Flatland assignment Report
Motion Planning Course

Student: Rushikesh Pramod Deshmukh

Student ID: 565298809

MS Robotics, WPI

Goal:

- Design and implement flatland world 2D Grid of 128x128 nodes
- Design and implement Breadth-First Search algorithm to find a path from North-West corner to South-East corner.
- Design and implement Depth First Search algorithm to find a path from North-West corner to South-East corner.
- Design and implement Dijkstra's Search algorithm to find a path from the North-West corner to the South-East corner.
- Design and implement Random Planner algorithm to find a path from North-West corner to South-East corner.
- Compare different motion planning algorithms and plot graphs to show the comparison

BFS:

Source Code: *bfs.py* and *obstacle_field.py*

Details:

This algorithm takes a randomly generated flatland grid from obstacle_field.py and using bfs() finds a path from the start node to the end node. Here flatland grid is a 2D NumPy array of user-defined data type *Nodes.* Nodes is a class defined in *obstacle_field.py* to represent a single node of the grid. Each node has various attributes associated with it like color and neighbors. The color attribute is used to denote if the node is a start node, end node, open node, or obstacle node. The neighbor attribute is used to store all the neighboring nodes that can be traveled to from the current node.

The algorithm uses a queue data structure to schedule the nodes to be explored in a breadth-first manner.

Observation:

This algorithm explores almost all the nodes in the flatland grid before it finds the end node. It does return a short path.

Images:

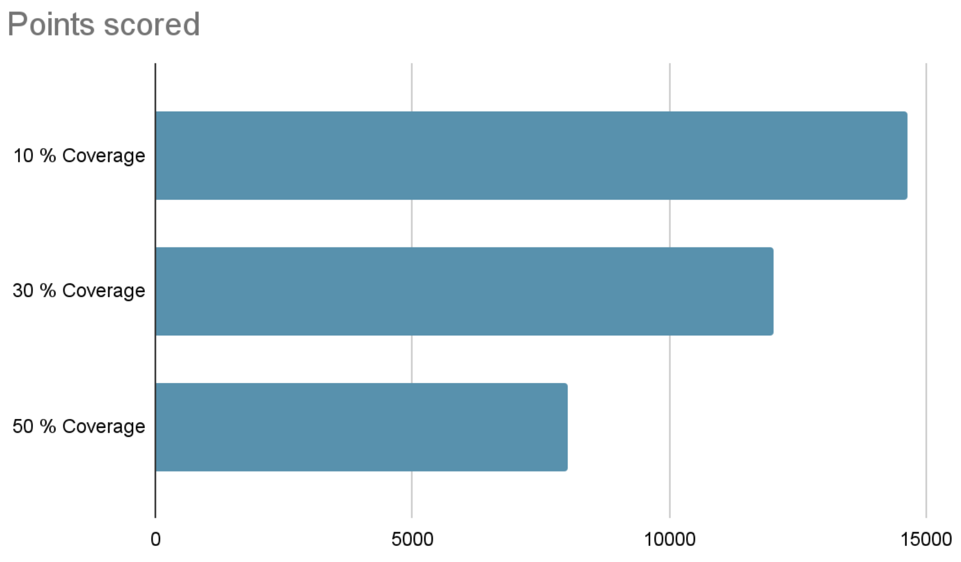

Figure: Nodes Explored vs Grid coverage % for BFS algo

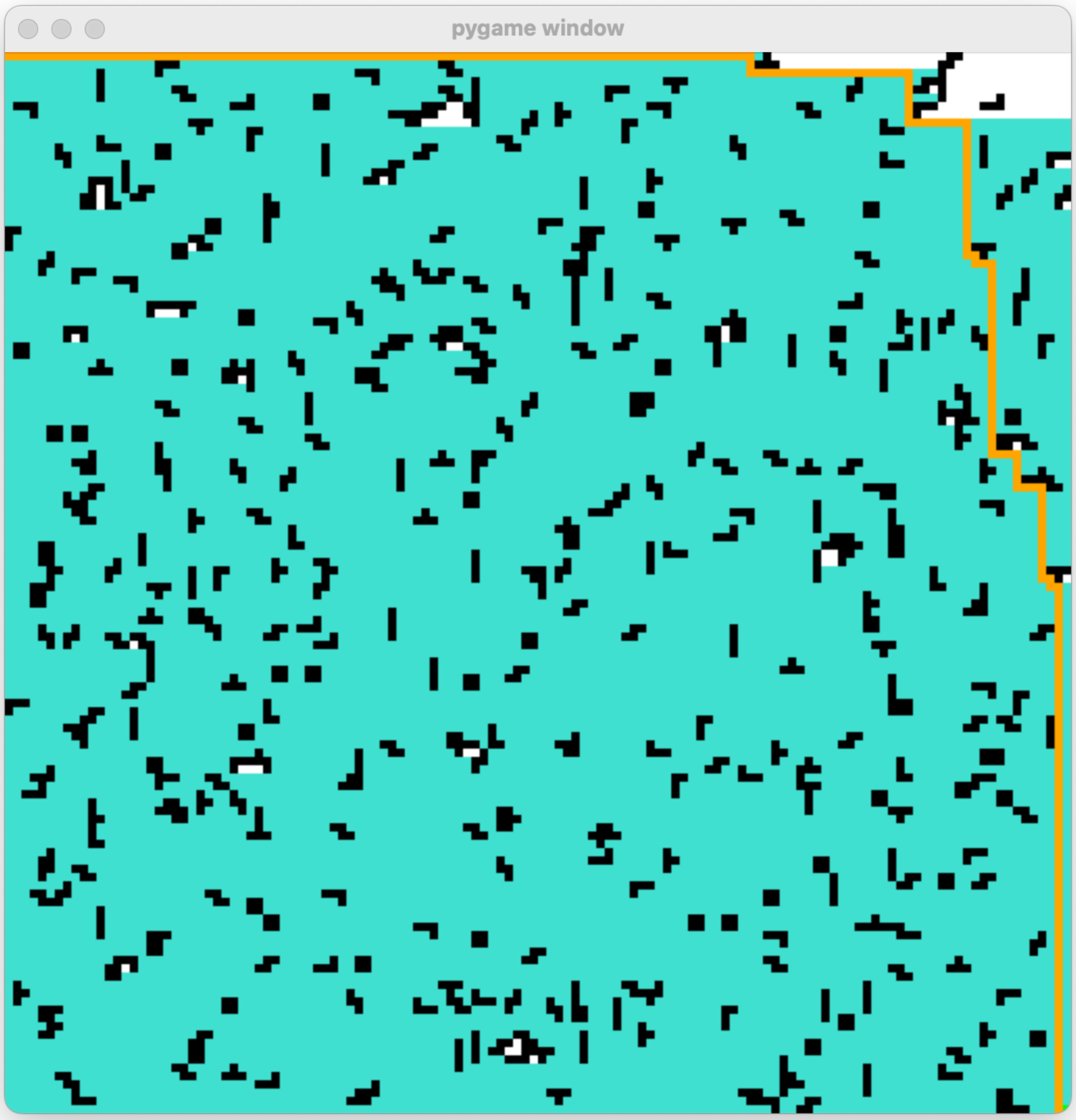

Figure: BFS algorithm for 10% coverage grid

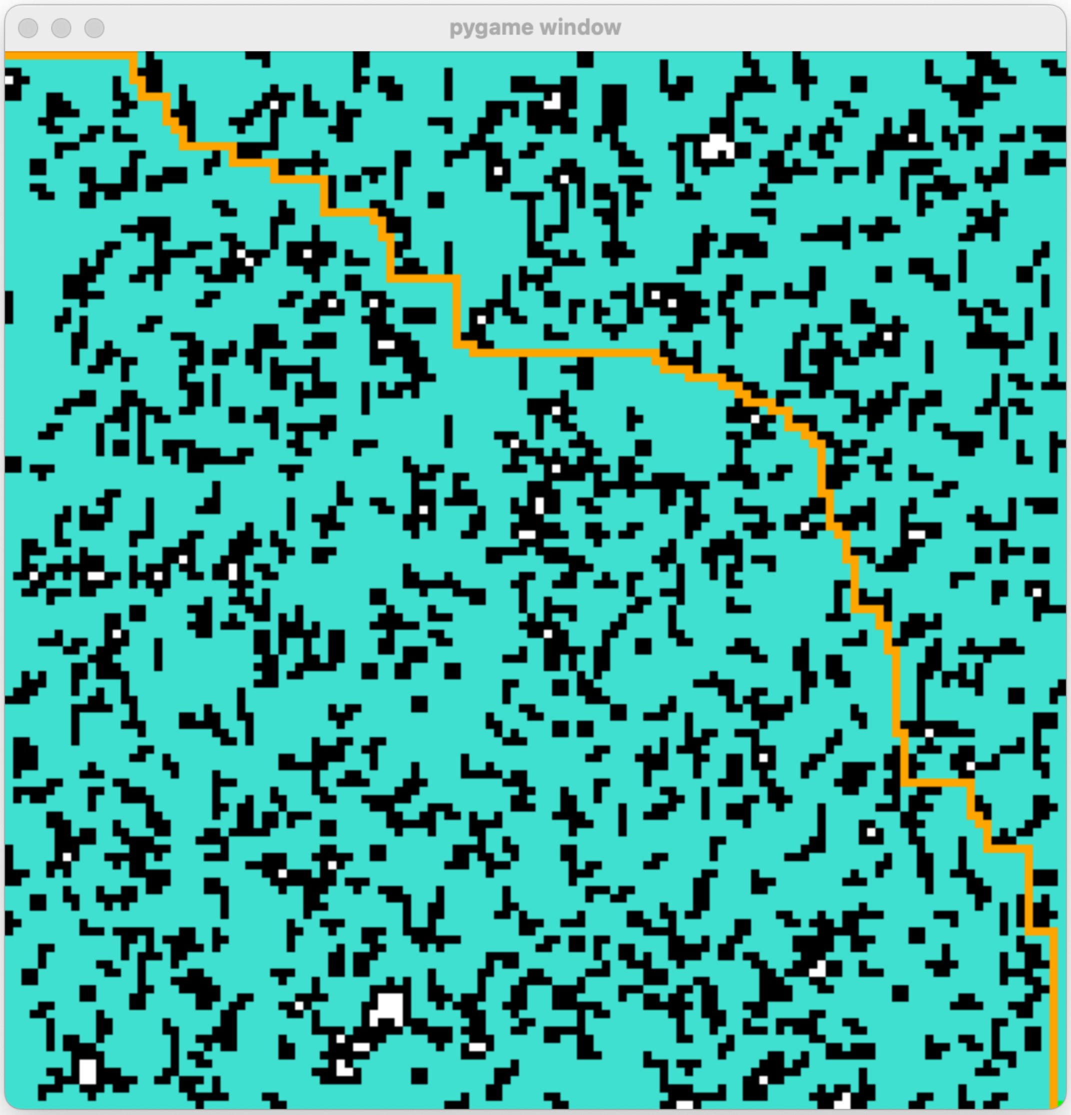

Figure: BFS algorithm for 30% coverage of the gird

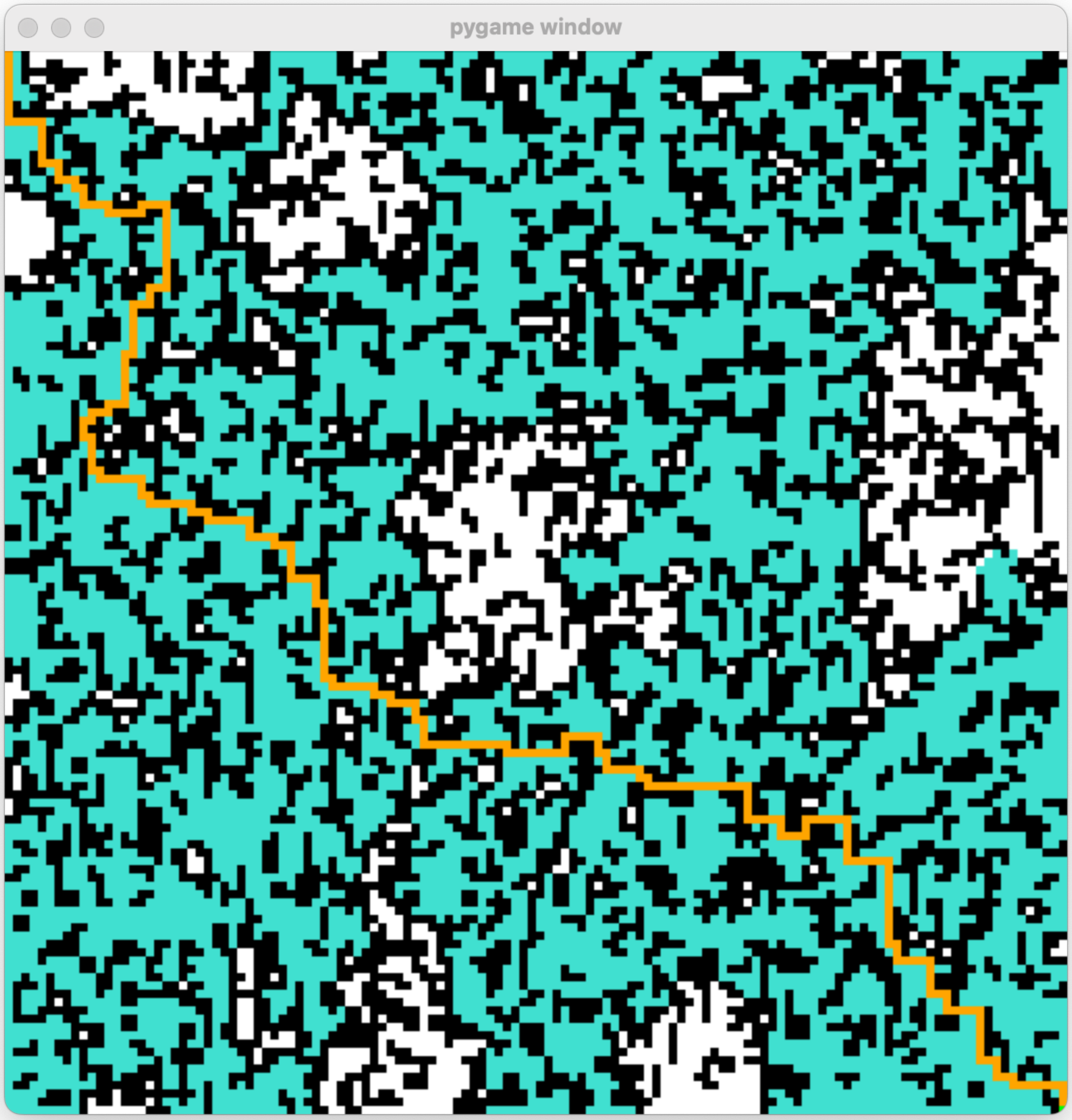

Figure: BFS algorithm for 50% grid coverage

DFS:

Source Code: *dfs.py* and *obstacle_field.py*

Details:

This algorithm takes a randomly generated flatland grid from obstacle_field.py and using dfs() finds a path from the start node to the end node. Here flatland grid is a 2D NumPy array of user-defined data type *Nodes.* Nodes is a class defined in *obstacle_field.py* to represent a single node of the grid. Each node has various attributes associated with it like color and neighbors. The color attribute is used to denote if the node is a start node, end node, open node, or obstacle node. The neighbor attribute is used to store all the neighboring nodes that can be traveled to from the current node.

The algorithm uses a stack data structure to schedule the nodes to be explored in a depth-first manner.

Observation:

This algorithm explores less number of the nodes in the flatland grid before it finds the end node than BFS algo. It does not return a short path. This can be changed by changing the sequence of neighboring nodes in which they are explored.

Images:

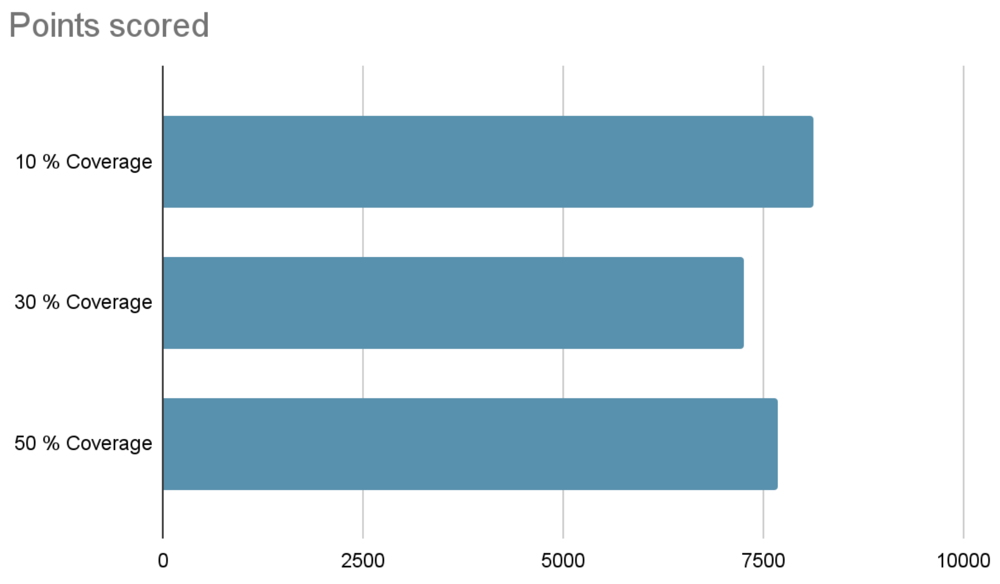

Figure: Nodes Explored vs Grid coverage % for DFS algo

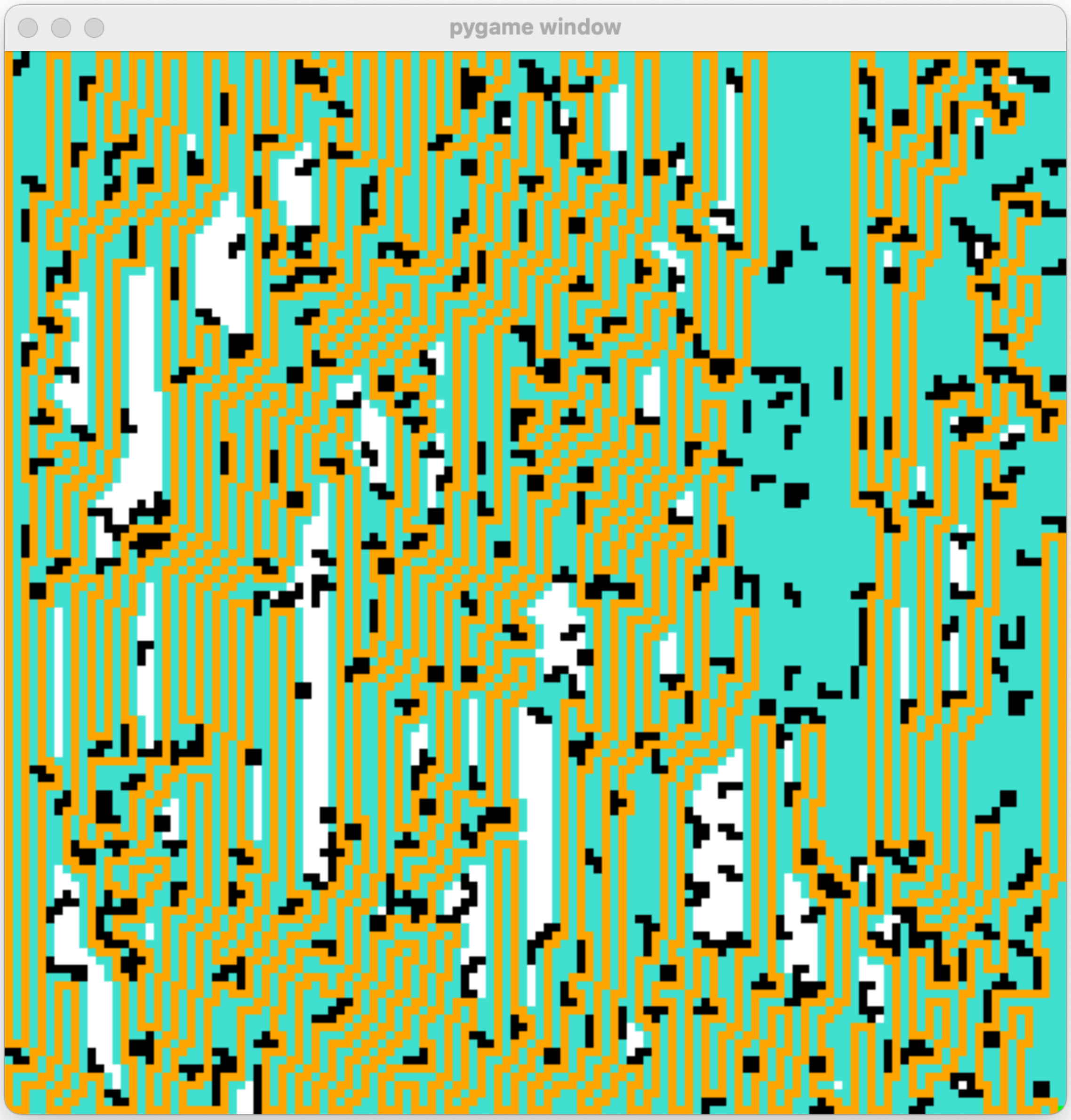

Figure: DFS algorithm for 10% Grid Coverage

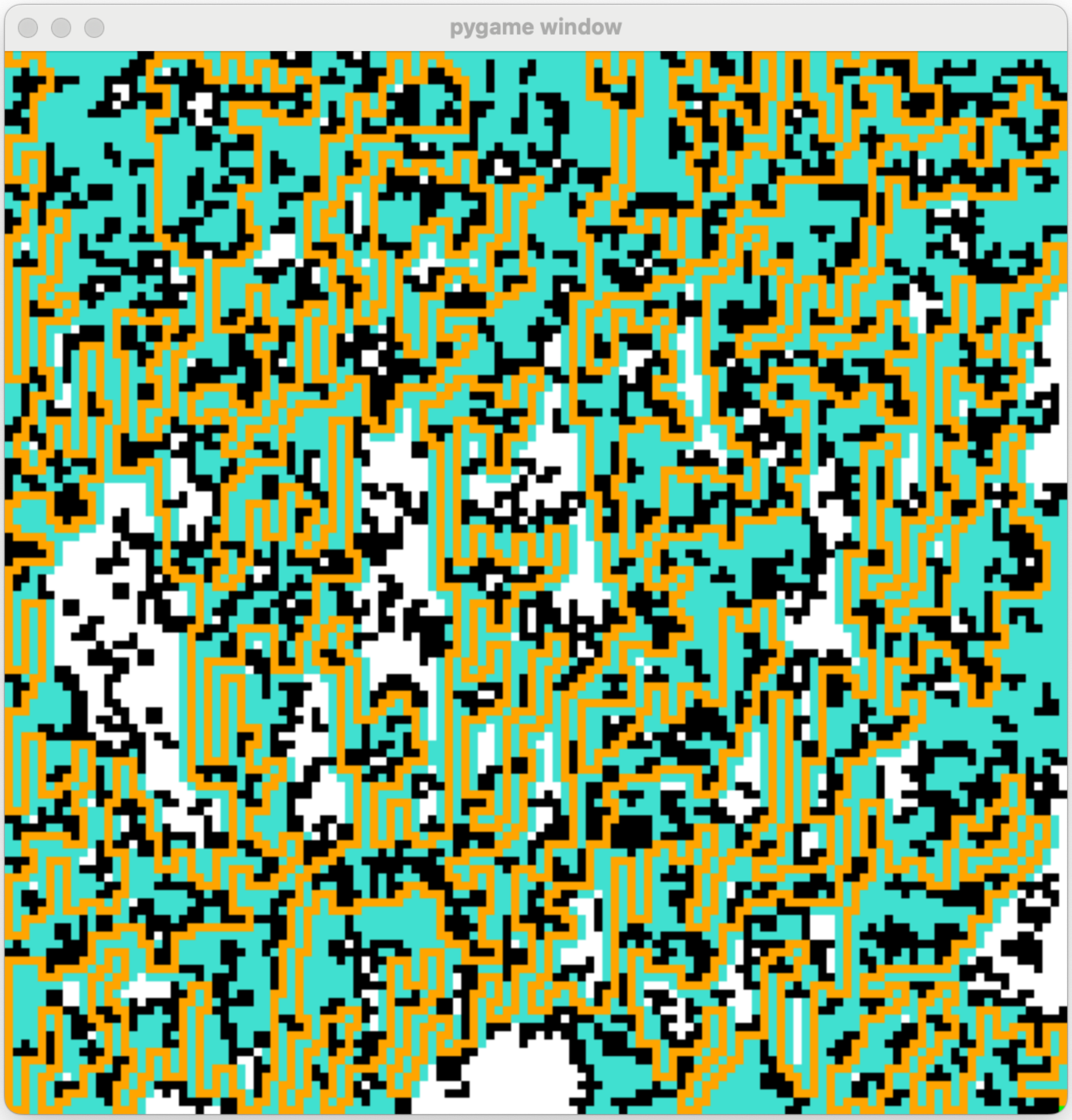

Figure: DFS algorithm for 30% grid coverage

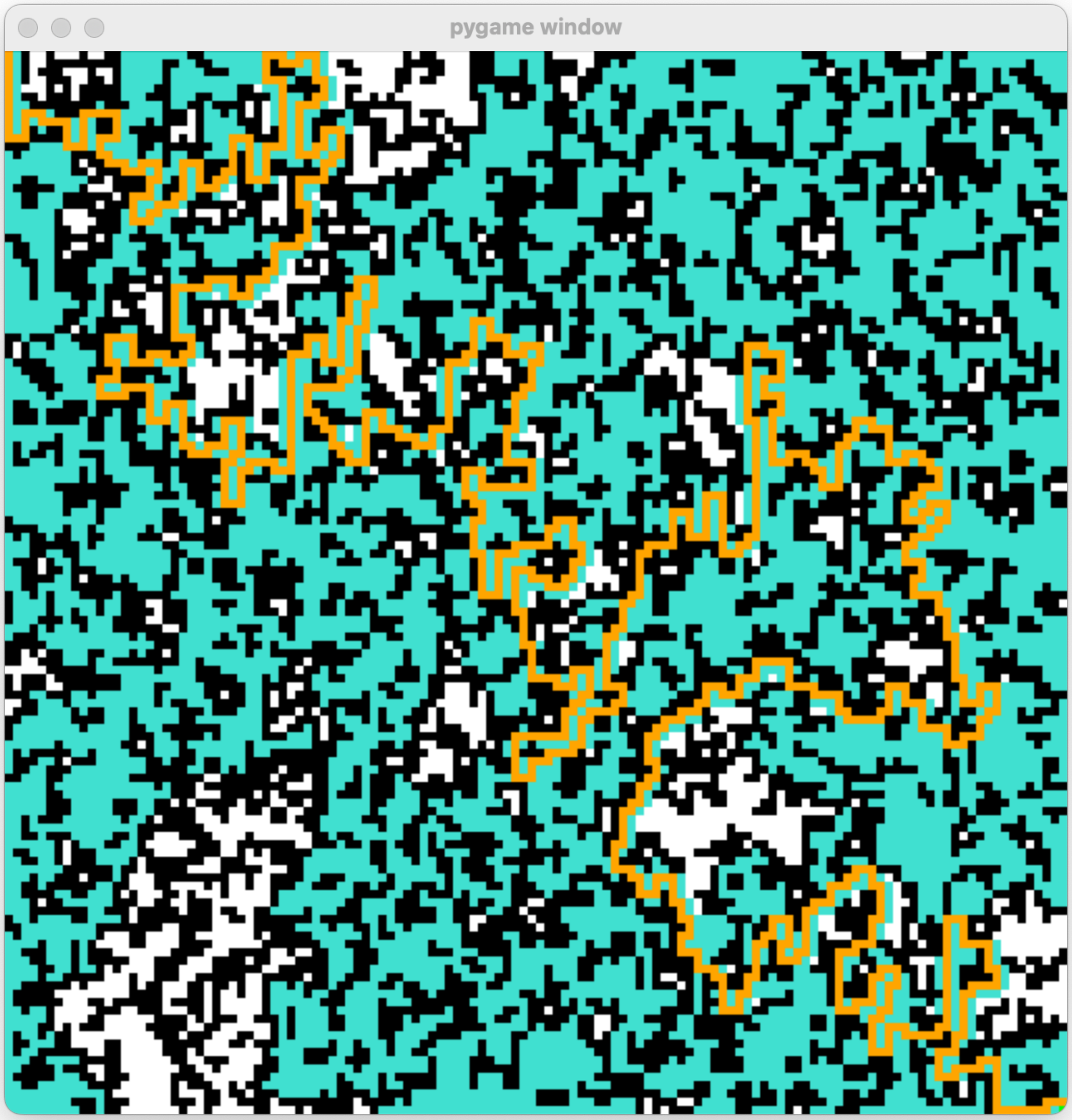

Figure: DFS algorithm for 50% grid coverage

Dijkstra’s Algorithm version 0 :

Source Code: *dijkstra.py* and *obstacle_field.py*

Details:

This algorithm takes a randomly generated flatland grid from obstacle_field.py and using dijkstra() finds a path from the start node to the end node. Here flatland grid is a 2D NumPy array of user-defined data type *Nodes.* Nodes is a class defined in *obstacle_field.py* to represent a single node of the grid. Each node has various attributes associated with it like color and neighbors. The color attribute is used to denote if the node is a start node, end node, open node, or obstacle node. The neighbor attribute is used to store all the neighboring nodes that can be traveled to from the current node.

The algorithm uses a priority queue data structure to schedule the nodes to be explored. In this version of algorithm, cost of the nodes connected to the obstacles is considered as 100. This will encourage the algorithm to not move towards the obstacles

Observation:

This algorithm explores almost all of the nodes in the flatland grid before it finds the end node like BFS algo. It does return a short path. This can be changed by changing the cost function of neighboring nodes.

Images:

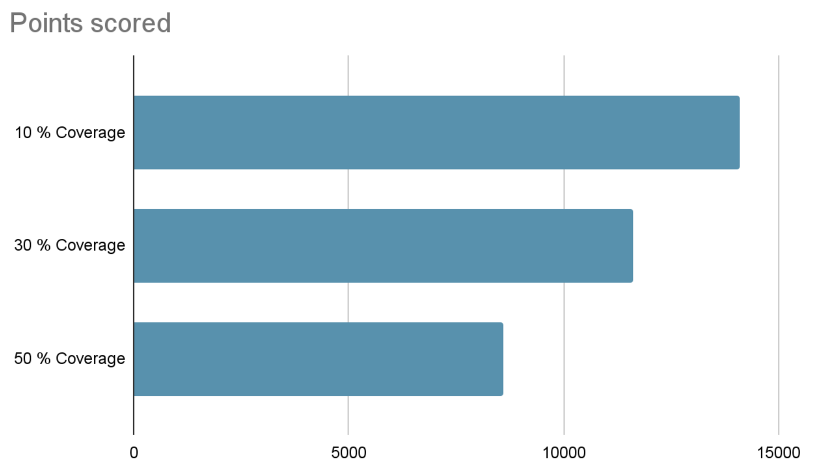

Figure: Nodes Explored vs Grid coverage % for Dijkstra algo version 0

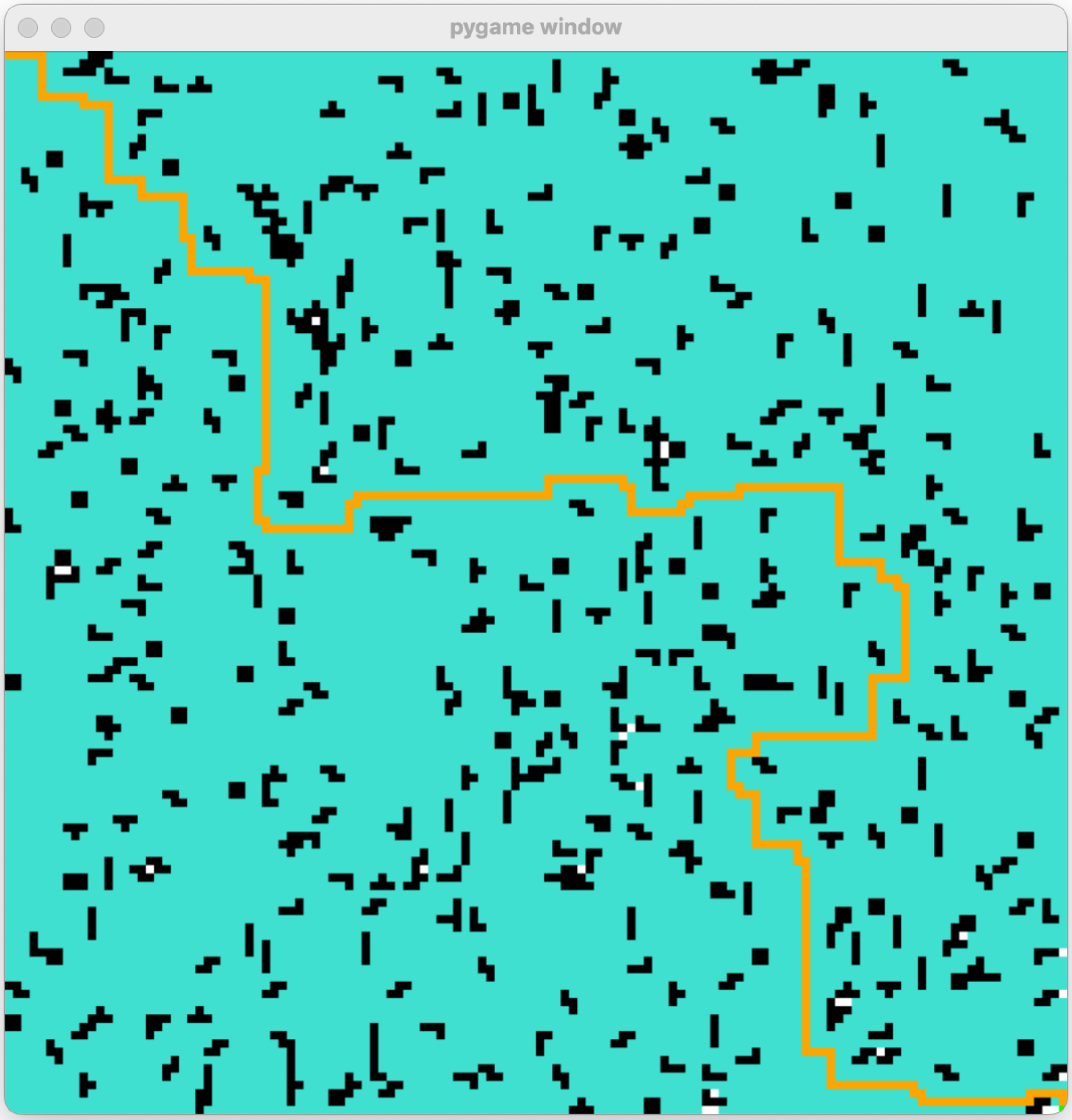

Figure: Dijkstra algorithm for 10% Grid Coverage

Dijkstra’s Algorithm version 1 :

Source Code: *dijkstra1.py* and *weighted_graph.py*

Details:

This algorithm takes a randomly generated flatland grid from weighted_graph.py and using Dijkstra1() finds a path from the start node to the end node. Here flatland grid is a 2D NumPy array of user-defined data type *Nodes.* Nodes is a class defined in *obstacle_field.py* to represent a single node of the grid. Each node has various attributes associated with it like color and neighbors. The color attribute is used to denote if the node is a start node, end node, open node, or obstacle node. The neighbor attribute is used to store all the neighboring nodes that can be traveled to from the current node.

The algorithm uses a priority queue data structure to schedule the nodes to be explored. In this version of algorithm, the cost of the neighboring nodes connected to the current node on the left and up side is considered as 10. This will encourage the algorithm to prioritize moving down and right.

Observation:

This algorithm explores very less number of the nodes in the flatland grid before it finds the end node. It does return a short path. This can be changed by changing the cost function of neighboring nodes.

Images:

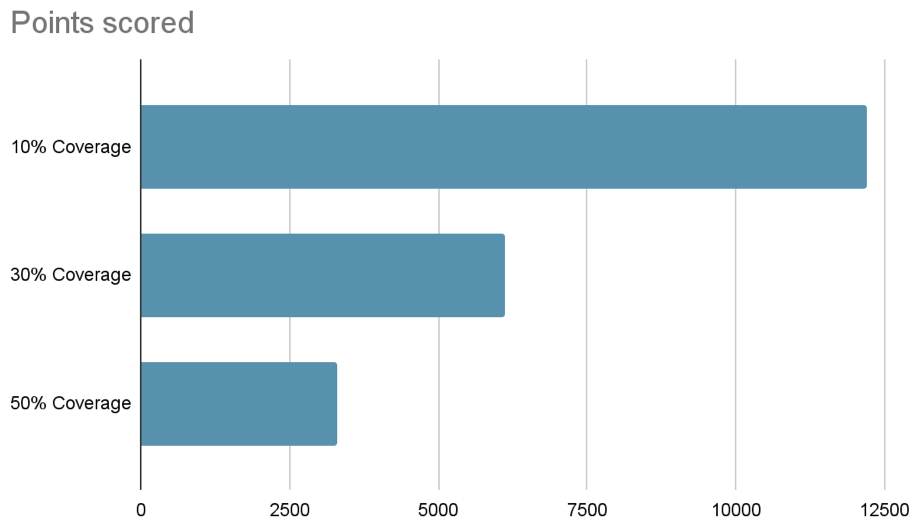

Figure: Nodes Explored vs % grid coverage for this version of Dijkstra’s Algo

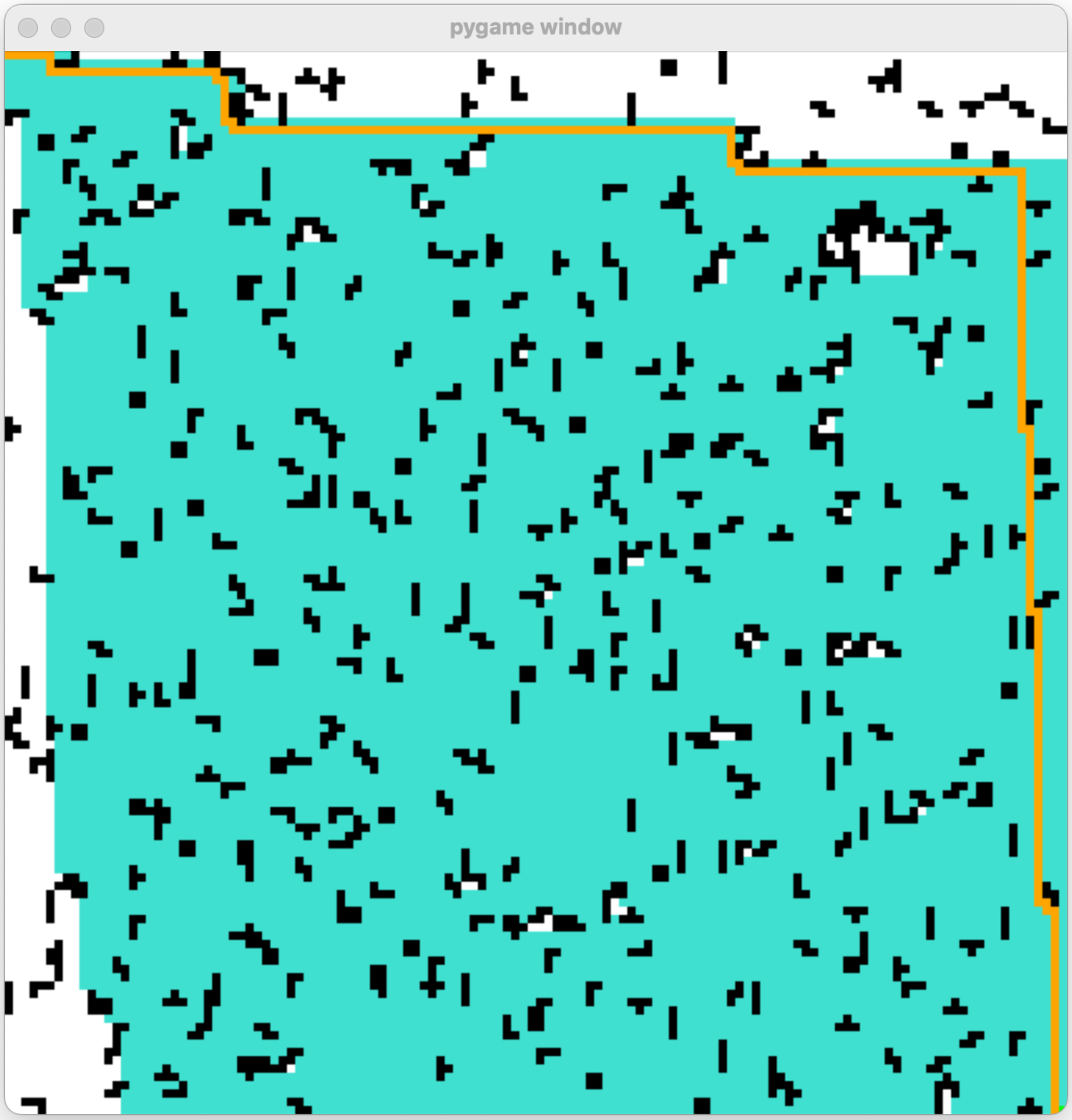

Figure: Dijkstra’s algo for 10% grid coverage

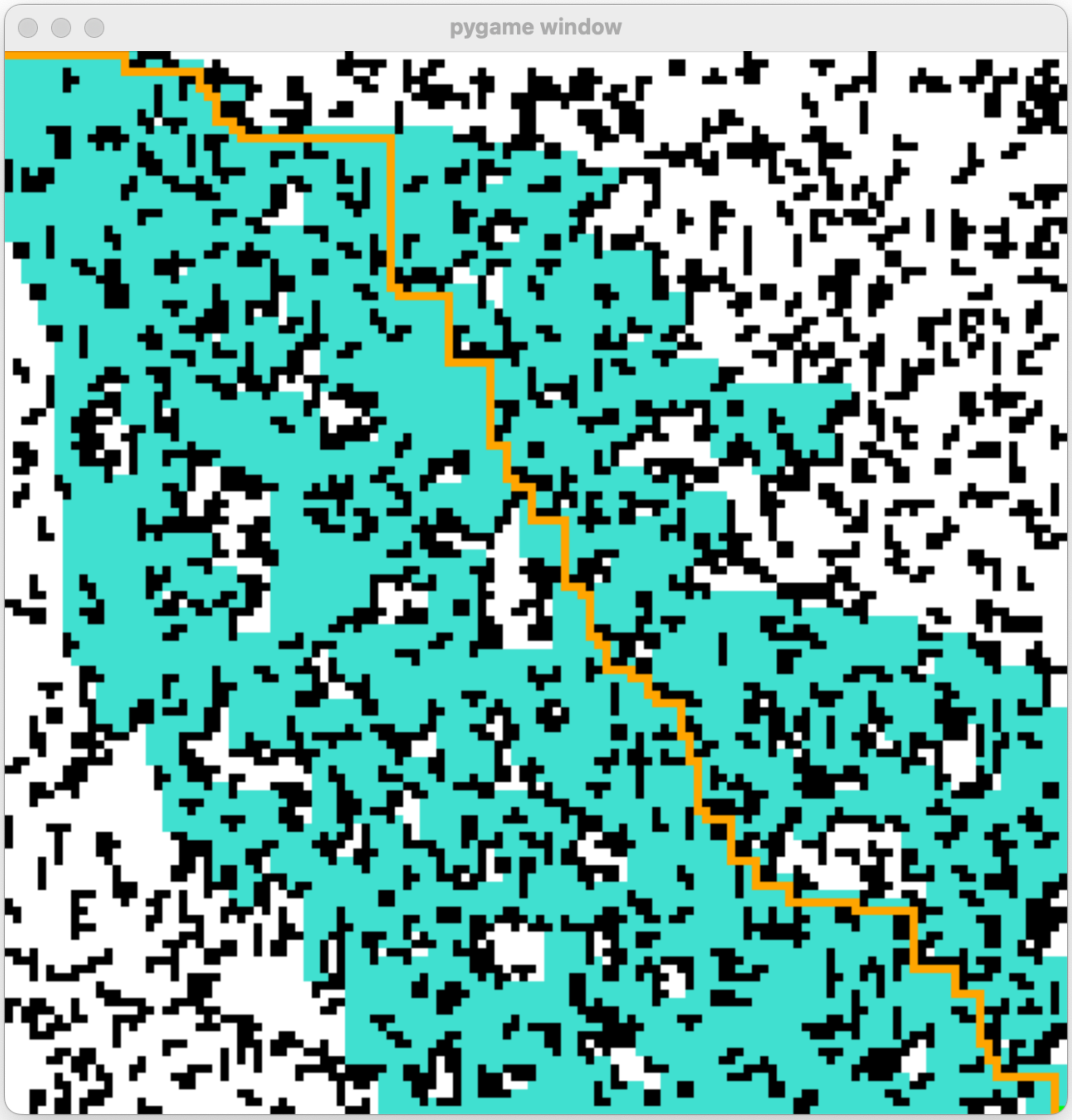

Figure: Dijkstra’s algo for 30% grid coverage

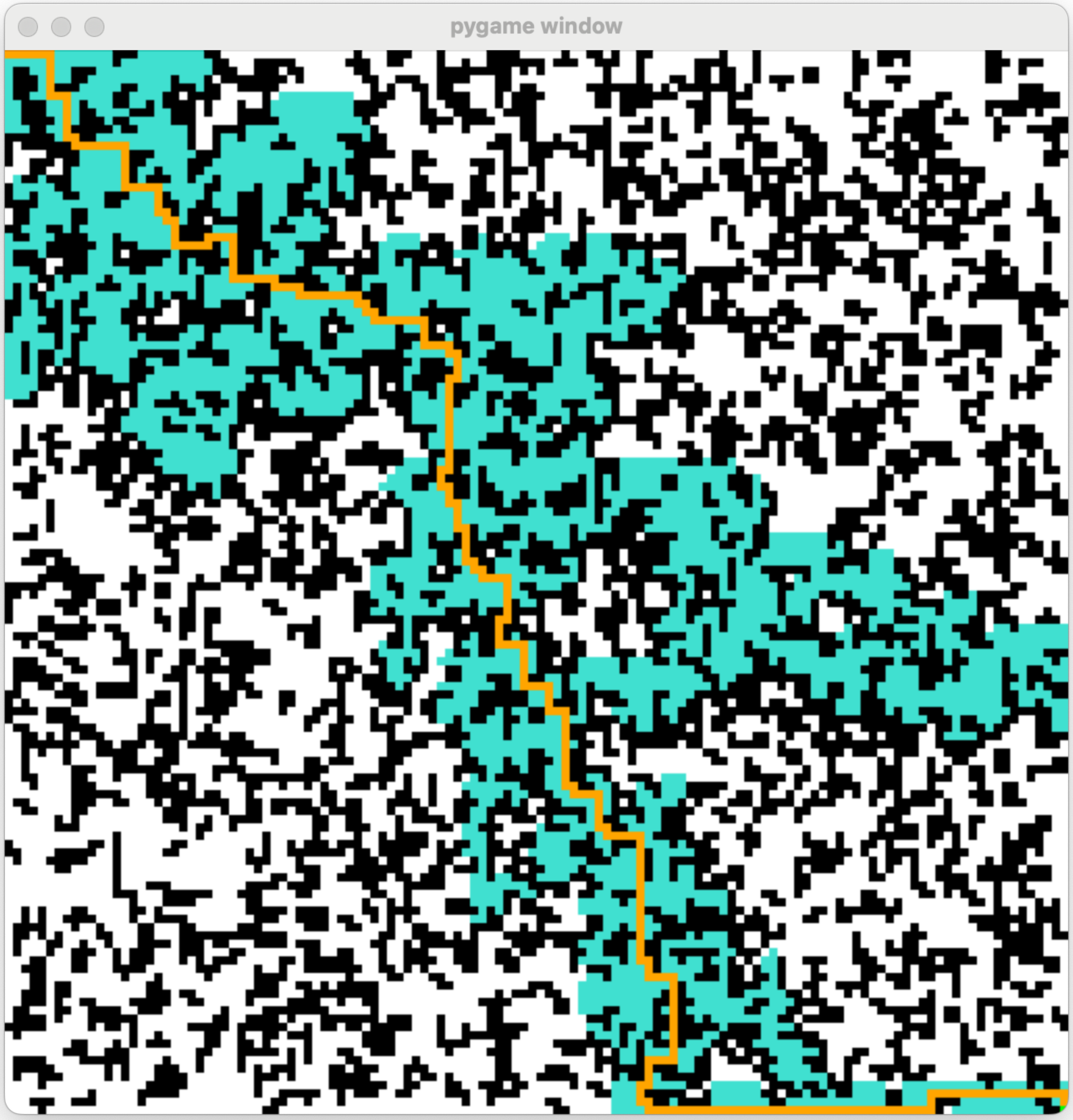

Figure: Dijkstra’s algo for 50% grid coverage

Random Planner Algorithm:

Source Code: *random_planner.py* and obstacle_field*.py*

Details:

This algorithm takes a randomly generated flatland grid from obstacle_field.py and using random_planner() finds a path from the start node to the end node. Here flatland grid is a 2D NumPy array of user-defined data type *Nodes.* Nodes is a class defined in *obstacle_field.py* to represent a single node of the grid. Each node has various attributes associated with it like color and neighbors. The color attribute is used to denote if the node is a start node, end node, open node, or obstacle node. The neighbor attribute is used to store all the neighboring nodes that can be traveled to from the current node.

The algorithm does not use any data structure to schedule the nodes to be explored. In this algorithm, random neighbor nodes are chosen to be explored

Observation:

This algorithm explores very high number of the nodes in the flatland grid before it finds the end node. It might or might not return a path.

Images:

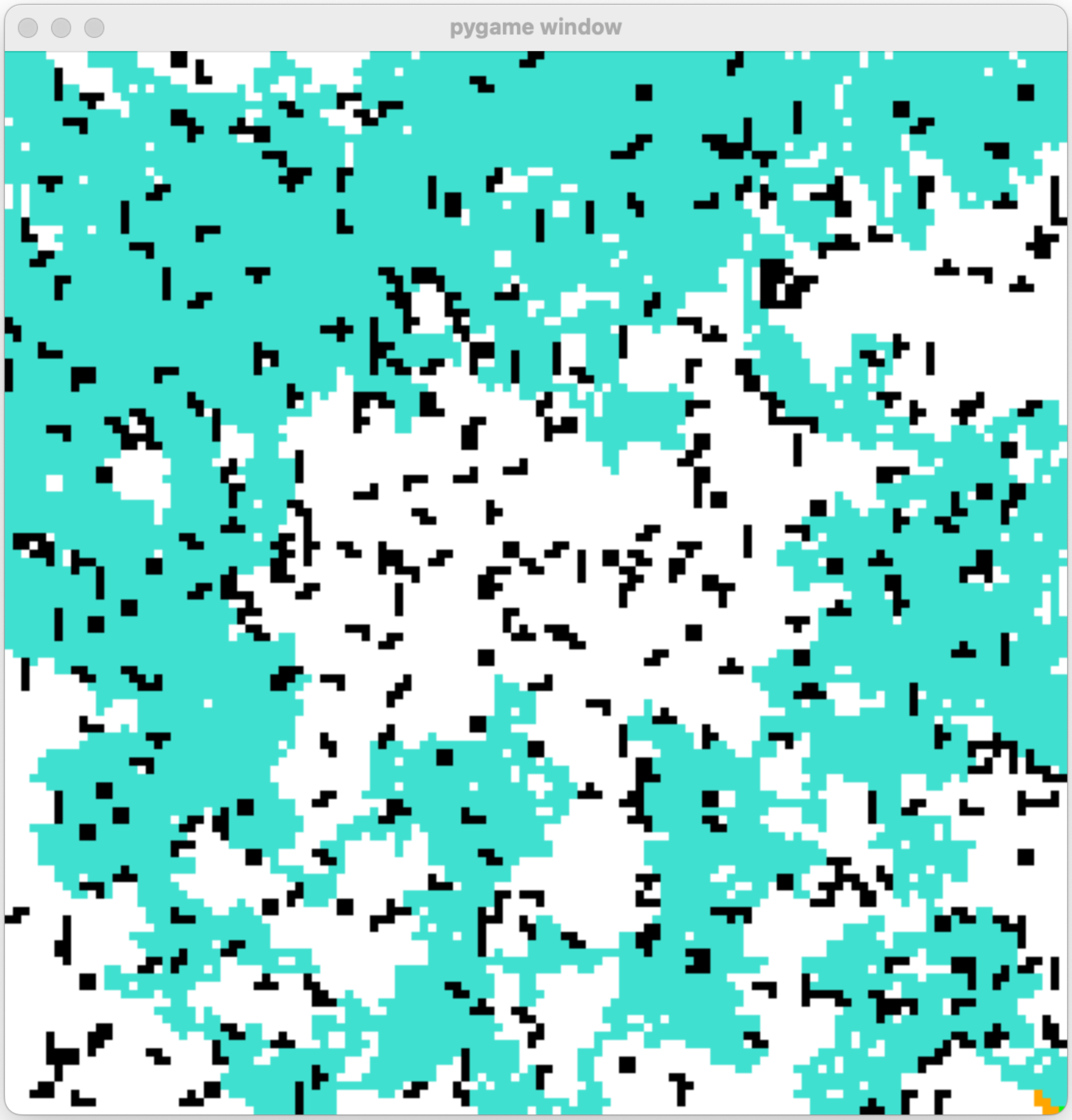

Figure: Random Planner for 10% grid coverage

Comparison of algorithms:

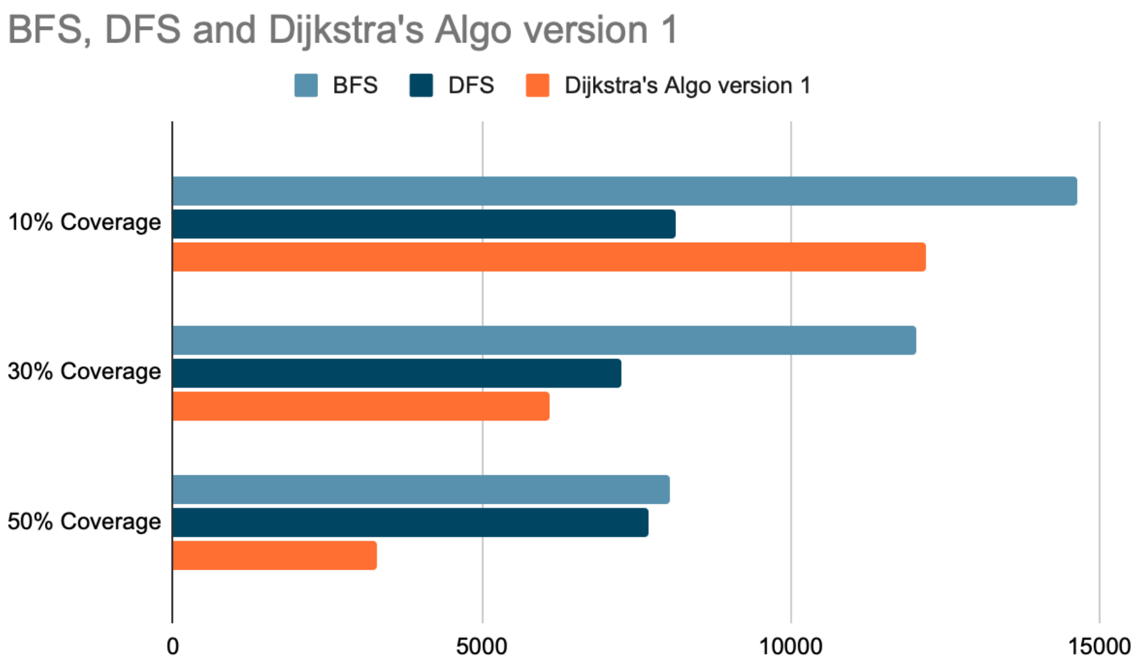

Figure: Comparison of BFS, DFS, Dijkstra’s Algo

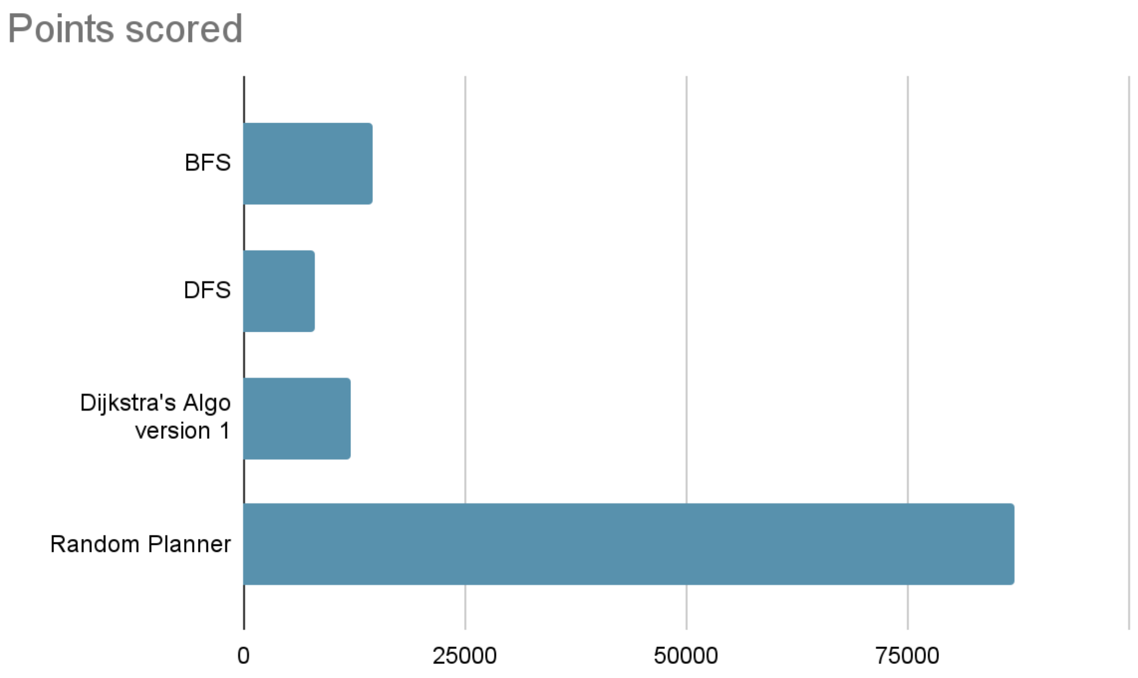

Figure: Comparison of random planner algo with other algorithms for 10% grid coverage