Aadiv Shah
WPI ID 413770228

Assignment 2

Flatland

Introduction

Playing *TETRIS* is hard enough, imagine wading through a sea of *TETRIS* blocks from the Northwest corner all the way down to the Southeast corner! Fortunately our PR (point robot) knows the location of every obstacle and its start and end point. PR also has a few cool tricks up its sleeve to navigate through this sea using 4 different planning algorithms:

1. Breadth First Search
2. Depth First Search
3. Dijkstra's Algorithm
4. Random Planner

Implementation

Environment Generation

Using **Pygame**, I simulated the obstacle course with randomly generated tetromino shapes strewn across the 128×128 grid. An example is shown in Fig 1. The program will generate a grid starting with a density of 5% and incrementally increases in steps of 5%.

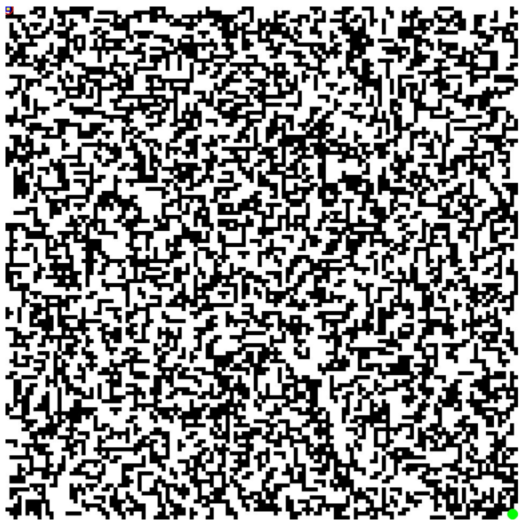

Figure 1: 128×128 grid with 55% obstacle coverage. In red: start point, in green: goal

Once the grid is finalised, 4 robots are spawned, moving according to their respective planning algorithms. After the robots have reached the goal, the program will reset their positions and generate a new grid with higher density.

Robot Framework

Keeping in mind modularity, I developed several classes keeping in mind the properties of the robot e.g *Robot Parameters* class, robot *Controller* class and robot *Telemetry* class. I also developed a *World* and *Visualiser* class to define the world and display the environment + robot on pygame. Finally all the classes and their respective methods are called sequentially in the main function.

This modularity allows me to spawn multiple robots simultaneously to compare the different planning algorithms. An example is shown in Fig 2 and in the submission video.

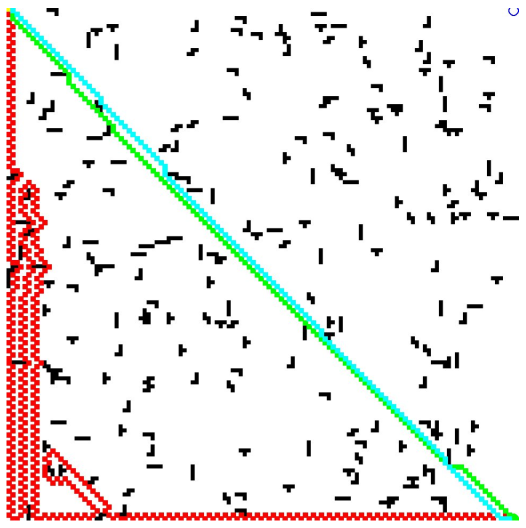

Figure 2: 3 PRs spawned, each using a different planning algorithm. Bright red - DFS, Green - BFS, Blue - Dijkstra's

Planning Algorithms

The 4 planning algorithms were inspired by Sakai, Atushi, et al. [1]. The robots may move to any $8 - connected$ neighbour with the cost equal to the euclidean distance of the current location and its neighbour.

Separate class files were created to keep the code organized. In the above example an object of each planner class is instantiated and the planning function is called. The final path is stored as a property of the controller class of each robot.

Inside a while loop, the controller is *stepped* and the position of the robots is updated. Finally the current state of the world and robots is updated and displayed. The obstacle density is incremented and then the process is repeated again.

Results

In general the **BFS & Dijkstra's** algorithm performed the best with the least number of steps needed to reach the goal. On the other hand the **DFS & Random** planner took seemingly arbitrary number of steps to reach the goal. The random planner maxed out the number of iterations without reaching the goal.

Another generalization that may be made for BFS and Dijkstra's is that the number of iterations increased as the obstacle density increased. The plots for *number of iterations* vs *obstacle density* is shown in Fig 3. In the graph, the plot for BFS and Dijkstra's **overlaps** hence only one plot is visible. Although the actual path taken by the robots may differ, the total number of steps for BFS and Dijkstra's remains the same.

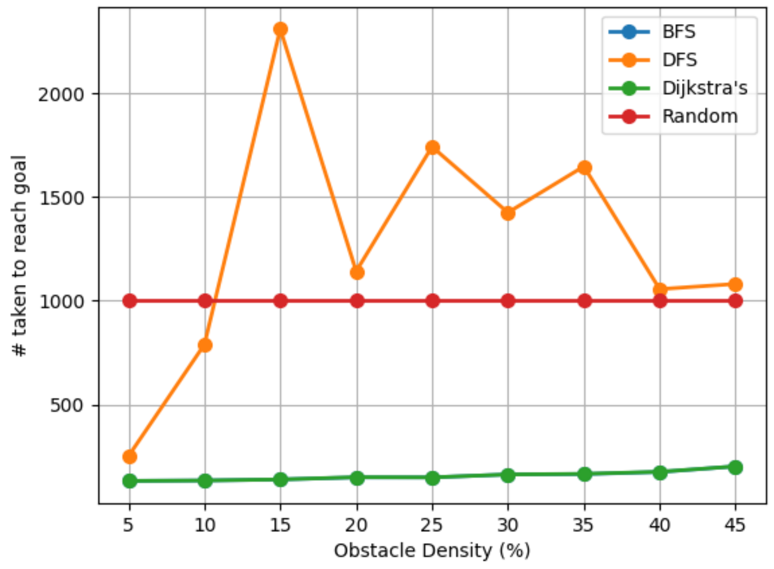

Figure 3: Number of iterations vs Obstacle Density for BFS, DFS, Dijkstra's and Random Planner

Shown in Fig 4 are the various environments produced with different obstacle density and the results of the planned paths. Note - the main function used is **test.py** and the function to plot the data from a saved .npy file is called **plot_data.py**. I have submitted the above 2 files, the virtual environment (as a zip) and the helper class files. The graphs required are in this document itself.

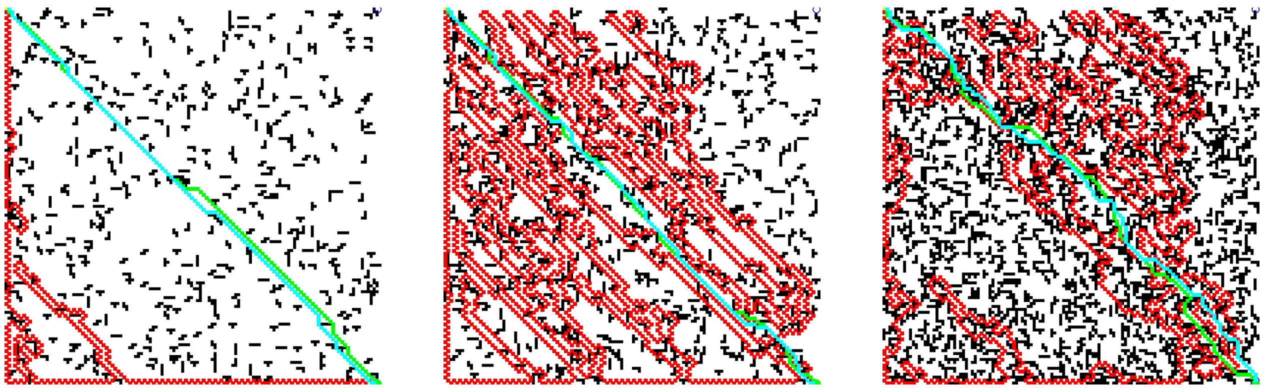
(a) 10% Density (b) 15% Density (c) 30% Density

Figure 4: Three figures with varying obstacle density and the respective planned paths. Bright red - DFS, Green - BFS, Blue - Dijkstra's

An example for the **Random Planner** is shown in Fig 5, it clearly shows the random planner cannot reach the goal, it would require a very large number of iterations.

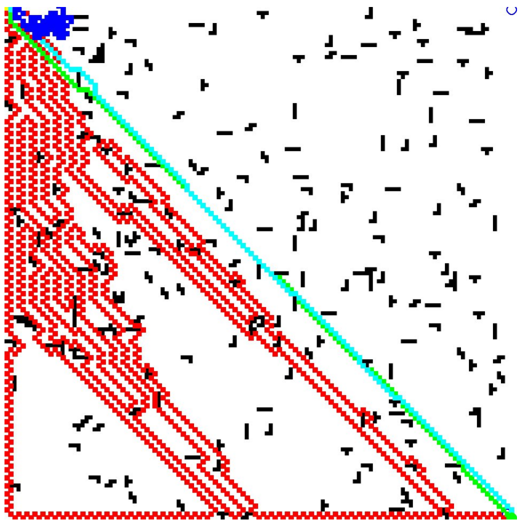
Figure 5: Random Planner, in blue. 5% density

Discussion

It is expected that BFS and Dijkstra's should give similar results as in the case of an *equally weighted* graph (or *unweighted* graph). This is proven as there is minimal difference between BFS and Dijkstra's as shown in Fig 3.

All in all, the BFS and Dijkstra's proved to be the best algorithm. Future scope could include creating a greedy algorithm to further reduce the number of iterations.

References

[1] Sakai, Atsushi, et al. "Pythonrobotics: a python code collection of robotics algorithms." arXiv preprint arXiv:1808.10703 (2018).