Part 3: Hard Maze

|  |  |
| --- | --- |
| A black and silver text on a tiled floor  Description automatically generatedRandomly Generated Maze | A close up of text on a white surface  Description automatically generatedHard Generated Maze |

We chose to implement a hill climbing local search algorithm. The reasoning behind our decision was that there are many different combinations of mazes with different levels of “difficulty”. For instance, assume that the X axis is the combinations of all the mazes and the Y – axis is the level of difficulty you can imagine hills and valleys being formed across the graph. Look at figure 1.

![A close up of a device

Description automatically generated]()

Figure 1.

Since there are so many different combinations had we picked a random maze to start with and changed it block by block and tried to make every single combination it would have taken us forever and it might be impossible to compute. So, the hill climbing algorithm gives us the ability to find local maximums for every time we run that algorithm some *nth* amount of iterations. Then we compare all the local maximums and pick the most “difficult” one and even though it might not be the most “difficult” out of all the possible combinations of mazes it is still a very difficult one that is close to the optimal.

We are representing the maze/environment as a 2D array. Every state is coordinate of (x, y) on that 2D array. We represent the source state as (0,0) and goal is the (dim-1, dim-1). Since we can move around the maze either by going left, right, up, or down. We can easily represent the different states by adding 1, 0, -1, to the appropriate X coordinate or Y coordinate to move horizontally or vertically. So, our hill-climbing algorithm generates a random 2d array with coordinates and some probability to add blocks. Once we have the maze, we clone it and change it by either adding one block or taking away a block randomly in each iteration. Then we compare the mazes using a compare function that keeps the hardest one. Then we repeat this process *n* amount of iterations. Then we store the “hardest” maze of that hill climb into a stack. Then we repeat the process with a new random generated maze which is essentially the start of a new hill climb. Finally, we compare all the hardest maze of each hill climb to find the hardest maze of them all.

Since its difficult to know when we have generated the hardest maze there are many conditions that you can apply to terminate the hill climbing algorithm. We decided to set the amount of max iterations per hill climb and set the amount of max hill climbs we would run. We could have chosen to make the simulated annealing algorithm, but we decided that the performance of a normal hill-climbing algorithm was worth more than the risk of not finding the most optimal solution that annealing algorithm might have found.

Our result for the DFS and A\* search on our hard mazes went against out intuition because we found that the “hardest” maze for a DFS with a maximal fringe size and Astar Manhattan with Maximal nodes expanded was basically a simple maze with low number of blocks. So, we learned that what is “harder” maze for an algorithm is not necessarily “harder” maze for a human mind.

Part 4: Fire Maze

A close up of text on a white background

Description automatically generated

Yellow block: where the fire started

Orange block: is the fire

Blue circle: is the agent

Black blocks: unwalkable spaces

White blocks: open spaces

**If you click right on the UI it will choose the next step in real time**

We formulated the problem by considering two main priorities. Don’t go into a fire of course. But also check the probability of a cell neighboring a fire will catch on fire next. In order to let the computer understand the states, we had to make a new “maze”, 2d array, called “fireMaze” which kept track of the blocks on fire, the unwalkable blocks, and the k values of each cell. We could now formulate the question by describing states as coordinates in the 2Darray, like[x][y], and then consider the new path had we chose to step on one of our neighbors next, such as[x-1][y], [x+1][y],[x][y-1], or[x][y+1]. After we check the 4 different paths, some or all of which can be unreachable paths with no chance to reach end goal, we choose the path with the lowest distance. If the paths had the same distance, then we would consider the sum of the k values of the whole path and choose the path with the least sum of k values.

We thought that by making a clone of the maze, 2d array, we were going to be able to keep track of the neighboring cells of a fire and keep track of their appropriate k values so we can determine where the fire might go next and choose our path accordingly. Then we could compare the paths from our 4 neighboring cells and choose the one that got us the closest to the goal. If they had the same distance, we would choose the least risky one, the one with less k values. This way we can quickly move towards the goal prioritizing time steps over the risk of catching on fire. We thought that we could survive more times without burning or having the fire engulf the end goal if we prioritize path distance over the risk of catching on fire.