CS 520: Assignment 1 - Search

All Figures are located at the end.

2.1 *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?*

A dim size of 100 seems a suitable candidate for such a range. As we can see in Figure 1, a dim of 50 is solvable for at least half of all generated graphs for all $p \leq .30$. Additionally, it works for a small section of graphs for $p = 0.35$. As solvability seems to remain relatively steady from dim = 10 onwards (the cutoff varies based on p), the solvability of 100 should be similar to the solvability of 50. Although we did not graph it, we tested the solvability at each level for 100 and it averaged roughly .02 below that of 50, so the difference is negligible. 100 requires a good amount of work, and such a large grid will mean that A* will get to experience a quite varied set of possible grids.

2.2 *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.*

(See Figure 2)

The results of BFS and DFS check out with their design. Since DFS operates as a stack, the nodes that are explored for the path are the ones that are closest to the currently explored node. This also explains why we see longer paths, as it does not explore the map space evenly. On the other hand, since BFS uses a queue, the nodes that are explored for the path happen one level at a time, more evenly distributing its search path. BFS is less affected by blocked paths than DFS, because of this exploration of nodes at the same level as the currently explored node before going to the next level. Since it both explores more nodes and is more evenly distributed, it is more consistent than DFS. A cluster of blocks hurts DFS more because as the current path gets longer, the more likely it is to be part of the solution path, whereas BFS's even exploration avoids this problem as it explores all nodes anyway.

The results of BDBFS also make sense. Compared to BFS, BDBFS has the added aspect that it explores from both the start and goal, enabling them to meet in the middle at the same rate. As seen in figure 2c, the path converges in the middle as expected.

The results of both A* Search algorithms make sense as well. The added heuristic of distance to the goal enables A* to explore nodes closer to the goal rather than every node at every level in BFS. The result path of Euclidean A* (2d) makes sense as it explores nodes that are closest in a straight-line distance to the goal. However, while Euclidean A* would usually seem most natural, it focuses on being straight diagonal despite the constraints of grid movement, and so its heuristic constantly corrects for straight distance rather than the true distance, which Manhattan A* uses as its heuristic.

The result path of Manhattan A* (2e) looks exactly like BFS (2b). So while the result may at first seem surprising, the nodes that the Manhattan heuristic prioritizes (true distance says to prioritize nodes more down and more to the right) are often the same nodes prioritized by our general queue mechanism (first node to look at is directly below, then directly right node). In this

way, the Manhattan A* finds a very similar if not the same path as BFS, but it is still valuable because its heuristic results in less expanded nodes than BFS.

For further analysis of these algorithms, see 2.5 and 2.6.

2.3 *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.*

Given dim, maze solvability decreases as p increases (p scales inversely with maze-solvability).

The best algorithm to use here is BD-BFS. This is because maze solvability is naturally more likely to be "killed" in the first few diagonals on each side, compared to the middle diagonals. Mazes are "unsolvable" when a path, consisting of adjacent or diagonal blocking elements, crosses the grid from one side to another such that it splits the grid in two, with goal in one half and start in the other. Consider the case of just the diagonal being blocking (ignoring every other type of blocking "path"): the first diagonal, starting at grid[0][1] and going to grid[1][0], will be able to block all possible paths from start to goal with just two squares (Figure 4). With a p of 0.5, the chance that both of these being blocked is 0.25. That means that a quarter of the time, the grid will be blocked before it even has a chance to explore most of the grid! As we can see in Figure 1, the first few dims drastically reduce the solvability, but eventually the rate of change decreases to near zero. This indicates that the earlier diagonals were much more impactful, in terms of solvability, compared to the later ones. For us, this means that the "blocking path" that makes the maze unsolvable is likely to be found earlier than later. However, since the maze can be blocked on both sides, near the start or the goal, using any other algorithm other than BD-BFS would cause us to have to explore the majority of the maze if a diagonal near the goal is blocked, but not a diagonal near the start. BD-BFS allows us to start in both places, so that we are likely to find the blocking path on either side quite early into our search, and avoid the expensive middle areas of the grid.

As an added bonus, if it's deemed too expensive to check the entire grid for blocking paths, we can also predict with high accuracy whether a grid is blocked or not after BD-BFS has expanded some, but not all, nodes.

To determine p_0, again observe Figure 1. Looking at the rate of change for each value of P, it seems reasonable that 50 is a good indicator for all dim > 50. While the graph gives you several possible values, I will be defining "most" as 85%.The solvability value for P = 0.20 was found through 10,000 iterations to be approximately 0.8431. This is very close to .85 (which is itself a somewhat arbitrary value, since defining "most" is somewhat subjective). Therefore, it seems that p=.2 is the point above which "most" grids are solvable.

It must also be noted that it is not true that “most” grids are not solvable below 0.2- a fair few are. To find the point where 85% of grids are *not* solvable, Figure 1 (and the underlying data) would tell us that this value appears to be approximately .38. For p values greater than this value, more than 85% (ie “most”) grids are unsolvable.

2.4 *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?*

From 0 to 0.2, the length of the shortest path is almost always equal to 2 * dim - 1. This can be clearly understood by observing the causes of an increase to shortest path. It is not enough for a blocked cell to lie on the path A* would otherwise take - in order to increase the shortest path, A* must be forced to move to the left or up (in an empty grid, it will never do this with Manhattan), which would require many blocked cells working in conjunction. The density vs expected shortest path graph, Figure 5, shows the relationship between density and shortest path length. For all values below p=0.3, there is little change. However, starting at p=0.3, the length of such a path greatly increases.

The most useful algorithm for solving this is A*, particularly with the Manhattan distance heuristic. This will most efficiently find us the shortest path, should one exist. For some high values of p, where solvability is low (see Figure 1), it may be useful to use BDBFS, as this will find the shortest path when it exists, and is better and can determine whether a grid is solvable earlier.

2.5 *Is one heuristic uniformly better than the other for running* $A*$ *? How can they be compared?*

Manhattan distance is superior to Euclidean distance, which is in turn superior to BFS. These can be compared by the number of nodes expanded (that is, the number of nodes visited and added to the closed list).

As we can see in Figure 3, for all values of p <= 0.4 (ie, all solvable grids), Manhattan outperforms Euclidean, which in turn outperforms BFS. As p approaches 0.4, the difference between Manhattan and Euclidean decreases. All of these observations make sense, as we will discuss in 2.6.

2.6 *Do these algorithms behave as they should?*

They do. It is reasonable to assume that as both Manhattan and Euclidean are consistent and relatively accurate, they would improve the ability of A* to avoid expanding nodes that are unlikely to be on the shortest path. In our observed data, they both accomplish this. Compared to BFS, which is A* with a heuristic of h(x) = 0, they decrease the total amount of nodes expanded. Additionally, Manhattan (which seems to be a closer lower bound to the

problem than Euclidean) outperforms Euclidean. This should be expected, as the “best” consistent heuristic is the one that most closely underestimates the remaining actual distance, and in an empty grid, Manhattan does this perfectly.

Understandably, the utility of the heuristics relative to BFS deteriorates as p increases. This also makes sense- the heuristics are now less accurate as a predictor, because they do not account for blocked squares. However, they still perform better than BFS.

The reason Manhattan increases as p increases, while Euclidean decreases, is due to the nature of the heuristics. A heuristic’s ability to save nodes from being expanded is derived from how accurate it is as an estimation. Manhattan starts out as a perfect estimation for p=0, and gets progressively worse as more and more blocked cells appear, which Manhattan doesn’t account for. Euclidean also gets worse and worse, relative to BFS (which shows the total number of cells able to be expanded per problem), but it decreases in raw number of expands, because there are less cells able to be expanded. Relative to BFS, both heuristics eventually decrease in utility as p increases, because they become less accurate predictors.

2.7 *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.*

Yes. DFS has 4 options when choosing where to go next: up, down, right, and left. One of these will obviously not be an option, because it is the node that DFS has just come from. Of the other three, down or right will generally be optimal choices on empty graphs. This is because we are looking for the most direct route (since DFS terminates when it reaches goal, the most direct route will be the most performant). The rooms that are most down and to the right are the most optimal in this case because the goal is down and to the right of the start. This is similar to a heuristic - we are telling DFS to move towards a specific direction whenever it can, similarly to how Manhattan and Euclidean heuristics tell A* to favor moving towards a specific point as well. As is the case for heuristics, this “optimal” order gets less reliable as p increases, but since it is similar to a heuristic, it will generally be an improvement over any DFS that prioritizes other directions.

2.8 *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, there are. Observe the case of Figure 8. A* and BD-BFS have returned the same path on an empty 5x5 array, but BD-BFS has expanded nodes that A* has not. This is because BD-BFS requires that on an empty grid of size 5x5, it cannot exclusively expand nodes that it will eventually include in the shortest path. BFS must expand all of the start’s and goal’s neighbors before it expands all their neighbors’ neighbor’s. Therefore, since the path will be of length 2*dim - 1, it must be expanding nodes that will not be included in the shortest path, since it expands more than 2*dim - 1 nodes. A*, on the other hand has the interesting feature of being

able to expand only nodes that will eventually be in the shortest path, in some cases. Since we use Manhattan distance and prioritize nodes in our frontier with higher g values, in the cases of ties, A* will follow the node it expands each time, and eventually reach the goal without expanding all the extra nodes BFS did.

Bonus: *How does the threshold probability p_0 depend on dim? Be as precise as you can.*

For small dims, p_0 (p = 0.20) decreases as dim increases. However, for larger dims, the relation fades off, since the rate of change for solvability approaches 0. Looking once again at Figure 1 (we're getting a lot of mileage out of that one!), we can see that p=0.2 reaches its "final" solvability rate at about dim = 7. If we use a higher p_0, it will reach its "final" solvability rate at a higher dim, but will not continue to scale with dim after that, for reasons discussed in Question 2.3. Ultimately, the relationship 'heavily scales inversely (as dim increases, p_0 decreases) early, then eventually flattens for larger dims.

3.1 *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 apply this search algorithm to this problem?*

The local search algorithm used was simulated annealing using random restart. It was used because while there weren't too many local maximums the search was getting stuck on, they were easy to implement and add to an otherwise basic hill-climbing approach. As for environment representation, up to four different grids were stored at once: the current maze, the potential next-step maze, the best maze in the current run, and the best maze overall. As the search moved forward, the current maze would be discarded and only hard enough mazes would be stored in the appropriate variable. In order to apply the search algorithm, the global search algorithms needed to be modified slightly to allow for measurement of "difficulty" (fringe size or number of expansions). Included in their return statements was data on their fringe size and explored nodes.

3.2 *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?*

A time variable was applied to the local search, since after a certain point most runs of the algorithm would get to a similar difficulty rating, and while spending more time would slowly give more results, execution was getting too long. Then, a small number of random restarts were added, to help try to find a higher local maximum, since restarts for better results would have been done manually anyway. The values chosen for these were selected only by trial and error, and could probably be optimized, as too small values results in finding weak mazes, but too high values would take unreasonable amounts of time to compute. It was found over time that less time should be devoted to random restarts and more time to further exploring specific scenarios, so we kept scaling the variables keeping the time about the same, but putting more effort into each random restart.

3.3 *Try to find the hardest mazes for the following algorithms using the paired metric:*
See Figure 7 for an example of each

3.4 *Do your results agree with your intuition?*

While the results are less extreme than expected, they still reflect what was expected. To get a DFS search to have a large fringe, the path it takes must be as long as possible so that it

passes as many potential cells as it can. Given more time, the DFS search would have produced paths that look increasingly folded over themselves. Meanwhile, to get A* to expand as many nodes as possible, the correct path must deviate away from the heuristic. Although the given maze in figure 7 doesn't look very complicated, attempting to move directly toward the goal (Manhattan distance style) runs into several large walls that must be pathed around. Having to path around these walls causes A* to have to search the space behind the wall nearest the goal first, until it eventually backtracks enough to go around it.

4.1 *How can you solve the problem (to move from upper left to lower right, as before) in this situation? As a baseline, consider the following strategy: simply find the shortest path in the initial maze, and run it; if you die you die. At density* p_0 *as in Section 2, generating mazes with paths from upper left to lower right and upper right to lower left, simulate this strategy and plot, as a function of q, the average success rate of this strategy.*

Figure 6 contains a graph of the survival rate of the baseline algorithm as a function of q, compared to our improved algorithm.

4.2 *Can you do better? How can you formulate this problem in an approachable way? How can you apply the algorithms discussed? Build a solution, and compare its average success rate to the above baseline strategy as a function of q. Do you think there are better strategies than yours? What would it take to do better? Hint: expand your conception of the search space.*

We can drastically improve upon this baseline algorithm. In earlier parts, our grid space was static, and so we treated it as such. Now that it is dynamic, we can no longer afford to treat it statically. Since we know our g value for any given coordinate, we can generate a probability value for each cell in the grid, determining the probability of it being on fire g steps in (similar to the result of a random walk/ stochastic matrix). We can then set a threshold for the value a cell's probability of being on fire, signifying that visiting any cell with a predicted probability over this threshold is a risk we won't take. If the value of an element is equal to or greater than this threshold when we would reach it, we treat this as if it is blocked, and avoid it. We then run A* for a range of thresholds, as precisely as required. This gives us the opportunity to observe the different tradeoffs between time and safety, and pick the one that satisfies our needs best (speed being determined by the length of the path). For this assignment, we prioritized safety over speed (always choosing the safest possible route), and this algorithm vastly outperformed our baseline in terms of safety. Often, our improved fire-adverse A* found alternative solutions that were nearly as fast (when they existed) on the opposite sides of the grid from the fire's starting cell, but significantly safer.

However, we can (and do) modify this algorithm further, by allowing A* to add it's observations into our conception of the grid space. In this way, the algorithm runs it's path while updating the graph state with it's observations of fire, and then recalculates A* from its current position to the goal when it, given these observations, realizes that it's about to walk into fire and die. By taking the new path, our algorithm can react to both good and bad luck, and pick paths that would otherwise be considered improbable, should they exist. The best results (shown in Figure 6) were discovered when combining our algorithm with this idea of updating the graph space with our observations and recalculating A* from intermediate points. In this way, we can evaluate what seems to be the best path, and then alter this path based on whatever specific scenario we encounter. This combined algorithm is able to backtrack down bad paths, determine whether risky paths are actually viable, and react the best to unexpected scenarios. At it's best,

compared to the baseline algorithm (around q = 0.3), our improved algorithm has almost twice the survival rate.

This algorithm, while effective, is still somewhat immature. In terms of ideas for future improvement, we could calculate an overall probability of being alive after completing each path, instead of just looking at it on a cell-by-cell basis. This would allow our algorithm to find paths that are extremely safe with a few exceptions, something our algorithm currently would not find. As an example, a path that goes through 9 cells with 0.1 probability of being on fire when we go through them, and then 1 cell with 0.5 probability, would be considered less safe than a path that could through 10 cells with 0.3 probability of catching on fire. This shows that our algorithm doesn't always necessarily capture the safest path, although the path we find is likely to be the safest, as this exception is a very rare edge case.

We could potentially create an algorithm to "rate" the paths we found, parameterized by the path's speed vs the path's safety. This would let our maze runner have a greater degree of autonomy, as currently it's up to the user to notice that a path of length 10 with safety 0.5 is better than a path of length 30 with safety 0.45. We currently have an algorithm that does this, but it just picks the path with the safest route, and does not consider speed. This algorithm would still be situational, however, because different scenarios would value tradeoffs differently.

Our algorithm is also relatively inefficient - we need to calculate the probability values many times, which can require a lot of resources for large grids. We could improve the efficiency of our algorithm by looking at some specific edge cases. If our g value is less than the manhattan distance of the fire start to our current cell, it's not possible for fire to exist there (it would not have spread in time), so we shouldn't have to generate probabilities for many of the cells near the start. If we wanted extreme efficiency, we could abandon the probability approach altogether and simply shift our heuristic to favor locations more downwards than before, so that we avoid solutions that go through the upper right corner. Such an algorithm would likely be less accurate, but would not require analyzing the potential probability of a cell being on fire.

One thing you may notice is that we have not considered the option of staying still. This is because there is no purpose in staying still. The fire won't die down, so staying still just makes the grid less survivable. Things will never get better if you wait, only worse.

Figures:

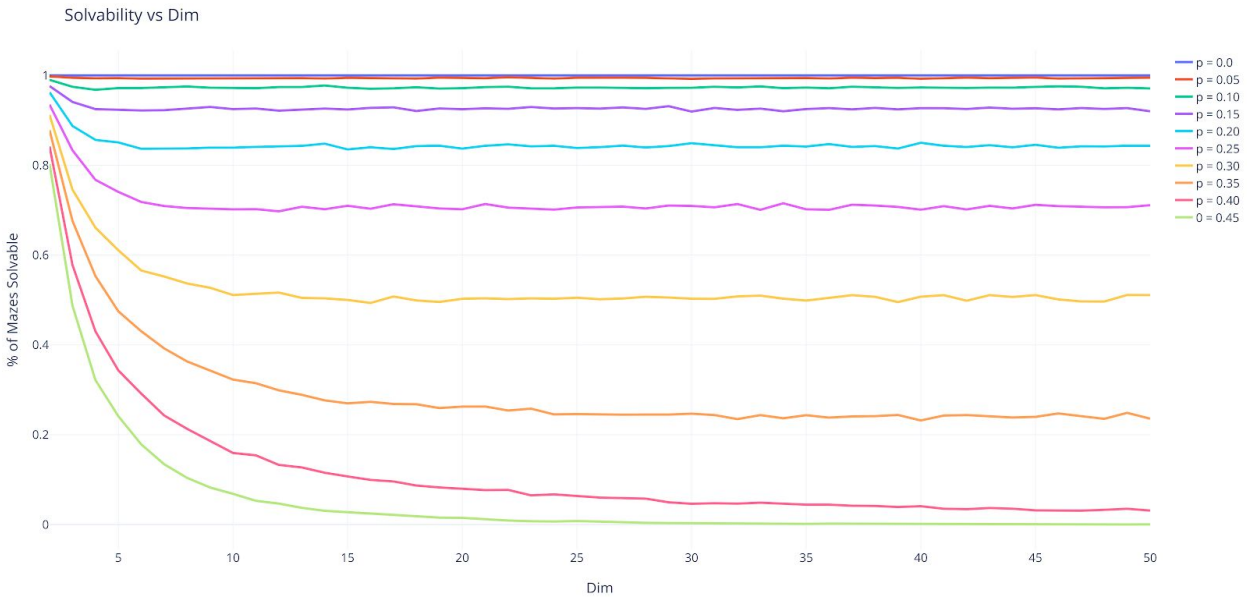

Figure 1: 10k iterations at each probability for each dim

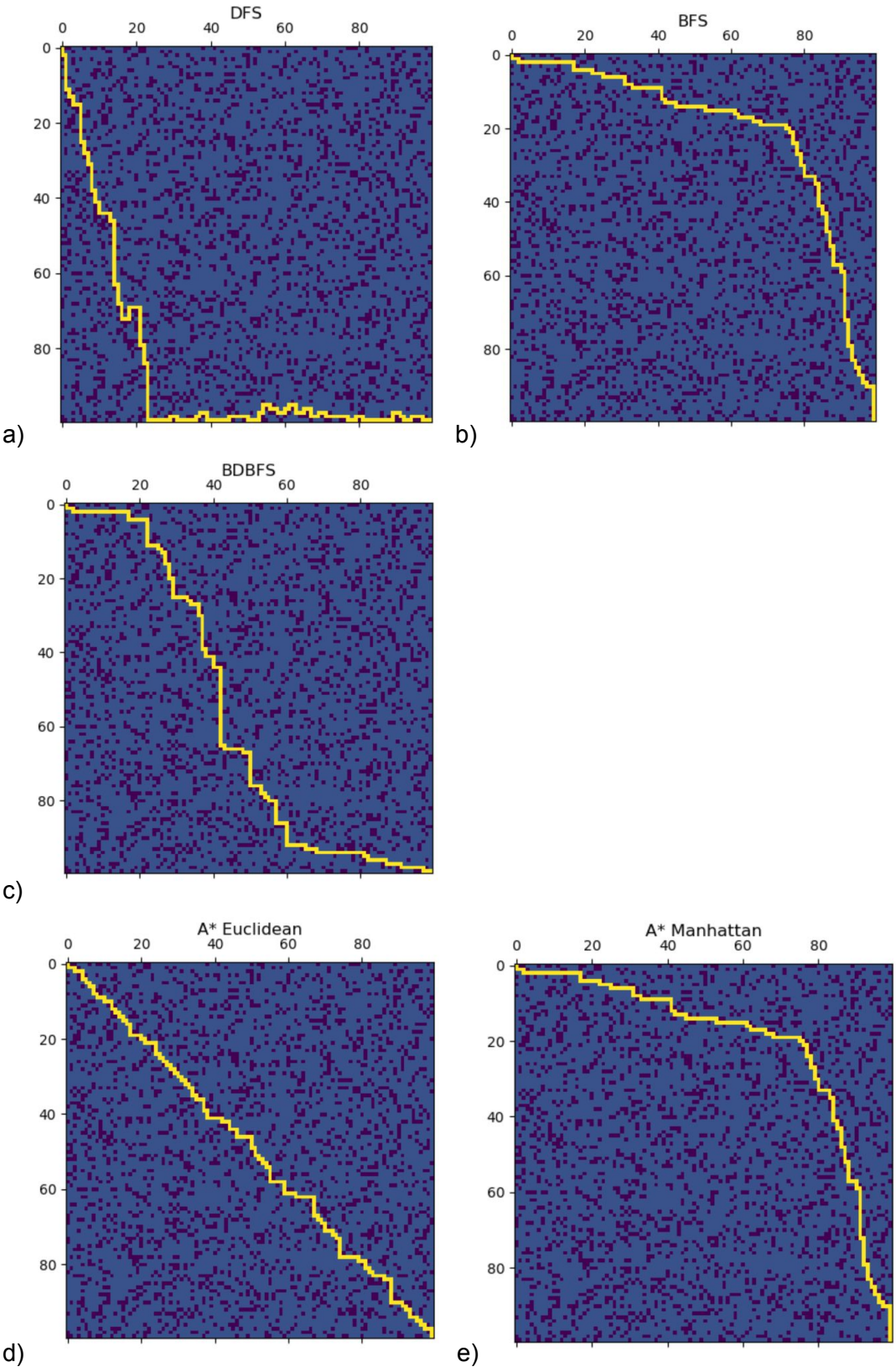

Figure 2: DFS, BFS, BDBFS, and A* variant algorithms run on a 100x100 map with p = 0.2

Figure 3
BFS vs Manhattan A* vs Euclidean A* for a range of p and dim

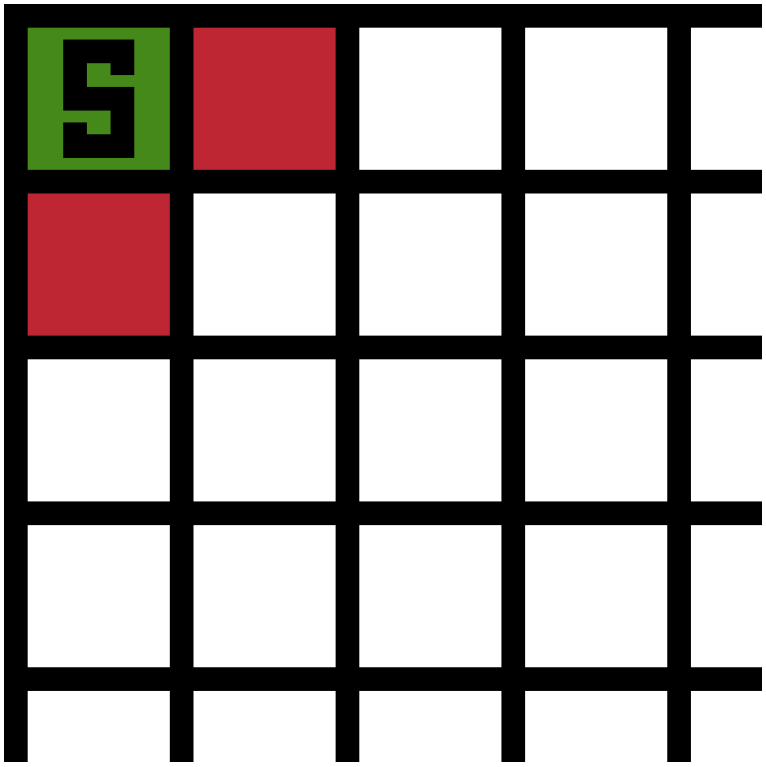

Figure 4
Indeterminately large grid being blocked by two cells. Green S is start, red cells are blocked.

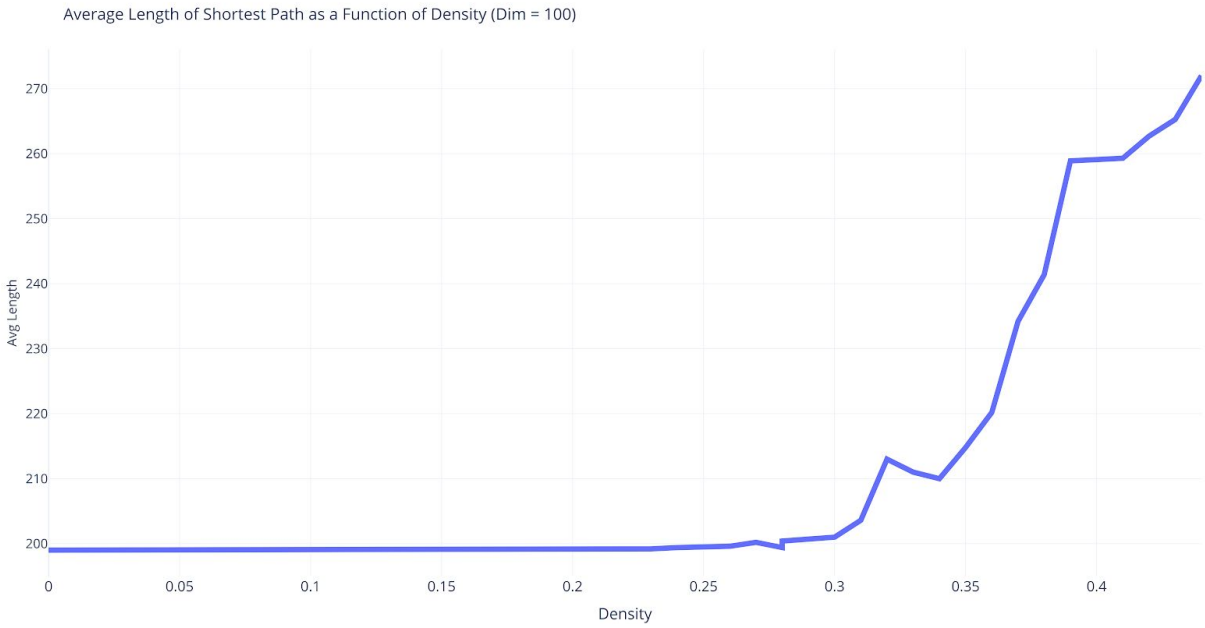

Figure 5
Density vs Shortest Path Length

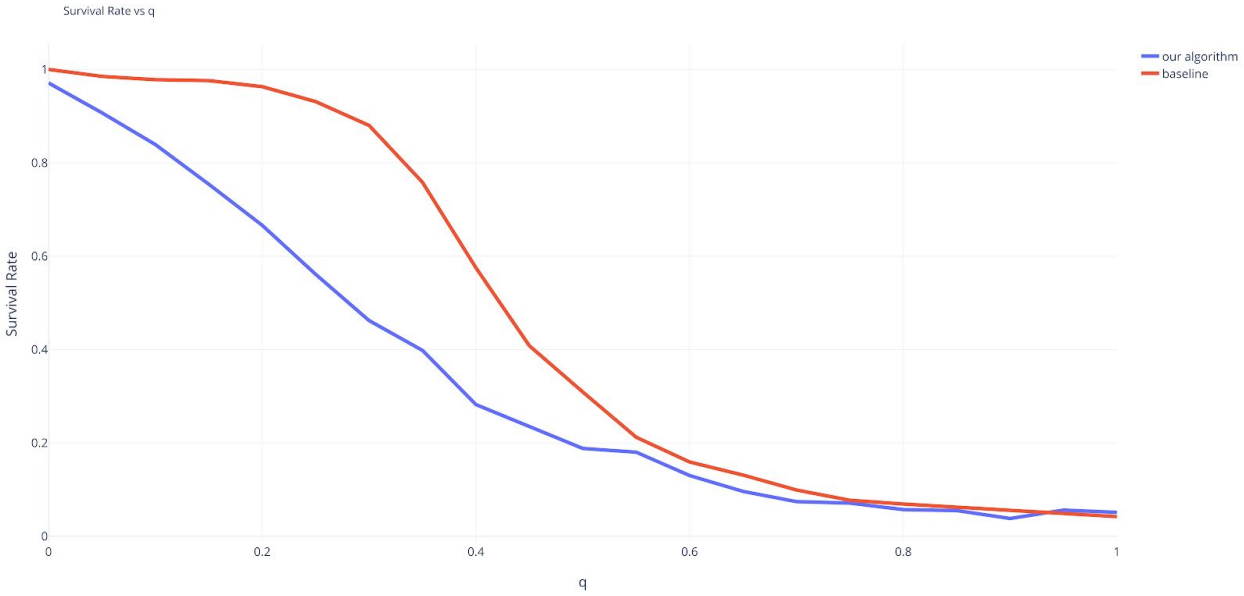

Figure 6
Survival rate of baseline and improved algorithms, parameterized by q

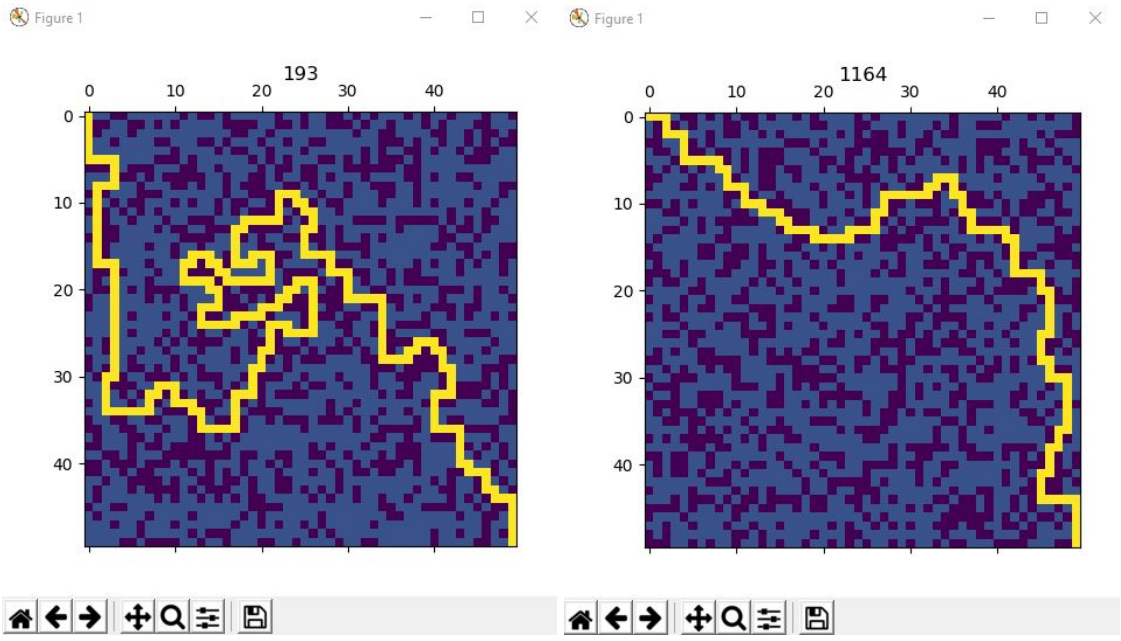

Figure 7
Hardest paths for DFS by maximal fringe size (left) and A*-Manhattan by nodes explored (right). The images only show the eventual path returned, not the space explored by the algorithms. The numbers at the top represent the measurement used to decide “difficulty” (maximal fringe size and nodes explored).

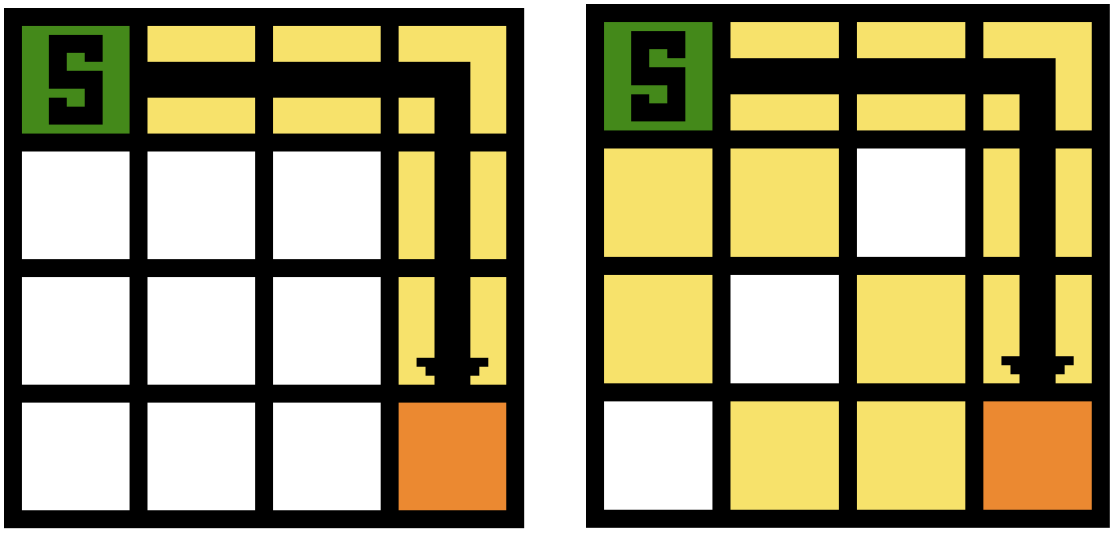

Figure 8
A* (with Manhattan distance) and BDBFS, (from left to right respectively). Yellow cells are expanded (in closed list), orange is the goal.

Contributions:

Eric Schneider (ejs229): Coded gridspace, DFS, BFS, A*, and some helper functions for navigating grid space. Also implemented fire grids, fire-adverse A*, and wrote answers for problems 2 and 4. Generated test cases/evidence of correctness/plotted some of the data for the aforementioned algorithms.

Ryan Cirincione (rjc355): Coded BDBFS, coded and answered problem 3

Basim Jaffer (baj61): Coded visualizations of paths on grid, answered 2.2, helped with 2.4, 2.8