RBE 550: Flatlands Assignment

Azzam Shaikh
February 5, 2024
GitHub repo: github.com/azzamshaikh/grid_search_algorithms

I. Introduction

The purpose of this assignment is to implement various grid search algorithms for a point robot in a 2D environment. The objective is to gain experience with discrete planning. For the 2D grid, the robot will start in the northwest corner of the grid and must travel to the southwest corner. There will be stationary obstacles placed randomly at a preset density which the robot must avoid.

A. Implementation Approach

The solution was implemented using Python and was built from scratch. The `matplotlib` library was utilized to create the figures which will contain the 2D grid. The visualization of the traversal was completed by using the `plot()` function to plot each point on the grid.

II. Obstacle Field

The obstacle field should place tetromino shapes randomly in a 128 x 128 two dimensional grid at a user defined density.

The `ObstacleField` class is the main class that creates the varying obstacle field. There are two subclasses, `ObstacleGenerator` and `GridObstacle`, that create random obstacles and store the location of all objects, respectively. Figure 1 shows varying density obstacle fields.

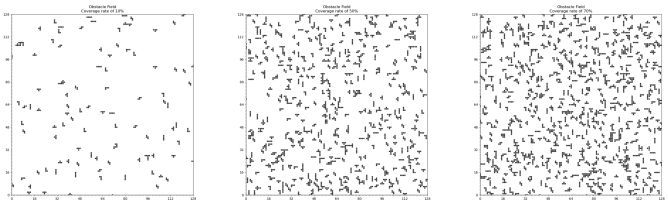

Fig. 1. Varying obstacle field configurations at 10%, 50%, and 70%.

III. Grid Search Algorithms

As mentioned in the introduction, the objective of this assignment is to gain experience with discrete planning. Thus, in order to achieve this, four search algorithms will be developed. Three of them are forward search approaches: a depth first search, a breadth first search, and Dijkstras algorithm. The fourth is a random planner, which will be discussed in more detail in a later section.

The main script that runs the grid search algorithms is the `RunGridSearchAlgorithms.py` file. This script calls the `ObstacleField` class in addition to the other classes required to run the four planners. The script contains a `main` function that executes the script.

For each of the mentioned planners, there is a separate script created for each algorithm. All the scripts contain two specific classes that highlight the architecture of the program. These are the `GridSearch` class and the `Node` class.

The `GridSearch` class is a general agent used to solve a grid search problem. It contains an initialization function with the starting and goal positions as well as an obstacle field parameter. Once initialized, the agent is able to determine its actions based on its current state. Based on the actions and the state, a resulting sequence of actions can be determined. In addition, the agent is able to determine if it has reached its goal state.

The `Node` class is a general object to store the information about a specific point in the 2D grid. The initialization of any node stores its current state (i.e. x and y coordinates on the grid), its parent node, its certain action, its path cost, and its depth. Each node also contains various functions. For example, it can expand itself to find its neighboring node with the help of the `GridSearch` agent. In addition, it is also able to find a solution from the goal node to the initial node.

A. Depth First Search

Based on the *Planning Algorithms* book by Steven LaValle, depth first search is a search approach which utilizes a last in, first out stack to search the grid [1]. This approach "dives quickly into the graph" [1]. This approach is accomplished by using a stack data structure to append the nodes. As the algorithm traverses through the graph, when a child node is discovered, that node gets added to the front of the stack. This continues until there are no other child nodes to be added to the front of the stack. Thus, resulting in the search of the deepest node first. The stack structure utilized is the `deque` class from the `collections` module.

The `GridSearchDFS.py` file manages the execution of this planner. This file contains the `GridSearch` agent class and `Node` class as well as various supporting functions such as checking validity of cells (i.e. is the new state/location out of bounds or in collision with an object), initialization of plots, plotting of obstacles, and the main `GridSearchDFS` class that runs the algorithm.

Figures 2, 3, and 4 show the output from the planning algorithm in varying obstacle field densities of 10%, 50%, and 70%, respectively. The color coding of the plot is such that, yellow refers to nodes/neighbors added to the frontier, green refers to visited nodes, and magenta refers to the solution path.

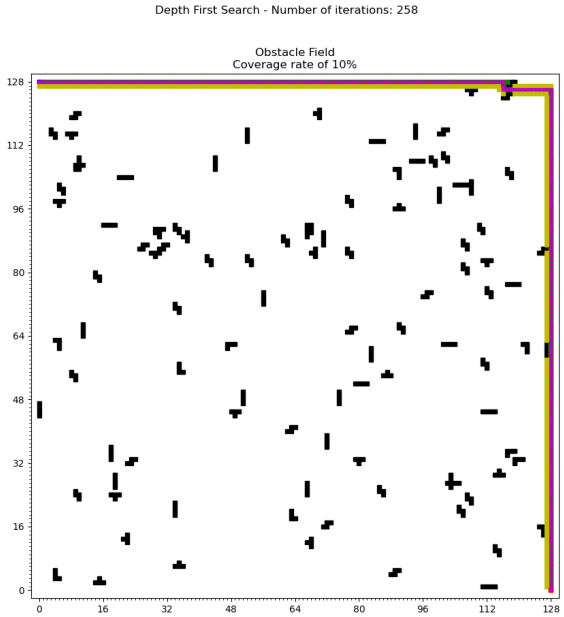

Fig. 2. Depth First Search algorithm searching through obstacle field with a coverage rate of 10%

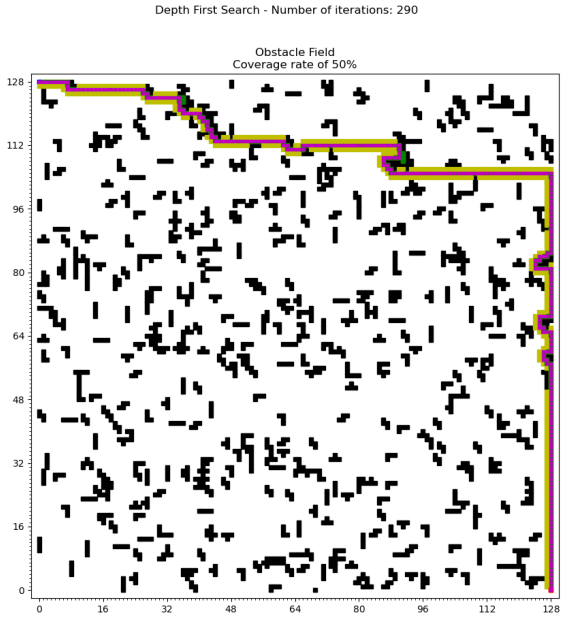

Fig. 3. Depth First Search algorithm searching through obstacle field with a coverage rate of 50%.

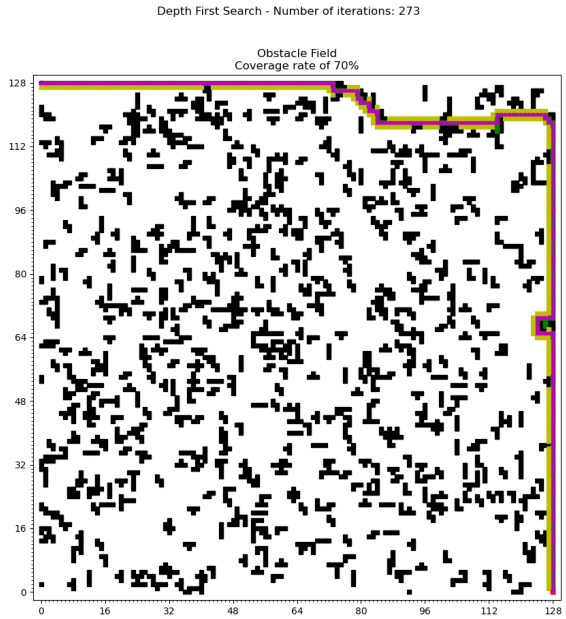

Fig. 4. Depth First Search algorithm searching through obstacle field with a coverage rate of 70%.

It should be noted that, based on how the algorithm works, the output shown is a unique scenario where the right most node is checked first, followed by the down most node. This results in the fastest search. However, if the algorithm followed any other order, such as right most followed by left most, the entire graph would be traversed before finding the solution. For example, if the goal is placed in the center of the grid, the search would take longer to solve. Figure 5 shows this example in practice. It can be seen that the number of iterations needed to reach the goal increased significantly. In addition, the solution is very difficult to follow.

B. Breadth First Search

Based on the *Planning Algorithms* book by Steven LaValle, breadth first search is a search approach which utilizes a first in, first out queue to search the grid [1]. This approach "causes the search frontier to grow uniformly" [1]. This approach is accomplished by using a queue data structure to append the nodes. As the algorithm traverses through the graph, when a child node is discovered, that node gets added to the back of the queue. Thus, those nodes will be discovered in order as they are entered into the queue and resulting in the a more uniform search. The queue structure utilized is the `deque` class from the `collections` module. The main difference between how this is implemented versus the `deque` implemented for the depth first search is the usage of the `appendleft()` function. This keeps the first in, first out order that is required.

The `GridSearchBFS.py` file manages the execution of this planner. This file contains the `GridSearch` agent class and `Node` class as well as various supporting functions such as

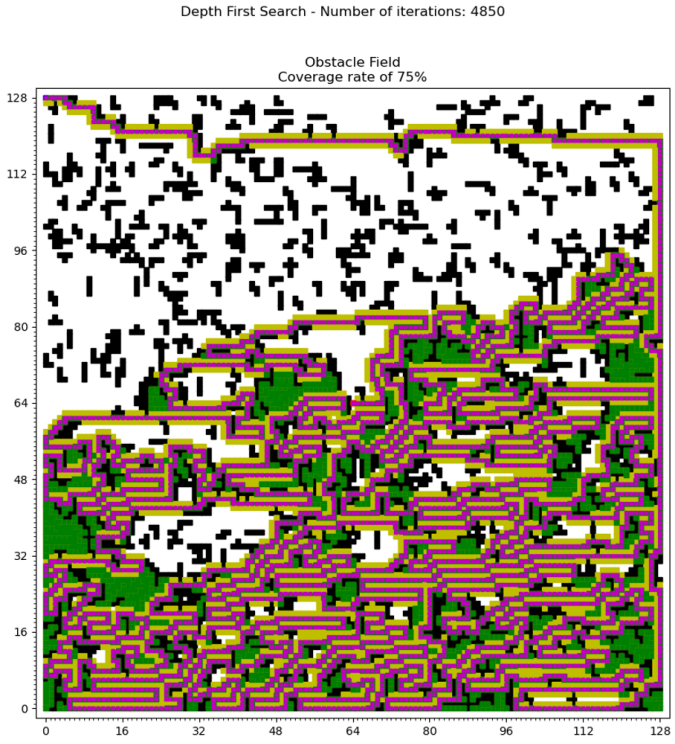

Fig. 5. Depth First Search algorithm searching through obstacle field with a coverage rate of 70%.

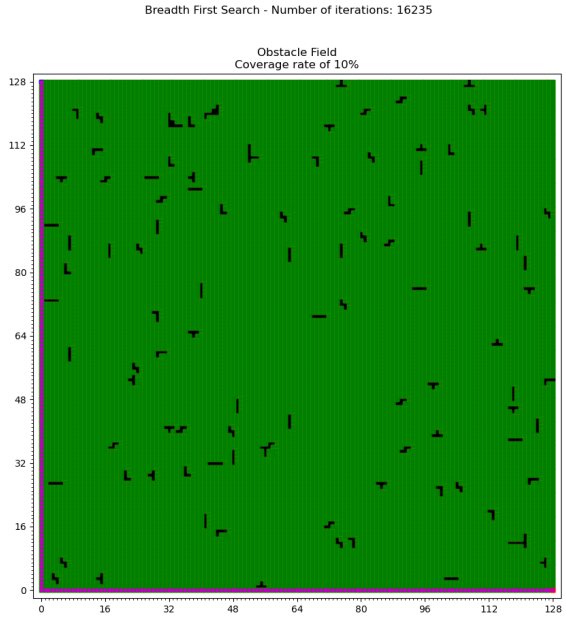

Fig. 6. Depth First Search algorithm searching through obstacle field with a coverage rate of 10%

checking validity of cells (i.e. is the new state/location out of bounds or in collision with an object), initialization of plots, plotting of obstacles, and the main `GridSearchBFS` class that runs the algorithm.

Figures 6, 7, and 8 show the output from the planning algorithm in varying obstacle field densities of 10%, 50%, and 70%, respectively. The color coding of the plot is such that, yellow refers to nodes/neighbors added to the frontier, green refers to visited nodes, and magenta refers to the solution path.

Compared to the depth first search, it can be seen that this approach visits more nodes in the graph in a systematic approach.

C. Dijkstra's

Based on the *Planning Algorithms* book by Steven LaValle, Dijkstra's algorithm is a search approach which utilizes a priority queue to search the grid [1]. The previous two methods used data structures that organized nodes based on the order they came into the structure. A priority queue organizes the structure of nodes based on the cost of a certain action. This model helps find the shortest path/solutions based on that specific cost criteria. As the algorithm traverses through the graph, when a child node is discovered, that node gets added accordingly to the queue based on the cost. Thus, those nodes with the lowest cost will be visited first. The queue structure utilized is the `PriorityQueue` class from the `queue` module.

The `GridSearchDijkstra.py` file manages the execution of this planner. This file contains the `GridSearch` agent class

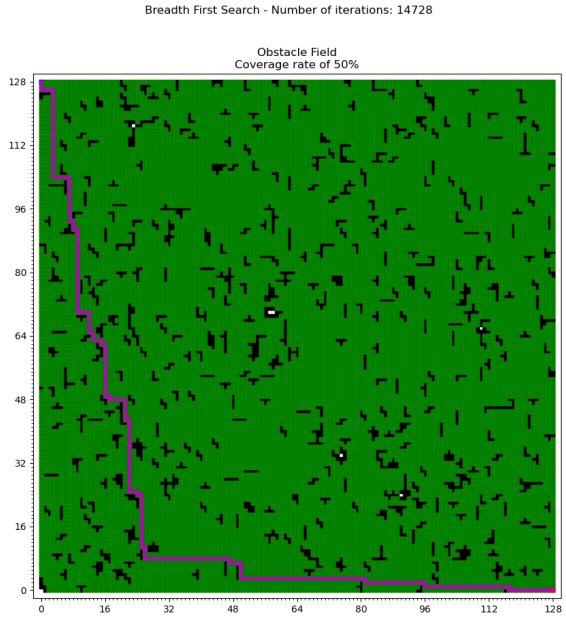

Fig. 7. Depth First Search algorithm searching through obstacle field with a coverage rate of 50%.

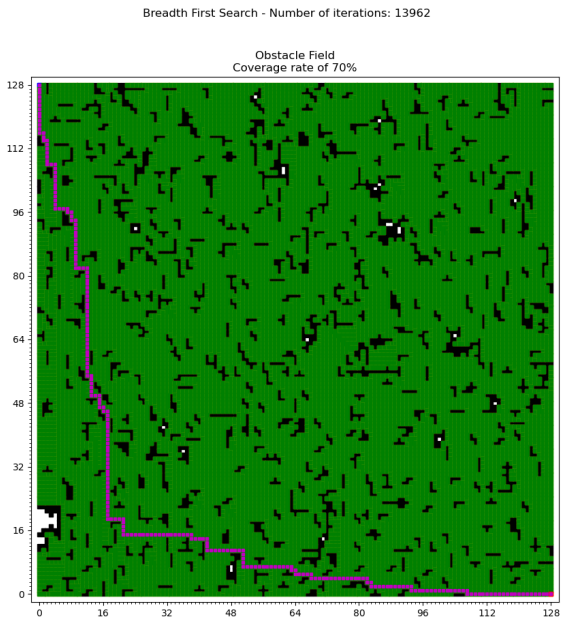

Fig. 8. Depth First Search algorithm searching through obstacle field with a coverage rate of 70%.

and `Node` class as well as various supporting functions such as checking validity of cells (i.e. is the new state/location out of bounds or in collision with an object), initialization of plots, plotting of obstacles, and the main `GridSearchDijkstra` class that runs the algorithm.

For Dijkstra's algorithm, since there is a cost based approach to organizing the queue, if there is a uniform cost for each action, then when a neighbor gets added to the queue, the item that gets added first will get removed. Thus, this would act like a breadth first search. Figure 9 shows a comparison between a breadth first search and Dijkstra's algorithm with uniform step costs. It can be observed that the solution paths are identical.

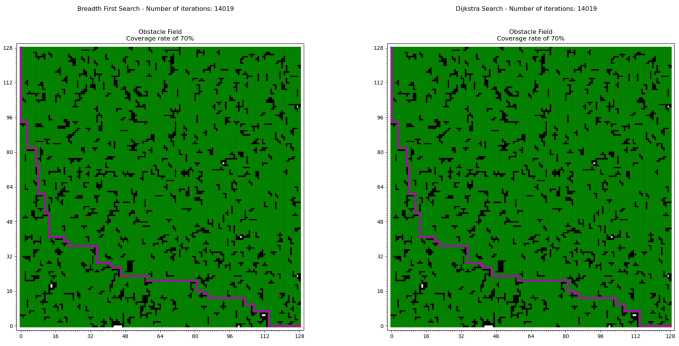

Fig. 9. Comparing results between breadth first search and Dijkstra's with uniform step costs.

If, for example, the step costs are adjusted such that the left and right actions are cheaper versus up and down actions, the result shown in Figure 10 occurs. It can be observed that the solution paths are now different.

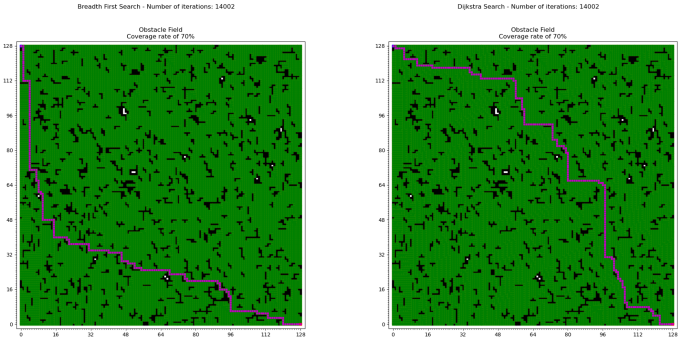

Fig. 10. Comparing results between breadth first search and Dijkstra's with cheaper left and right actions.

Another way to test the application of the priority queue is to implement diagonal steps. Figure 11 shows a comparison of Dijkstra's with and without diagonal steps. With the diagonal steps, there is a smaller cumulative cost generated with the diagonal approach as compared to the standard movement actions. Thus, this results in the shortest path between the start and the goal position.

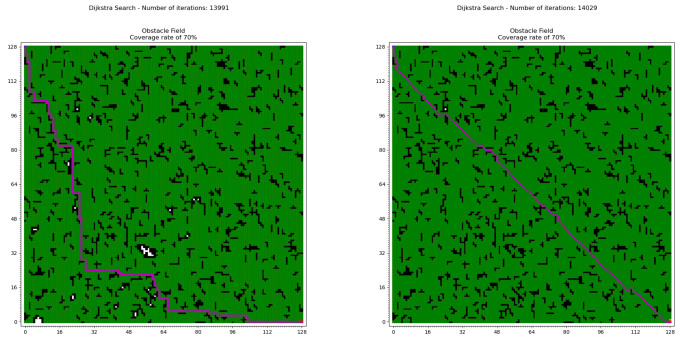

Fig. 11. Comparing results between Dijkstra's with standard movements versus diagonal movements.

D. Random Planner

An additional, random planner was required to be implemented as well. The purpose of this planner was for the planner to randomly move to a neighboring cell every iteration. In this implementation, instead of using a queue/stack/priority queue to decide which neighbor gets searched first, the planner will randomly select a node from the frontier. The random selection is implemented with the usage of the `choice()` function from the `random` class. The `frontier` object which stores unvisited nodes gets passed as an argument to the `choice()` function. The output of the function is a randomly selected node.

The `GridSearchRandom.py` file manages the execution of this planner. This file contains the `GridSearch` agent class and `Node` class as well as various supporting functions such as checking validity of cells (i.e. is the new state/location out

of bounds or in collision with an object), initialization of plots, plotting of obstacles, and the main `GridSearchRandom` class that runs the algorithm.

Figure 12 shows the output of the random planner with no max iteration. This was executed out of curiosity to see how many iterations would be required to find the goal cell. Interestingly, the search took about three hours to complete a total of 700,000 iterations and used 8 GB of RAM to solve the solution. Due to the length required to run this planner, a max search limit of 16,000 iterations was set. Due to this, the planner will stop searching before reaching the goal at the other side of the grid.

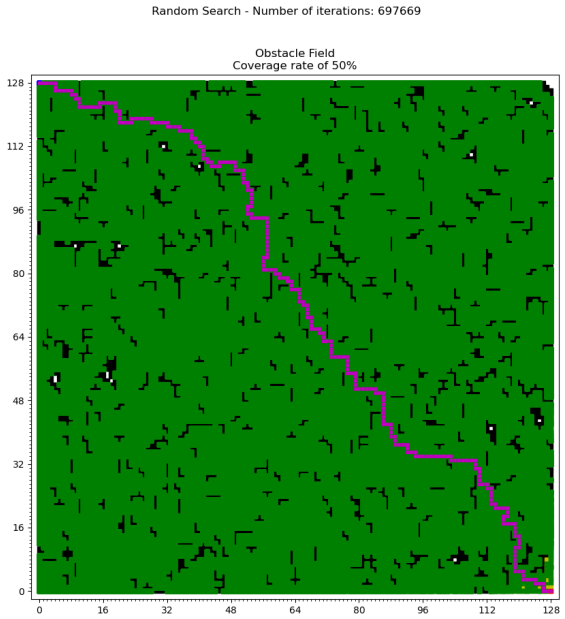

Fig. 12. Random planner searching with no maximum iteration through obstacle field with a coverage rate of 50%.

IV. Results

With the implementation of the four planners, the performance of the different planners can be compared side by side. For this plot, as shown in Figure 13, the number of iterations to reach the goal state are shown at an obstacle density of 50%. The variations in iterations can be seen with the random planner taking the longest time while the depth first search taking the shortest. As mentioned in Depth First Search, the depth first search solution happens to find the shortest possible path by moving right first and then down. In another scenario, the algorithm could move left and right downward until reaching the goal node. Because of this case, the result from the depth first search isn't a fair comparison. In theory, it could take just as many iterations as the breadth first search or Dijkstra's.

Three additional figures have been added to compare the three forward search approaches with the same obstacle set at

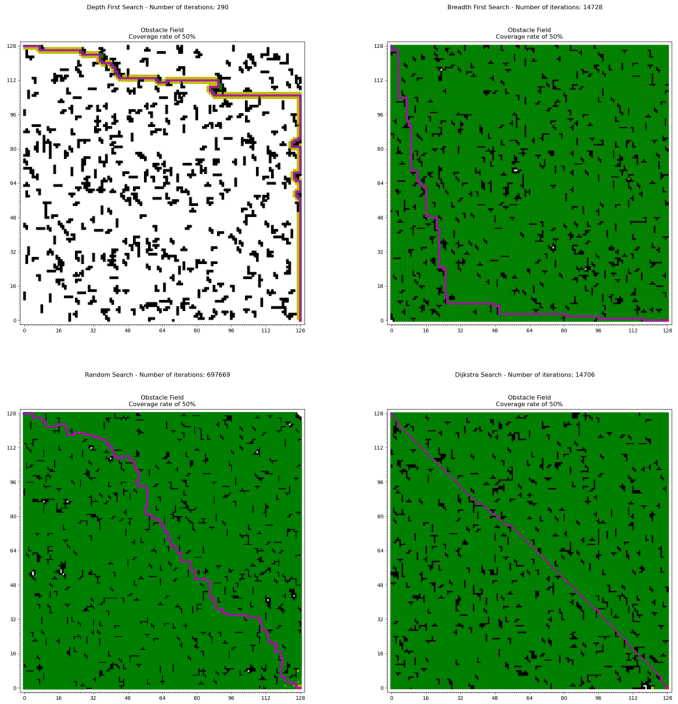

Fig. 13. Performance comparison of the four planners with number of iterations required to reach the goal at a obstacle field density of 50%.

varying densities. The objective is to provide a performance comparison between the different approaches. The random planner will not be utilized in this comparison due to the time required to run the solution. A total of three runs will be executed for the three planners. Each run will constitute a different obstacle field density. For the Dijkstra algorithm, two variations will be ran, one with standard movements and the other with diagonal movements.

Figure 14 shows a comparison of the three algorithms searching through a obstacle field with a coverage rate of 25%. Breadth first search and Dijkstra's all require the same number of iterations to find the goal. In addition, the standard actions for Dijkstra's and breadth first search both find the same solution path, as mentioned previously. Once Dijkstra's has diagonal movement implemented, there is a shorter path found. Depth first search, as mentioned earlier, happens to find the shortest possible path by moving right first and then down. Thus, the result from the depth first search isn't a fair comparison. Figure 15 shows a plot of the number of iterations required for each algorithm versus time. As shown in the plot, breadth first search and Dijkstra's algorithm take virtually the same number of iterations. For depth first search, as previously mentioned, the fastest path is found very quickly. This trend is consistent with increasing obstacle field densities as well. Figure 16 and 17 show the forward search algorithms traversing through an obstacle field density of 50% and the performance plot, respectively. Figure 18 and 19 show the forward search algorithms traversing through obstacle field densities of 75% and the performance plot, respectively.

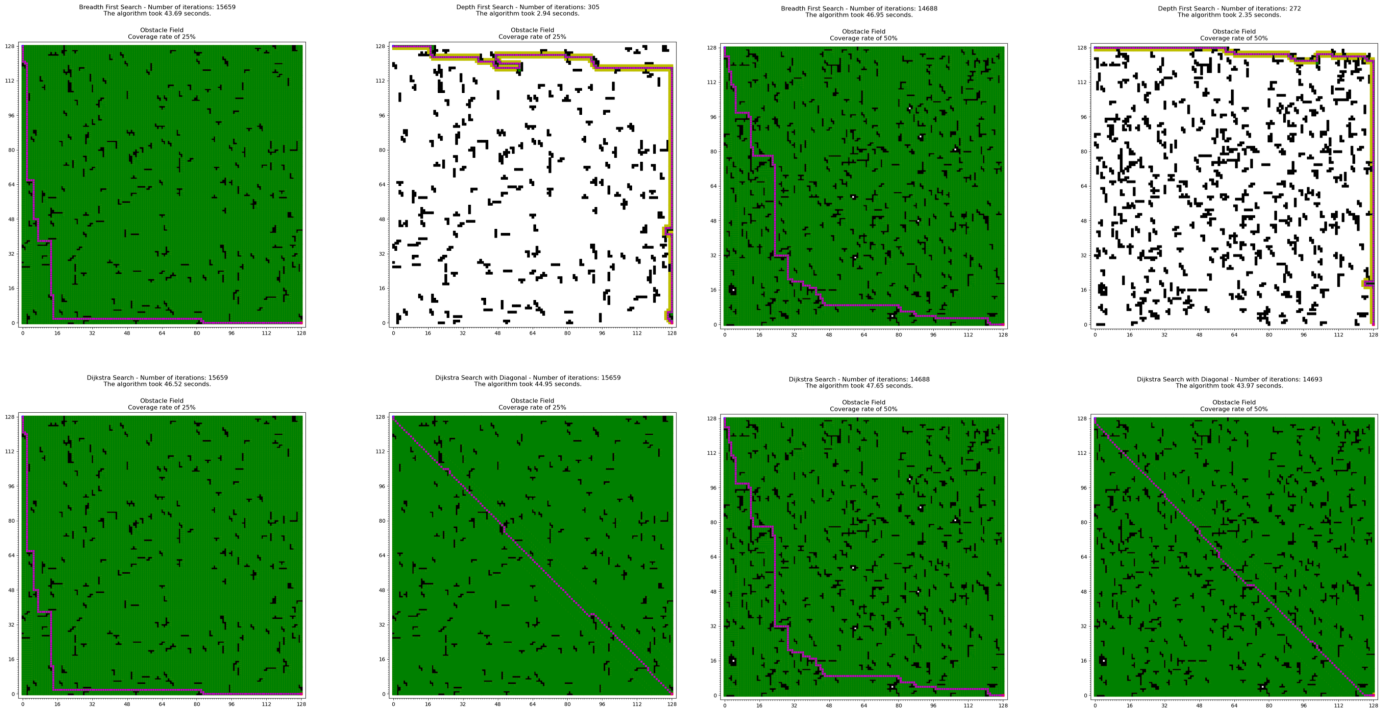

Fig. 14. Performance comparison of the three forward search algorithms with an obstacle field with a coverage rate of 25%.

Fig. 16. Performance comparison of the three forward search algorithms with an obstacle field with a coverage rate of 50%.

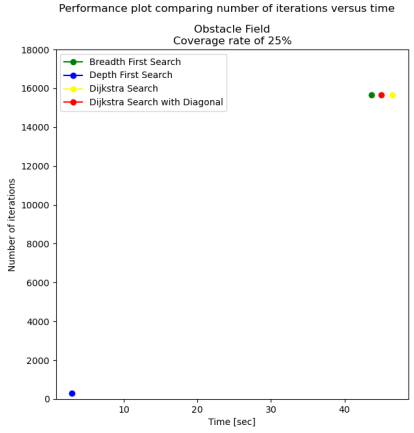

Fig. 15. Iteration versus time performance comparison of the three forward search algorithms with an obstacle field with a coverage rate of 25%.

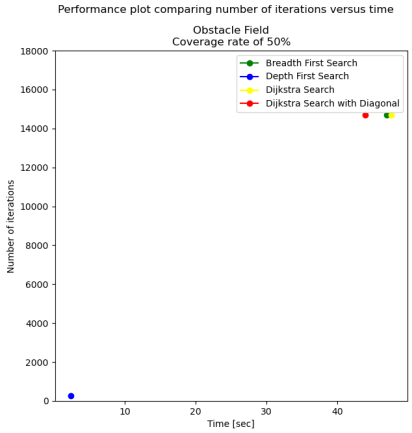

Fig. 17. Iteration versus time performance comparison of the three forward search algorithms with an obstacle field with a coverage rate of 50%.

Figure 20 shows a performance plot comparing the number of iterations for the different planners versus the obstacle density. It can be seen that, as the density increases, the number of iterations decrease. This is expected since there are greater number of obstacles in the field, and thus, there are in turn fewer valid nodes for the algorithm to visit.

For the purposes of comparison, the search algorithms were tasked to find the goal when it is set to the center of the grid. Figure 21 shows the results of the planners paths when the goal is set to the center. Figure 22 shows the iterations vs time performance plot comparison. Based on these results, it can be seen that, the fastest and shortest search was indeed completed the Dijkstra's search with diagonal steps. The breadth first/regular Dijkstra's searches both found solutions relatively fast. While they had the highest iterations, the overall solution is found faster. For the depth first search, even though it had lower iterations than breadth first/regular Dijkstra, the time took to find the goal was significantly high.

To further support the previous example, if the goal is placed very close to the start, it would be expected for a uniform search approach to find the solution fairly quickly. However, for a depth first search, this would not be expected as it will

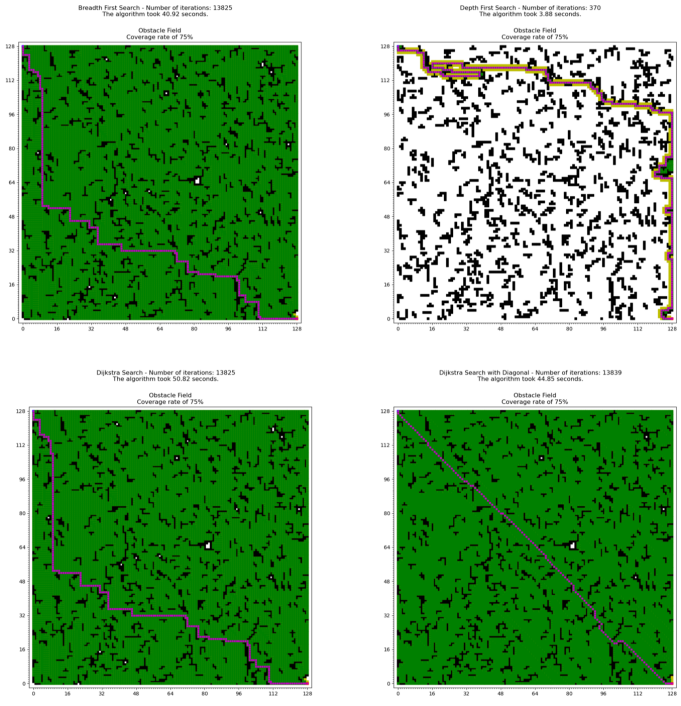

Fig. 18. Performance comparison of the three forward search algorithms with an obstacle field with a coverage rate of 75%.

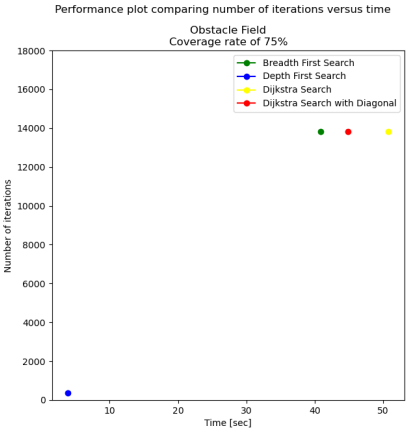

Fig. 19. Iteration versus time performance comparison of the three forward search algorithms with an obstacle field with a coverage rate of 75%.

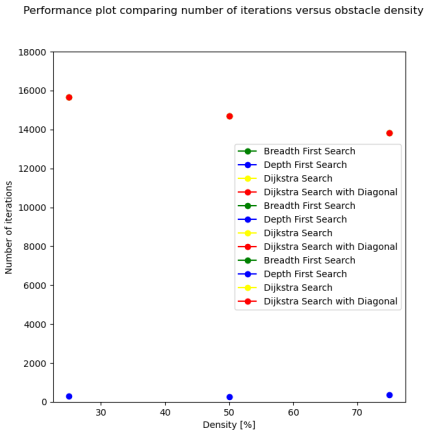

Fig. 20. Iteration versus density performance comparison.

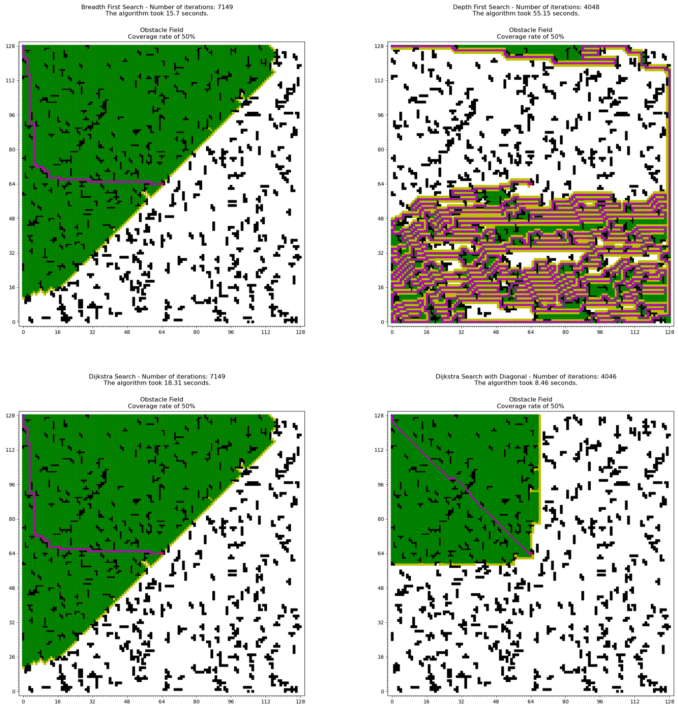

Fig. 21. Performance comparison with a center goal of the three forward search algorithms with an obstacle field with a coverage rate of 50%.

search the deepest node first before attempting another route. Figure 23 visualizes this example and Figure 24 plots the performance of the algorithms. It can be seen that the breadth first search and both Dijkstra search implementations find the goal in less than 2 seconds because of how close the goal is. However, for the depth first search, it takes 150 seconds to find the solution as the search traverses the I, III, and IV quadrants of the grid first before searching near itself.

V. Discussion

It can be noted that these planners, while efficient, generally visit a large portion of the graph in order to find the correct solution. If the goal state is known by the planner, the search can be optimized by making informed decisions on where to proceed next. This informed nature of the search can be implemented with a heuristic, in addition to the cost. In future work, the implementation of informed search approaches, such as A* search, will provide an interesting comparison against these forward searches.

References

[1] S. M. LaValle, Planning Algorithms. Cambridge: Cambridge University Press, 2006.

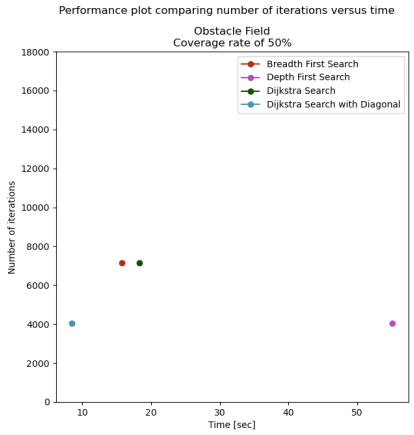

Fig. 22. Iteration versus time performance comparison with a center goal of the three forward search algorithms with an obstacle field with a coverage rate of 50%.

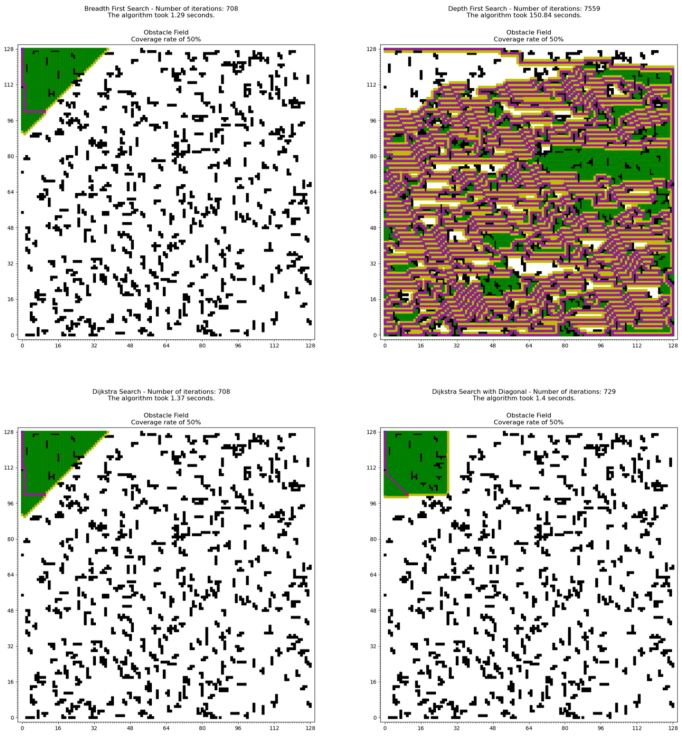

Fig. 23. Performance comparison with a close goal of the three forward search algorithms with an obstacle field with a coverage rate of 50%.

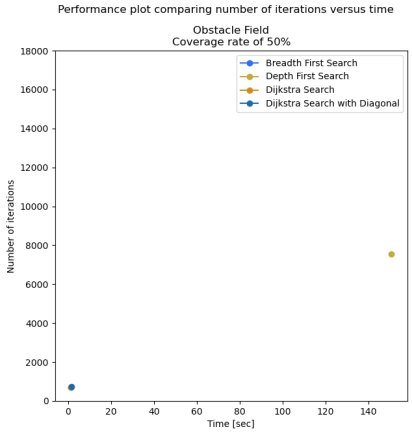

Fig. 24. Iteration versus time performance comparison with a close goal of the three forward search algorithms with an obstacle field with a coverage rate of 50%.