RUTGERS
UNIVERSITY | NEW BRUNSWICK

Utsav Patel (RUID - 211009880, NetID - upp10)
Manan Vakta (RUID - 206003851, NetID - mv651)

Master of Science in Computer Science

Voyage Into the Unknown

September 2021

The school of Graduate Studies

Contents

List of Figures

Chapter 1

Question 1,2 and 3

1.1 Question 1

The main reason why we only re-plan when blocked cells are discovered is because re-planning when unblocked cells are discovered will not change the shortest path from start to goal. Which means spending resources to calculate the new shortest path when we have no new information will never give us a path shorter than the one we are already following.

In the best case our path to the goal will not be blocked at all and thus moving without re-planning will lead us to the goal in the least amount of time possible.

Even though we are collecting information about the environment while traversing the path, it's not useful to re-plan because we may end up at the goal while following the current planned path. In the case that we do have a block on our path to the goal, getting as much information about the maze as possible before re-planning will again save us computation time.

1.2 Question 2

Our agent will **never** get stuck in a solvable maze because of the rules that we have set for it.

On the first run our agent will consider the maze to be completely unblocked and then plan a shortest path to the goal for this maze. It will continue to move on this path until it encounters a blocked cell or reaches the goal. On encountering a blocked cell it will update its own version of the maze with any blocked cells that it has encountered in its field of view. After this it will re-plan a new path to the goal from its current position. If it encounters a blocked cell it will

repeat the previous listed steps.

This makes it so that even in the worst case scenario where our agent has to find every blocked cell; in a solvable maze it will always be able to create a path to the goal once all blocked cells are encountered in the maze.

The agent will also never get stuck in a loop since it will always follow a path given to it by the A* algorithm and A* does not return looped paths.

1.3 Question 3

This will not be an optimal path in the **complete** grid world. This is simply because after completing the whole maze just once there is no guarantee that the agent has explored all the cells in the maze which might cause it to assume a certain path is optimal whereas there might be a more optimal path in the part of the grid world that it has not discovered yet. Let us explain this with an example.

Our maze is shown in Fig 1.1. Our agent assumes all the cells are unblocked and will attempt to follow a straight path to the goal, Fig 1.2 shows what this path might look like and how it will adapt based on the blocked cells it encounters. After completing this full run, we consider the undiscovered part of the maze to "blocked" and plan an optimal path in the discovered Grid World. So when we run our agent from the starting position again, it will attempt to follow a path as shown in Fig 1.3 . This is not the optimal path in the full Grid World. If we run our agent through the maze after it has encountered all the blocks we will get a path shown in Fig 1.4 which is the actual optimal path.

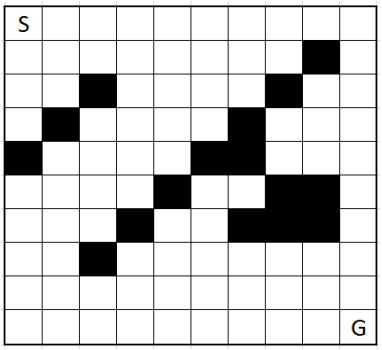

Figure 1.1: Original Maze

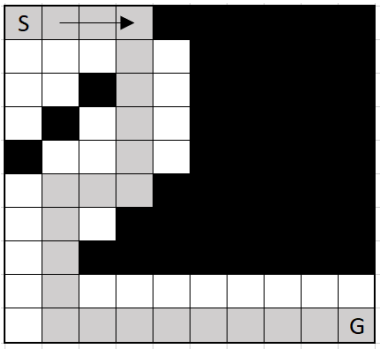

Figure 1.3: Optimal Path in Discovered Grid World

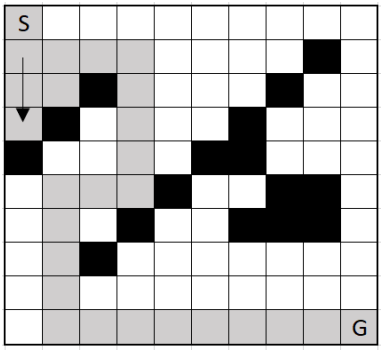

Figure 1.2: First Path

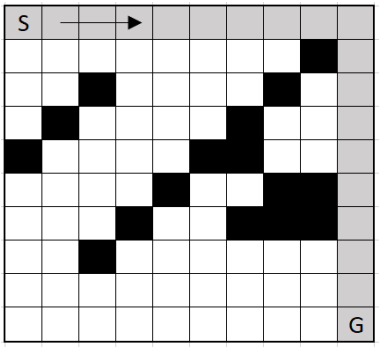

Figure 1.4: Optimal Path in Full Grid World

Chapter 2

Implementation, Questions 4 and 5

2.1 Implementation

2.1.1 A* search

- We have used *visited_nodes* of type *set* to maintain a record of all visited cells , *sorted_set* of type *SortedSet* as our fringe and *parent* of type *dict* to store the parent of each cell.
- Here, we have used *SortedSet* instead of *PriorityQueue*. Time complexity for the *SortedSet* to add or remove an element is *O(logN)* and to check whether the set is empty or not is *O(1)*.
- First, we add our initial starting point to our *sorted_set* as well as to *visited_nodes*. We will keep running a while loop until we will get the goal node or the *sorted_set* is empty. In each iteration of the loop, we pop the minimum element from the *sorted_set* based on the following order: *f(n) >h(n) >g(n)*. If these three values are equal, we randomly pop one of the equal elements from the fringe. Then, we look into valid neighbours (Not outside of the maze and not blocked node according to agent's knowledge). If neighbour is not in the *visited_nodes*, we add this neighbour to *visited_nodes* and *sorted_set* and update its parent.
- We also update its *g(n)* and *f(n)* value. If a neighbour is present in the *visited_nodes*, we first compare the current value of *f(n)* of that neighbour to its previous *f(n)* value. If current value of *f(n)* is higher, we will just ignore it otherwise we will remove this neighbour from the *sorted_set* and add its updated value. Also, we will update value of

f(n), *g(n)* and *parent* accordingly. Using this algorithm, A* will be able to give us the shortest path to goal if it exists.

2.1.2 Repeated Forward A* search

- In repeated forward A*, we keep using A* to move towards the goal and update the agent's environment with new information we have explored as of now.
- We have created a list named *final_paths* in which we append the path returned by our A* algorithm. Also, we have stored the total number of explored nodes in each A* call. As mentioned in A*, we have the parent information for each node that has been processed by A*. We have also created another dictionary for child information.
- Now, we move from the starting node and will keep moving until we reach the goal node or we encounter a blocked cell in our path. If we have found a goal node, we just return the *final_paths* and total number of nodes explored by each A* run. If we have found a block node, we will append a path from starting position to the last unblocked node to *final_paths* and then change the starting node to the current node and call A* again from this new starting node. This loop keeps running until we will find a path to goal state or we can say that no such path to goal exists.

2.2 Question 4

In this question we have taken dimensions of maze = 101×101 and probability = $0.0 \leq p \leq 1.0$. This means that if we have a value of $p = 0.05$ for a certain maze then each cell has a 5% chance of being blocked $\implies$ that 5% of the total cells in the grid will be blocked because we are using uniform distribution to assign probability to each cell. We have tested for different probabilities in increments of 0.01 and also tested 3000 different mazes for each value of p. We get a plot as shown in Fig 2.1. Here we also define solvability as the percentage of mazes out of 3000 that were solvable.

We get an interesting graph that shows us that **solvability decreases as density (p) increases**.

We define "most mazes" as being solvable as solvability $\geq 80\%$ and thus we get $P_0 = 0.2$.

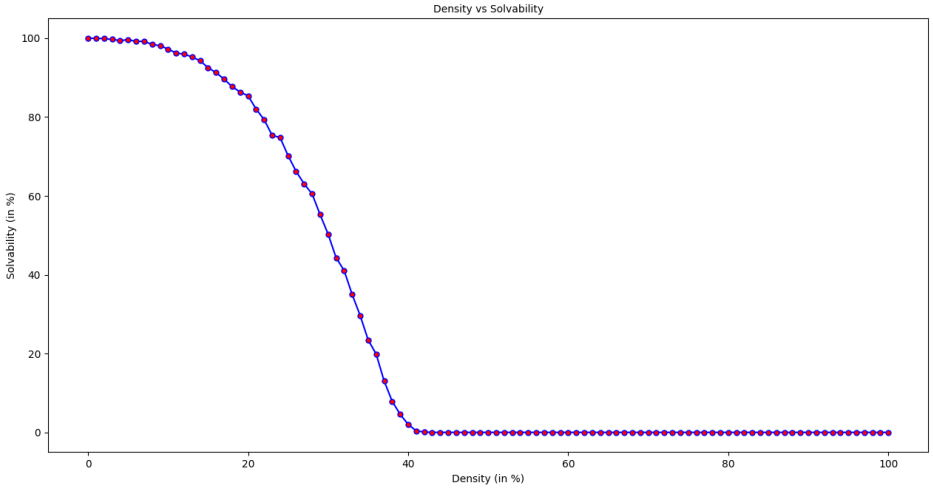

Figure 2.1: Density vs Solvability

In this scenario, A* is the best algorithm to use to test for solvability since the computation time for A* is very low as long as we use a good heuristic. A* will also always detect a path from start to finish in a solvable maze. Like A* a few other algorithms like UCS will also give us a good idea of the solvability of mazes but A* is much faster than these other algorithms.

2.3 Question 5

For grids that are solvable, we unequivocally get the **Manhattan heuristic** to be better than the other heuristics. For this question we ran tests on dimensions of maze = 101×101. We picked 101 different probabilities distributed uniformly between 0.0 and 1.0 . We also ran 3000 different mazes for each probability. We compare the given heuristics in the following ways -

2.3.1 Number of Nodes Explored

Here we check the number of nodes that each heuristic has to explore in the maze before giving us the shortest path.

Fig 2.2 shows us that on an average Manhattan distance has to explore much fewer nodes than Euclidean or Chebychev before returning the shortest path.

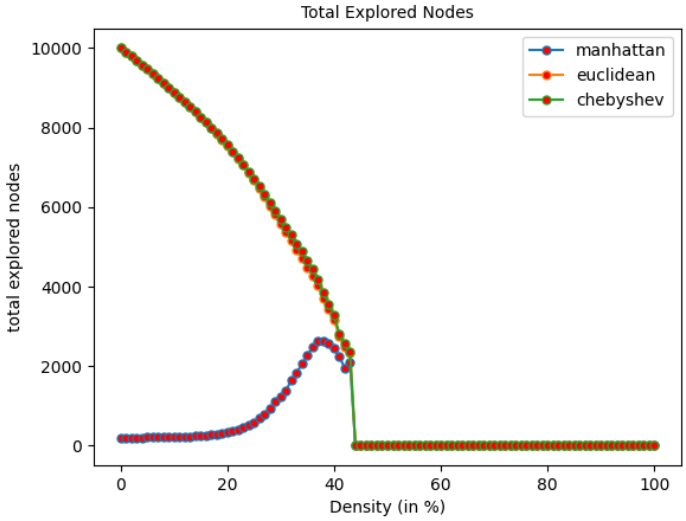

Figure 2.2: Density vs Nodes Explored

There are a few interesting things to note here -

- For very low density values, Chebychev and Euclidean explore almost the whole available grid before returning the shortest path to the goal. To explain this let us take an example of a 5×5 completely unblocked grid where we list the $f(h)$ for all the cells for each heuristic. Fig 2.3 (Manhattan distance), Fig 2.4 (Euclidean distance) and Fig 2.5 (Chebychev distance).

8	8	8	8	8
8	8	8	8	8
8	8	8	8	8
8	8	8	8	8
8	8	8	8	8

Figure 2.3: Manhattan $f(h)$

5.7	6	6.5	7.1	8
6	6.2	6.6	7.2	8
6.5	6.6	6.8	7.2	8
7.1	7.2	7.2	7.4	8
8	8	8	8	8

Figure 2.4: Euclidean $f(h)$

4	5	6	7	8
5	5	6	7	8
6	6	6	7	8
7	7	7	7	8
8	8	8	8	8

Figure 2.5: Chebychev $f(h)$

From this we can see that because we remove items from our priority queue primarily based on $f(h)$, Euclidean and Chebychev will explore almost all nodes with $f(h) \leq f(goal\_node) = 8$ which causes the number of nodes explored by Euclidean and Chebychev to be $\approx$ total number of nodes in our grid.

- As the density increases, the total nodes explored by Manhattan distance increases since it will encounter more blocks as it tries to find the optimal path. But the nodes explored by Euclidean and Chebychev decrease almost linearly since our algorithm does not explore

blocked nodes. Thus causing the number of nodes explored by these two heuristics to decrease based on the number of blocked nodes in the maze.

2.3.2 Computation Time

Here we test the amount of time each heuristic takes to finish running.

Again we see from Fig 2.6 that Manhattan distance has a significantly shorter computation time on average when compared to Euclidean or Chebychev. We get a very similar graph to Fig 2.2 because computational time is directly dependant on the total number of nodes that the algorithm has to explore. Another subtle difference that we can see is that the Euclidean Heuristic takes slightly more computation time that Chebychev. This is because the calculation of the Euclidean Heuristic requires more complex float operations, unlike in Chebychev; where we do simple integer operations.

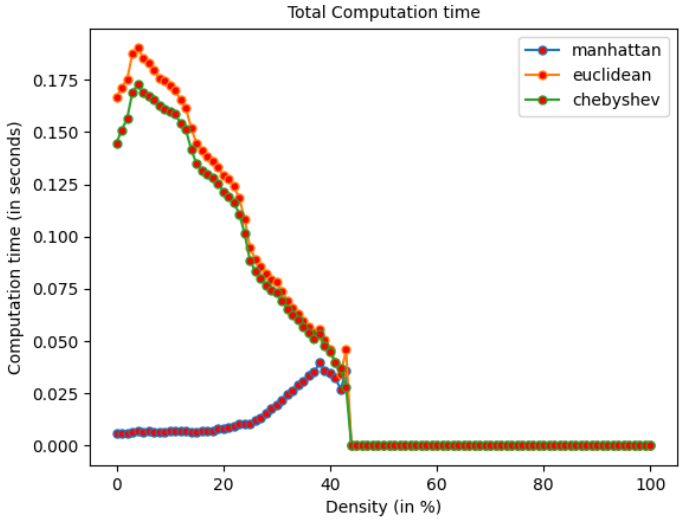

Figure 2.6: Density vs Computation Time

The reason why Manhattan outperforms Chebychev and Euclidean heuristics is because of the kind of maze we are using and the constraints that we have set on it.

- Movement is only possible in the 4 compass directions.

This constraint makes it so that the heuristic calculated by Manhattan distance is actually very close to the actual shortest path to the goal thus giving Manhattan distance an edge over most

other admissible heuristics. Also

$$h(n)_{Manhattan} \geq h(n)_{Euclidean} \geq h(n)_{Chebyshev}$$

2.3.3 Length of Trajectory

Here we define trajectory as the shortest path to goal. As we can see in Fig 2.7, all three heuristics give us very equal trajectory lengths which is expected since all 3 heuristics will be able to give us the shortest path to goal for our maze. So, length of trajectory is not a good way to compare the given heuristics.

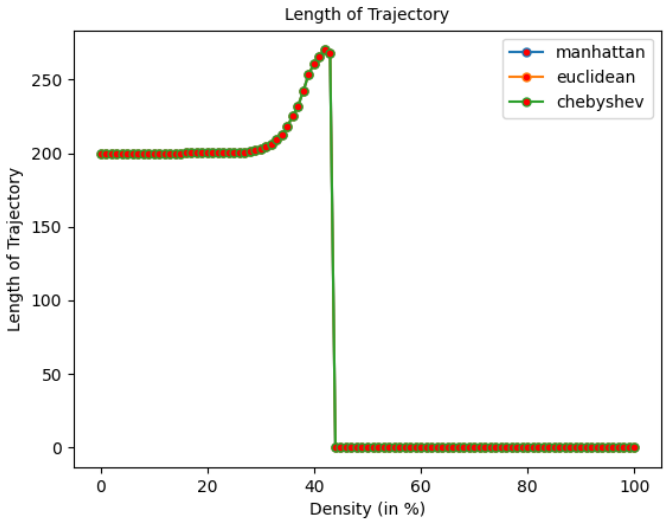

Figure 2.7: Density vs Length of Trajectory

Chapter 3

Questions 6, 7 and Extra Credit

3.1 Question 6

For the analysis of Question 6 and 7, we have taken $0 \leq p \leq 0.2$ as our values for density. We ran tests on 101 different densities in the range of 0 and 0.2 (including 0 and 0.2) and generated 1000 different grids for each density. We have taken the Manhattan heuristic as our de facto choice since it is the best performing heuristic. We get the following results -

3.1.1 Density vs Average Trajectory Length

We get the following graph -

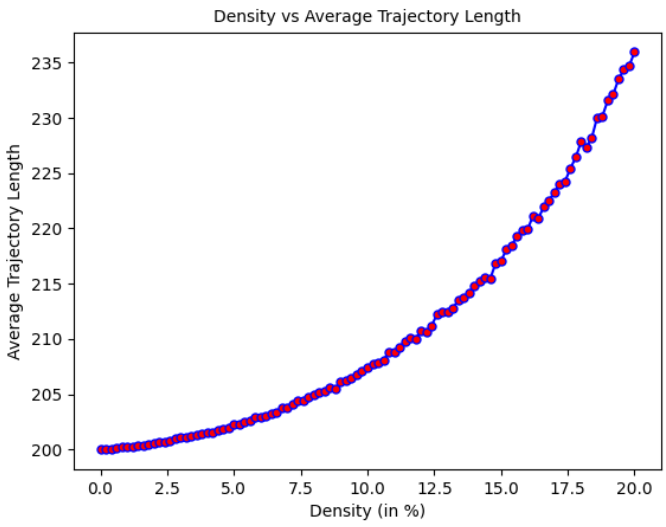

Figure 3.1: Density vs Average Trajectory Length

This is exactly as expected since as the density of our mazes increases, our agent will get blocked an increasing number of times and thus have to traverse more cells to be able to reach the goal node.

3.1.2 Density vs Average (Trajectory Length / Length of Shortest Path in Final Discovered Grid World)

We get the following graph -

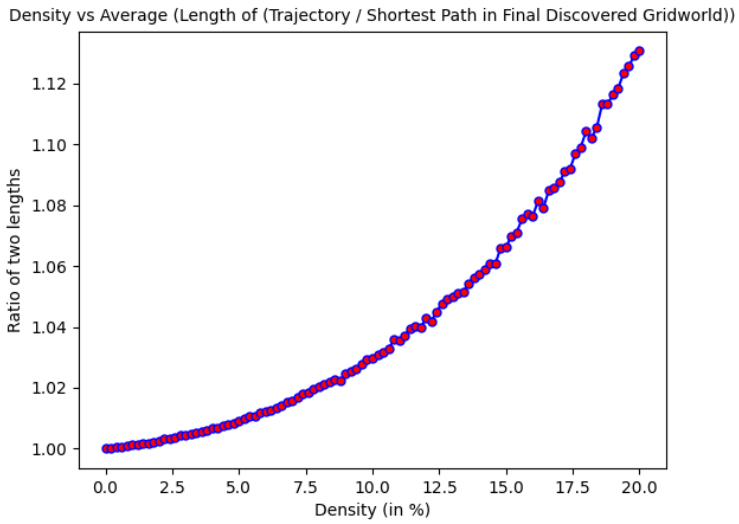

Figure 3.2: Density vs Average (Trajectory Length / Length of Shortest Path in Final Discovered Grid World)

From this graph we can see that as density increases, Trajectory length increases by around 12% more than the shortest path in the discovered grid world. This is as expected and looks very similar to Fig 3.1. We expect this because the path taken by the agent must be greater than or equal to the shortest path in the final discovered grid world. As density increases, our agent may get stuck on an increasing number of blocked cells but when calculating the shortest path in the final discovered grid world, our agent already knows the positions of a large number of blocked cells. This causes our path planned in the final discovered grid world to be on average shorter than the Length of the Trajectory as density increases.

3.1.3 Density vs Average (Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World)

We get the following graph -

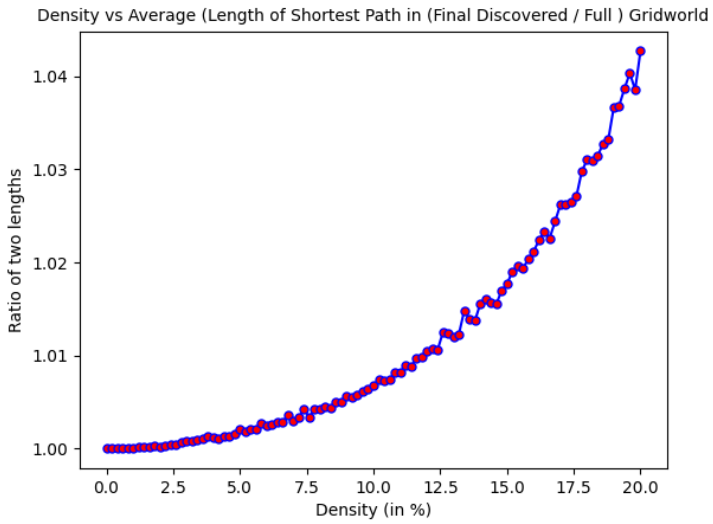

Figure 3.3: Density vs Average (Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World)

Here we see that as density increases, the length of the shortest path in the final discovered grid increases about 4% more than length of the optimal path in the full grid world. This is expected and a very nice way to show graphically; the answer to the question asked in **Question 3**

3.1.4 Density vs Average Number of Cells Processed by Repeated A*

We get the following graph -

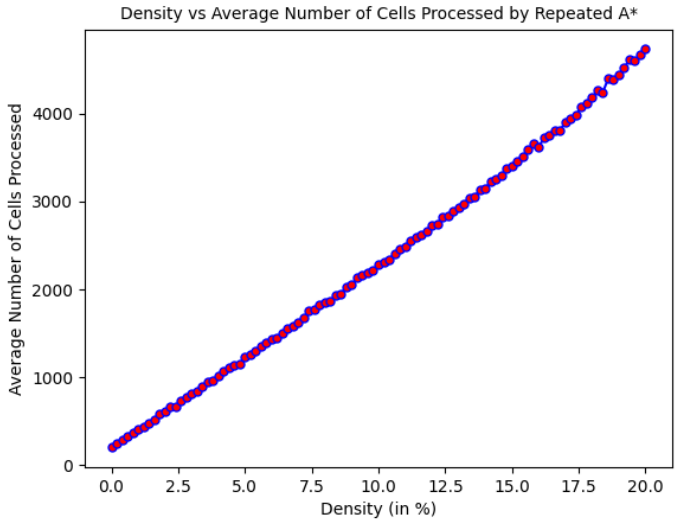

Figure 3.4: Density vs Average Number of Cells Processed by Repeated A*

Our result is interesting because this graph shows an almost perfectly linear relation between Density and average number of cells processed by repeated A*. This is expected because as the

number of blocked cells in the grid increases, the agent will get blocked and have to explore more cells to be able to find a path to the goal.

3.2 Question 7

After limiting the agents field of view to only be in the direction of motion, we re-plotted our data to get the following graphs -

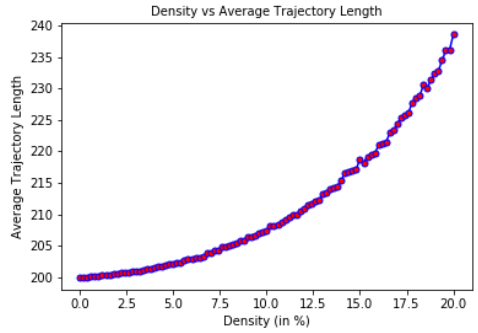

Figure 3.5: Density vs Average Trajectory Length

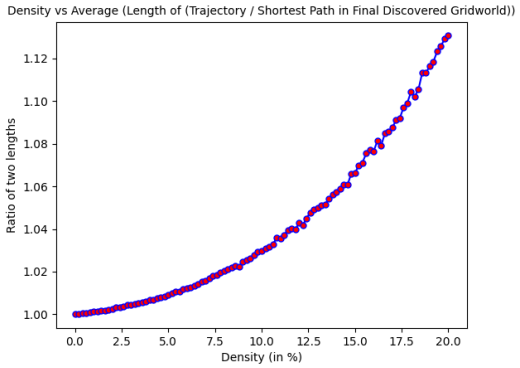

Figure 3.6: Density vs Average (Trajectory Length / Length of Shortest Path in Final Discovered Grid World)

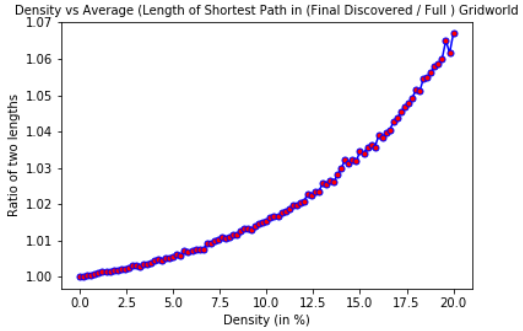

Figure 3.7: Density vs Average (Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World)

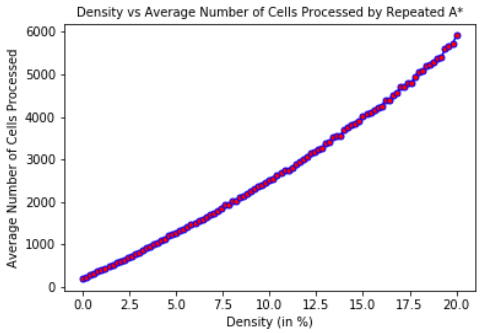

Figure 3.8: Density vs Average Number of Cells Processed by Repeated A*

These are interesting and expected results because most of the new graphs look around the same as the graphs that we obtained with a full field of view. The differences in these graphs can be largely explained by the severely limited field of view of the agent. There are also some other interesting observations that can be drawn -

- We see that the values in Fig 3.5, Fig 3.6, Fig 3.7(Question 7) are not very different than the values in Fig 3.1, Fig 3.2, Fig 3.3(Question 6). Even though we have a reduction in visibility of around 75%, our average trajectory length remains the same.

  This is because with a restricted field of view , our agent would encounter a block more

often and thus have to re-plan its path to goal multiple times. In the end, our restricted agent would slowly end up discovering the same part of the grid that our unrestricted agent did and would still end up giving us an optimal path through the maze. So, overall trajectory and shortest path in discovered grid world by both agents will be the same. This causes the above mentioned graphs to look very similar to each other.

- For the Density vs Average Number of Cells Processed by Repeated A* graphs we see that the restricted agent ends up exploring around 0 to 50% more cells than its counterpart which is mainly because the restricted agent gets much less information about the graph as it passes through it and thus must explore more of the maze before being able to find the shortest path to goal. And because our restricted agent encounters more blocked cells, it will run A* search more times than its unrestricted counterpart which in turn will cause it to explore a lot more nodes.

3.3 Extra Credit with Repeated Forward BFS

For the extra credit question at the end of the Assignment we used Repeated Forward BFS for 101 probabilities between 0, 0.2 and tested each probability on 100 graphs. The reason why we tested on much fewer graphs than in our other Questions is because the run-time for repeated forward BFS increases and becomes very high as density increases on 101 $\times$ 101 grids.

3.3.1 Question 6

On comparing these with the results we obtained for Question 6 when using Repeated Forward A*, we can see a few interesting things -

- The first 3 figures for Question 6 and Extra credit Question 6, (Fig 3.1, Fig 3.2, Fig 3.3 and Fig 3.9, Fig 3.10, Fig 3.11) all seem to be almost equal with a maximum of 1-2% difference. This difference can be attributed to our smaller sample size of only 100 graphs for this extra credit question.

- On comparing the 4^{th} graphs (Fig 3.4 and Fig 3.12), we come across something startling. Repeated Forward BFS explores **50 to 70 times** more nodes than Repeated Forward A*. Which in turn means that Repeated Forward BFS takes a lot more time to run than Repeated Forward A* search.

Following are the graphs of our results -

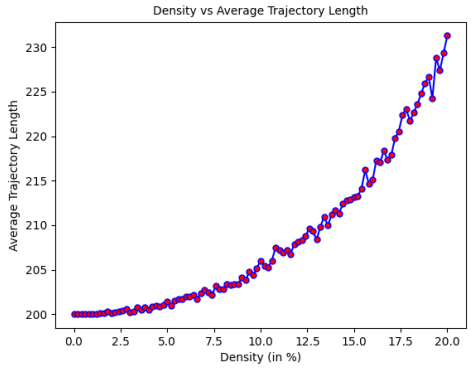

Figure 3.9: Density vs Average Trajectory Length

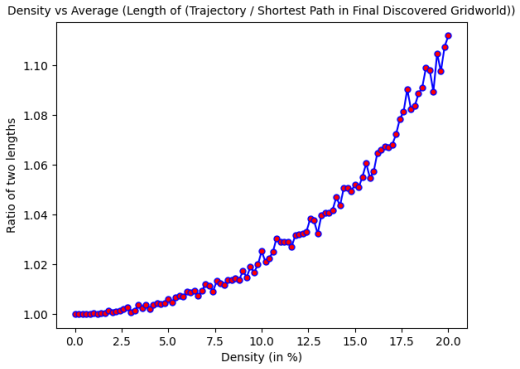

Figure 3.10: Density vs Average (Trajectory Length / Length of Shortest Path in Final Discovered Grid World)

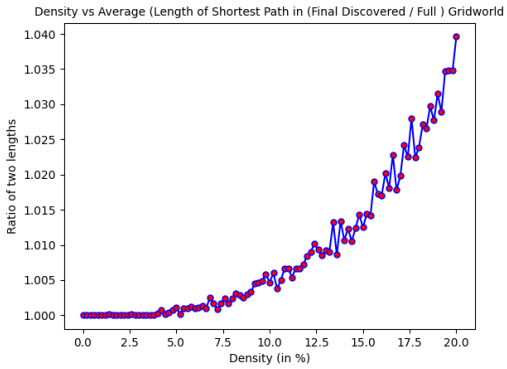

Figure 3.11: Density vs Average (Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World)

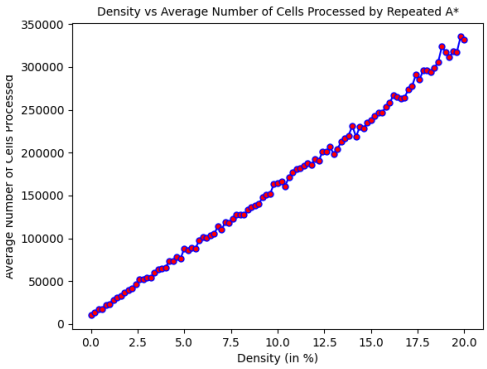

Figure 3.12: Density vs Average Number of Cells Processed by Repeated Forward BFS

Before forming any conclusions let us take a look at -

3.3.2 Question 7

On comparing these results with the graphs from Question 7 we can see that the results of our comparison are very similar to the ones above,

- The first 3 figures (Fig 3.5, Fig 3.6, Fig 3.7 and Fig 3.13, Fig 3.14, Fig 3.15) show us that Restricted Repeated forward BFS and Restricted Repeated Forward A* are almost equal.
- On comparing the 4^{th} graphs (Fig 3.8 and Fig 3.16), we see that Restricted Repeated Forward BFS explores **45 to 66 times** more nodes than Repeated Forward A*. Which in turn means that Repeated Forward BFS takes a lot more time to run than Repeated Forward A* search.

From the above two experiments we can firmly conclude that Repeated Forward BFS does not

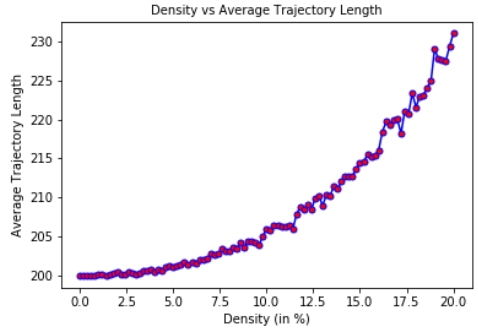

Figure 3.13: Density vs Average Trajectory Length

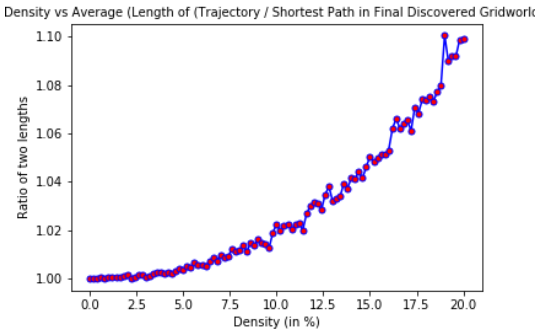

Figure 3.14: Density vs Average (Trajectory Length / Length of Shortest Path in Final Discovered Grid World)

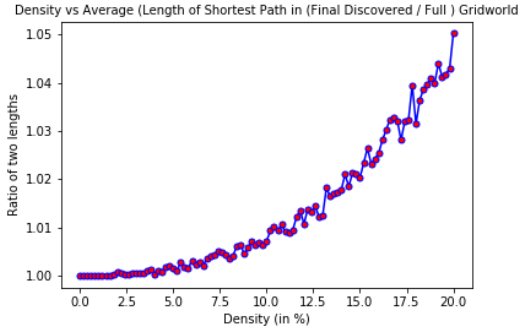

Figure 3.15: Density vs Average (Length of Shortest Path in Final Discovered Grid World / Length of Shortest Path in Full Grid World)

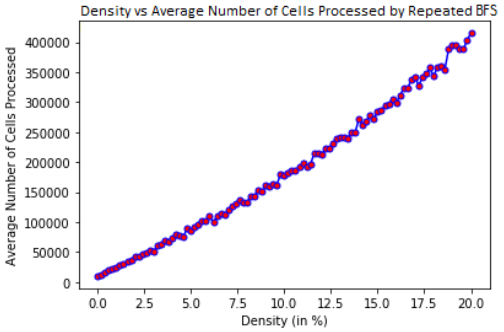

Figure 3.16: Density vs Average Number of Cells Processed by Repeated Forward BFS

give us any results that are better than Repeated Forward A*, the extremely high computation time on large grids makes Repeated Forward BFS very undesirable to use.

One of the only cases where Repeated Forward BFS seems useful is if the heuristic that we are using with Repeated Forward A* is inconsistent.

Chapter 4

Questions 8 and 9

4.1 Question 8

For Question 8 we had a hard time figuring out what kind of backtracking to use and when we should make our algorithm backtrack. The main issue that we were facing was that any complicated optimal backtracking that we came up with would just increase the run-time of our algorithm. This is because testing for the optimal amount of backtracking always took more computation time than it cut down.

To deal with this problem, we decided to implement backtracking in the exact case that is talked about in the Question (As show in Fig 4.1). Whenever the agent noticed it was walking into a tunnel, we would note down the start position of that tunnel (1,0 in this case). If the agent reached a dead end while inside a tunnel, instead of re-running A* search from inside the tunnel, we would manually make the agent backtrack to the start of the tunnel (1,0), add the length of the backtrack to our trajectory length and then re-run A* search from the start of the tunnel now armed with the knowledge of the blocked tunnel.

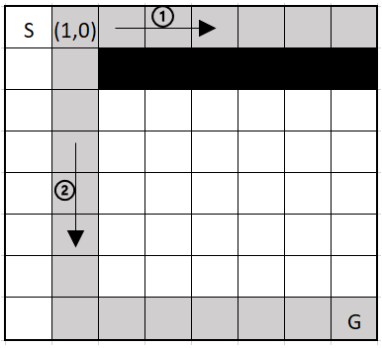

Figure 4.1: Sample Tunnel

We get the following graphs as results when comparing A* search with and without backtracking. For these tests we ran tests on 101 probabilities between 0 and 0.4. We also tested each probability on 300 different grids.

4.1.1 Trajectory Length, Computation time and Number of Nodes Processed

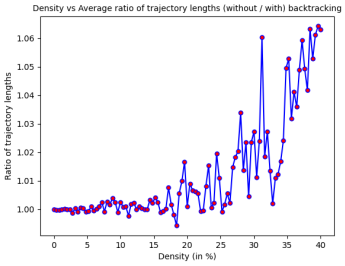

Figure 4.2: Ratio (Without/With backtracking) of Trajectory Lengths

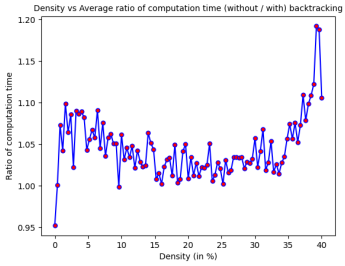

Figure 4.3: Ratio (Without/With backtracking) of Computation Time

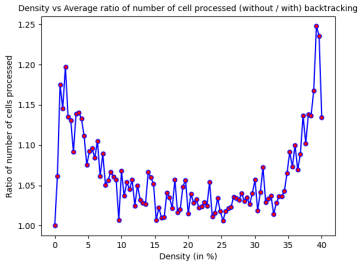

Figure 4.4: Ratio (Without/With backtracking) of Number of Cells Processed

- As we can see from the ratio of trajectory length (Fig 4.2), our backtracking algorithm does not change the length of final trajectories very much at all. This is because we always add the number of cells backtracked to our trajectory length.
- The ratio of computation time also shows us clearly that as density of the graph increases, the computation time without backtracking is around 20% higher than the computation time with backtracking.
- The reason why computation time decreases is because as shown in (Fig 4.4), the number of cells processed by our algorithm with backtracking is significantly lower than the number of cells without backtracking. Our manual backtracking makes it so that those extra cells do not need to be processed by our fringe when we run A* search again. Making A* with backtracking faster than A* without backtracking.

We also obtained an interesting graph showing how the average number of backtracks we execute in our maze increases with increasing density -

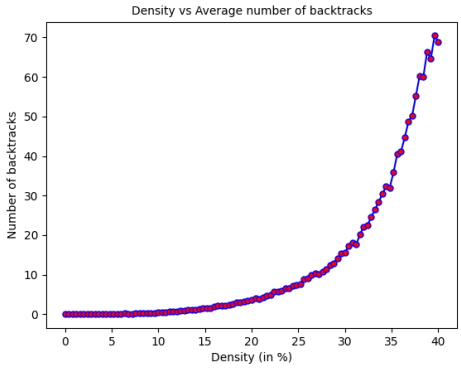

Figure 4.5: Average backtracks vs Density

4.2 Question 9

For this question we wanted to test a few different kinds of weighted inadmissible heuristics. For the following tests we plotted our graphs to compare Heuristics $\times$ 1 and Heuristics $\times$ 2. We tested on 101 different densities between 0, 0.2 and We tested on 100 different Grid Worlds for each probability and obtained the following graphs as results -

4.2.1 Trajectory Length

Here we wanted to compare the trajectory lengths for the normal and weighted heuristics vs Density.

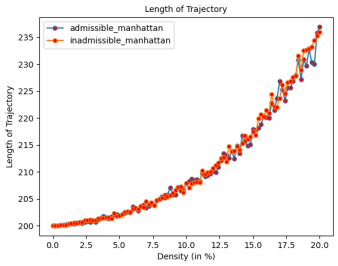

Figure 4.6: Original Manhattan, Weighted Manhattan VS Density

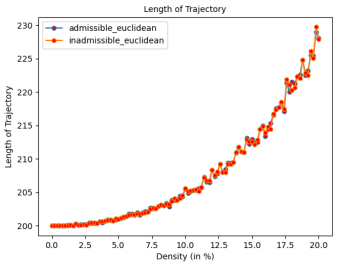

Figure 4.7: Original Euclidean, Weighted Euclidean VS Density

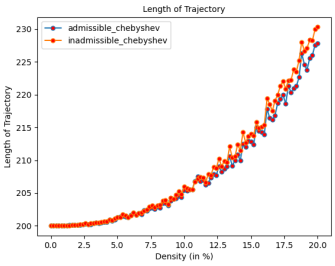

Figure 4.8: Original Chebychev, Weighted Chebychev VS Density

From our results we can see that the length of Trajectory increases slightly for all of our heuristics. This is expected because the new Heuristic that is being used is inadmissible. This makes it so that the heuristic actually overestimates the distance to goal from the current cell, which makes our A* algorithm take a longer than necessary path to the goal.

4.2.2 Total Number of Cells Processed (Admissible vs Inadmissible)

Here we wanted to compare the number of nodes processed for the normal and weighted heuristics vs Density.

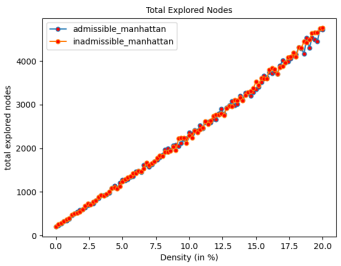

Figure 4.9: Nodes Explored by Original Manhattan, Weighted Manhattan VS Density

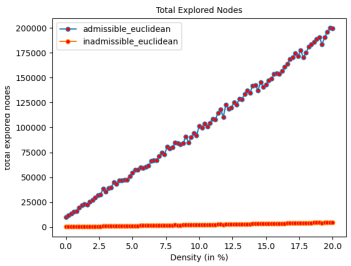

Figure 4.10: Nodes Explored by Original Euclidean, Weighted Euclidean VS Density

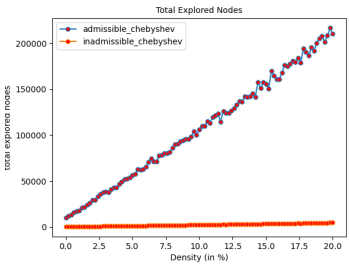

Figure 4.11: Nodes Explored by Original Chebychev, Weighted Chebychev VS Density

We see some of the advantages of using Weighted heuristics. The weighting causes a reduction in number of nodes explored by a very big margin for Euclidean and Chebychev heuristics, and causes little to no difference in number of nodes explored for the Manhattan heuristic. To explain this let us take an example of a 5×5 grid with Weighted $f(n)$ values.

16	15	14	13	12
15	14	13	12	11
14	13	12	11	10
13	12	11	10	9
12	11	10	9	8

Figure 4.12: Weighted Manhattan F(n)

11.3	11	10.9	11.3	12
11	10.5	10.2	10.3	11
10.9	10.2	9.66	9.47	10
11.3	10.3	9.47	8.83	9
12	11	10	9	8

Figure 4.13: Weighted Euclidean F(n)

8	9	10	11	12
9	8	9	10	11
10	9	8	9	10
11	10	9	8	9
12	11	10	9	8

Figure 4.14: Weighted Chebychev F(n)

As we can see from these grids for Manhattan, Euclidean and Chebychev distances, our grids now promote exploration along the diagonal between the start node and the goal node a lot more than before. This makes it so that a lot less nodes need to be explored on average before reaching the goal node.

An interesting thing to note is how much lower the number of explored nodes are for inadmissible Euclidean and Chebychev when compared to admissible Euclidean and Chebychev. We can also see that Inadmissible Euclidean and Chebychev are almost as efficient as Admissible Manhattan heuristic.

4.2.3 Ratio of Computation time

Here we wanted to compare the computation time for the normal and weighted heuristics vs Density. Since the difference in computation time is very large, we have decided to show this difference as a ratio of computation times.

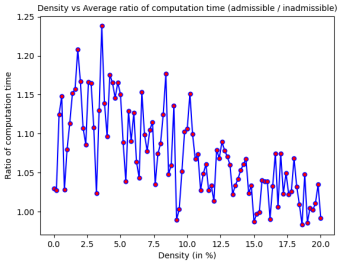

Figure 4.15: Original Manhattan/Weighted Manhattan VS Density

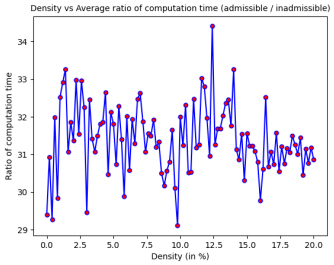

Figure 4.16: Original Euclidean VS Euclidean/Weighted Density

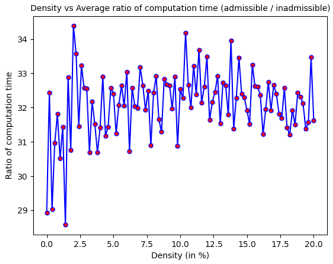

Figure 4.17: Original Chebychev VS Chebychev/Weighted Density

As we can see, the results we have obtained are very similar to those shown for the total explored nodes. This is because Computation time is directly proportional to the number of nodes explored by our algorithm. Which is why the reasoning for these ratios in computation time is very similar to our reasoning for the difference in number of nodes explored. All three graphs show that using an inadmissible heuristics speeds up computation time significantly for all of our heuristics.

Apart from these tests, we ran tests by using a combination of heuristics and also tried different weights for each heuristic. None of the results were as promising as the ones shown above which is why we have demonstrated the advantages of weighted heuristics with the above given data.

Appendix A

A* Search Code[1]

```
def astar_search(maze: Maze, start_pos: tuple, goal_pos: tuple):
"""
Function to compute A* search
:param maze: Maze object
:param start_pos: starting position of the maze from where we want to start A*
                                        search
:param goal_pos: Goal state (position) where we want to reach
:return: Returning the path from goal_pos to start_pos if it exists
"""

# Initialize a set for visited nodes
visited_nodes = set()

# Initialize a sorted set to pop least value element from the set
sorted_set = SortedSet()

# Initialize a dictionary to store a random value assigned to each node. This
                                        dictionary would help us know the value
                                         of a node when we want to remove it
                                        from the sorted set.
node_to_random_number_mapping = dict()

# Initialize another dictionary to store parent information
parents = dict()
```

[1]We have only put code for A* and Repeated Forward A* here, please refer to the attached winrar file for the code used in Questions 4 to 9

```
21
22 # Initialize g and f for the starting position
23 maze.maze[start_pos[0]][start_pos[1]].g = 0
24 maze.maze[start_pos[0]][start_pos[1]].f = maze.maze[start_pos[0]][start_pos[1]]
                                            .h
25
26 # Assigning a random number to start position to the starting position and
                                            adding to visited nodes
27 node_to_random_number_mapping[start_pos] = random.uniform(0, 1)
28 visited_nodes.add(start_pos)
29
30 # Add start position node into the sorted set. We are giving priority to f(n),
                                            h(n), and g(n) in the decreasing order.
                                             Push random number for random
                                            selection if there is conflict between
                                            two nodes (If f(n), g(n), and h(n) are
                                            same for two nodes)
31
32 sorted_set.add(((maze.maze[start_pos[0]][start_pos[1]].f,
33 maze.maze[start_pos[0]][start_pos[1]].h, maze.maze[start_pos[0]][start_pos[1]].
                                            g, node_to_random_number_mapping[
                                            start_pos]), start_pos))
34 parents[start_pos] = start_pos
35 num_explored_nodes = 0
36
37 # Running the loop until we reach our goal state or the sorted set is empty
38 while sorted_set.__len__() != 0:
39     # Popping first (shortest) element from the sorted set
40     current_node = sorted_set.pop(index=0)
41
42     # Increase the number of explored nodes
43     num_explored_nodes += 1
44
45     # If we have found the goal position, we can return parents and total
                                            explored nodes
46     if current_node[1] == goal_pos:
47         return parents, num_explored_nodes
48
49     # Otherwise, we need to iterate through each child of the current node
```

```
50      for val in range(len(X)):
51          neighbour = (current_node[1][0] + X[val], current_node[1][1] + Y[val])
52
53          # Neighbour should not go outside our maze and it should not be blocked
                                        if we want to visit that particular
                                        neighbour
54          if check(neighbour, maze.num_cols, maze.num_rows) and (
55                  not maze.maze[neighbour[0]][neighbour[1]].is_blocked):
56
57              # If neighbour is being visited first time, we should change its g(
                                        n) and f(n) accordingly. Also, we need
                                        to assign a random value to it for the
                                        time of conflict. In the end, we will
                                        add all those things into the sorted
                                        set and update its parent
58              if neighbour not in visited_nodes:
59                  maze.maze[neighbour[0]][neighbour[1]].g = maze.maze[
                                        current_node[1][0]][current_node[1][1]]
                                        .g + 1
60                  maze.maze[neighbour[0]][neighbour[1]].f = maze.maze[neighbour[0
                                        ]][neighbour[1]].g +  maze.maze[
                                        neighbour[0]][neighbour[1]].h
61                  node_to_random_number_mapping[neighbour] = random.uniform(0, 1)
62                  visited_nodes.add(neighbour)
63                  sorted_set.add(((maze.maze[neighbour[0]][neighbour[1]].f, maze.
                                        maze[neighbour[0]][neighbour[1]].h,maze
                                        .maze[neighbour[0]][neighbour[1]].g,
                                        node_to_random_number_mapping[neighbour
                                        ]),neighbour))
64                  parents[neighbour] = current_node[1]
65
66              # If a particular neighbour is already visited, we should compare
                                        its f(n) value to its previous f(n)
                                        value. If current computed f(n) value
                                        is less than the previously computed
                                        value, we should remove previously
                                        computed value and add new value to the
                                         sorted set
67              else:
```

```
68                  neighbour_g = maze.maze[current_node[1][0]][current_node[1][1]]
                                  .g + 1
69                  neighbour_f = maze.maze[neighbour[0]][neighbour[1]].h +
                                  neighbour_g
70                  if neighbour_f < maze.maze[neighbour[0]][neighbour[1]].f:
71
72                      # The following if condition is needed only when the
                                  heuristic is inadmissible otherwise a
                                  neighbour has to be in the sorted set
                                  if we are able to find out less value
                                  of f(n) for that particular neighbour
73                      if ((maze.maze[neighbour[0]][neighbour[1]].f, maze.maze[
                                  neighbour[0]][neighbour[1]].h,
74                      maze.maze[neighbour[0]][neighbour[1]].g,
                                  node_to_random_number_mapping[neighbour
                                  ]),neighbour) in sorted_set:
75                          sorted_set.remove(((maze.maze[neighbour[0]][neighbour[1
                                  ]].f, maze.maze[neighbour[0]][neighbour
                                  [1]].h,maze.maze[neighbour[0]][
                                  neighbour[1]].g,
                                  node_to_random_number_mapping[neighbour
                                  ]),neighbour))
76                      maze.maze[neighbour[0]][neighbour[1]].g = neighbour_g
77                      maze.maze[neighbour[0]][neighbour[1]].f = neighbour_f
78                      node_to_random_number_mapping[neighbour] = random.uniform(0
                                  , 1)
79                      sorted_set.add(((maze.maze[neighbour[0]][neighbour[1]].f,
                                  maze.maze[neighbour[0]][neighbour[1]].h
                                  ,maze.maze[neighbour[0]][neighbour[1]].
                                  g, node_to_random_number_mapping[
                                  neighbour]),neighbour))
80                      parents[neighbour] = current_node[1]
81
82  return parents, num_explored_nodes
```

Appendix B

Repeated Forward A* Search Code

```
1
2
3  def repeated_forward(maze: Maze, maze_array: np.array, start_pos: tuple,
                                        goal_pos: tuple,
                                        is_field_of_view_explored: bool = True,
                                         backtrack_length: int = 0, algorithm:
                                        str = 'astar', is_backtrack_strategy_on
                                        : bool = False):
4      """
5      This is the repeated forward function which can be used with any algorithm
                                        (astar or bfs). This function will
                                        repeatedly call corresponding algorithm
                                         function until it reaches goal or
                                        finds out there is no path till goal.
6      :param maze: Maze array of agent
7      :param maze_array: Original (Full) Maze array
8      :param start_pos: starting position of the maze from where agent want to
                                        start
9      :param goal_pos: goal state where agent want to reach
10     :param is_field_of_view_explored: It will explore field of view if this
                                        attribute is true otherwise it won't.
11     :param backtrack_length: How many times you want to backtrack for each run.
12     :param algorithm: Either A* or BFS
13     :param is_backtrack_strategy_on: If you want to run strategy 2, this
                                        attribute should be set to true
```

```
14      :return: This function will return final paths on which agent moved to
                                        reach goal or empty list if agent can't
                                         find path to goal. Second is total
                                        number of processed nodes while running
                                         the algorithm.
15      """
16
17      # defining the following two attributes to find which would be useful to
                                        return values
18      final_paths = list()
19      total_explored_nodes = 0
20      num_backtracks = 0
21
22      # Running the while loop until we will get a path from start_pos to
                                        goal_pos or we have figured out there
                                        is no path from start_pos to goal_pos
23      while True:
24          # Choose which algorithm you want to use for search
25          if algorithm == 'astar':
26              parents, num_explored_nodes = astar_search(maze, start_pos,
                                        goal_pos)
27          elif algorithm == 'bfs':
28              parents, num_explored_nodes = bfs_search(maze, start_pos, goal_pos)
29          else:
30              raise Exception("algorithm should be either astar or bfs")
31
32          # Adding up number of nodes explored (processed) in the last call to
                                        algorithm.
33          total_explored_nodes += num_explored_nodes
34
35          # If goal_pos doesn't exist in parents which means path is not
                                        available so returning empty list.
36          if goal_pos not in parents:
37              return list(), total_explored_nodes, num_backtracks
38
39          # parents contains parent of each node through path from start_pos to
                                        goal_pos. To store path from start_pos
                                        to goal_pos, we need to store child of
                                        each node starting from start_pos.
```

```
40          cur_pos = goal_pos
41          children = dict()
42
43          children[cur_pos] = cur_pos
44
45          # Storing child of each node so we can iterate from start_pos to
                                        goal_pos
46          while cur_pos != parents[cur_pos]:
47              children[parents[cur_pos]] = cur_pos
48              cur_pos = parents[cur_pos]
49
50          # Setting current position to starting position so we can start
                                        iterating from start_pos
51          cur_pos = start_pos
52
53          current_path = [cur_pos]
54          last_cell_which_is_not_in_dead_end = cur_pos
55
56          # Iterating from start_pos to goal_pos if we won't get any blocks in
                                        between otherwise we are terminating
                                        the iteration.
57          while cur_pos != children[cur_pos]:
58
59              if is_backtrack_strategy_on:
60                  path_exist_from_the_last_point = 0
61
62              # Explore the field of view and update the blocked nodes if there's
                                         any in the path.
63              if is_field_of_view_explored:
64                  for ind in range(len(X)):
65                      neighbour = (cur_pos[0] + X[ind], cur_pos[1] + Y[ind])
66                      if (check(neighbour, NUM_COLS, NUM_ROWS)) and (maze_array[
                                        neighbour[0]][neighbour[1]] == 1):
67                          maze.maze[neighbour[0]][neighbour[1]].is_blocked = True
68
69                      # Here, we are finding whether the current node is a part
                                        of the dead end or not. If there is a
                                        path exists other than its child and
                                        parent, then this node should not be
```

```
                                    part of dead end because there is
                                    another path available which you can
                                    explore.
70                  if is_backtrack_strategy_on and (check(neighbour, NUM_COLS,
                                     NUM_ROWS)) and \
71                          (children[cur_pos] != neighbour) and (parents[
                                    cur_pos] != neighbour) and \
72                          (maze_array[neighbour[0]][neighbour[1]] == 0):
73                      path_exist_from_the_last_point += 1
74
75              if is_backtrack_strategy_on:
76
77                  # If we can find such a node which we can explore later using
                                    current node, then this node should not
                                     be part of the dead end path.
78                  if path_exist_from_the_last_point > 0:
79                      last_cell_which_is_not_in_dead_end = cur_pos
80
81              # If we encounter any block in the path, we have to terminate the
                                    iteration
82              if maze_array[children[cur_pos][0]][children[cur_pos][1]] == 1:
83                  break
84              cur_pos = children[cur_pos]
85              current_path.append(cur_pos)
86
87          # If we are able to find the goal state, we should return the path and
                                    total explored nodes.
88          if cur_pos == goal_pos:
89              final_paths.append(current_path)
90              return final_paths, total_explored_nodes, num_backtracks
91          else:
92
93              # Change the start node to last unblocked node and backtrack if it
                                    is set to any positive integer.
94              maze.maze[children[cur_pos][0]][children[cur_pos][1]].is_blocked =
                                    True
95              num_backtrack = min(len(current_path) - 1, backtrack_length)
96              cur_pos = current_path[-1]
97
```

```
98              while num_backtrack > 0:
99                  cur_pos = parents[cur_pos]
100                 current_path.append(cur_pos)
101                 num_backtrack -= 1
102
103             if is_backtrack_strategy_on:
104
105                 # We keep backtracking cell until we reached a cell from where
                                        we can explore a new path. Also, we are
                                         manually blocking those dead end nodes
                                         because they are not useful anymore.
106                 while cur_pos != last_cell_which_is_not_in_dead_end:
107                     num_backtracks += 1
108                     maze.maze[cur_pos[0]][cur_pos[1]].is_blocked = True
109                     cur_pos = parents[cur_pos]
110                     current_path.append(cur_pos)
111
112             final_paths.append(current_path)
113             start_pos = cur_pos
```