Jordan Grotz

2/6/2023

Flatlands Assignment

Methods

General Structure

The Obstacle generator from the first assignment was ported to python for this assignment. The field is stored as a 2D array, with a zero representing a free space, and a 1 representing a filled space. The obstacle generator also has two functions for setting the start and end point. These functions ensure that those spots are not filled, but they do not ensure that there is a viable path between them.

The main file of the project imports the obstacle_field, breadth_first_search, depth_first_search, dijkstras, and random classes. One field is generated as passed into the creation of the search algorithm objects. Each algorithm object then can be run independently with a singular field for comparison. The results are plotted with regular pyplots and the pyplot.imshow() command for the field.

Search Algorithms

Every search class has a general structure where it can be initiated with the field and a color it should display on the visualization. Then each function has a search with will populate internal class features with the results of the class. The search function will return true if the goal was found, and false if the goal was never found. To obtain the searched map and path, each search class has a get_searched_map() function.

For searching, the algorithms only considered the four neighbors that were up, down, left, or right. The corner spots were not considered. Each algorithm also had internal functions to ensure that the neighbors taken were not obstructed by obstacles or out of the bounds of the field. With these base functions, the actual implementation of each as simple as following the provided pseudo code for each algorithm.

Depth First Search

The key feature of depth first search is using a stack to store the next cells to search. This is achieved by using the base list structure in python. The list already has a .pop() command that will return the most recent point added to the list. Once the neighbors are found, they are checked to be valid points and added to the stack only if they haven’t been searched already. Points are added to the stack at the same time they are added to the searched list. This ensures that a point isn’t added to the stack multiple times.

Breadth First Search

The key difference from depth first search to breadth first search is the use of a queue instead of a stack. This is achieved using python’s built in queue class. Otherwise, the implementation is the same as DFS.

Dijkstra's

Dijkstra's algorithm has a largely different implementation than the other search algorithms discussed so far. The key features are a priority queue, a cost map, and a way to store the previously visited point for each current point. The priority queue ensures that you are exploring the path with the least distance to the starting point. This is achieved using python's heapq class. The heapq works similarly to a queue, but a priority must be assigned to the point. This can cause issues in the object if multiple points have the same priority. The avoid this, a priority and item count are assigned to a point. Therefore if two points have the same priority, the point that was added first is popped from the queue.

The cost map is created as a 2D array the same size as the field. Every value is initialized to infinity. Lastly, the previously visited map is created as a 3D array with dimensions: field_size X field_size X 2. This allows a point to be stored at the location of another point. To create the final path the program easily iterates through this previous point map from the goal to the start.

Random

The random search randomly chooses one of the four directions to go until it reaches the end goal. It doesn't track previously visited spaces, so it is possible that it never finds the goal. Therefore it has a check to see if its run too many iterations, and cancels out of the search. The iteration cap was set to be three times the number of spaces in the map.

Results

The following graphs and images show the results of each search algorithm based on different densities of obstacle placement. The start was placed at point [0.0] and the goal was placed at point [127,127]. Figure 1, shows a plot of the number of iterations each search algorithm needed to find the target. This is plotted over obstacle density. For the random search, a limit of three times the size of the board was set for iterations, so the search algorithm didn't run forever. Lastly, the graph goes to zero around 50% density, this is due to there not being a valid path to the goal. This is expected as the obstacles take up most of the space on the board.

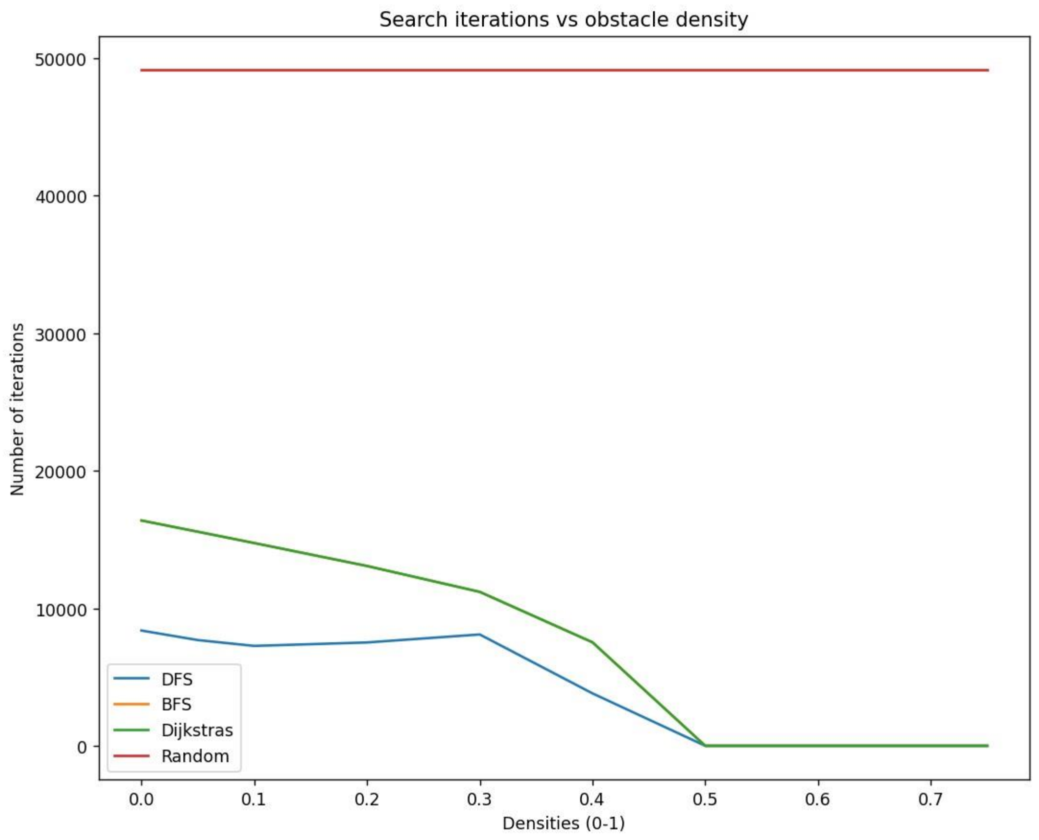


Figure 1: Search iterations vs density for up to 75% density

Figure 2 shows the length of the path for Dijkstra's algorithm for these same plots. Dijkstra's algorithm is the only one of the search algorithms that produces an optimal path. However, as you can see from figure 1, it has the same number of iterations as breadth first search, as it searches nearly the entire map to find the goal. The benefit is that it then produces the shortest path to the goal.

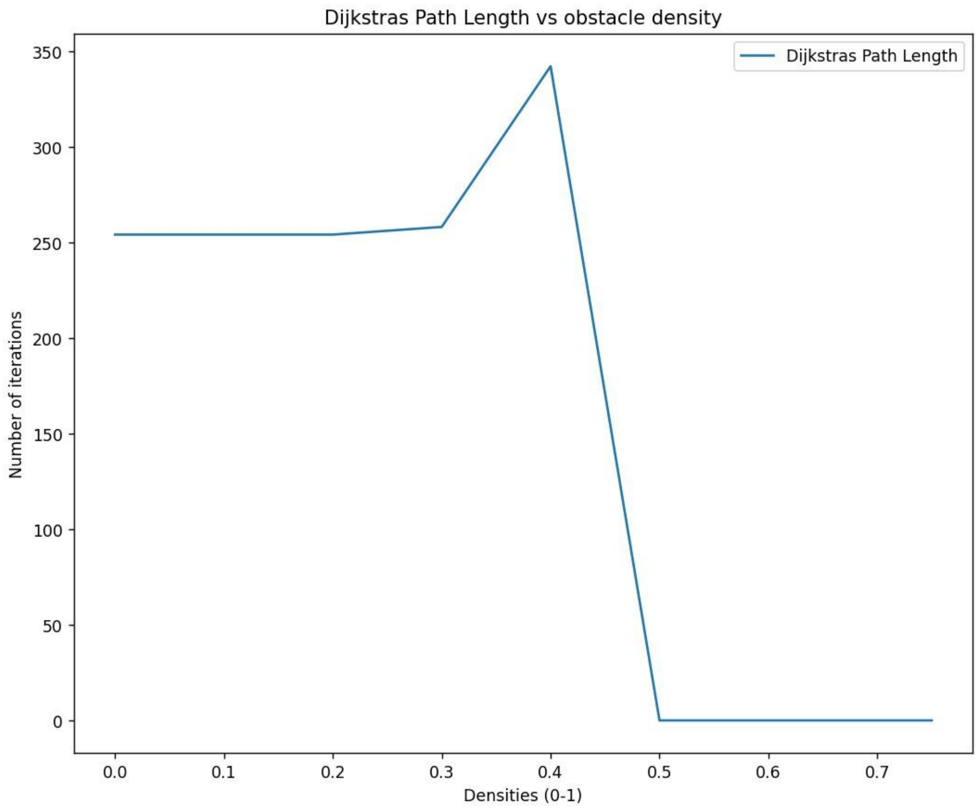


Figure 2: Dijkstra's Path Length vs density

Figures 3-5 showcase the spaces searched by each algorithm overlayed on each other for three different densities. Figure 6 shows the number of iterations for these graphs.

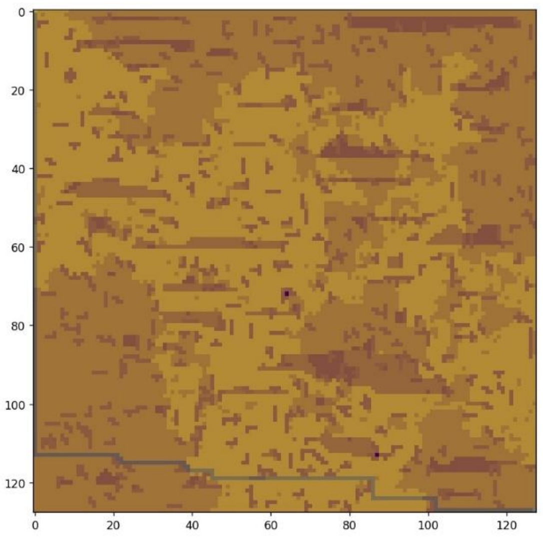


Figure 3: 10% density search

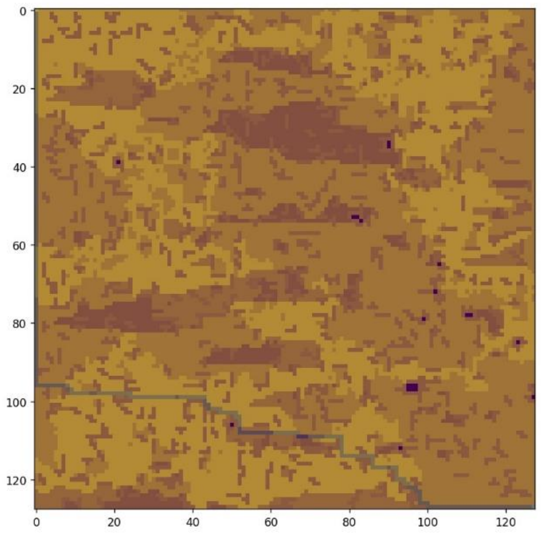


Figure 4: 15% density search

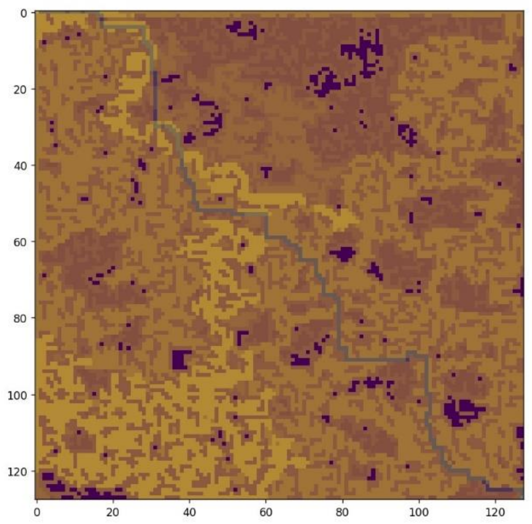


Figure 5: 35% density searched

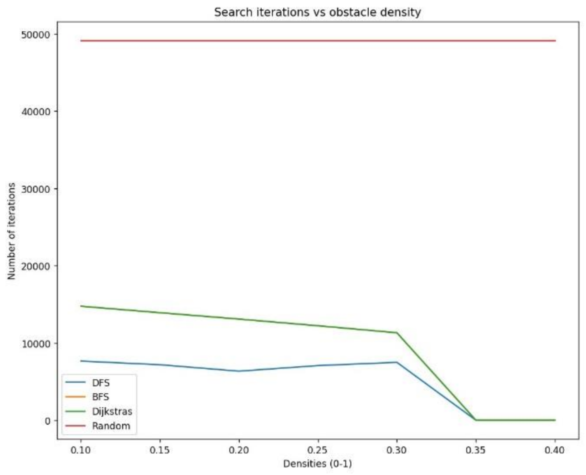


Figure 6: Search iterations for 10-40% density

Figures 7-10 show one map type with each search algorithm shown on a separate graph. It is interesting how the random search tends to go towards the goal, and it has the potential to have less iterations than dfs and bfs. I assume this is some of the reasoning behind the RRT algorithm, which I haven’t learned anything about.

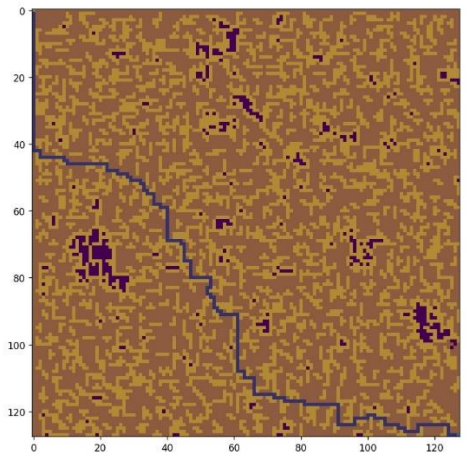


Figure 7: BFS Searched (Lighter colored tiles)

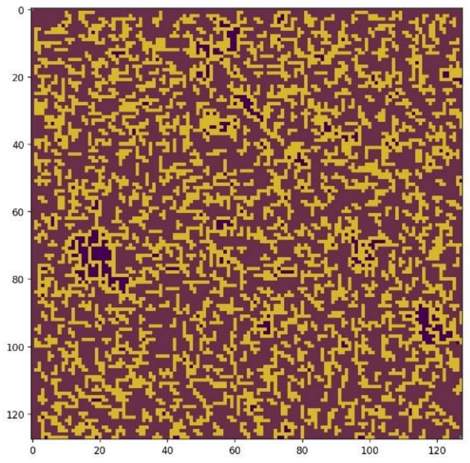
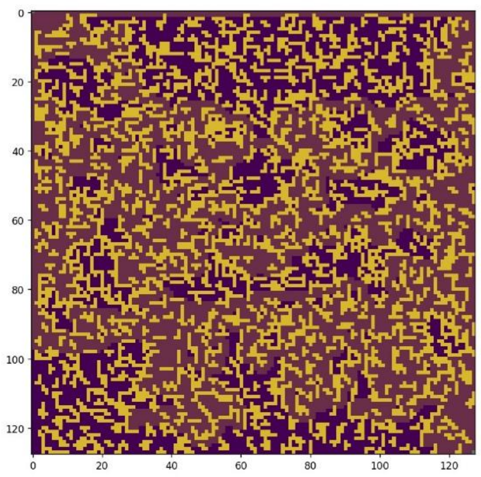


Figure 8: Dijkstra's Searched and Path



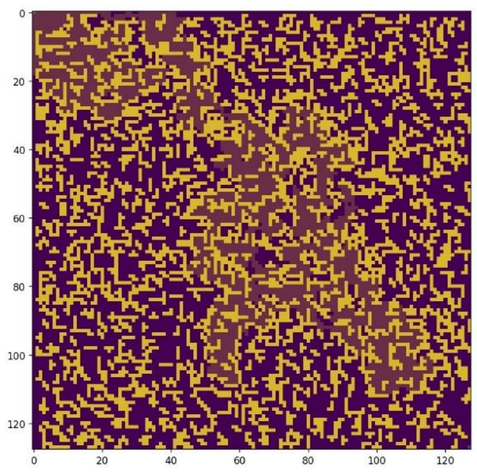


Figure 9: Random Searched (Lighter color tiles) *Figure 10: DFS Searched (Lighter colored tiles)*

Lastly, Figure 11 shows 60% obstacle density, visualizing how the goal is unreachable.

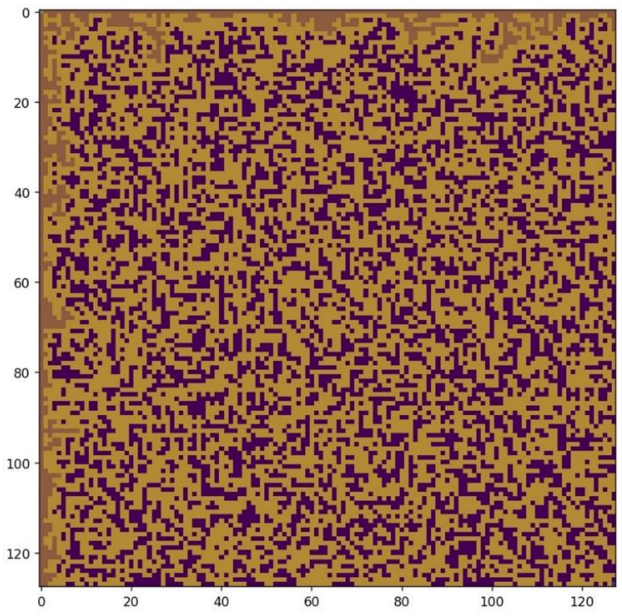


Figure 11: 60% Density Obstacles