RBE502 - Homework Set 9

Keith Chester

Due date: November 3 2021

Introduction

In this assignment, we are charged with exploring various path planning algorithms - specifcally depth-first-search (DFS), breadth-first-search (BFS), and Dijkstra's search. We also explore A* and greedy best-first search (greedy) search, and completey random neighbor selection (random) algorithms for comparison.

In this project, we are looking at a discrete space of a square grid. The grid consists of individual cells that are represented as occupied by an obstacle (a black square), as empty (a white square), the robot's starting position (a red square), and the goal (a green square). In our provided examples, an additional light blue square is used to show areas that were considered during path planning, but not utilized, and a dark blue square for the final generated path. For movement, we only consider orthogonal movement, not diagonal.

How Each Planner Works

For our initial planners, BFS and DFS, we operate exactly the same but store our considered neighbors into different queues. For BFS, we utilize a FIFO (first in, first out) queue to explore the breadth of each neighbor before diving a level deeper. For DFS, we utilize a LIFO (last in, first out) queue to explore the depth of possible children before considering breadth of other adjacent neighbors.

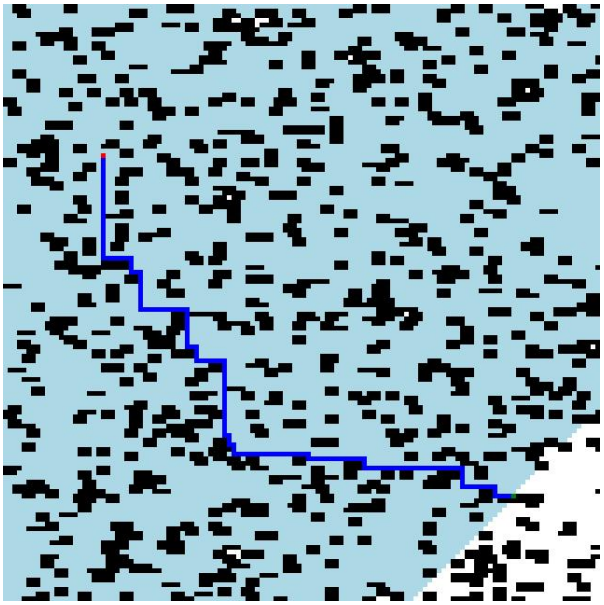
For our random planner, we simply store all neighbors in a list and choose from our queue randomly each time we consider a new node.

For our remaining planners, we use priority queues with slightly different scoring techniques. Defining a $g(x)$ as a cost function for moving to each cell, we increment the cost to reach each cell by one for each cell we've moved from the start. When we encounter a cell that we have encountered prior as a possible new neighbor, we check to see if we have a lower cost to reaching this neighbor than before; if so, we note the new cost and mark the node as having a new "parent" for once a path is found. For Dijkstra's, the cost $g(x)$ is utilized alone.

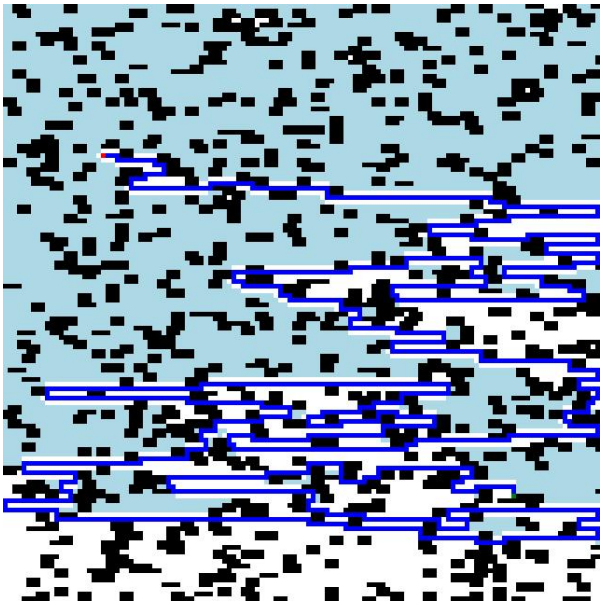
We then introduce a heuristic function $h(x)$ - which, for our example, is a simple Euclidean distance function bewteen g goal and c current $d = \sqrt{(x_g - x_c)^2 - (y_g - y_c)^2}$. For our greedy function, we use the priority queue with this alone defining the priority of each cell. For A*, we utilize a priority defined by $g(x) + h(x)$ - or, the cost to reach each cell with the estimated distance to the goal. Cells that are easier to reach that are closer to the goal are thus prioritized.

Examples

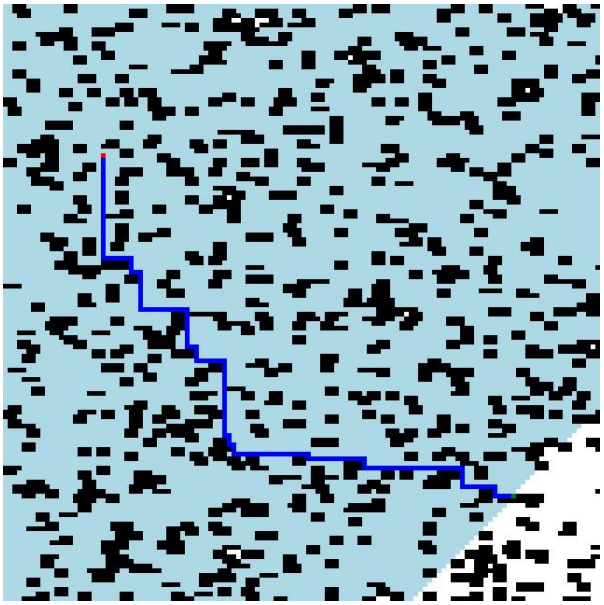
Here we see an example map of 128x128 grid cells. In this example, we fill the map with obstacles until we hit a desired coverage percentage - in this case, 35% coverage. We see the total considered cells and final path from the robot's starting position to its goal.



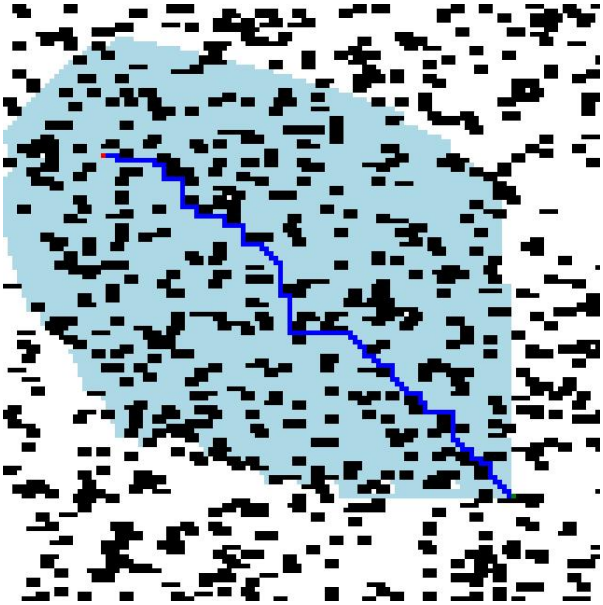
(a) Breadth first search (BFS)



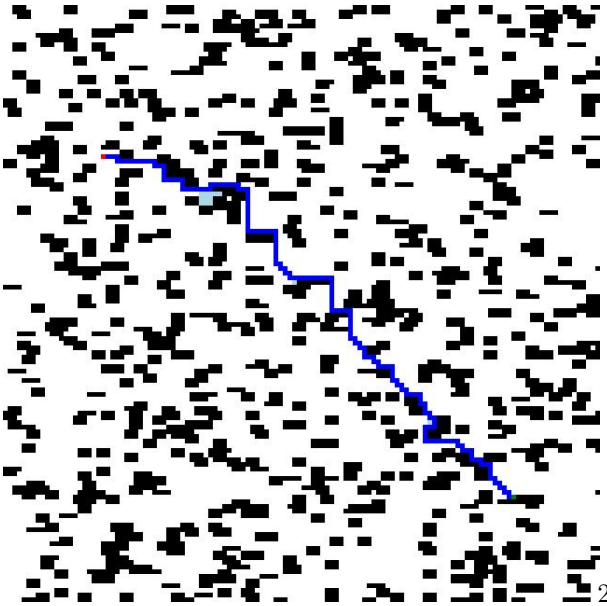
(b) Depth first search (DFS)



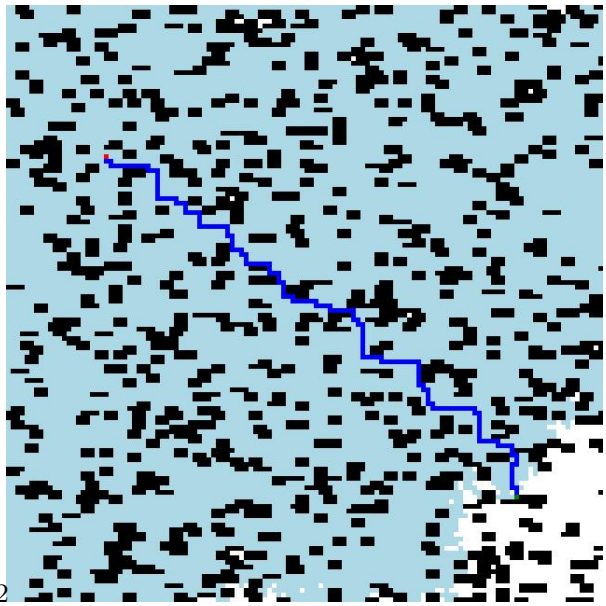
(c) Dijkstras



(d) A*



(e) Greedy



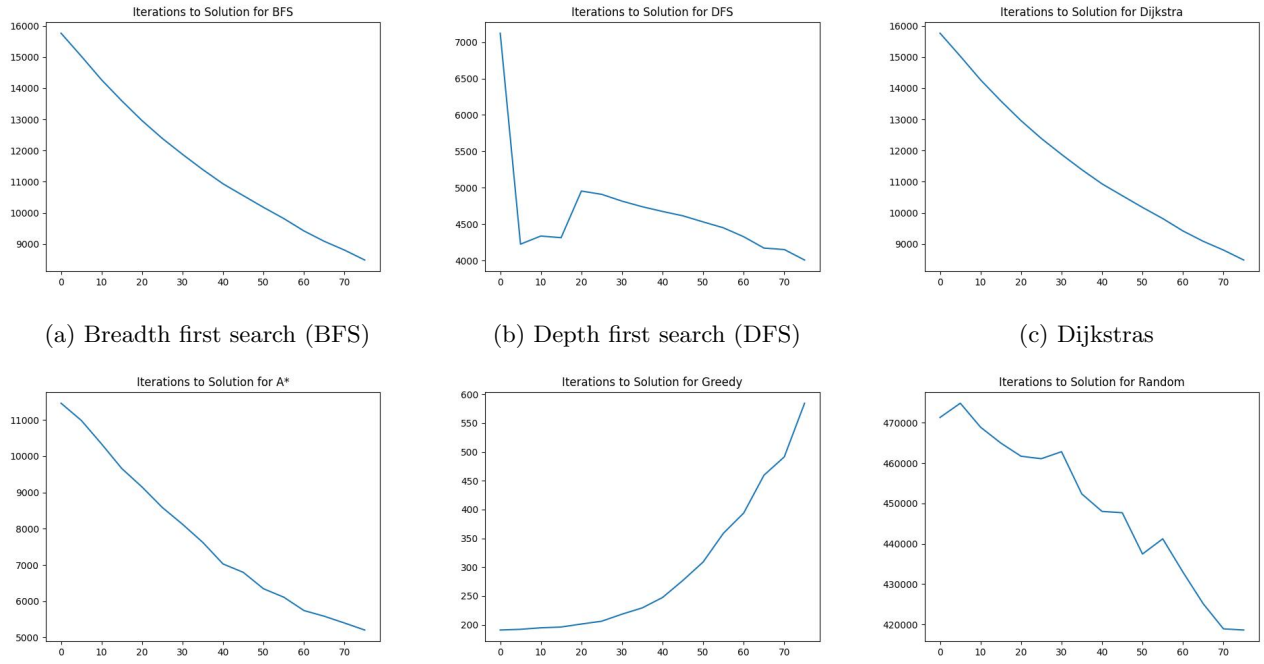
(f) Random

In our above figure of example maps, light blue cells emit outward from the start (red) until it reaches the goal (green), and a dark blue path directs us to the goal. For BFS we see the majority of the map covered, and then the optimal path towards the goal. DFS is far crazier - since the algorithm continually digs deeper into the children of each neighbor before electing to explore an earlier child, we see a wild exploration before we see an equally wild and inefficient route. Dijkstras matches the BFS image, which we discuss later in the **Results and Conclusions** section. For our A* search, we see a similar outward branching present in the BFS and Dijkstra examples, but to a lesser extreme due to its use of a heuristic function (in our case, Euclidean distance to the goal from current position). The optimal path is still found. Our greedy search - which only looks at the heuristic of distance to goal and not at all the to-reach cost that Dijkstra's and A* would, seems to explore less space but finds a non optimal route to the goal. Finally our random search it appears to be just that - random until it bumps into the goal and attempts to form its path from its chaos.

Performance

We took each algorithm and plot total iterations taken to find a solution and the resulting path length, as well as all algorithms overlayed. We explored each algorithm from 0% obsacle coverage to 75% coverage with increments of 5%. For each obstacle percentage coverage, we randomly generate 500 solvable maps and test them upon each algorithm. If a generated map was not solvable (no path existed), we would ignore that attempt for our total of 500. With this approach we have a large body of equivalent tests for each algorithm and can draw some conclusions.

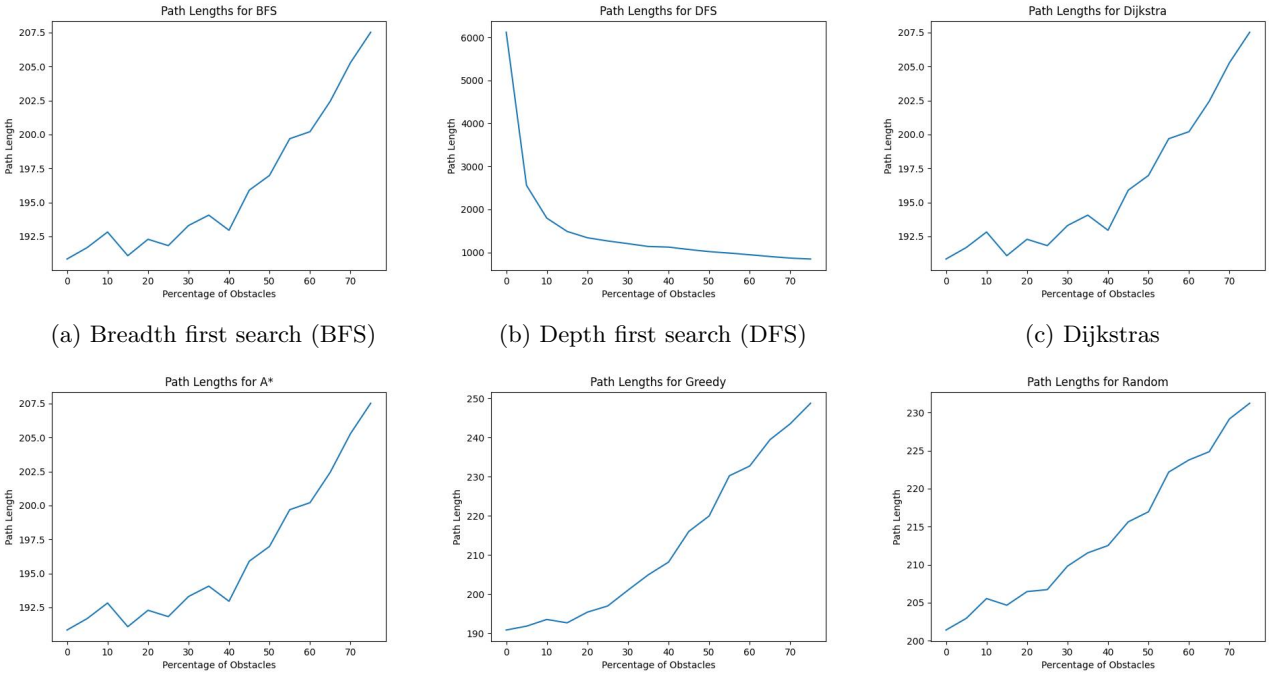
The individual performance of each algorithm is shown below:



(a) Breadth first search (BFS) (b) Depth first search (DFS) (c) Dijkstras

(d) A* (e) Greedy (f) Random

Figure 2: Iterations per planner algorithm



(a) Breadth first search (BFS) (b) Depth first search (DFS) (c) Dijkstras

(d) A* (e) Greedy (f) Random

Figure 3: Path length averag per each planner algorithm

Finally, we overlay the data for planner for comparison:

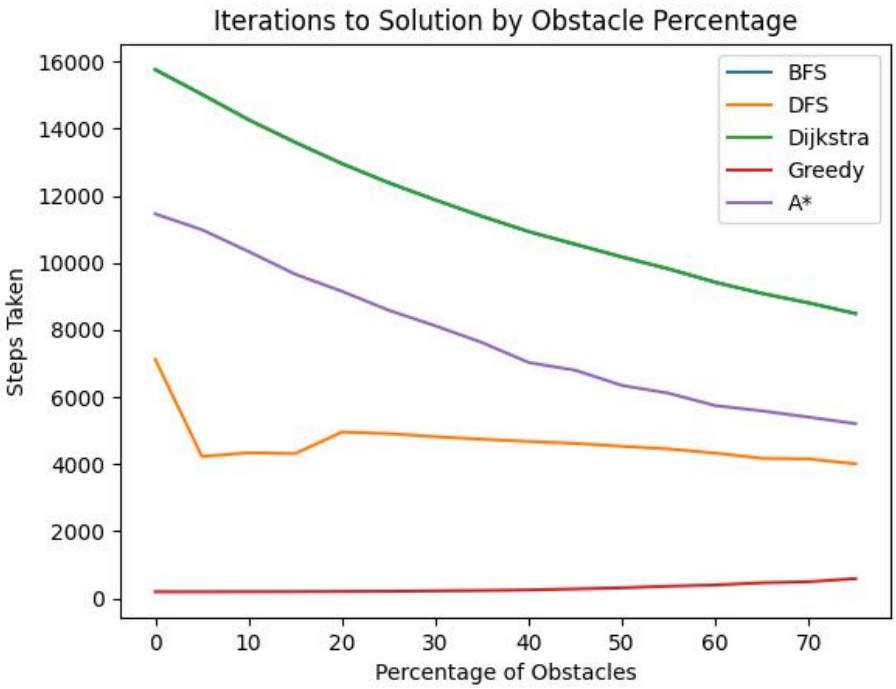


Figure 4: Iterations for each planner algorithm overlayed

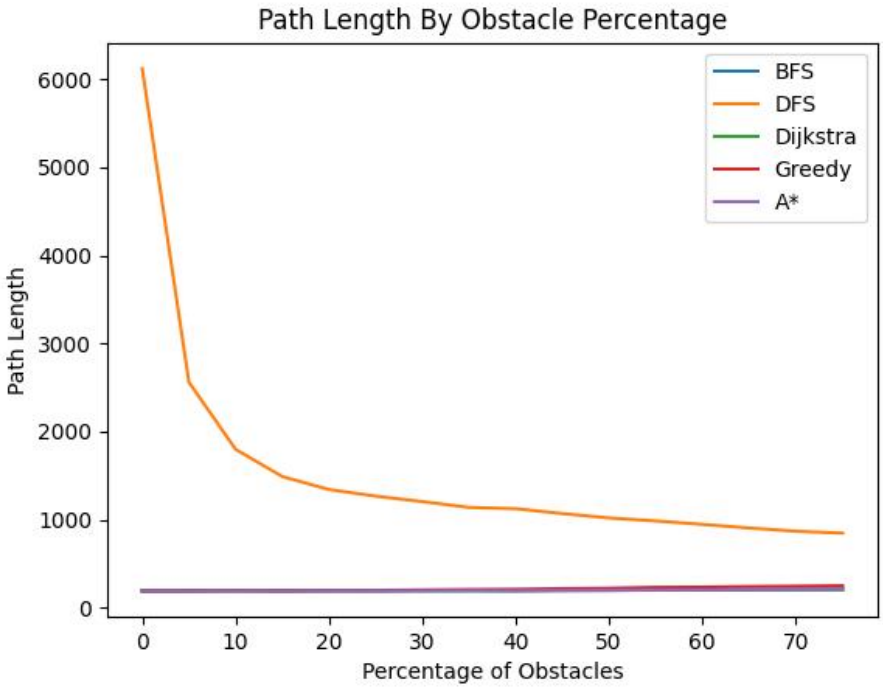


Figure 5: Iterations per planner algorithm

...and since DFS creates such inefficient routes, we look at that last chart again with DFS removed:

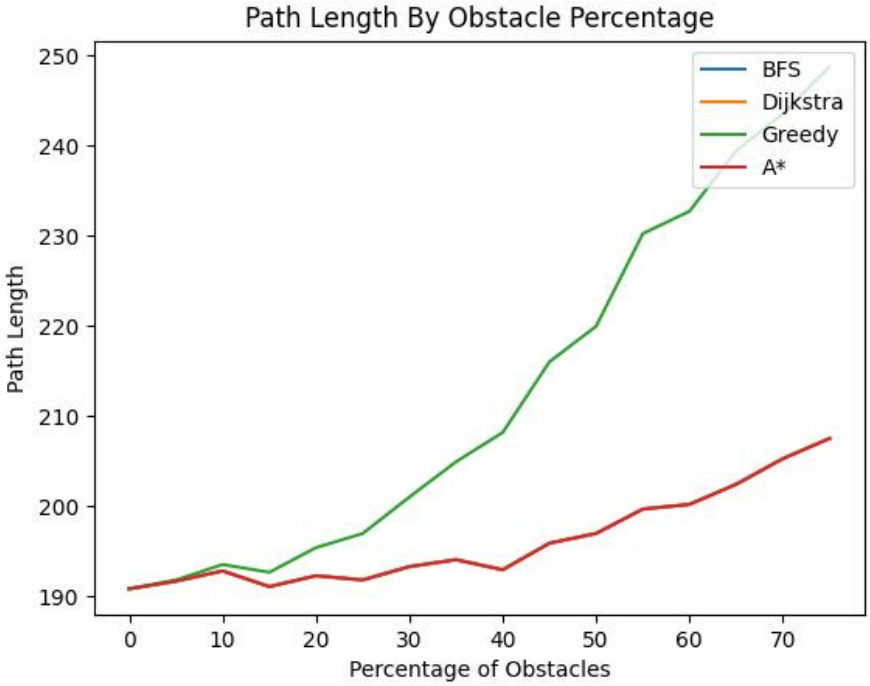


Figure 6: Iterations per planner algorithm, sans DFS

In these figures we see on our overlayed algorithms several algorithms overlap, resulting in some algorithms appearing as if they are missing. This is explained in our below section.

Results and Conclsuions

In this assignment, it appears the BFS and Dijkstras perform the same, but we know their implementations are different. Why is this? Note that our grid represents a discrete space with equal cost between all neighboring cells - which in our algorithms would be displayed as an unweighted graph. For an unweighted graph, BFS and Dijkstra's algorithm perform identically. If the graph was weighted, ie varying costs between neighboring cells driven by some ruleset, Dijkstra's would have less iteratons to find the optimal solution than BFS.

BFS, Dijkstra's, and A* all find the optimal path, though A* is dependent upon an admissible heuristic function. With the simplicity of our example we utilized Euclidean distance to the node for our heuristic.

DFS as an algorithm creates wildly inefficient routes, but can quickly discover if a path exists. However, it does find routes with at a fairly steady rate irregardless of obstacle prolificness. Our greedy algorithm, however, which has a queue that prioritizes entirely off a heuristic of Euclidean distance to the goal, proved to be even quicker a path to the goal. The greedy approach does not gaurantee the optimal route that BFS, Dijkstra's, or A* does, but performs similarly until obstacle percentage raises.

In all, if the graph is unweighted, there is no discernable difference between BFS and Dijkstra's, and a noticeable improvement with A*. If the graph is weighted, A* would still be the preferred choice. If we were aiming to find if any path to the goal exists as quickly as possible, and we are confident in a low probability of obstacles between us and the path, we may find utility in applying the greedy approach for a more rapid check for a possible, albeit non-optimal, path.