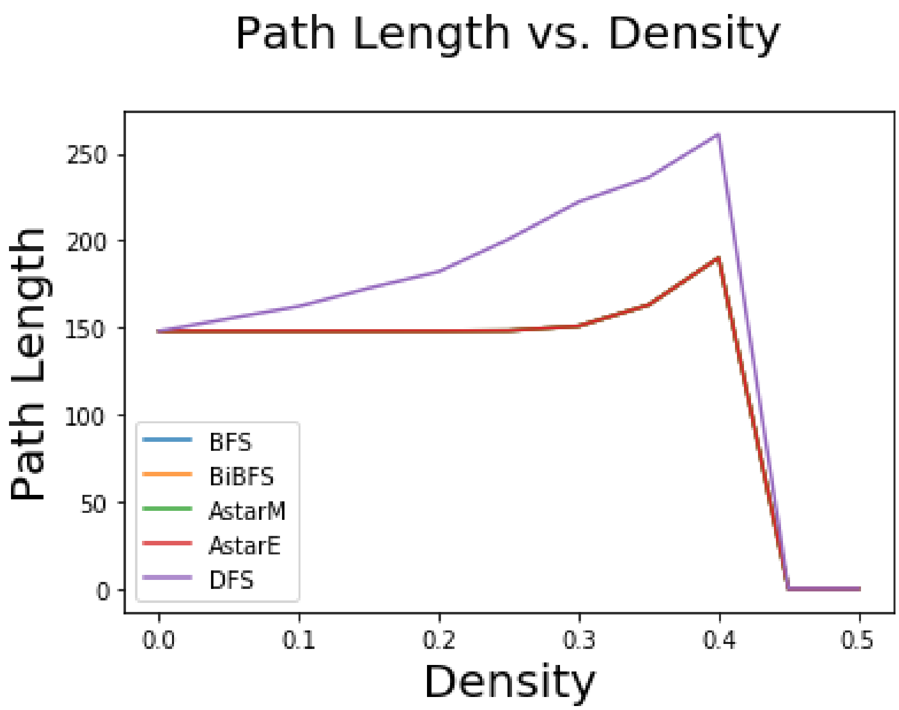

Figure 7

5) Is one heuristic uniformly better than the other for running A*? How can they be compared? Plot the relevant data and justify your conclusions.

We found that the best heuristic for running A* is Manhattan. Since both Manhattan and Euclidean return the shortest path length for a given maze, the best way to compare the two would be to look at their run time for different dimension sizes. As seen in Figure 8, both heuristics have approximately the same performance time. However, as the dimension size continues to grow, Euclidean becomes less time efficient compared to Manhattan. We know that both heuristics are admissible as it never overestimates the true remaining cost from a given node to the goal. The difference between the two is that the Manhattan heuristic is closer to the true cost, which results in us expanding less nodes than with the Euclidean heuristic.

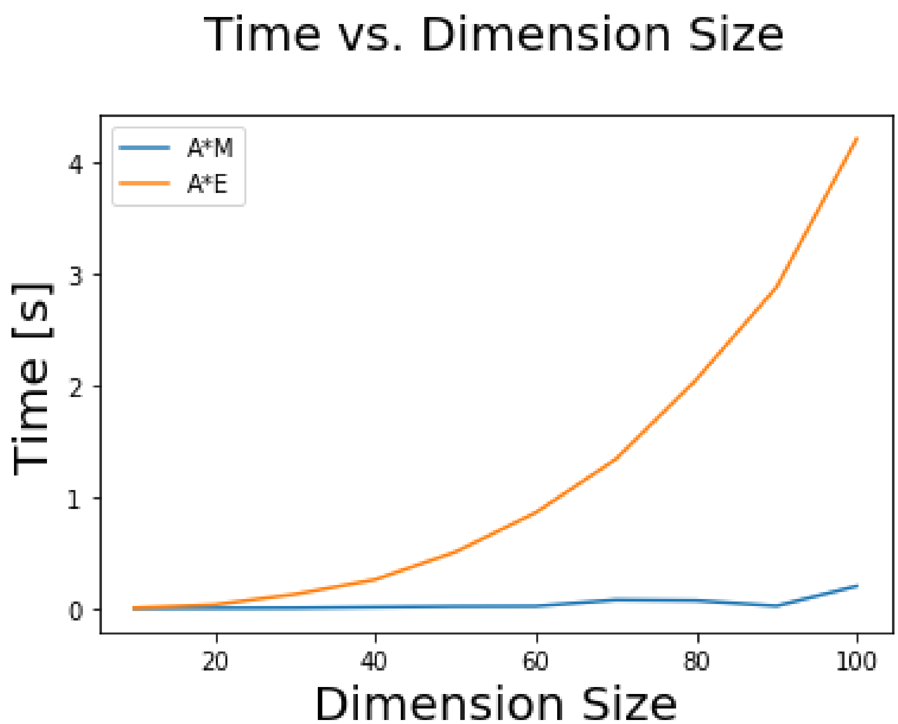

Figure 8

6) Do these algorithms behave as they should?

We expected DFS to be inefficient because it tries to explore only one branch of the fringe instead of expanding its possibilities outwards. DFS ends up missing the most efficient path unlike the other algorithms, which do return the most efficient path in every run as seen in Figure 7. BFS and Bi-Directional BFS will expand most of the maze as will the two heuristics.

7) For DFS, can you improve the performance of the algorithm by choosing what order to load the neighboring rooms into the fringe? What neighbors are 'worth' looking at before others? Be thorough and justify yourself.

From the beginning, we realized that DFS would be different from the algorithms in the sense that the order in which moves were loaded into the fringe would matter greatly. Since DFS wants to pick a branch and keep pushing all the way through until it gets stuck, it would make the most sense to prioritize pushing it down branches that advance us towards the goal node with the least amount of moves. We decided to prioritize moving downwards and then to the right. DFS only checked possible upwards and left if it wasn't able to move to the left or downwards.

8) On the same map, are there ever nodes that BD-BFS expands that A* doesn't? Why or why not? Give an example, and justify.

Yes, the algorithms are meant to explore for the shortest path in different ways. When comparing Bi-Directional BFS with A*-Manhattan on a grid with no obstacles, Bi-Directional

BFS explores almost the entire grid before returning the shortest path, while A*-Manhattan strictly explored the left side of the graph and the bottom before returning the shortest path. Bi-Directional BFS will try to explore the entire grid because starting from the upper left corner and the bottom right corner, the two paths will explore every child added to the fringe before the two paths meet in the middle. A*-Manhattan will prioritize moves that can be made downwards and to the right, which is why the nodes that are explored are mostly limited to those two directions and the total nodes explored are much less than that of Bi-Directional BFS (see BiBFS.mov and AstarM.mov).

9) What local search algorithm did you pick, and why? How are you representing the maze/environment to be able to utilize this search algorithm? What design choices did you have to make to make to apply this search algorithm to this problem?

We picked the hill climbing search algorithm because it is the one that we best understood and it will definitely find a local maximum. In order to apply the hill climbing algorithm, we created the same maze that was previously generated for all of the previous parts and a copy of this same maze was generated. First, one maze was run through, DFS for example (see Figures 9 and 10). We calculated the maximal fringe size for the original, randomly generated maze and saved that value. Then, we took a copy of that very maze, but randomly added an obstacle. We ran DFS through this edited maze and calculated the maximal fringe size. If the maximal fringe size for the edited maze was larger than the original maze, we determined that this was a harder maze to solve. Therefore, through a recursive function, this edited, harder maze was saved as the new original maze and we continued the process of comparing original mazes and ones with an added obstacle. If the edited maze was not more difficult than the original, then we repeated the aforementioned process with the same maze, not the edited one. We continued this process until we found ten mazes that were consistently not more difficult. The same process was used for A*-Manhattan except instead of analyzing the maximal fringe size, we analyzed the maximum number of nodes explored (see Figures 11 and 12)

10) Unlike the problem of solving the maze, for which the 'goal' is well-defined, it is difficult to know if you have constructed the 'hardest' maze. What kind of termination conditions can you apply here to generate hard if not the hardest maze? What kind of shortcomings or advantages do you anticipate from your approach?

We choose the terminating condition as 10 consecutive failures to produce a harder maze. We predict that this number will lead us to the local maximum that we are looking for. We think that for any state space of dim by dim, the number 10 would be significant enough to indicate that the mazes would not be increasing in difficulty.

11) Do your results agree with your intuition?

The results agree with our intuition. When looking for the shortest path, DFS will typically display a path that travels mostly around the left side and bottom of the grid as a result of the way it expands its children on the fringe— exploring an entire path before backtracking. When executing our hill climbing algorithm with DFS we see that the path that DFS is taking isn't as direct, traveling throughout the grid instead of taking its usual path. This is because in order for our hill climbing algorithm to progress, the fringe size of DFS needs to become increasingly larger. As this happens, more children in the fringe need to be expanded causing DFS to explore other routes around obstacles in order to reach the goal cell. The hill climbing algorithm followed the same logic for A*-Manhattan. In order to make the mazes harder, more obstacles had to be added. As more obstacles were added, A*-Manhattan began to not travel along the most optimum path, which meant that it was expanding more nodes. Whenever the newest state of A*-Manhattan had explored more nodes, the hill climbing algorithm continued until our aforementioned terminating condition was met.

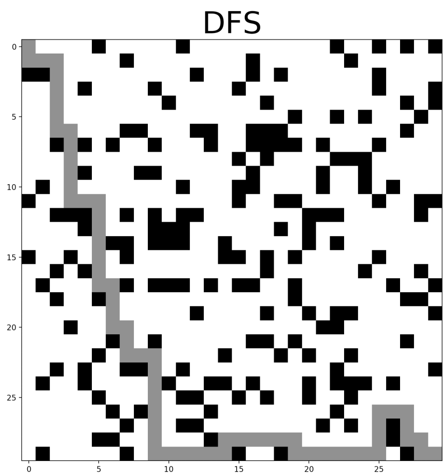

Figure 9

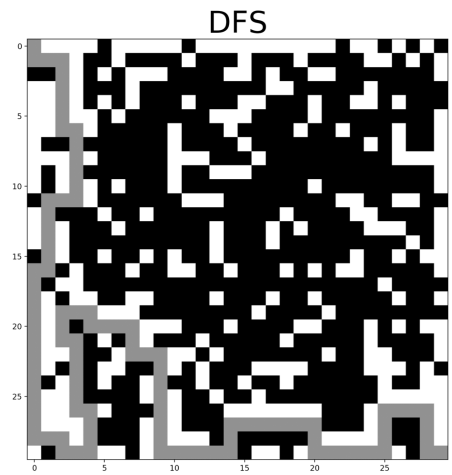

Figure 10

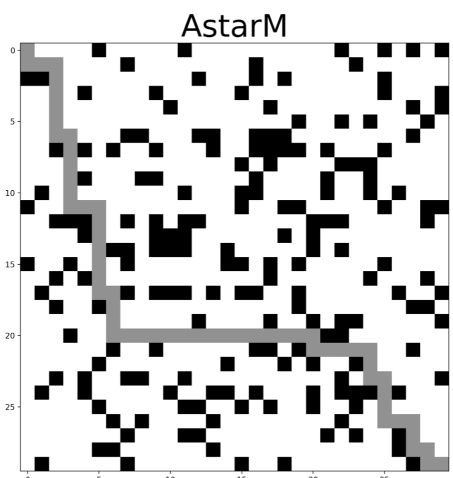

Figure 11

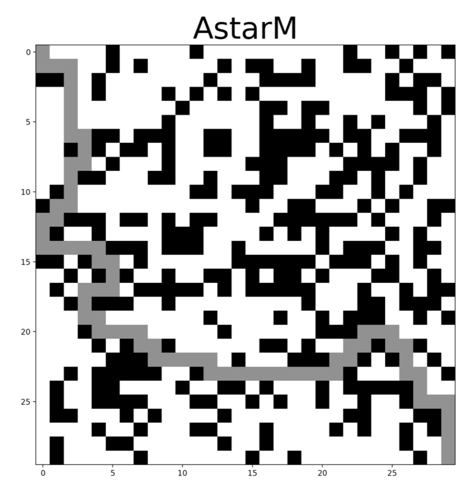

Figure 12

12) Generate a number of mazes at the dimension **dim** and density p_0 as in Section 2. Be sure to generate a new maze and a new starting location for the fire each time. *Please discard any maze where there is no path from the initial position of the agent to the initial position of the fire - for these mazes, the fire will never catch the agent and the agent is not in any danger.* For each strategy, plot a graph of 'average successes vs flammability q'. Note, for each test value of q, you will need to generate multiple mazes to collect data. Does re-computing your path like this have any benefit, ultimately?

This absolutely helps, as seen in Figure 13. Even though there is not a significant improvement between strategies two and three, both are still much better than strategy one, where the path doesn't account for the fire at all.

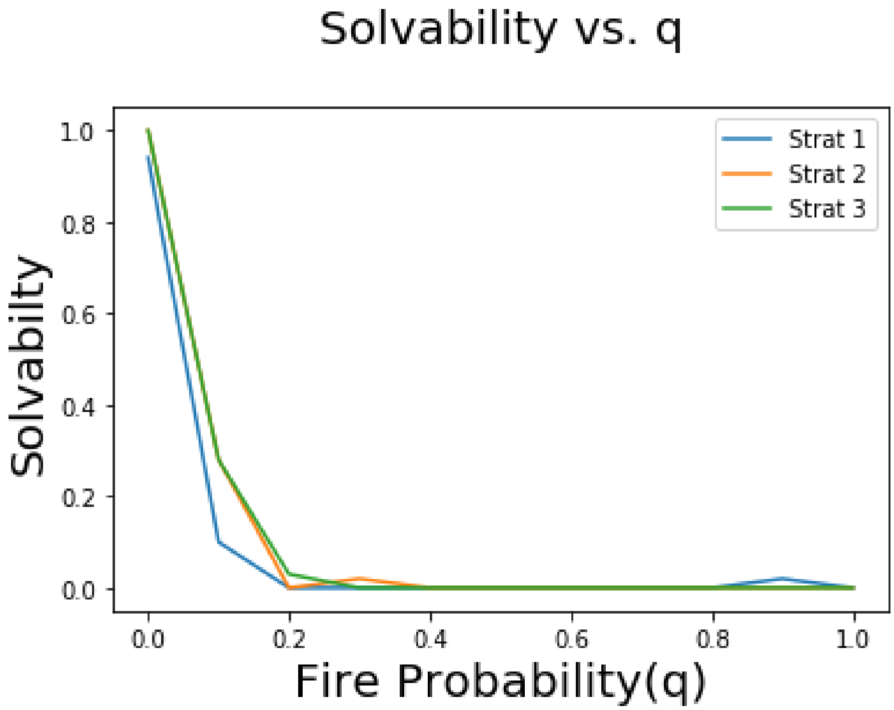

Figure 13: Strat 2 and 3 are better than Strat 1. Strat 2 and 3 perform similarly until 0.2 and then Strat 3 performs the best overall.

13) Come up with your own strategy to solve this problem, and try to beat both the above strategies. How can you formulate the problem in an approachable way? How can you apply the algorithms discussed? Note, Strategy 1 does not account for the changing state of the fire, but Strategy 2 does. But Strategy 2 does not account for how the fire is going to look in the future. How could you include that?

To improve strategy one, we included a check to make sure that the maze does not move into a box that is on fire and we calculated the shortest path to the end each time using the Manhattan hueristic. Strategy three predicts where the fire will be using the probability formula given in the assignment. We added this probability as a part of the heuristic to decide which square we should travel to. Ideally, one would want to choose a spot that has 0 flammability. The Manhattan heuristic gives us the distance and then p is added to that to create a fire heuristic. For example, consider if the following two moves can be made: down and right. The Manhattan heuristic for these two moves is the same but the probability of the right block catching on fire is 0.866% and the probability of the down block catching on fire is 0.333%. The heuristic allows the program to make the decision to go down instead. The fastest path still has most of the heuristic weight, but the added flexibility incentive that helps predict future fires allows for the program to have a higher general success rate.

☺Member Contributions:☺

Part 1: William
Part 2: William, Ilana, Abinaya
Part 3: Ilana
Part 4: Abinaya