Assignment 1

Farhat Alam, Karthik Paka, Matan Shachnai, Andrea Wu
198:520 Intro to Artificial Intelligence
Rutgers University

February 10, 2019

1) Code is attached for question 1.

2)

2.1) In order to choose a map size that allows us to experiment and reach valid theoretical conclusions, we had a few criteria that needed to be met:

1) The map size had to be small enough so that we could visually see the maze on our screens.

2) The runtime of each algorithm over 100 iterations would remain around 1 second so that we could test a wide range of P values.

3) The time it would take a human to solve the maze would be roughly between 10-20 seconds.

Based on these criteria, we found **dim** = 30 sufficient to satisfy our needs. All tests were done with this dimension unless noted otherwise.

2.2) Our implementation of BFS adds unblocked/unvisited cells to a queue, checking that a given cell's neighbors are unblocked/unvisited in the following order: right, down, up, left. Cells in the queue are checked in the order that they are added until the goal cell is reached. Given the enqueue/push ordering of each different algorithm, we can see (in the visual below) that these algorithms work as intended.

Our implementation of DFS adds unblocked/unvisited cells to a stack, checking that a given cell's neighbors are unblocked/unvisited in the following order: up, left, right, down. Cells in the stack are checked with the most recently added cell being checked first until the goal cell is reached. As a result, it makes sense that the final path favors going down and going right, in that order.

Our implementation of A*-Manhattan adds unblocked/unvisited cells to a queue, ordering the queue based on the lowest estimated distance from the start to the goal using the sum of the distance traveled to a cell and the Manhattan distance from the cell to the goal and checking that a given cell's neighbors are unblocked/unvisited in the following order: right, down, up, left. As a result, it makes sense that the final path limits the Manhattan distance from the start cell to the goal cell.

Our implementation of A*-Euclidean adds unblocked/unvisited cells to a queue, ordering the queue based on the lowest estimated distance from the start to the goal using the sum of the distance traveled to a cell and the Euclidean distance from the cell to the goal and checking that a given cell's neighbors are unblocked/unvisited in the following order: right, down, up, left. As a result, it makes sense that the final path limits the Euclidean distance from the start cell to the goal cell.

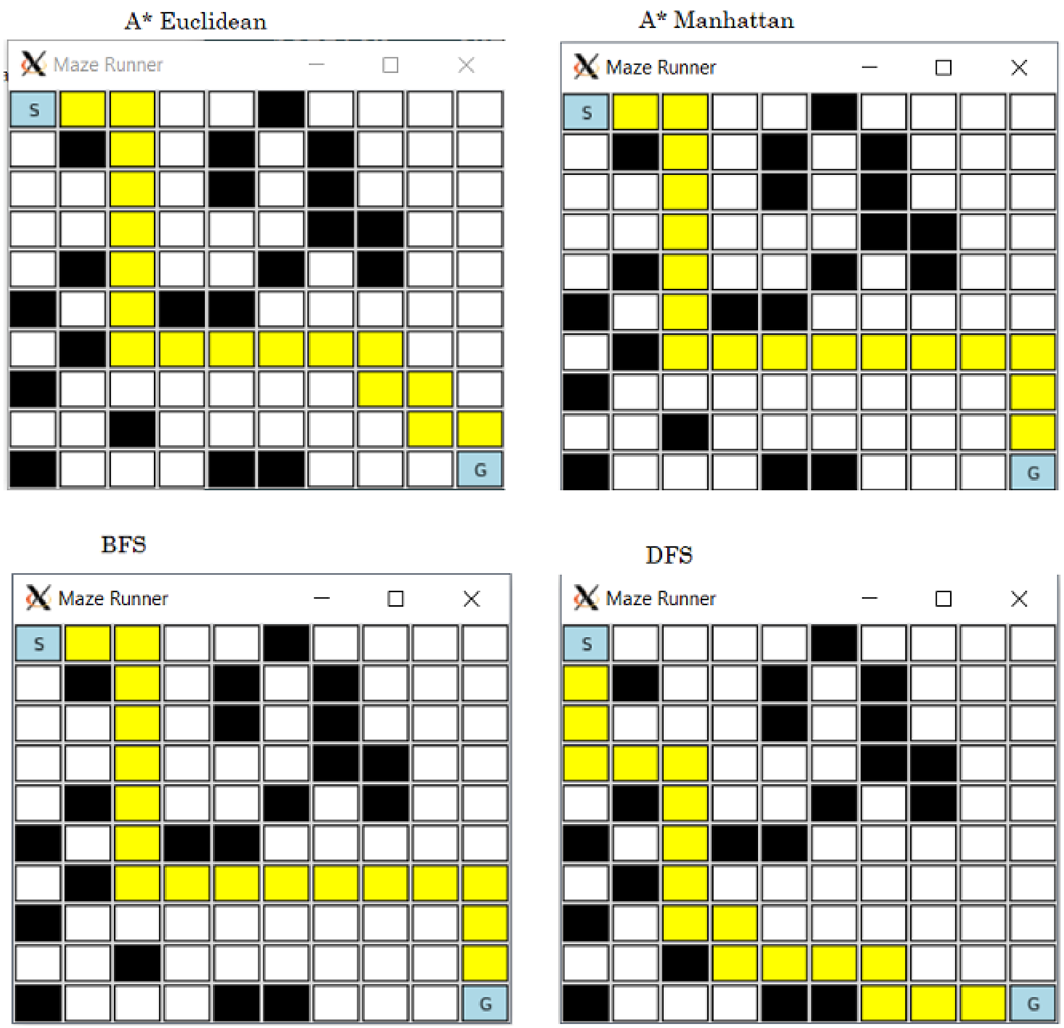

Figure 1: Solution Paths

2.3) Given dim = 30, Figure 2 depicts how maze-solvability depends on the value of p.

Methodology to generate graph for $p = 0.0, 0.1, 0.2, \ldots 0.9$: run DFS on a maze with dim

30. Repeat this for a total of 100 tests and keep track of how many were solvable. Then that ratio will give the estimate for the probability that a maze of the corresponding dim and p will be solvable.

The best algorithm to use here is DFS because it runs the fastest. We dont need optimality if all we need to know is if the maze is solvable or not, and DFS is guaranteed to find a path to the goal if one exists. We take 30 to be a good sample size that is representative of the true ratio of solvability for a given p.

We can see that the maze-solvability drastically decreases as the value of p increases. The more interesting question is, what is the threshold value, p_0, where for all $p < p_0$, most mazes are solvable (here we define most as any value greater than 50%), and for all $p > p0$, most mazes are not solvable. Based on the tests and the results depicted in this graph, that threshold value is $p_0 = 0.3$.

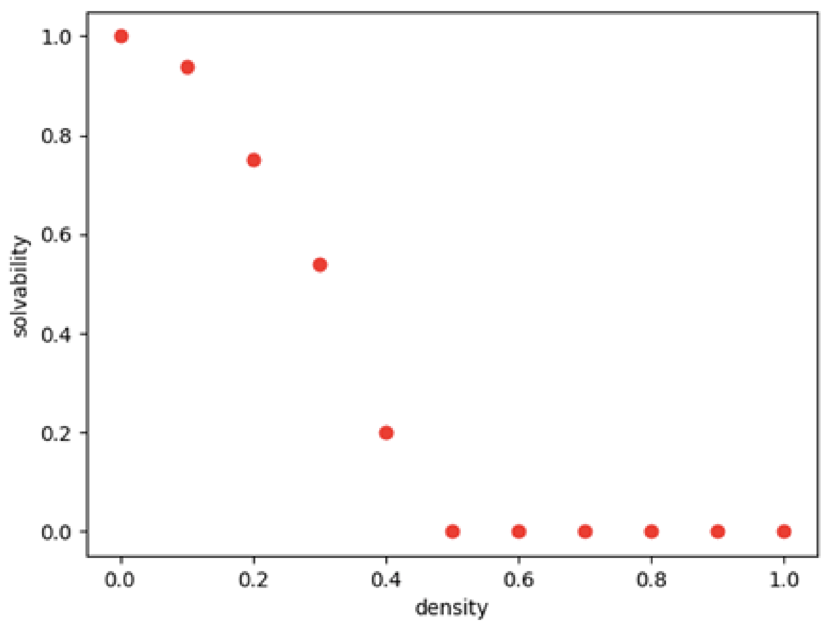

Figure 2: Solvability vs. Density

2.4) For each p in $[0, p0]$, we generated and solved 30 mazes, and found the average shortest path (see Figure 3). We chose a step size of 0.2. The BFS and A* algorithms are the most useful for this problem, because they return the shortest path by default.

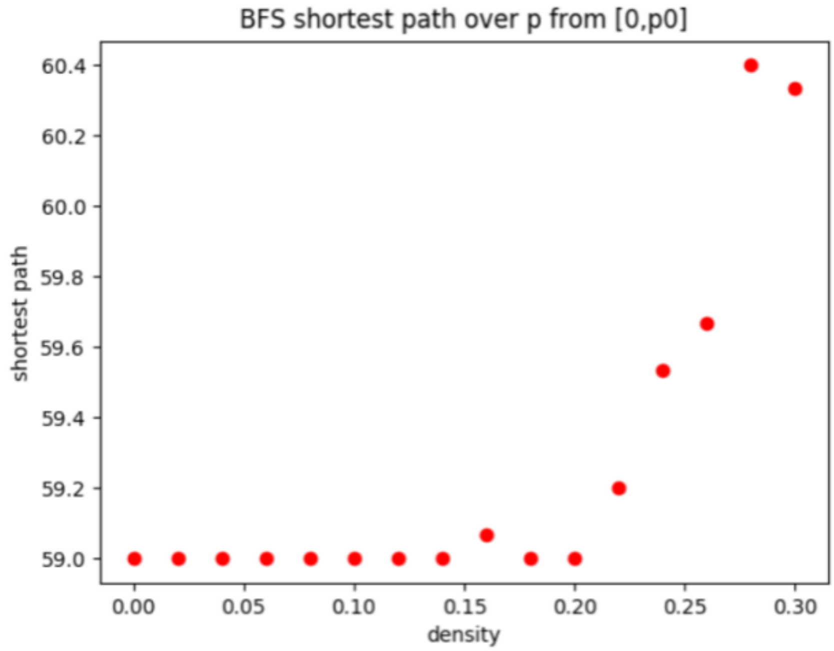

Figure 3: Expected Shortest Path vs. Density

2.5) The Euclidean distance heuristic helps produce a more direct path to the goal, however it is more computationally heavy. The Manhattan distance heuristic leads to more stretches of straight line paths within the overall final path, or in other words - less frequent changes of direction (less zig-zagging), but the path from the start to the goal is not necessarily direct and almost linear, as the Euclidean heuristic would have provided us.

Computing A* with the Euclidean distance heuristic takes much longer than computing with the Manhattan distance because it typically requires visiting more nodes. This is because when we estimate $h(n)$ using Euclidean distance, that results in lower values for $h(n)$, at some nodes, than they would have been with the Manhattan distance heuristic, therefore, we are more likely to visit those nodes.

Figure 4 shows how the two algorithms behave in comparison to each other, in terms of nodes expanded and runtime, with respect to p. In the left graph of Figure 4 the average runtime of A* Euclidean is vastly greater than the average runtime of A* Manhattan, for p values that yield a high percentage of solvable mazes.

This is supported by the results shown in the right graph in Figure 4, where we can see that A* Euclidean expands more nodes on average, than A* Manhattan does, (for p values that yield a high percentage of solvable mazes) except for when $p = 0.0$ because both algorithms then expand every node in the maze.

So, depending on the problem we want to solve, there will be trade offs for using either heuristic. The Euclidean distance is useful to generate more direct paths, but at the cost of increase computation time. The Manhattan distance is useful if you want quicker computation, but will give you paths that are suited for a grid-like area and not necessarily a straight-shot from start to goal with zero changes in direction.

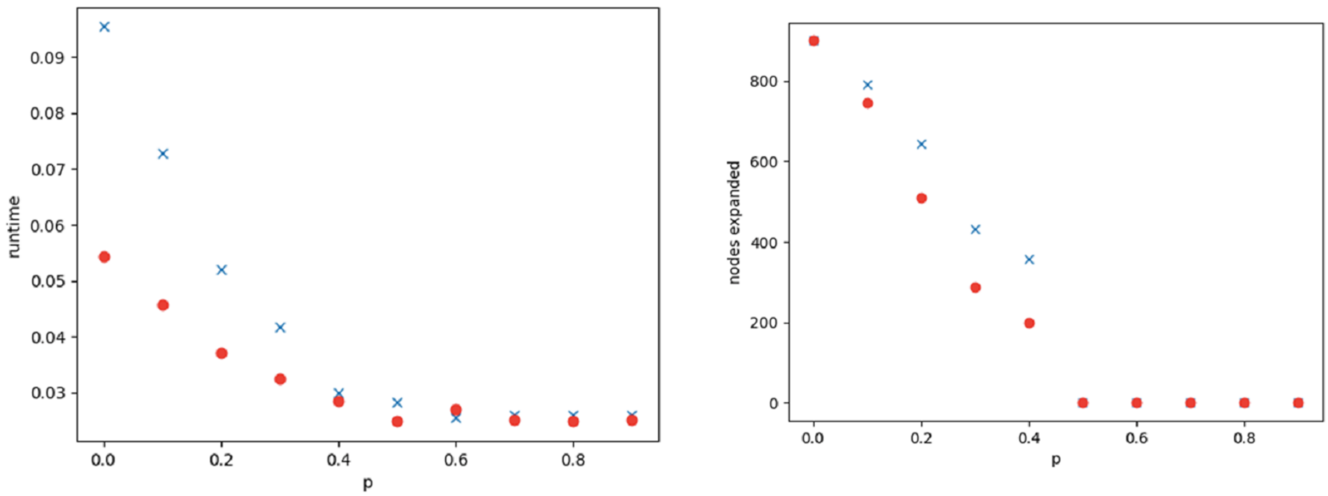

Figure 4: Manhattan vs. Euclidean (Red Dot: Euc, X: Man). Note: runtime measured in seconds

2.6) We generate a graph for average path length at various P to show that BFS is better than DFS at finding the shortest path. For the left graph in Figure 5, we see that when P is 0, BFS and DFS both produce solutions of equal length. However as P increases, the average solution lengths produced by DFS get longer, while the lengths produced by BFS remain stable (around 60). We also generated a scatter plot (the right graph of Figure 5) to represent the difference in length of individual maze solutions. The y-axis represents the length difference between DFS and BFS paths. We can see that there are no instances where the difference is less than 0 (we represent positive differences with red dots, and negative ones with black dots), thus there are no instances where DFS finds a better path than BFS.

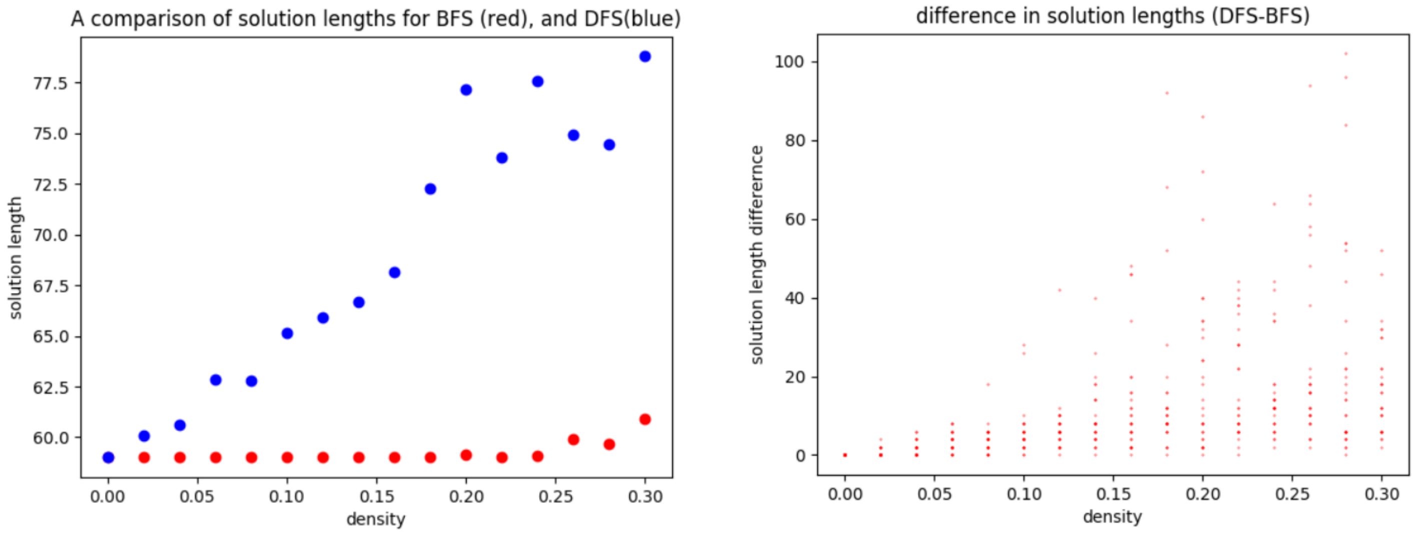

Figure 5: BFS vs. DFS

2.7) All four algorithms behave as expected. BFS uses a queue to find a path from the start cell to the goal cell, in addition to being optimal. DFS uses a stack to find a path from the start cell to the goal cell, and as a result of the nature of the algorithm and its implementation, is not optimal. A*-Manhattan and A*-Euclidean are modified versions of BFS, ordering the queue such that the lowest estimated distance from the start cell to the goal cell through a given cell is prioritized within the queue. However, the differing heuristics result in differing final paths for the two algorithms.

2.8) DFS performance can be improved by modifying the order in which neighboring cells are added to the fringe. For example, neighboring cells can be added to the stack such that the closest cell to the goal cell is the most recently added cell, so that each set of additions to the fringe prioritizes getting closer to the goal cell. DFS is not optimal, and as a result can expand more nodes than is necessary, so it is important that any modifications to the order of adding neighboring cells to the fringe intend to limit the number of nodes expanded and return a shorter path to the goal cell.

2.9) The threshold probability p_0 varies as dim changes. At lower values of dim (dim < 30), the majority of mazes become unsolvable at lower values of p, typically near 0.24 - 0.27. Conversely, as you increase the value of dim (dim > 30), the threshold value, p_0, where most mazes become unsolvable, converges at 0.3.

3) answers::

a)

b)

c)

d)

e)

4) Intuitively, the notion of a thinning A* seems like a viable efficient approach to searching. We found, however, that this is not the case. We sought to find the overall cost of using a thinning A*, using A* Euclidean as our base search algorithm, when compared to using A* Euclidean without the thinning process. We define cost as the total amount of nodes the search algorithm must expand. The thinning A* approach can be described as a sort of optimal directed DFS search; we solve a simpler maze, use the solution path as our heuristic for solving the original maze, and take detours if blocked cells are encountered along the path. This process by itself is faster than running any of the previously used search algorithms, yet when combined with the cost of obtaining the heuristic, this technique becomes less efficient as can be seen in Figure 6.

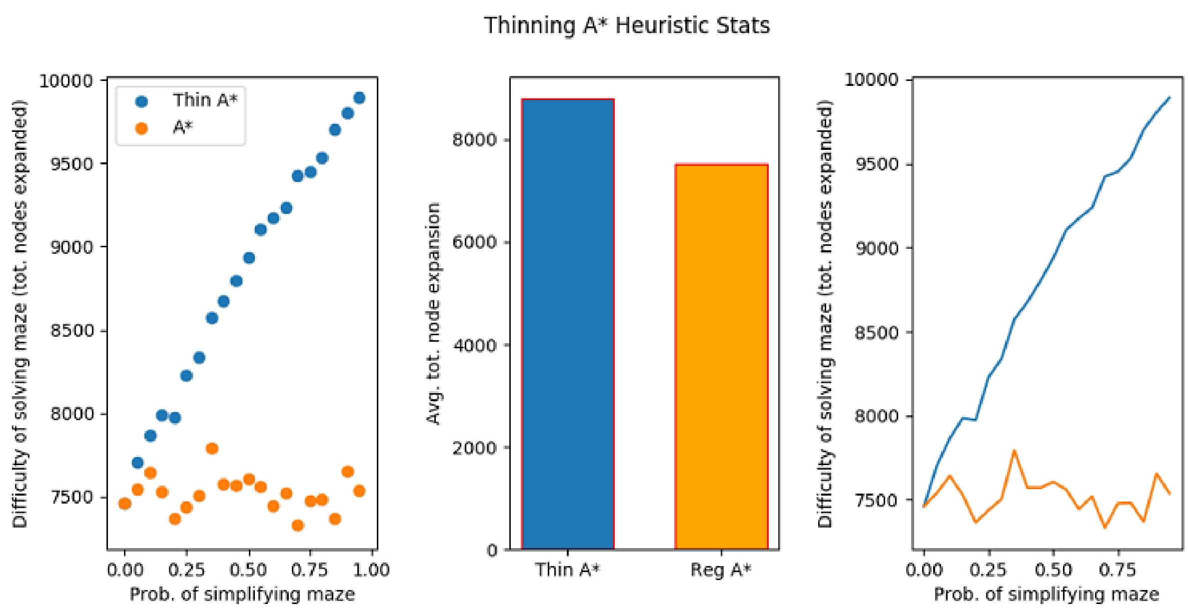

Figure 6: Thinning A* Heuristic

When considering the cost of computing the thinning A* heuristic, we quickly run into a problem; the overall cost of computing this heuristic get significantly larger, in terms of overall nodes expanded, as the probability (q) of simplifying the maze gets larger. That is, the easier the maze we solve, the more work the thinning algorithm has to do. It seems that having blocked cells in a given maze reduces the amount of work needed to search through that maze because these blocked cells constraint the search space - they eliminate paths from the search tree. A thinning algorithm can still be valuable, however, if the thinned maze is $\approx 5\%$ easier than the original. It is worth noting that we did not evaluate all the possible way this heuristic can be used or calculated and thus there might be other ways to exploit this idea. For example, if we know that the shortest path of the thinned maze is 10 cells, then we might be be able to use this as a

lower limit when solving the original maze by DFS limiting the search depth to 10 or so. Yet, we would still run into the problem of computing the heuristic.
The heuristic by itself is useful, yet the cost of calculating it makes it comparable, and usually less favorable, to the other A* algorithms for this specific problem.

5) In the previous examples our main goal was to find an optimal path, shortest in most cases, from start to finish. In this problem, however, there is a question of safety versus optimality. We are given that the probability of a cell catching on fire is $P(A) = 1 - (\frac{1}{2})^k$ where k is the number of neighbor cells on fire.
For clarity, we assume that neighbors only count as adjacent cells and not diagonals. The main idea behind the proposed algorithm (shown below) is creating a safest/optimal using a safety heuristic - the robot, or player, will move to any neighboring cell with a probability of $P(A^{\complement}) = 1 - (1 - (\frac{1}{2})^k)$. Essentially, this is a greedy best first search where the safest neighbor will be prioritized. We propose the following algorithm:

Input: A maze (two-dimensional array) M and starting vertex v (this is the starting cell) with blocked, fiery, or open cells as described by the problem .
Output: A valid path through the maze - a list L of coordinates.
1) Let fringe be a queue
2) enqueue v onto fringe
3) **while** fringe is not empty
    set node = fringe.dequeue()
    if node is goal state
        backtrace path to v through parent nodes and return list of cell coordinates L.
    else if node has not been visited
        label node as visited and record parent
        for all child nodes (neighboring cells) **do**
            discard any node that is blocked or on fire
            evaluate safety heuristic $P(A^{\complement})$ for each open node
            enqueue nodes from safest to least safe (highest to lowest probability)*
    else (node has been visited before or is on fire/blocked)
        discard node and continue
4) **if** no path is found : return none

note:* if nodes have equal safety heuristic value, enqueue in any order (clockwise, left to right, counterclockwise...).