RBE 550: Assignment 2 - Flatland

Mandeep Singh
M.S. Robotics Engineering
Worcester Polytechnic Institute (WPI)
Email: msingh2@wpi.edu

***Abstract*—In this assignment, basic motion planning search algorithms have been deployed for a point robot in a grid world having randomly placed obstacles. The algorithms discussed are Breadth-First search, Depth-First search, Dijkstra's shortest path algorithm and a Random planner. A detailed comparison has been done between all algorithms in terms of shortest path and time/exploration needed to reach that shortest path.**

***Index Terms*—Motion planning, Search algorithms.**

I. Breadth First Search

A. Introduction

Breadth first search is an algorithm for searching a tree/-graph data structure for a node. This search starts at the root of the tree and explores all nodes present at a given depth and then moves on to the nodes at the next depth level. Usually, a queue data structure is used to keep track of all child nodes which are found but not yet explored. BFS generally explores all possible nodes in the path before reaching the goal position which sometimes is very time taking.

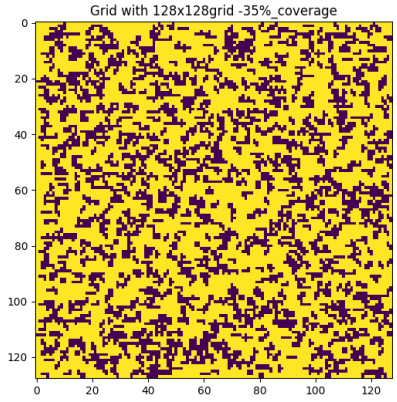


Fig. 1: Grid with 35% obstacle density

B. Application in Grid World

In our 2D grid world, it is assumed that if the point is at a current position of (x,y) then its child nodes will be its four immediate neighbours (right, left, top and bottom). We have not assumed diagonal nodes in our case for simplicity. Now, in such a case BFS will first explore all these four neighbours to the point robot (this is 1st level) one by one and then again further explore neighbours of each of the above found neighbours one by one (this is 2nd level). This process is repeated again and again until the desired goal is reached.

C. Implementation in code

We start with a given start position (0,0) and goal position for the point robot. Then possible child nodes of the current position are explored. A queue is maintained for all the explored child nodes. Once all neighbour child are stored in queue, we pop the first element of the queue and mark it as visited and store it in another list of visited nodes. Again, child nodes of this popped first element are found and stored in queue. This process of taking the first element from the queue and checking its neighbours is repeated again and again until we reach our desired goal position. Also, when we are exploring the child nodes it is checked that whether this node is already visited or not and it is already in queue or not, if visited or in queue it is not added in queue again.

D. Results

Results of BFS implementation can be seen in Fig 2. BFS tries to find the optimal path between start and goal node.

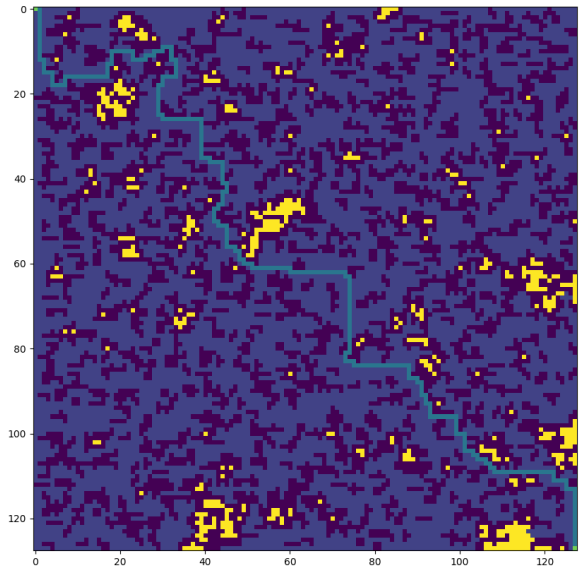


Fig. 2: BFS - 128x128 grid - 35% coverage

II. Depth First Search

A. Introduction

Depth first search is an algorithm for traversing a tree/graph data structure for a node. This search starts at the root of the tree and explores all nodes in a particular branch first and then start backtracking. Usually, a stack data structure is used to keep track of all child nodes which are found but not yet explored. DFS can be helpful in case goal position is in the earlier branches of the tree/graph, but the goal position is in later branches but at less depth then DFS algorithm will not be optimum to use.

B. Application in Grid World

In our 2D grid world, it is assumed that if the point is at a current position of (x,y) then its child nodes will be its four immediate neighbours (right, left, top and bottom). We have not assumed diagonal nodes in our case for simplicity. Now, in such a case DFS will first explore only one of these child nodes (in our case last one) and then find neighbours of this node again. Again it will select the last node of these child nodes and thus this process will repeat again and again till the desired goal ins not reached.

C. Implementation in code

We start with a given start position (0,0) and goal position for the point robot. Then possible child nodes of the current position are explored. A stack is maintained for all the explored child nodes. Once all neighbour child are stored in stack, we pop the last node which was pushed in the stack and mark it as visited. Again, child nodes of this popped last element are found and stored in stack again. This process of taking the last element from the stack and checking its neighbours is repeated again and again until we reach our desired goal position. Also, when we are exploring the child nodes it is checked that whether this node is already visited or not and it is already in stack or not, if visited or in stack it is not added in stack again.

D. Results

Results of DFS implementation can be seen in Fig 3. Path found by DFS is not optimal.

III. Dijkstra's Shortest Path Algorithm

A. Introduction

As the name suggest Dijkstra's algorithm is used to find the shortest path between the source node to all other nodes in a graph. The main principle of this algorithm is that it keeps track of the distance of all the nodes from the source node and when the goal node is reached it returns the shortest path between the source node and the goal node on the basis of the least minimum distance out of all nodes.

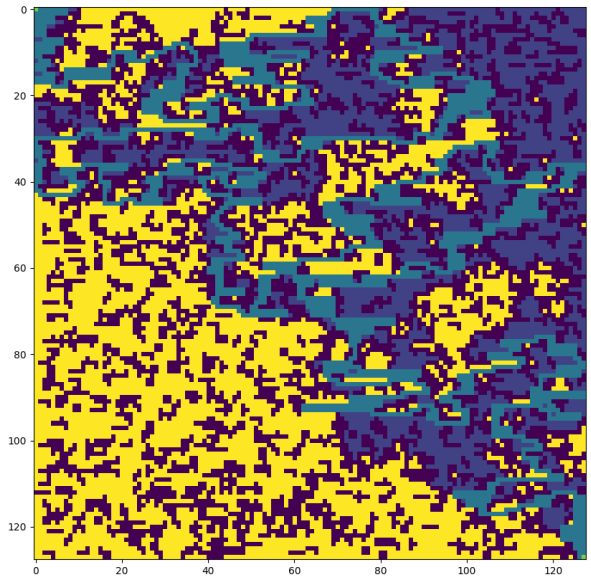


Fig. 3: DFS - 128x128 grid - 35% coverage

B. Application in Grid World

In our 2D grid world, it is assumed that if the point is at a current position of (x,y) then its child nodes will be its four immediate neighbours (right, left, top and bottom). We have not assumed diagonal nodes in our case for simplicity. Now, in such a case Dijkstra's algorithm will first explore all these four neighbours to the point robot (this is 1st level) one by one, calculate the distance (Manhattan distance in our case) of each node from the source node. Then it selects the node having least distance from the source node and find its child nodes (neighbours) and add them in the list of all explored nodes. Now, again Dijkstra's algorithm checks the minimum distance node out of all these nodes and repeats this process until the goal node is reached.

C. Implementation in code

We start with a given start position (0,0) and goal position for the point robot. Then possible child nodes of the current position are explored. A heap queue data structure is maintained for all the explored child nodes. In heap queue our prioritised nodes (nodes with least distance) are sorted in increasing order of distance. Once all neighbour child are stored in heap queue, we pop the first element of the heap queue and mark it as visited and store it in another list of visited nodes. Again, child nodes of this popped first element are found and stored in heap queue. This process of taking the first element from the queue and checking its neighbours is repeated again and again until we reach our desired goal position. Also, when we are exploring the child nodes it is checked that whether this node is already visited or not and it is already in heap queue or not, if visited or in heap queue it is not added in queue again.

D. Results

Results of Dijkstra implementation can be seen in Fig 4. Dijkstra tries to find the optimal path between start and goal

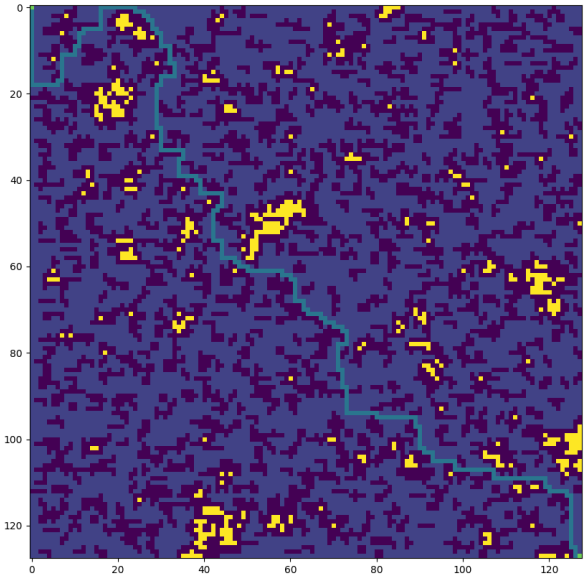


Fig. 4: Dijkstra - 128x128 grid - 35% coverage

IV. Random Planner

A. Introduction

As the name suggest a Random planner traverses randomly in our grid world. It doesn't follow any said rules or algorithm. It finds its nearest neighbour nodes and randomly selects one of those nodes to explore the graph further.A random planner will not give the best results always but sometimes in small grids a random planner may sometimes give the result by exploring minimum nodes.

B. Application in Grid World

In our 2D grid world, it is assumed that if the point is at a current position of (x,y) then its child nodes will be its four immediate neighbours (right, left, top and bottom). We have not assumed diagonal nodes in our case for simplicity. Now, in such a case a Random planner will first explore all these four neighbours to the point robot one by one, select any node randomly out of these nodes and then again further explore neighbours of the selected node. This process is repeated again and again until the desired goal is reached.

C. Implementation in code

We start with a given start position (0,0) and goal position for the point robot. Then possible child nodes of the current position are explored. A queue is maintained for all the explored child nodes. Once all neighbour child are stored in a list, we pop any random node from this list of nodes mark it as visited and store it in another list of visited nodes. Again, child nodes of this randomly popped element are found and stored in list. This process of taking the random node from the list and checking its neighbours is repeated again and again until we reach our desired goal position. Also, when we are exploring the child nodes it is checked that whether this node is already visited or not and it is already in list or not, if visited or in list it is not added in the list again.

D. Results

Results of Random planner implementation can be seen in Fig 5. Path found by our random planner is not optima lwhich is expected because of the random nodes chosen.

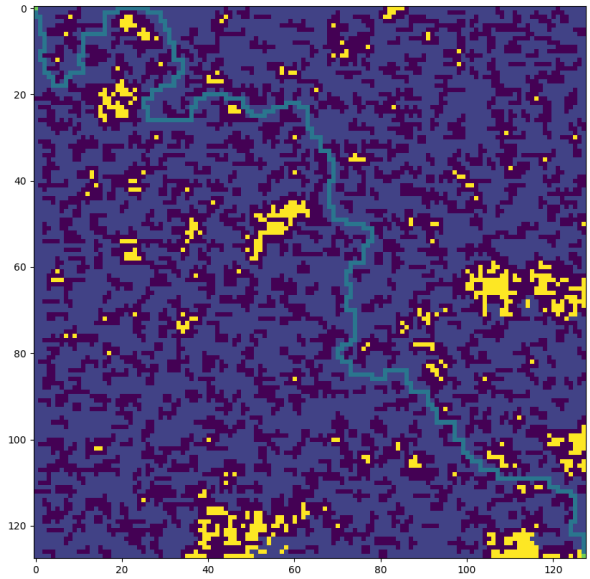


Fig. 5: Random planner - 128x128grid - 35% coverage

V. Comparison

A detailed comparison on the basis of number of iterations done to reach the goal position for all the search algorithms with respect to obstacle density (10%, 20%, 30% and 35%) on 50x50 and 128x128 grid can be seen in Fig. 6 and 7. **Obstacle densities greater than 35 are not implemented because there was no path being available from start to end node.** Also, a plot of path length found for various algorithms versus obstacle densities has been plotted in Fig 8.

It can be seen that apart from DFS all other algorithms are almost exploring the same number of nodes. It is because all the algorithms are trying to cover the entire grid from start to goal position and in the process they are covering all the nodes. If we would have considered diagonal movement as well and our goal position is somewhere in the middle of the grid (not in the southeast corner), then our results could have been different. Also, it can be seen that Random planner though exploring same nodes as BFS and Dijkstra doesn't provide optimal paths like them which is expected as its movement

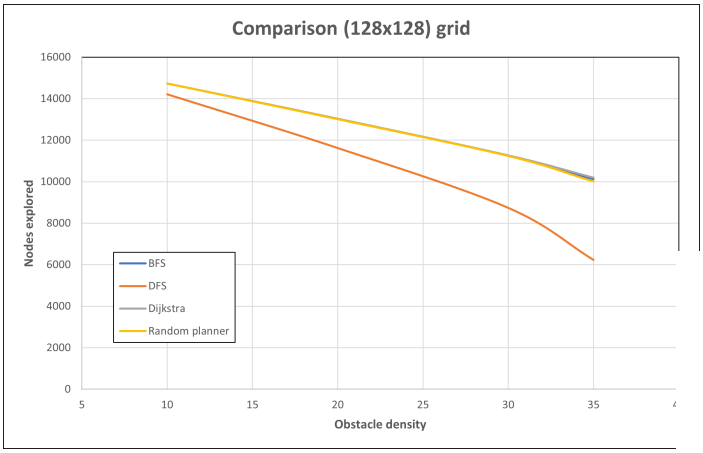


Fig. 6: Iterations vs obstacle density on 128x128 grid

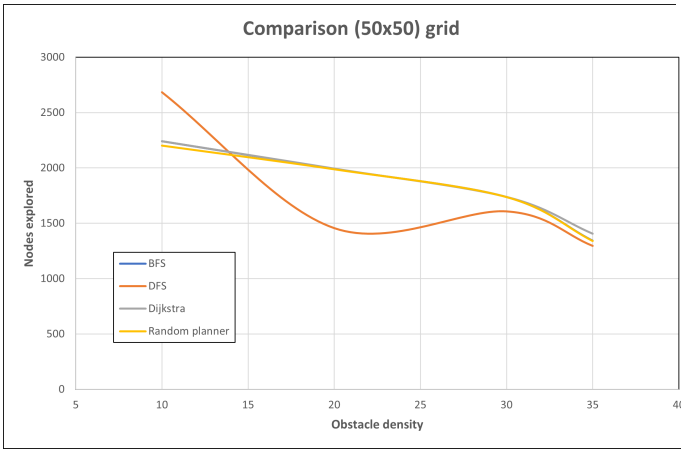


Fig. 7: Iterations vs obstacle density on 50x50 grid

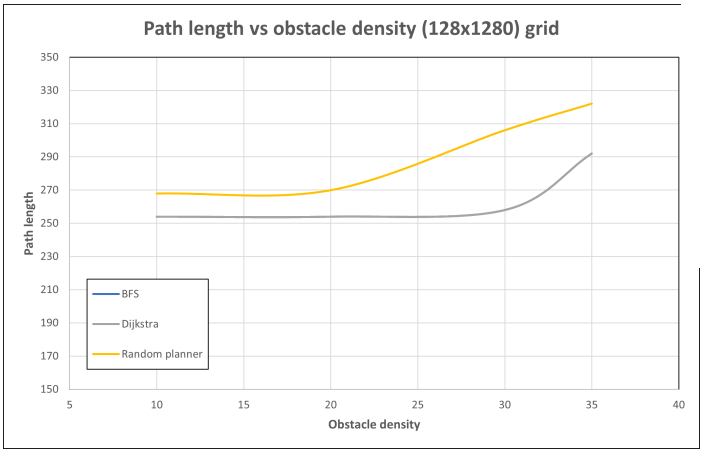


Fig. 8: Path length vs obstacle density on 128x128 grid (DFS not plotted because of too large iterations.

is random. On the other hand, DFS which is exploring least number of nodes is giving the longest path from start to goal position. So, it can be established that the algorithm exploring the least number of nodes cannot be assumed to be working best. In our case, BFS and Dijkstra are giving the optimal shortest paths from start position to goal position. **The results of BFS and Dijkstra are same as our graph is unweighted. If the graph is weighted Dijkstra would have performed better.**

VI. Results on different grid sizes and obstacle densities

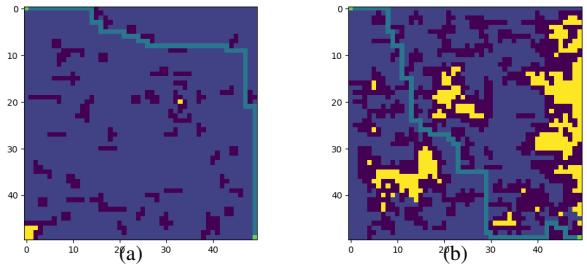
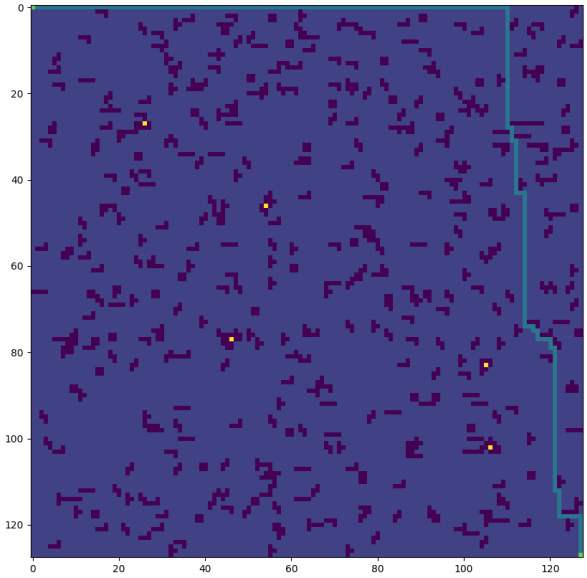
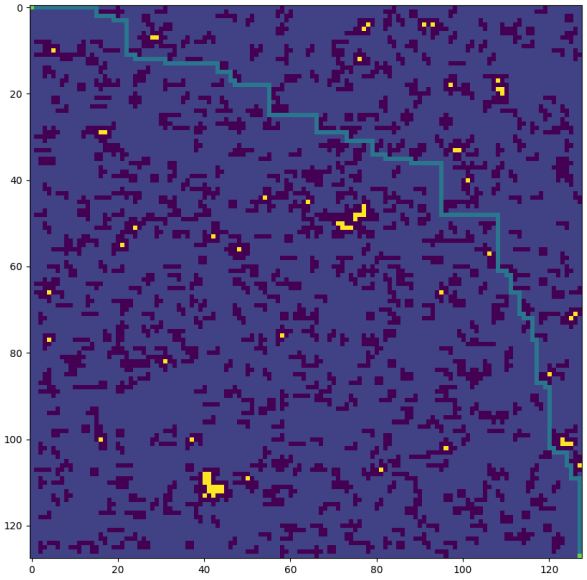


Fig. 9: BFS - 50x50 grid - 10% and 35% coverage



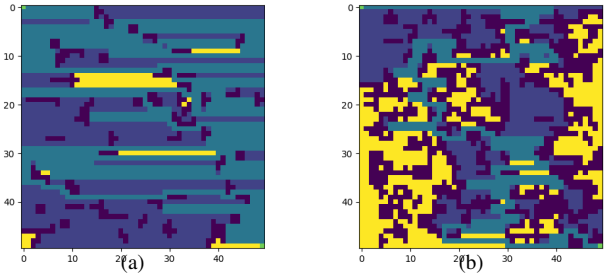
(a)

Fig. 10: BFS - 128x128 grid - 10% coverage



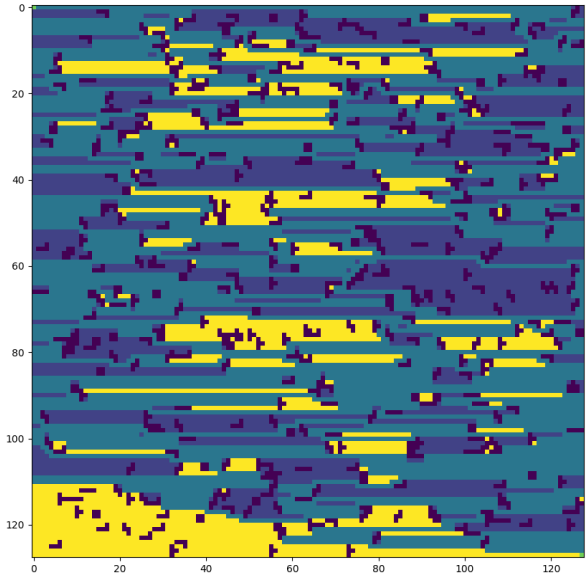
(a)

Fig. 11: BFS - 128x128 grid - 20% coverage



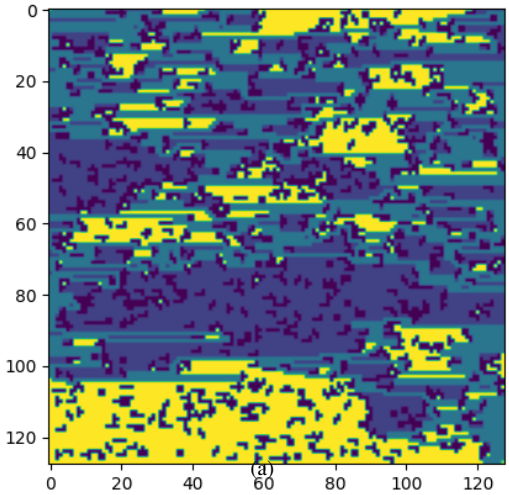
(a) (b)

Fig. 12: DFS - 50x50 grid - 10% and 35% coverage



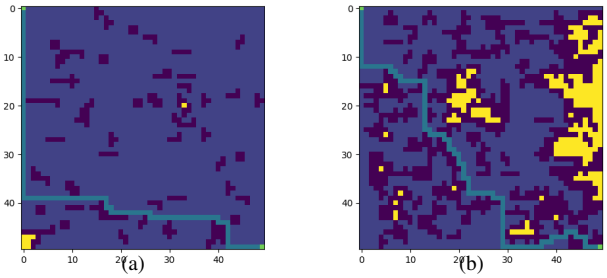
(a)

Fig. 13: DFS - 128x128grid - 10% coverage



(a)

Fig. 14: DFS - 128x128grid - 20% coverage



(a) (b)

Fig. 15: Dijkstra - 50x50 grid - 10% and 35% coverage

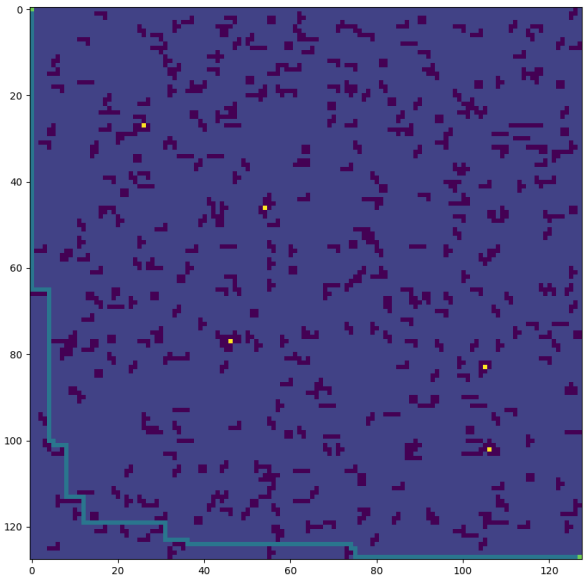


Fig. 16: Dijkstra- 128x128 grid - 10% coverage

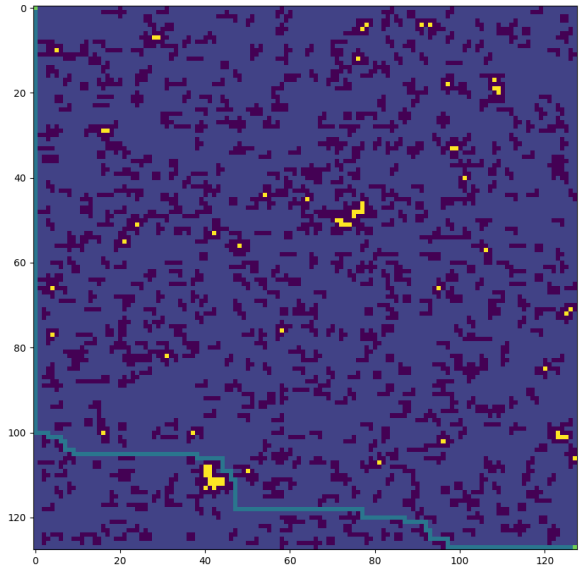
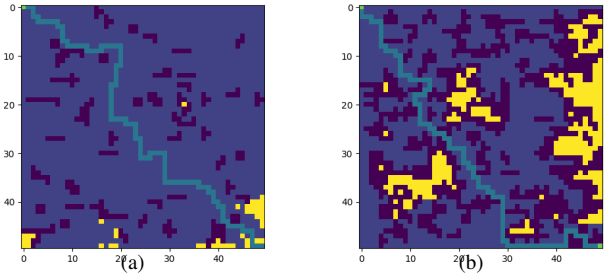
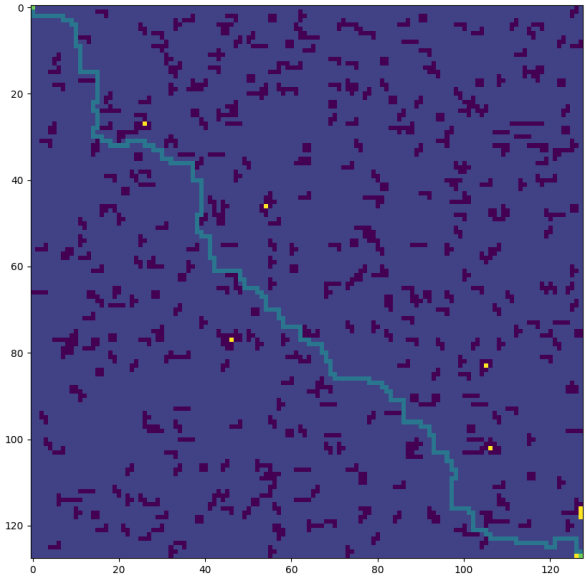


Fig. 17: Dijkstra- 128x128 grid - 20% coverage



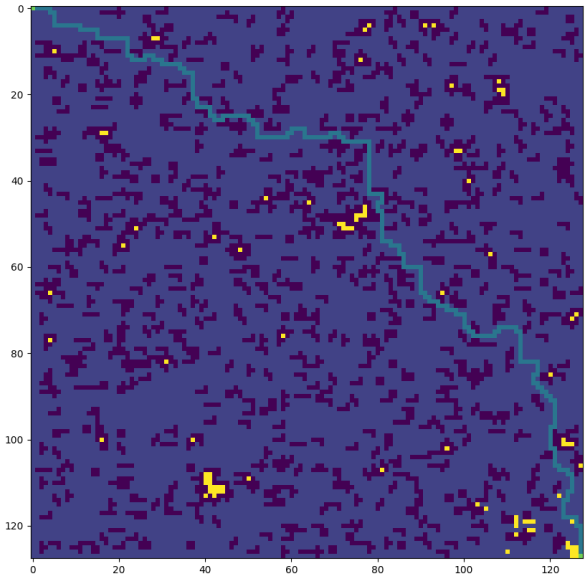
(a) (b)

Fig. 18: Random planner - 50x50 grid - 10% and 35% coverage



(a)

Fig. 19: Random Planner 128x128 grid - 10% coverage



(a)

Fig. 20: Random Planner 128x128 grid - 20% coverage