CS440 - Spring 2021 - Project 1

Gal Zandani and Vaishnavi Manthena

February 19, 2021

Problem 1

The code for the algorithm for generating a maze with a given dimension and obstacle density p, the definition function create_maze, generates a valid 2-dimensional array such that the obstacle density p is randomized in each index of this array created. A valid 2D array which can be used as a maze has a dimension strictly larger than 1. The start and goal indexes of the 2D array are labeled S and G, which are S-(0,0) and G-(dim-1,dim-1), and the rest of the elements of the array are randomly decided whether the should be empty, 'e', or filled, 'f'. The filled elements in the array would be chosen randomly and depends on the obstacle density p given to us when creating a maze and is assumed to be a value which is $0 \leq p \leq 1$. If the value of p is smaller, less places in the maze would be blocked/filled and more places would be empty, and if the value of p larger, more places in the maze would be blocked/filled and less places would be empty. For each place in the 2D array, which is not the start or the goal, the algorithm chooses a random number in the interval of $[0, 1]$ so that if the number is smaller or equal to the value of the obstacle density p given, this space would be filled. After going through the entire 2D array in this way, the final maze is returned. When creating a fire maze, the places of the maze which are on fire, would be marked as 'x'.

General Note About This Search

In this first part of the project, we are concerned with finding a path from a start state to a destination state. The problem of searching through this maze can be formulated as below:

State space: The set of empty blocks, the start block, and the goal block in the 2D square array.

Action Space: To go from one state to another we can take one step to the left, or one step to the right, or one step upward, or one step downward.

Initial state: In this problem, usually the initial state is the upper left state: '(0,0)'. However, when we are interested in finding a path between two other states, then the start state in that situation will be the initial state.

Restricted states: States that are filled or out of the bounds of the square array.

Goal state: In this problem, usually the goal state is the lower right state: '(dim-1,dim-1)'. However, when we are interested in finding a path between two other states, then the end state in that situation will be the goal state.

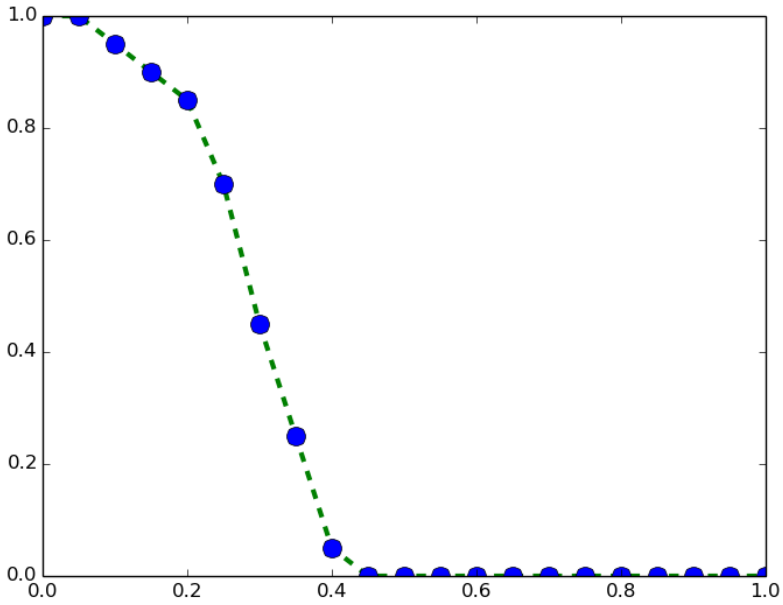
Problem 2

The code for the DFS algorithm that takes a maze and two locations and determines whether one is reachable from the other, the definition function isPath, uses a stack to store the fringe. It takes as an input the maze and two points on that maze formatted like '(x,y)' where both x and y are values from 0 to (the dimension of the maze - 1). We assume that both of the points given are valid points in the given maze. The algorithm follows a certain path determined by the principle of DFS, which is storing the fringe used in the search as a stack, and determines whether there exists a path from one location in the maze to another. The stack is implemented using a python list. Using a stack ensures that at any point we explore the newest state on the fringe. We also maintained a closed set. This is because it is possible to revisit a previously explored state and we want minimize redundant explorations by using a graph search algorithm as compared to a tree search algorithm.

DFS is a better choice than BFS here because there is no need to find the shortest path between these two points, but to find if it is possible to reach one from the other. Moreover, BFS has a high space complexity since it traverses through the graph search tree level by level. The space complexity of BFS is about O(b^k) whereas that of DFS is O(bk), which is much better. Here b is the branching factor and k is depth of the goal. Since optimality of BFS is not required we can go with DFS due to its better spacial complexity.

The exact code to generate the below graph is specified in the function 'part2Graph'.

Figure 1: Obstacle Density p vs Probability that S can be Reached from G



Index: X-axis: obstacle density p, $0.0 \leq p \leq 1.0$. Y-axis: Probability that S can be Reached from G in DFS.

Problem 3

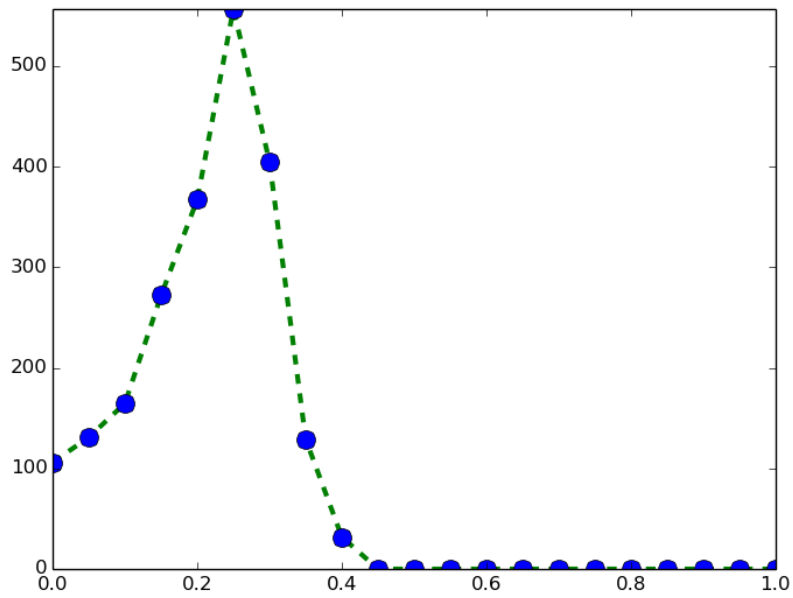
The code for the BFS algorithm that takes a maze and two locations (which can be defined as the S and G) and determines the shortest path between them if one exists, the definition function shortestBFS, uses a queue to store the fringe. This would ensure that the oldest value is explored first and that we traverse the graph level by level. It takes as an input the maze and two points on that maze formatted like '(x,y)' where both x and y are values from 0 to (the dimension of the maze - 1). We assume that both of the points given are valid points in the given maze. If we want these points to be S and G, then S = (0,0) and G = (dim-1, dim-1). If a path exists, it returns True, the path between the two points, and the number of nodes explored.

The code for the A* algorithm that takes a maze and two locations (which can be defined as the S and G) and determines the shortest path between them if one exists, the definition function shortestA, uses a priority queue to store the fringe and computes the heuristic for each index of the 2D array maze to the goal index. It takes as an input the maze and two points on that maze formatted like '(x,y)' where both x and y are values from 0 to (the dimension of the maze - 1). We assume that both of the points given are valid points in the given maze. If we want these points to be S and G, then S = (0,0) and G = (dim-1, dim-1). If a path exists, it returns True, the path between the two points, and the number of nodes explored.

If there is no path from S to G, both of the algorithms BFS and A* would try to explore all possible paths to get from S to G which would result in exploring a similar number of node. The difference should be 0 is those cases. We can observe that happening in the graph for p values of 0.45 or larger, the difference there is 0 since the obstacle density increases which lowers the probability of having a reachable path from S to G.

The below graph has been obtained from the 'part3Graph' function in the code.

Figure 2: Obstacle Density p vs The Average Number of Nodes Explored by BFS - Number of Nodes Explored by A*



Index: X-axis: obstacle density p, $0.0 \leq p \leq 1.0$. Y-axis: The Average Number of Nodes Explored by BFS - Number of Nodes Explored by A*.

Problem 4

We created a function, maxDims, to compute those results of the dimensions by running those algorithm on mazes in increasing sizes with p=0.3. By running that function several times, we took the maximum dimension resulted for each algorithm to conclude that.

run number	DFS	BFS	A*
1	255	248	244
2	254	242	245
3	255	243	242
4	252	243	242
5	249	242	243

The largest dimension we can solve using DFS at p = 0.3 in less than a minute is 255.

The largest dimension we can solve using BFS at p = 0.3 in less than a minute is 248.

The largest dimension we can solve using A* at p = 0.3 in less than a minute is 245.

By this relationship between these three different search algorithms, we can see that since DFS reaches to the largest dimension in less than a minute, it takes the least time to compute. And since A* reaches the smallest dimension within a minute, we can observe that it takes the most time to compute. So, these algorithms have different advantages which helps us use the all effectively for different scenarios of mazes or situations.

Problem 5

While constructing our improved strategy 3, it was important for us to keep in mind the advantages and challenges of each of the searching algorithms implemented in this project, DFS, BFS, and A* as well as the advantage of considering the representation of the predictions of the future in this simulation learned in class. The strategy 3 is implemented in the function 'strategy3' which used three helper functions. These are right before strategy3 function in the code.
We gave our strategy 3 a creative name: Heat Gradient.
Our improved strategy 3 is using a modified version of A star, where extra weights are added to positions next to the fire based on the distance from the fire. So, we basically used an extra heuristic along with the euclidean distance. The extra heuristic is constant for any position in a particular maze configuration. So, we used the method in our code called 'fireheuristics()' to initially compute the extra heuristic for all positions. Value of this extra heuristic are described below:
If the position was filled or on fire then 0.
If q value is 0 then the heuristic was also 0, since there is no way any empty position will catch fire.
If the position was right next to a fire position, then:

1) A weight of 3*2*dim was added if q = 1

2) For all other q values we added a weight of k*2*dim, where k is the number of neighbors which are on fire.

If the position had a shortest distance of 'm' between itself and the fire (this was determined through the function 'mod_BFS'):
Then the heuristic was dim/m.

In this way we added weights to nodes based on their distance from the fire (with increasing distance we added less weight).
Also, we took into consideration the number of neighbors on fire for those nodes which were adjacent to the fire. In this way, the weight increases with increasing number of adjacent fire nodes.
So, strategy 3 is like strategy 2 except instead of recomputing each time with A star we recompute with the modified version.
Although this may cause the overall heuristic to not be admissible, this might be okay in this case. This is because here we are not concerned about an optimal path. A heuristic which is not admissible may in fact even be helpful since it scares the A star algorithm away from areas near the fire and this

promotes survivability.

It accounts for the unknown future since we adjust the weights of the neighbor nodes of the fire nodes so we take into account the behavior of the fire (the number of neighbors that are on fire, the value of q, dimension factor, the distance from a certain node of the fire). We also adjust weights for neighbors of neighbors of fire and so on in a level by level fashion using the principles of BFS. So, it makes the agent to stay away from the fire or other nodes which can be on fire in future states of the maze as much as possible.

Since we decrease the cost of positions with distance from fire, we called our algorithm Heat Gradient.

Run time Analysis

Time complexity of strategy 1:

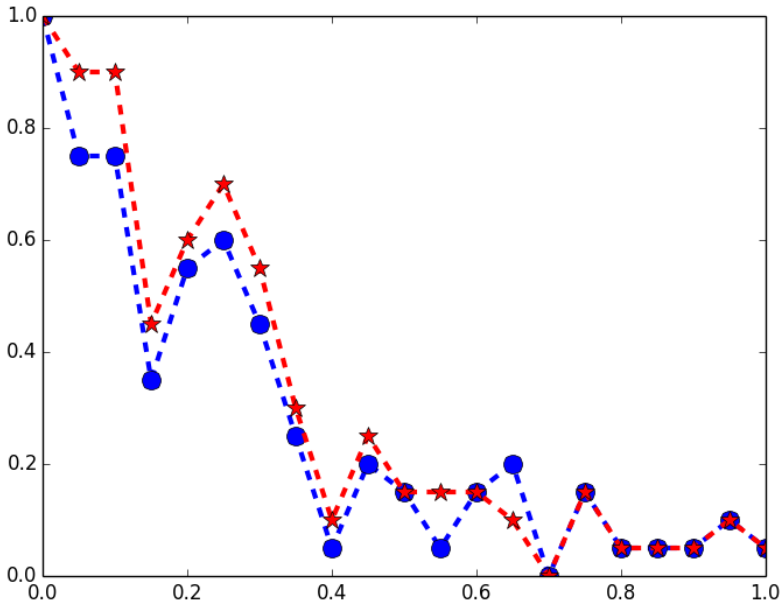
Initially using A star to calculate the shortest path: $O(n+n\log n)$. Here n is equal to dim^2. The linear time n is used for n removals from the fringe, and also to generate all heuristics. Each insert takes $logn$. There are about 'n' inserts. This takes $O(n)$. Then there is a linear traversal along the generated path. Lets say the length of the path is 'k'. For each step, we advance the fire as well, which is linear time. So, this part takes $O(kn)$.

Total complexity: $O(kn + n)$, where k is length of path generated by A star. And n is dim^2.

Time complexity of strategy 2:

Starting out to get the path of a maze and then after each move of the agent we had to advance the fire and recompute the path after every step the agent did. The running time of computing the path using A* algorithm takes $O(n + n\log n)$. The running time of advancing the fire of a maze is $dim^2 = O(n)$. It is possible for the path to be recomputed, in the worse case which is if the fire is frozen and there are not many obstacles in the maze, the number of places in the maze which is $dim * dim = dim^2 = O(n)$. It could take less than that since there are obstacles and fire spreading in most generated mazes. Then, multiplying those running times since after every step we generate a possibly new path to reach the goal in the maze, the running time of strategy 2 is $(O(n + n\log n) + O(n)) * O(n) = O(n^2)$.

Figure 3: Average Strategy Success Rate vs Flammability q at p = 0.3



Index: Blue- strategy 1 and red- strategy 2. X-axis: flammability q, $0.0 \leq q \leq 1.0$. Y-axis: Average Strategy Success Rate.

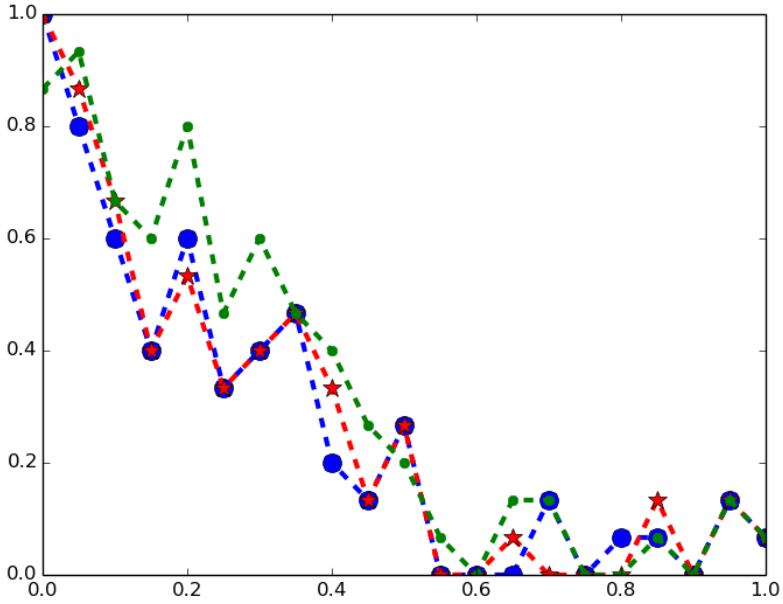
Time complexity of strategy 3:

The only extra amount of work in strategy 3 would be to generate the heuristics of each position in the maze based on the fire, each time we re-compute the path.

Extra time cost in strategy 3:
the number of times we recompute the path times the time complexity of creating heuristic table for fire.
Worst case number of nodes for which heuristic needs to be calculated: dim^2. For each heuristic the main complexity occurs from BFS to see the shortest distance between the point and the fire. The worst case runtime of BFS will occur if we need to visit every node in the search graph. This would be $O(b^{m+1})$. Here b is the branching factor (in this case at most 4, since each state has at most 4 valid neighbors). 'm' is the maximum depth of the tree, which in this case should be lower than $2 * dim$.

Problem 6

Figure 4: Average Strategy Success Rate vs Flammability q at p = 0.3



Index: Blue- strategy 1, red- strategy 2, and green- strategy 3. X-axis: flammability q, $0.0 \leq q \leq 1.0$. Y-axis: Average Strategy Success Rate.

The different strategies perform the same in $q = 0.3$ and $0.9 \leq q \leq 1.0$. Where the fire in the maze does not spread, the agent succeeds to get from the start to the goal of the maze in these three strategies. And when the fire does spread at a high flammability level q, it gets harder for the agent to succeed to get from the start of the maze to the goal in which all the strategies perform the same.

The different strategies perform differently in $0.0 \leq q \leq 0.25$ and $0.35 \leq q \leq 0.85$. Between 0.35 and 0.85, we can observe that strategies 1 and 2 perform differently at some points on the graph, such that the average success of strategy 2 is bigger than the average success of strategy 1 in most of the points plotted. Then, in most values of q, strategy 3 is more successful that strategy 2 and 1. This is because strategy 1 computes a path only initially and does not adjust to the spreading fire. Strategy 2 performs a bit better than strategy 1 in certain areas due to recomputing and adjusting to the spreading fire. Strategy 3 tends to perform better than the other two throughout this region as it tries to push the agent into a safer path away from the fire at each step.

Initially between 0.0 and 0.05 strategy 3 performs weaker, potentially due to the fact that it may lead the agent into longer paths to the goal by trying to stay away from the fire although the fire has a low flammability rate. This might be inefficient when the fire is not spreading rapidly. Strategy 3, by considering potential future states of the maze, is mainly a prediction of the future states of the mazes, regrading the way the fire spreads across it. Those kind of predictions might be proven to be wrong in future states of the maze.

Problem 7

If we had unlimited computational resources at our disposal, we could improve strategy 3 by thinking about all the possibilities of scenarios which can occur within the maze. We can think of the maze with a start, S, and a goal, G, empty spaces, e, and filled spaces, f, (which is controlled by obstacle density p) as a two-player game with an agent, a, which needs to get from the S to G and survive and fire, x, which spreads across the maze (controlled by flammability q).

In that way, as we are not limited in the computational resources at our disposal, we can build a game tree for that game. Every level of the tree will represent a turn of either the agent or the fire alternatively with the agent starting. Then, as we construct all the scenarios of the maze we are given, we would probe all the way to the terminal nodes of the tree so we have some resulting cases when the agent either get to the goal or does not make it. These states would be marked with the decision points of the game, if the agent got to the goal, 1, if the agent did not, 0. So that we can figure out the decision that should be taken at any turn of the game. During the agent's turn, the agent will always choose the next configuration which has maximum utility. During the fire's turn, it will choose the next configuration based on chance. The agent can take have at most 4 next possible configurations (maximum branching factor of agent = 4). The branching factor for a fire can be exponential. The picture below is just a sample. The terminal nodes are all not necessarily in the kth level. They can be found in previous levels as well. In the figure below a node which has 'Agent' next to it means that it is the agent's turn to choose next step. Same for this Fire. Since we are not limited in the amount of memory here, we can use either DFS or BFS to find the ideal path the game should go because the fringe is not limited to a certain amount of nodes.

Mathematical explanations:

Formula for branching factor at a fire node:

Let n be the number of nodes which have neighbors that are on fire. Then, if q is between 0 and 1, then:

branching factor = 2^n

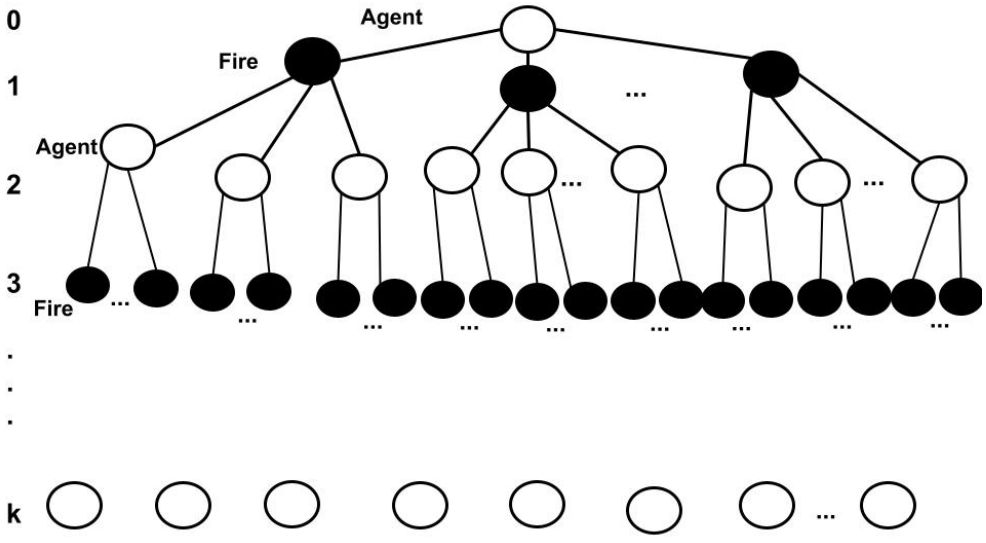
For big dimensions this is only practical with unlimited computational resources.

Formula for utility at a fire node (state say 'f'):

Say the fire has children states : c1, c2, ..., cm, with utilities U(c1), U(c2), ... U(cm)

utility at fire node = $P(c1|f)*U(c1)+P(c2|f)*U(c2)+...+P(cm|f)*U(cm)$

The probability of going to state 'c' given a state 'f', $P(c|f)$, can be computed using the q value. Say the number of nodes adjacent to nodes on fire are 'n' in state 'f'. Say in state 'c', of these n nodes a set of nodes 'A' are on fire and a set of nodes 'B' are not on fire. The probability of this occurring is the probability that all nodes in A catch on fire multiplied by the probability that all nodes in B do not catch on fire. The probability that any node catches on fire depends on q and number of neighbors on fire as per the project description.



Problem 8

If we could only take ten seconds between moves (rather than doing as much computation as we like), that would change our strategy since pruning a whole game tree, as explained in problem 7, would take a lot more than ten seconds.

A potential strategy 4 is using the game tree we generated in the previous problem and looking at only a certain number of levels, not all k of them. So, we can have a game tree for a maze and generate only the first 3 levels, so that we have an understanding of possible moves for 2 maze states ahead. This would take less time than probing the entire game tree and it can return probabilities of the agent succeeding to get to the goal in the options resulted. This could give a good indication of the path the agent should go or the ones it should avoid. We could use tricks like alpha beta pruning to further reduce the number of nodes we need to explore. Moreover, a good heuristic should also be used to calculate the good estimates for the utilities of the level 3 nodes. We could then percolate these utilities upwards. The utilities could be determined by factors like closeness to the fire, closeness to the goal, existence of a path to the goal, etc...

Another potential strategy 4 is a variation of local search. Since, there isn't enough time, it may be beneficial to just look at neighboring positions and choose the position with the best value of a function. Say, we prefer to minimize the value of the function at each step. Then the value of the function should decrease with decreasing distance from the goal, and should decrease with increasing shortest distance from the fire. So, the function could be something along the lines:
$F = constantofproportionality * (distancefromgoal/distancefromfire)$
Notice that F becomes 0 (minimized to the maximum extent) at the goal.

Summary

We both worked on the logic and code for the problems in this project, we brain stormed ideas, and agreed on the solutions we got it. We met on Zoom to write together the code for this project to generate the mazes, different search algorithm, and strategies. We collaboratively found solutions to issues we faced while testing and running the code. We both were running test cases on the code outside of our Zoom meetings (generating graphs, running different mazes as input, making sure the strategies make sense). Mostly it was completely collaborative but the below are some individual things we did:

Gal: Getting the latex report ready, and filling in the first 4 questions. Generating the graphs and debugging if there are any issues in generating the graphs. Added extra points to Vaishnavi's answers in the next 4 questions. Running test cases. Editing comments on the code.

Vaishnavi: Getting python project ready and doing two or three basic helper functions and DFS to get ourselves familiar with python. Added in to Gal's answers in the first 4 questions and typed in points for the next 4 questions. Running test cases and debugging where necessary. Writing comments on the code.

We both wrote some methods of the code, debugged separately, and contributed to the answers on this report. This write up was written by both Gal and Vaishnavi, each one of us was responsible for some questions and we added on to each other's answers.

I read and abided by the rules laid out, I have not used anyone else's work for our project, my work is only my own and my partner's.

Gal

Vaishnavi