CS 520 : Project 1 - Voyage Into the Unknown

Farhan Kapadia - fk232, Jayashree Domala - jd1552

1 Question 1: Why does re-planning only occur when blocks are discovered on the current path? Why not whenever knowledge of the environment is updated?

The aim of the agent is to reach the goal state from the start state using the shortest path. While finding the route during the planning phase, it is possible that the agent finds the shortest path with no blocked cells and reaches the goal state in the most optimum way. In this case if the agent tries to consider the environment too while traversing and re-plans every time it encounters blocks in its field of view, then this optimum result could be lost and the running time would increase. The re-planning occurs only when blocks are discovered on the current path because while traversing the path, only when blocked cells are encountered it makes sense to re-plan otherwise the computation cost increases when considering the environment too.

2 Question 2: Will the agent ever get stuck in a solvable maze? Why or why not?

The maze is not solvable if is there no path to the goal and all the cells to reach the goal state from the start state have been exhausted in the fringe. The agent will never get stuck in a solvable maze. The reason being that the algorithm is designed is such a way that even if the agent encounters a blocked cell it will re-route to obtain a new path and if all its forward routes are blocked, it will backtrack to a previous state and re-plan from there so as to reach the goal state. Therefore, if a path exists to the goal state, the agent will always find it with this algorithm.

3 Question 3: Once the agent reaches the target, consider re-solving the now discovered gridworld for the shortest path (eliminating any backtracking that may have occurred). Will this be an optimal path in the complete gridworld? Argue for, or give a counter example.

Once the agent reaches the target, re-solving the gridworld for the shortest path by eliminating backtracking that may have previously occurred may not give an optimal path. This is because the agent only has knowledge of the path and the neighbours it encountered during traversal but no knowledge of the cells it didn't encounter during traversal. It could be possible that there is a better and more optimal path through the unvisited grid world that the agent hadn't discovered previously and is unaware of.

4 Question 4: Solvability - A gridworld is solvable if it has a clear path from start to goal nodes. How does solvability depend on p? Given dim = 101, how does solvability depend on p? For a range of p values, estimate the probability that a maze will be solvable by generating multiple environments and checking them for solvability. Plot density vs solvability, and try to identify as accurately as you can the threshold p0 where for p < p0, most mazes are solvable, but p > p0, most mazes are not solvable. Is A* the best search algorithm to use here, to test for solvability? Note for this problem you may assume that the entire gridworld is known, and hence only needs to be searched once each.

The range of p values considered go from 0.01 to 0.99 and the values are equally spaced in this range. For each value of probability, 30 gridworlds are created of dimension 101. These gridworlds are solved to get the value of solvability and accordingly the graph is plotted. The dependence of solvability and probability is depicted in the figure 1. For dim=101, as probability increases the solvability decreases. After a point, the value of solvability becomes stagnant. The threshold p0 from the graph turns out to be 0.23 at which the solvability is around 70%. The maze is solvable if p<=0.23 and not solvable if p>0.23. To test for solvability, A* is the best search algorithm to use here. When compared to other search algorithms like BFS and DFS, A* algorithm maintains a fringe which is a priority queue. On the other hand, BFS uses a normal FIFO queue and DFS uses a stack. This means that that nodes are popped off the fringe in the most optimal manner and not randomly throughout the search space.

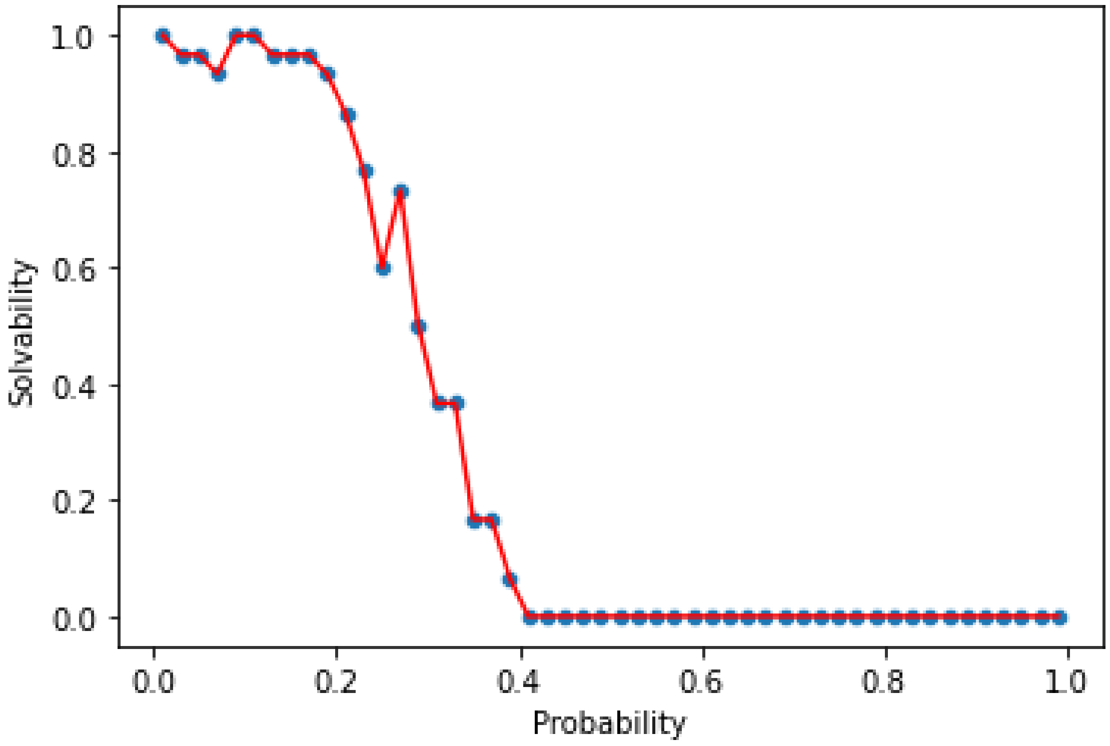

Figure 1: Solvalibity v/s Probability

5 Question 5: Heuristics- (a) Among environments that are solvable, is one heuristic uniformly better than the other for running A$*$? Consider the following heuristics: – Euclidean Distance – Manhattan Distance – Chebyshev Distance How can they be compared? Plot the relevant data and justify your conclusions. Again, you may take each gridworld as known, and thus only search once.

Among the environments that are solvable, the Manhattan distance heuristic is uniformly better than the other two heuristics- Euclidean distance and Chebyshev distance. They have been compared by the number of nodes processed for each value of density from 0.01 to p_0=0.23 . As seen in figure 2, the number of nodes processed by Manhattan is significantly less for all the distances as compared to Euclidean and Chebyshev. It is also worth noting that the Euclidean distance heuristic outperforms the Chebyshev distance heuristic by a very small margin.

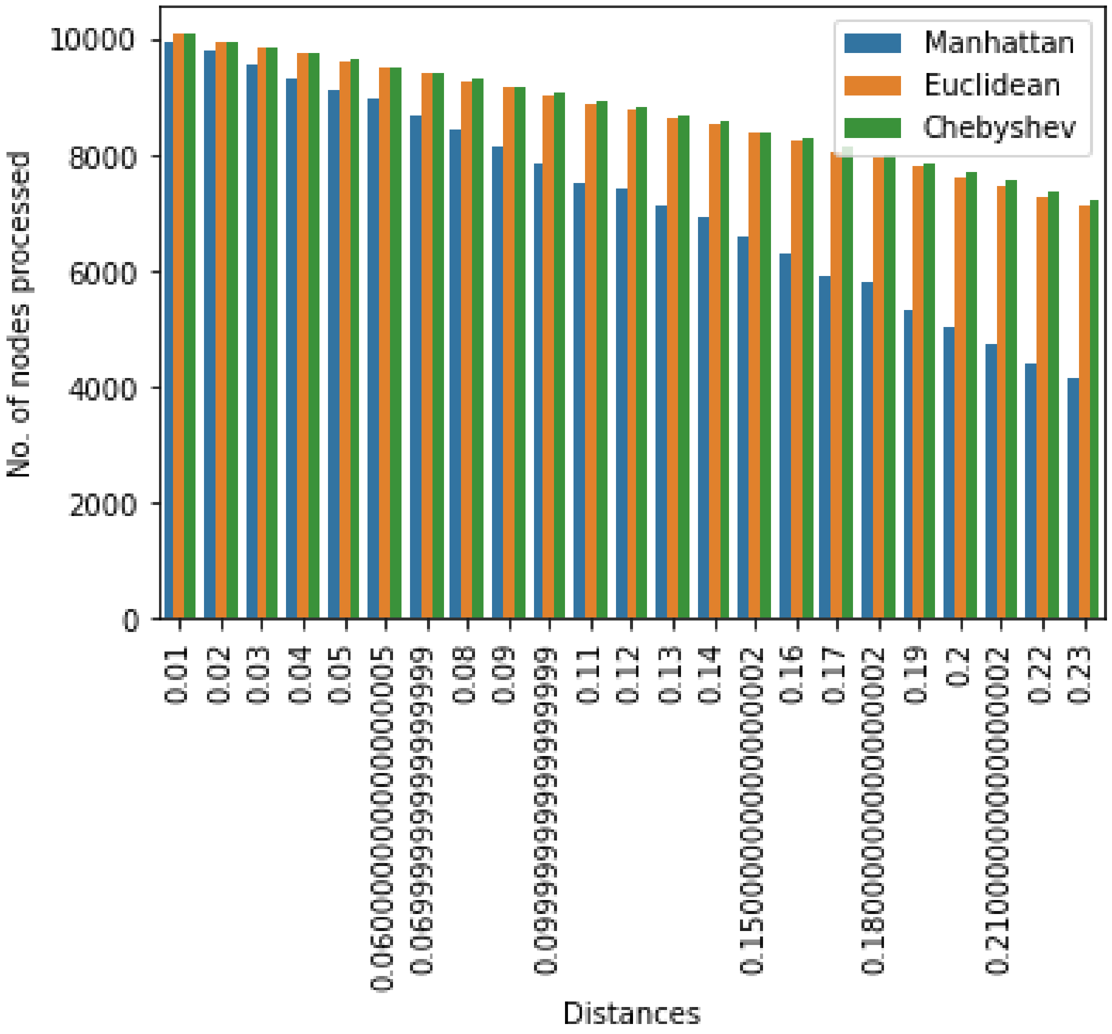

Figure 2: No. of nodes processed v/s Distances

6 Question 6: Performance Taking dim = 101, for a range of density p values from 0 to min(p0, 0.33), and the heuristic chosen as best in Q5, repeatedly generate gridworlds and solve them using Repeated Forward A*. Use as the field of view each immediately adjacent cell in the compass directions. Generate plots of the following data: – Density vs Average Trajectory Length – Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Gridworld) – Density vs Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) – Density vs Average Number of Cells Processed by Repeated A*. Discuss your results. Are they as you expected? Explain.

Figure 3 depicts the relationship between the density and the average trajectory length after applying repeated A*. From the graph we can make out that as the density increases the average trajectory length also generally increases. Though it is worth noting that there are a few anomalies where there is a dip at few points like for density=0.09 and density=0.17 approximately. The results are as expected because as the density increases, the number of blocks in the grid increases and hence the A* is called more times, which results in a lot of re-planning, thereby increasing the average trajectory length.

The figure 4 depicts the relation between the density and the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld). As the density increases, the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) also increases. But there are dips at the same points of density as seen in the graph prior. The results are as expected because as the density increases, the number of blocks increases due to which the length of trajectory increases.

Figure 5 depicts the relation between the density and Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) where in as the density increases, the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) is steady for the initial values of density and then has a downward trend. The result is as expected because initially when the density is less, the blocks are less and therefore the probability of the length of the path being equal in each of the case viz agent with acquired knowledge and full knowledge of the gridworld will be more. But as the density increases, a sharp dip is observed because the number of blocks in the grid increase due to which the length of shortest path in the discovered gridworld is more than the length of the shortest path in full gridworld.

Figure 6 depicts the relation between density and Average Number of Cells Processed by Repeated A*. As the density increases, a general upward trend is observed. Although there are sharp drops at some points of density showing some anomaly. This is again an expected result since the number of blocks encountered increases with density.

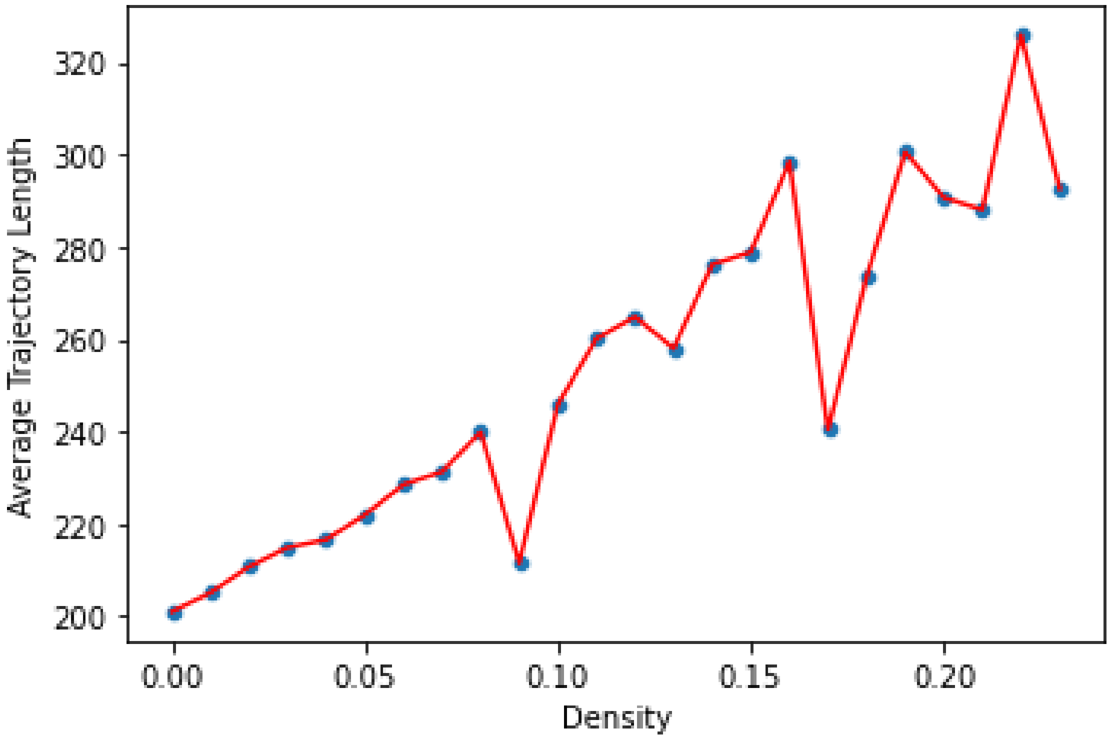

Figure 3: Density v/s Average trajectory length

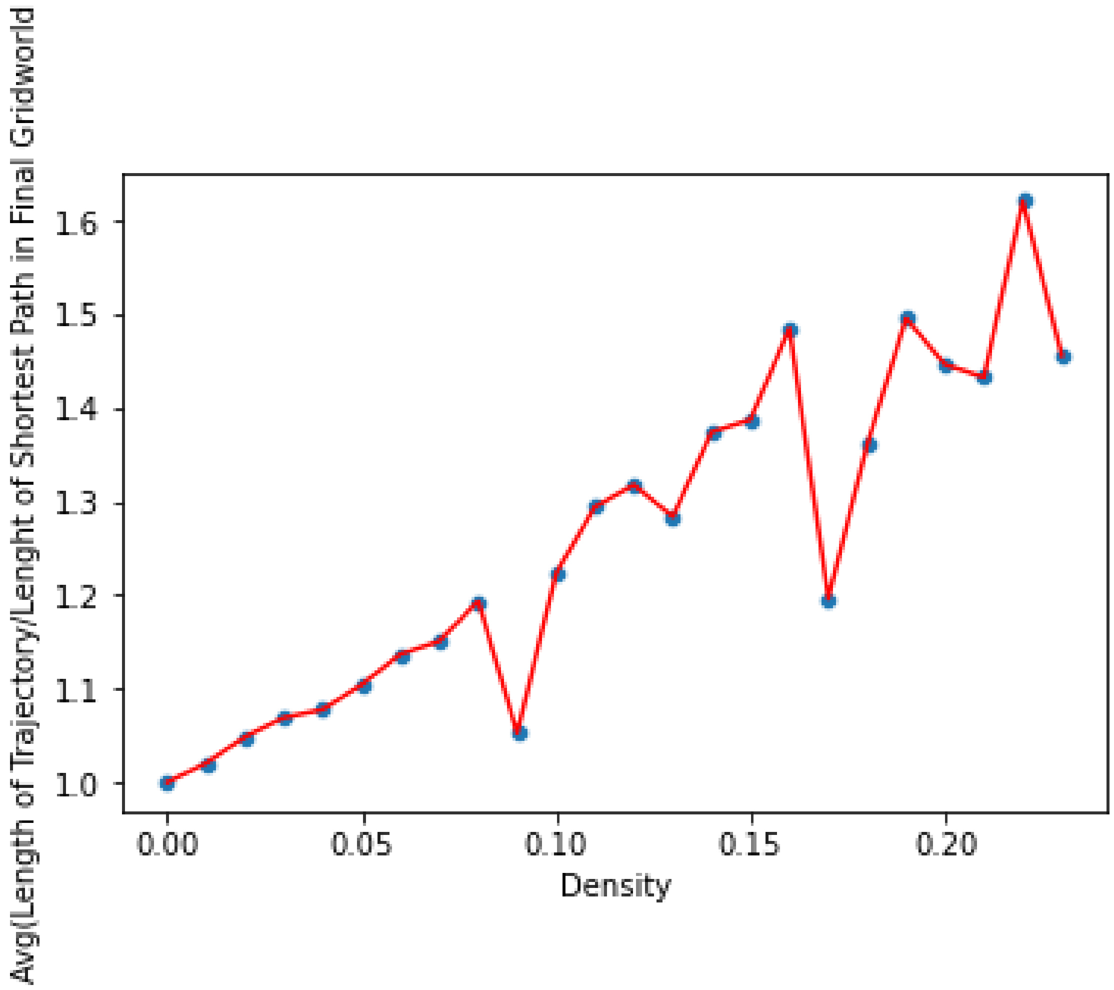

Figure 4: Density v/s Average(Length of trajectory/Length of shortest path in final gridworld)

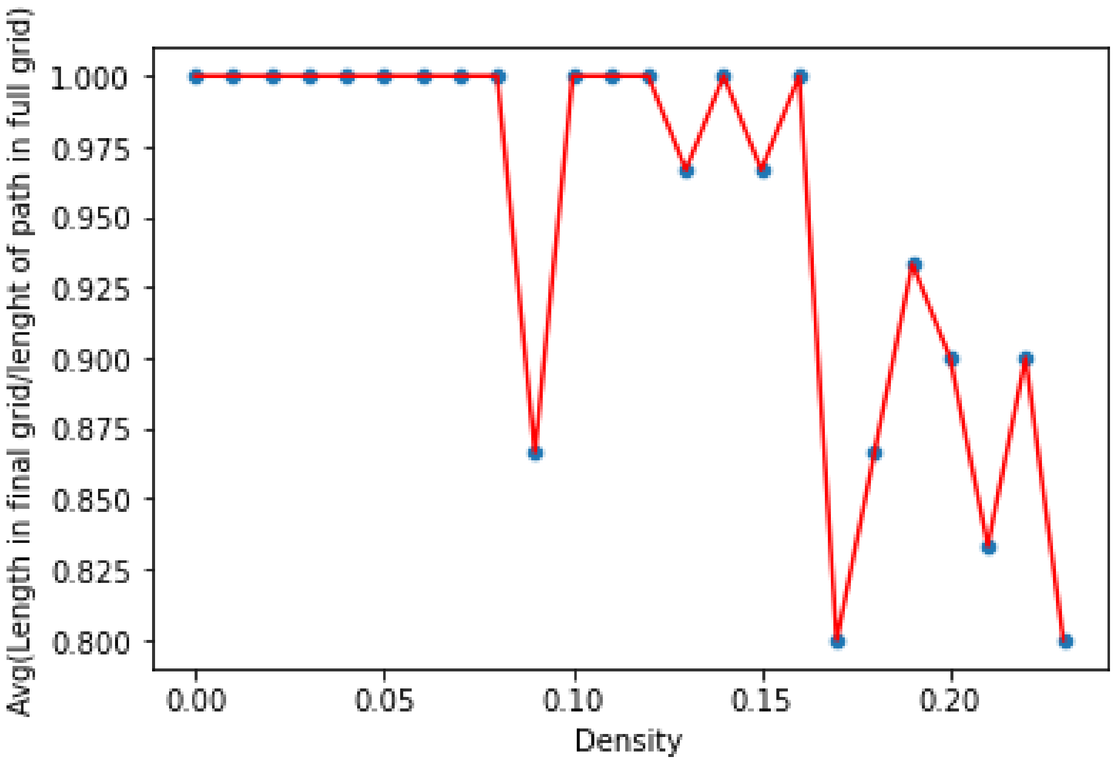

Figure 5: Density v/s Average(Length in final grid/length of path in full grid)

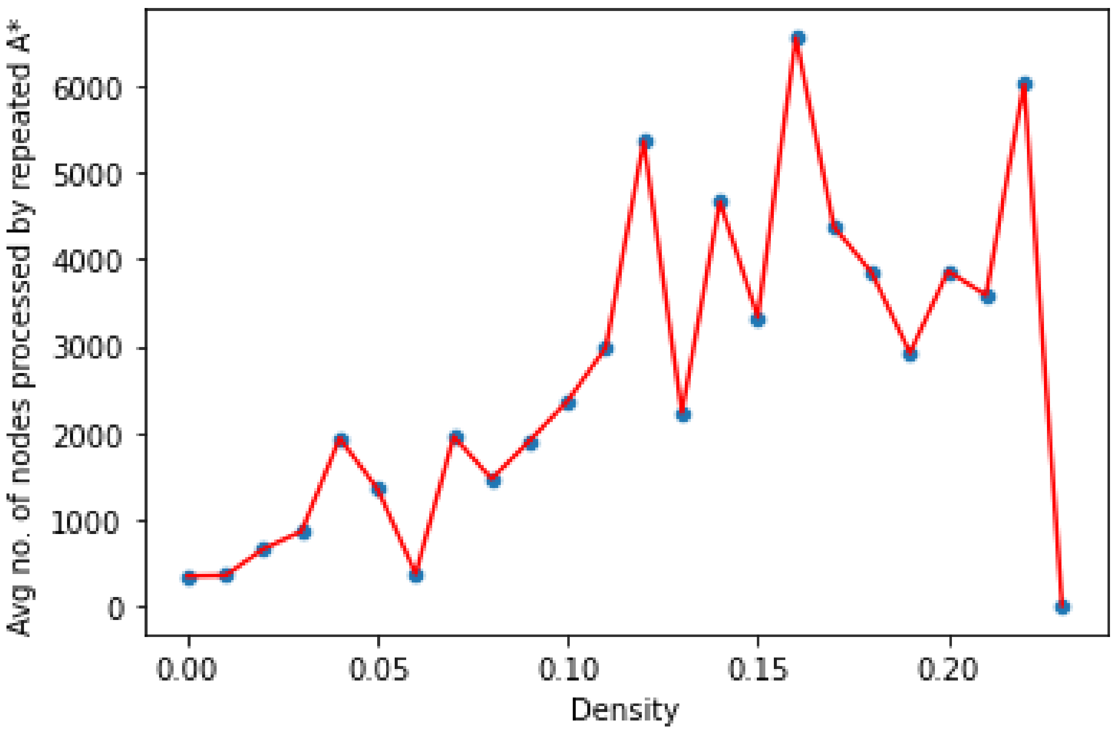

Figure 6: Density v/s Average no. of nodes processed by repeated A*

7 Question 7: Performance - Part 2 Generate and analyze the same data as in Q6, except using only the cell in the direction of attempted motion as the field of view. In other words, the agent may attempt to move in a given direction, and only discovers obstacles by bumping into them. How does the reduced field of view impact the performance of the algorithm?

Figure 7 depicts the relationship between the density and the average trajectory length after applying repeated A*. From the graph we can make out that as the density increases the average trajectory length also generally increases. But it can also be observed that there is a lot of anomaly due to multiple troughs.

The figure 8 depicts the relation between the density and the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld). As the density increases, the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) also increases with a few dips.

Figure 9 depicts the relation between the density and Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) where in as the density increases, the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) has a downward trend.

Figure 10 depicts the relation between density and Average Number of Cells Processed by Repeated A*. As the density increases, a general upward trend is observed along with some unusual dips.

When the graph for the previous question is seen, the number of nodes processed is 6000 where as when the field of view is reduced the number of nodes processed is 8000. Therefore when the field of view is reduced, it take the algorithm takes more time to run.

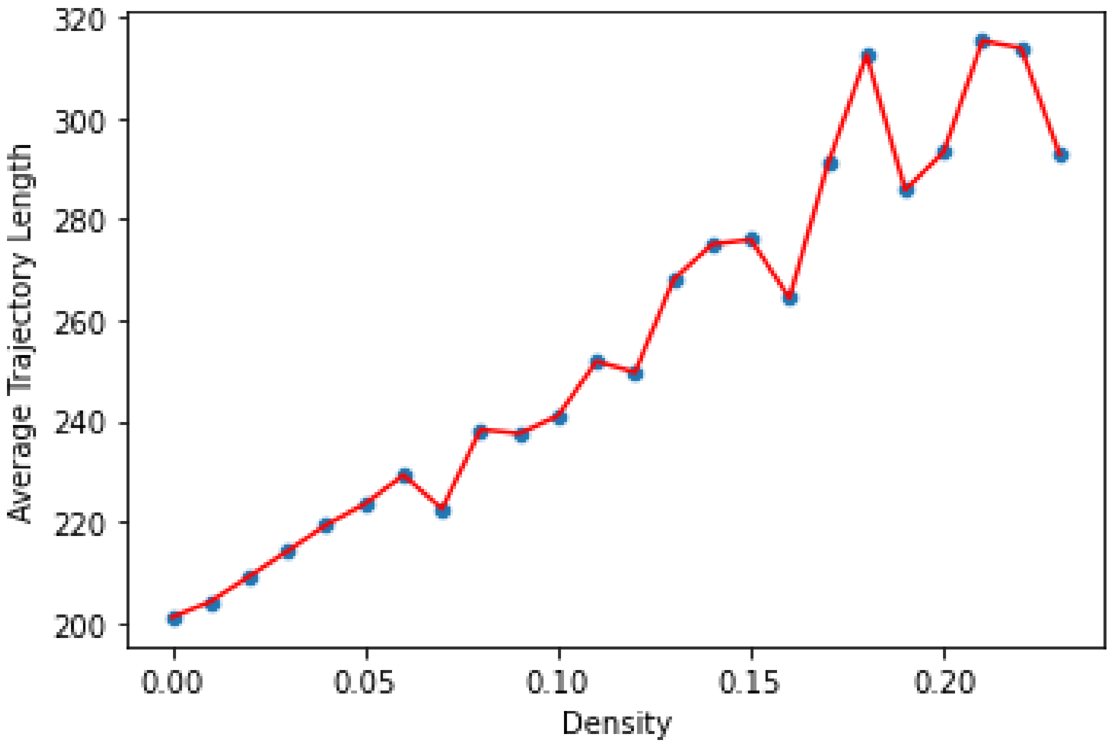

Figure 7: Density v/s Average trajectory length

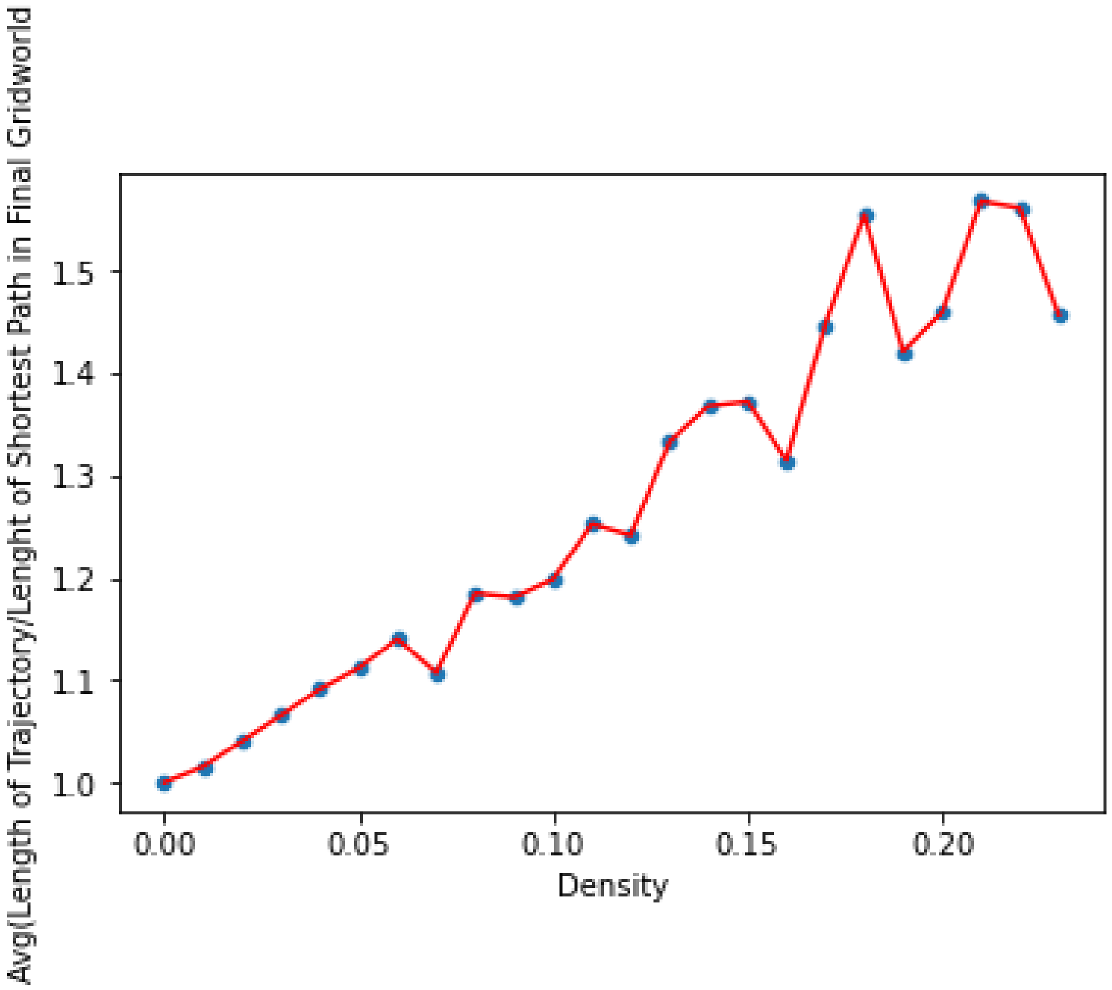

Figure 8: Density v/s Average(Length of trajectory/Length of shortest path in final gridworld)

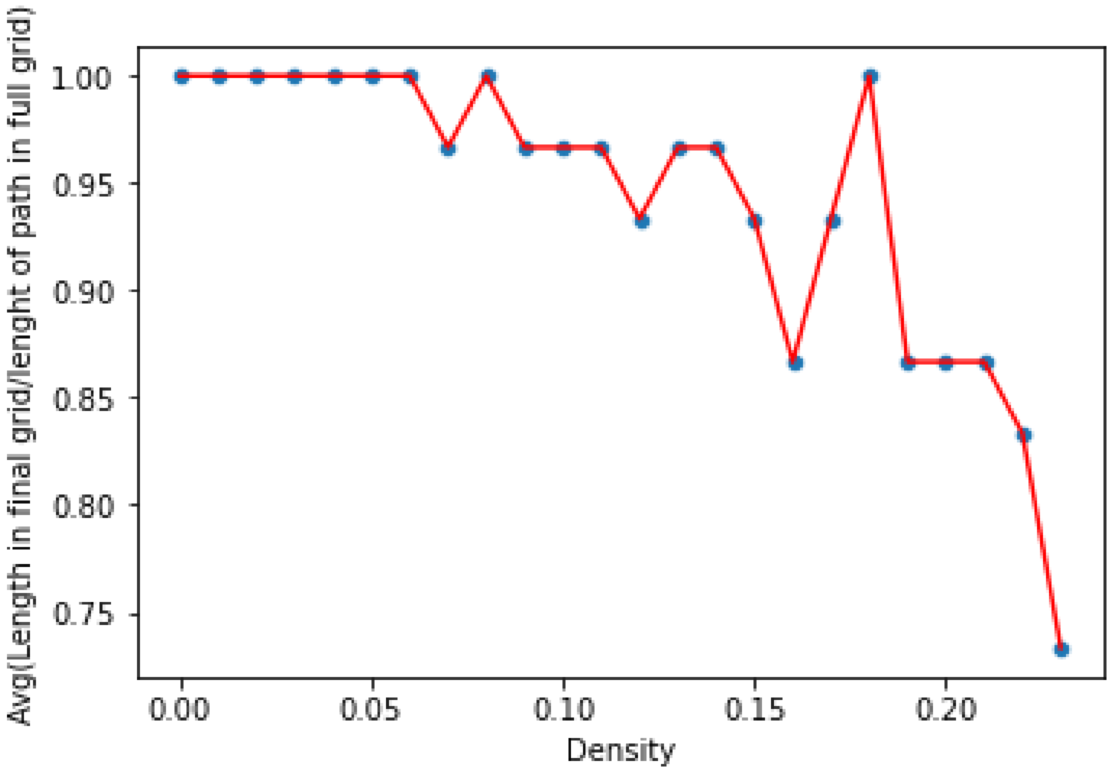

Figure 9: Density v/s Average(Length in final grid/length of path in full grid)

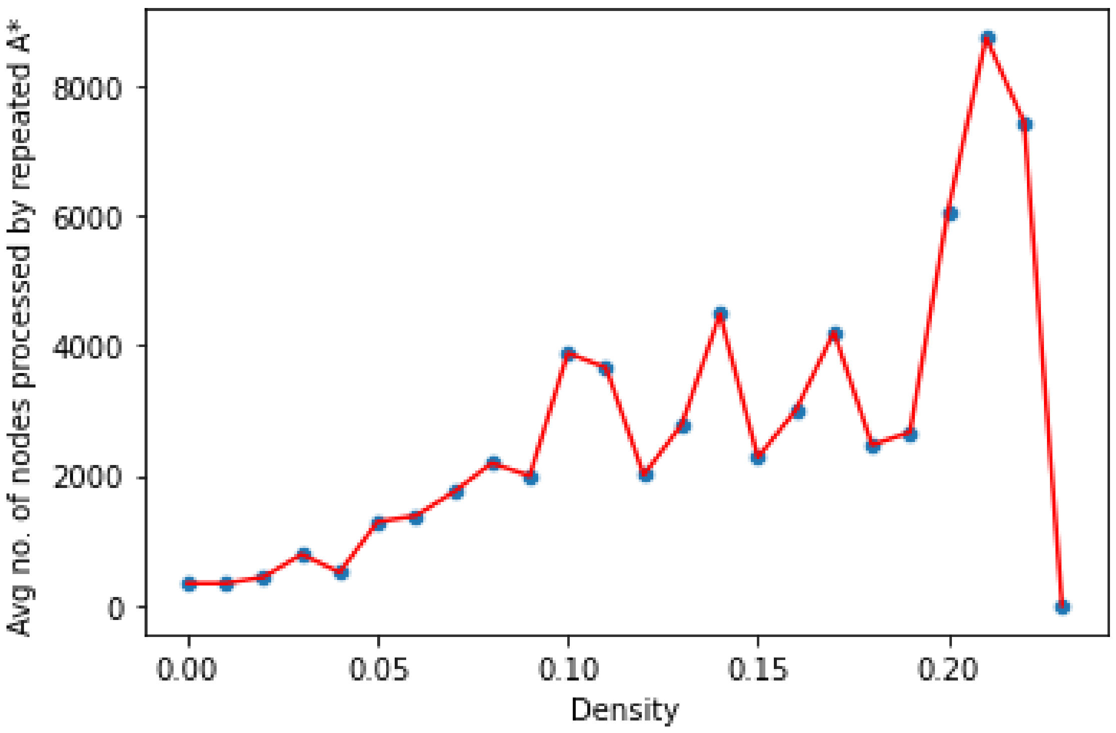

Figure 10: Density v/s Average no. of nodes processed by repeated A*

8 Question 9: Heuristics - A* can frequently be sped up by the use of inadmissible heuristics - for instance weighted heuristics or combinations of heuristics. These can cut down on runtime potentially at the cost of path length. Can this be applied here? What is the effect of weighted heuristics on runtime and overall trajectory? Try to reduce the runtime as much as possible without too much cost to trajectory length.

By using weighted heuristics (multiplying the heuristic by numbers in the range 1, meaning normal heuristic calculation, to 3, with interval steps of 0.2), from figure 11 and 12, it can be seen that as the weights increase the length of the path decreases and the time increases for Manhattan distance. Therefore, the idea that weighted heuristics can cut down the runtime at cost of the path length does not apply for Manhattan distance. On the other hand, for Euclidean distance, as the weights increase, the length of the path decreases along with the time. Same goes for Chebyshev distance where as the weights increase, the length of the path and time decreases. Thus, weighted heuristics have a negative effect in terms of runtime on the Manhattan heuristic, but a positive effect in terms of runtime on the Euclidean and Chebyshev heuristic. The length of the trajectory decreases in all three cases.

9 Extra Credit: Repeat Q6, Q7, using Repeated BFS instead of Repeated A*. Compare the two approaches. Is Repeated BFS ever preferable? Why or why not? Be as thorough and as explicit as possible.

9.1 Repeated BFS - Q6

The relationship between the density and the average trajectory length is shown in figure 17. As the density increases the average trajectory length also increases. The results are as expected because as the density increases, the number of blocks in the grid increases and hence the repeated BFS is called more times thereby increasing the average trajectory length.

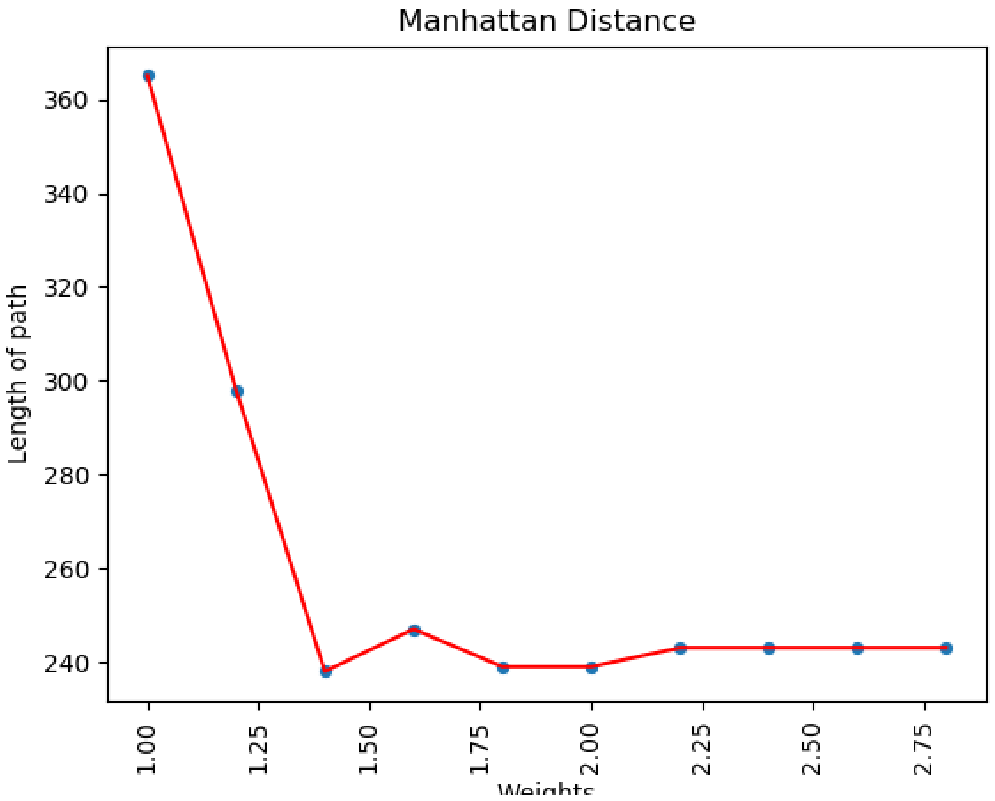

Figure 11: Manhattan distance - Weights v/s Length of path

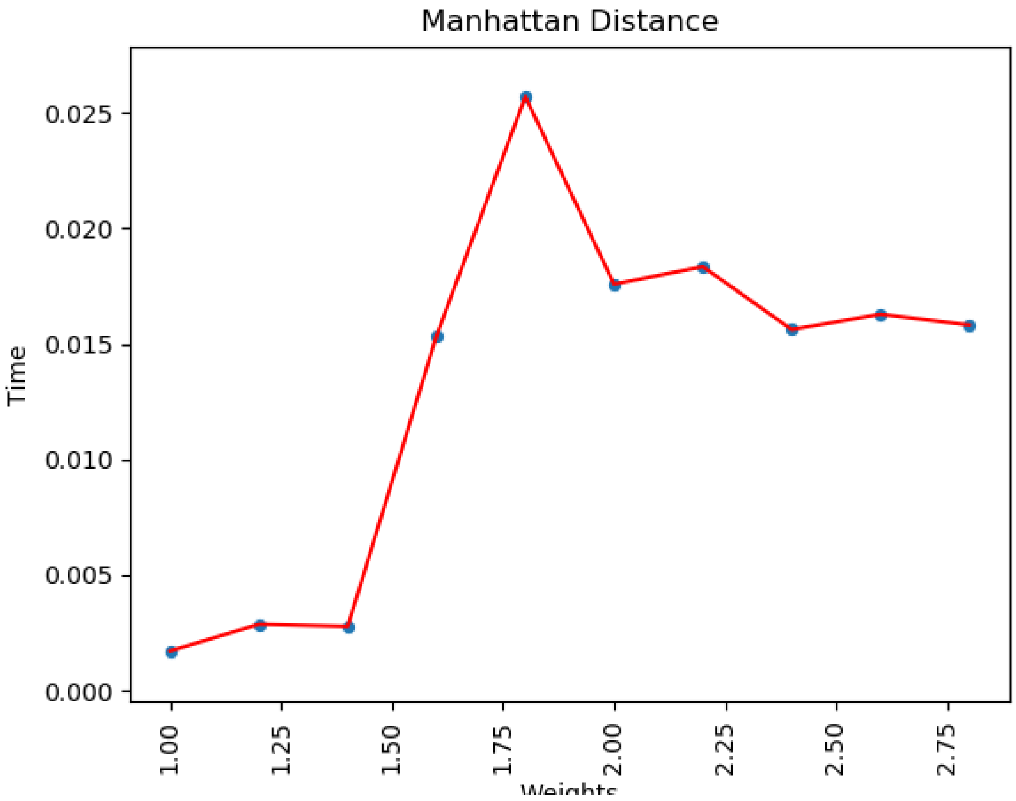

Figure 12: Manhattan distance - Weights v/s Time

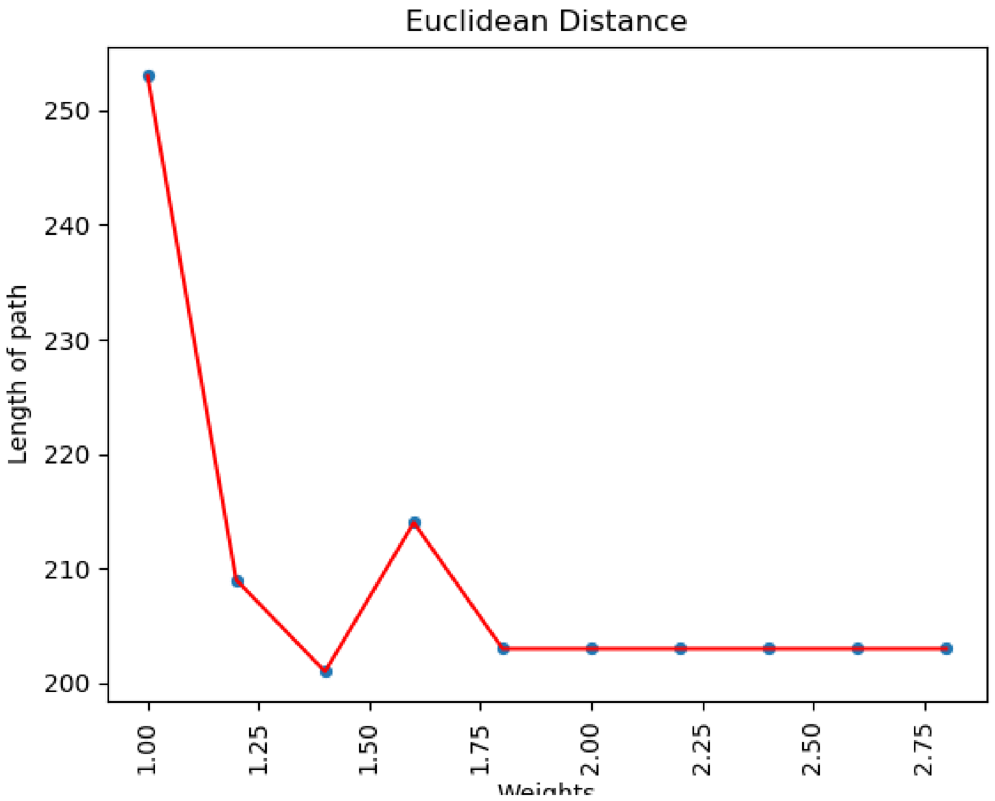

Figure 13: Euclidean distance - Weights v/s Length of path

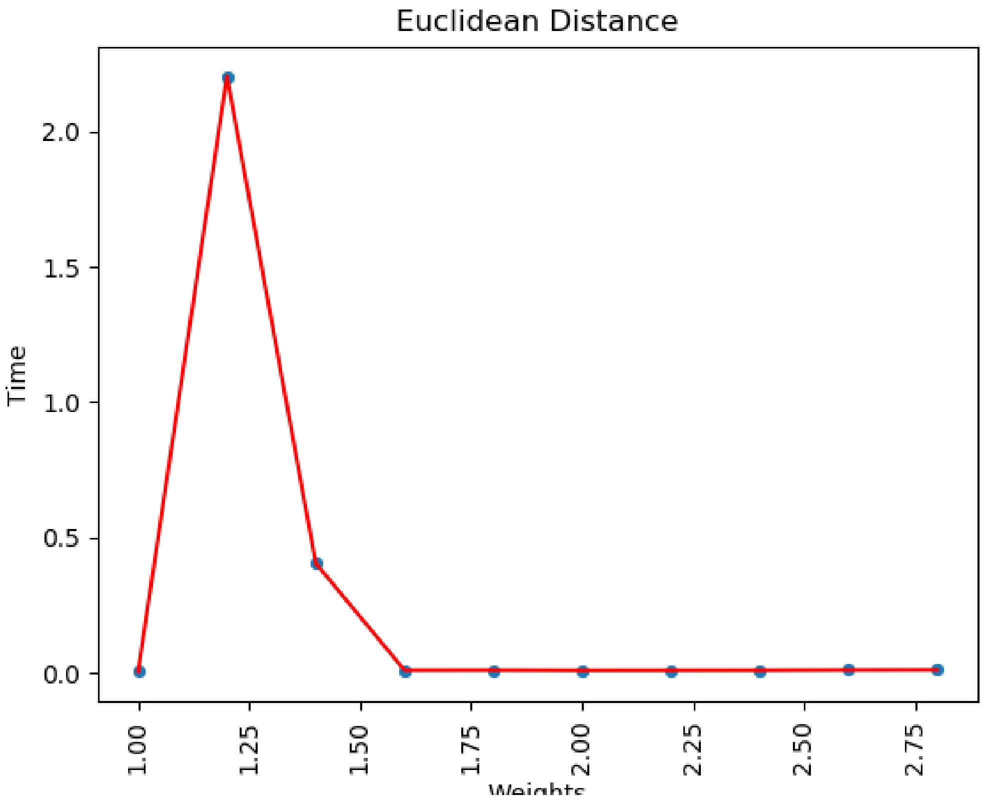

Figure 14: Euclidean distance - Weights v/s Time

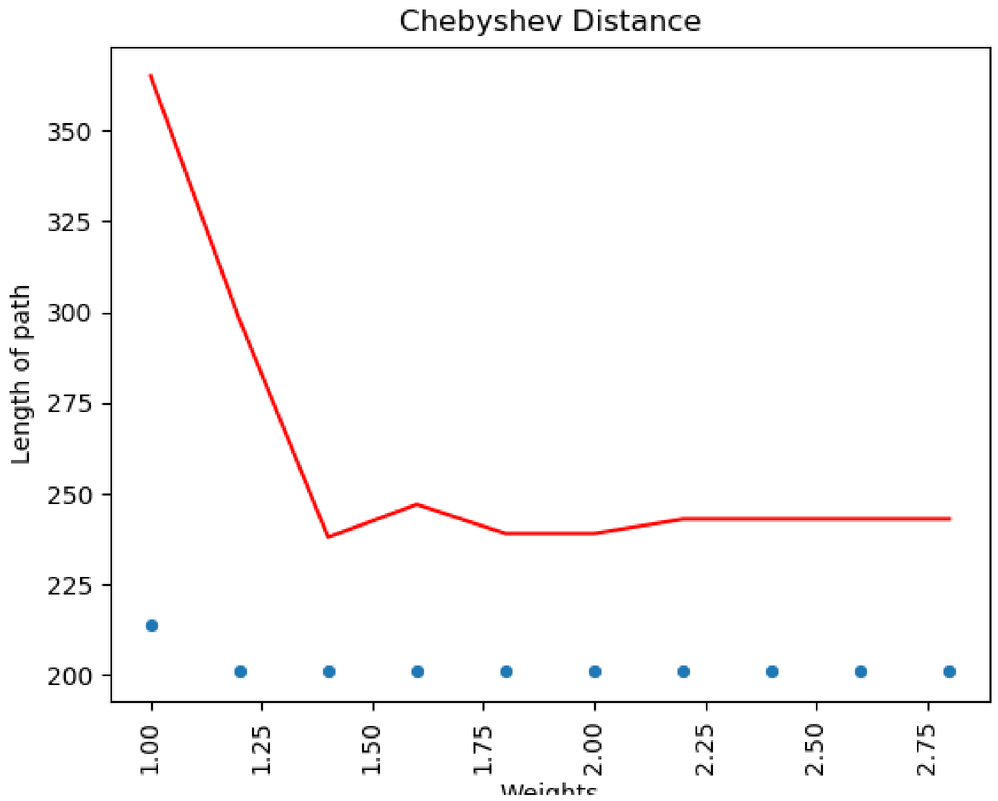

Figure 15: Chebyshev distance - Weights v/s Length of path

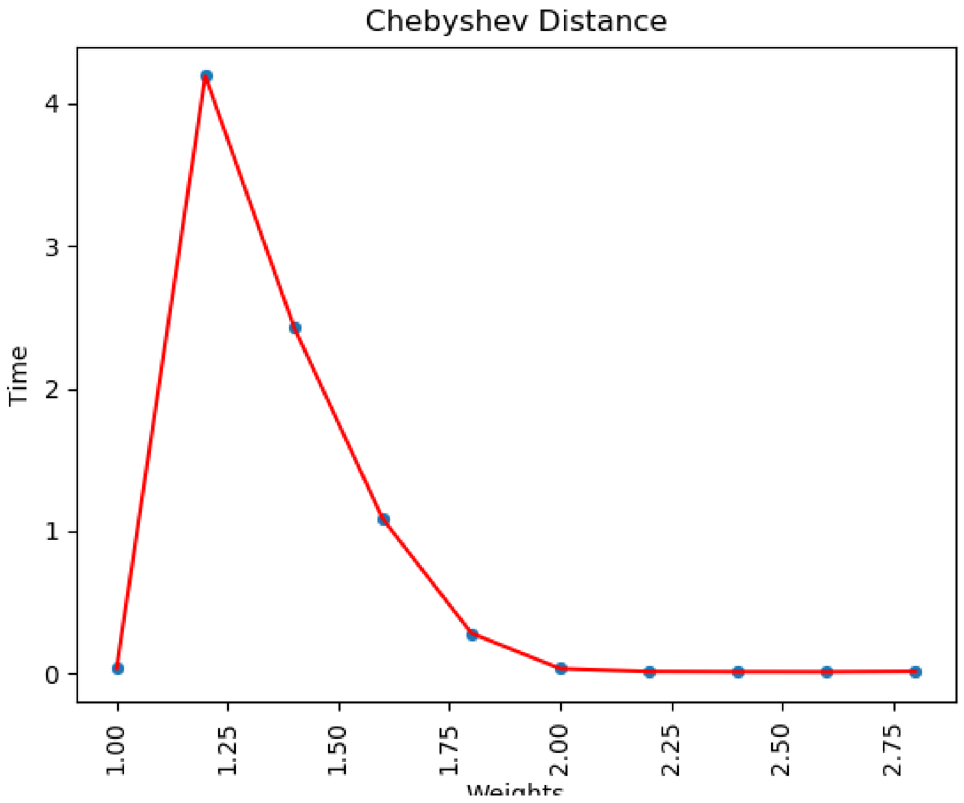

Figure 16: Chebyshev distance - Weights v/s Time

The relation between the density and the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) is shown in figure 18. As the density increases, the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) also increases. The results are as expected because as the density increases, the number of blocks increases due to which the length of trajectory increases.

The relation between the density and Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) is shown in figure 19. The Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) is steady for the initial values of density and then has a sharp drop as the value of density increases.

Figure 20 depicts the relation between density and Average Number of Cells Processed by Repeated BFS. As the density increases, a general upward trend is observed.

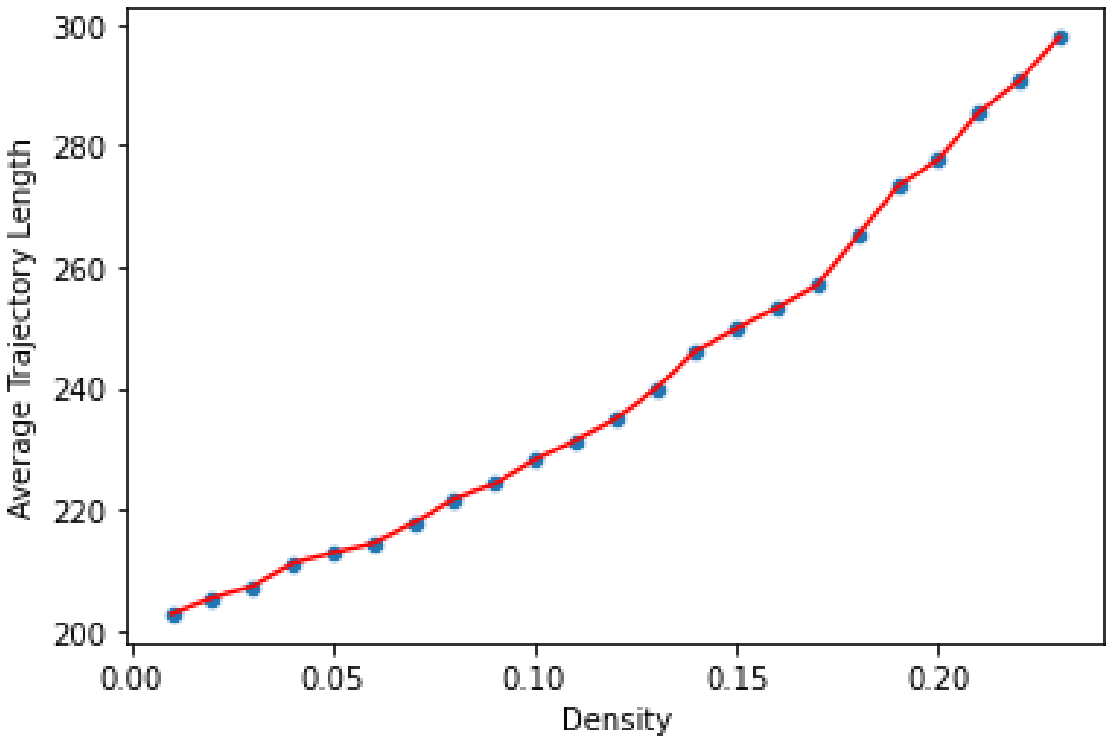

Figure 17: Density v/s Average trajectory length

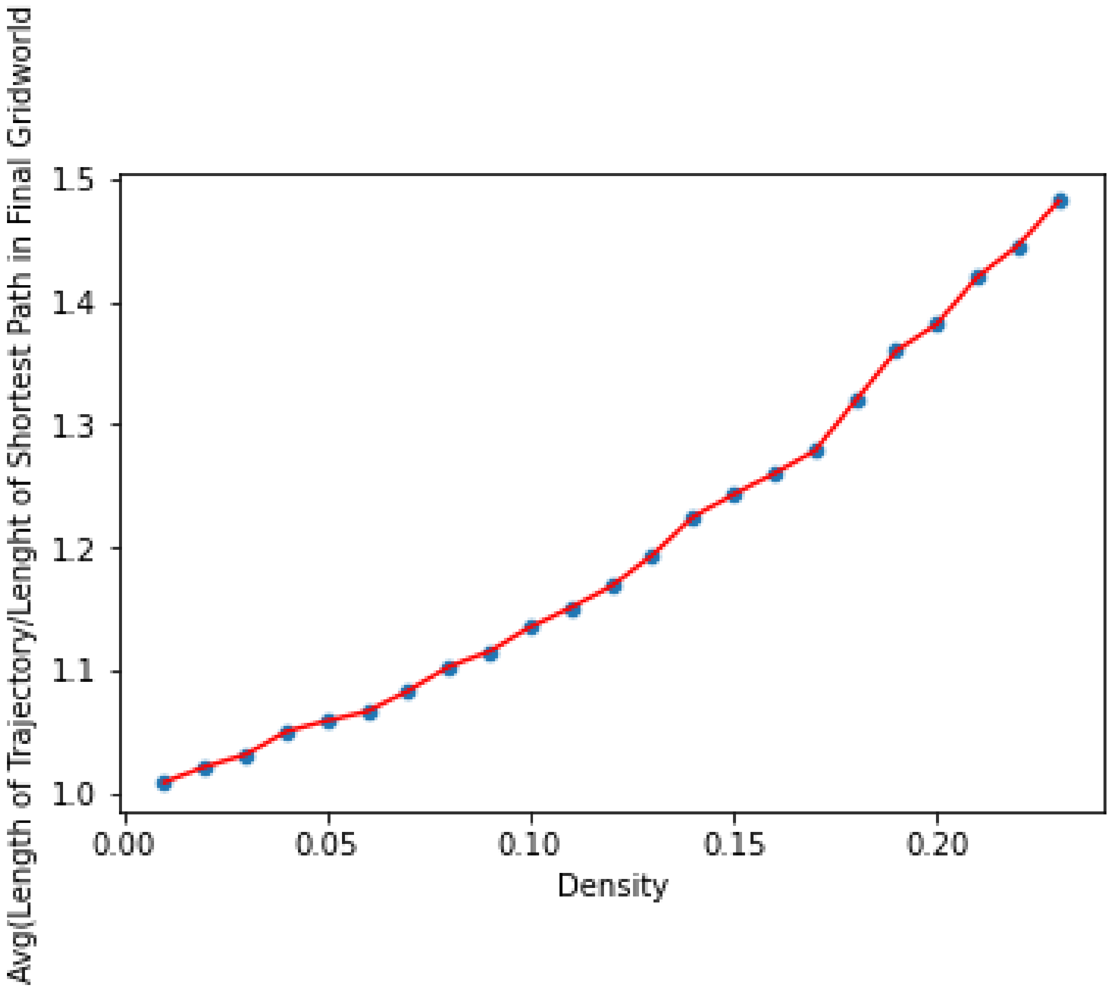

Figure 18: Density v/s Average(Length of trajectory/Length of shortest path in final gridworld)

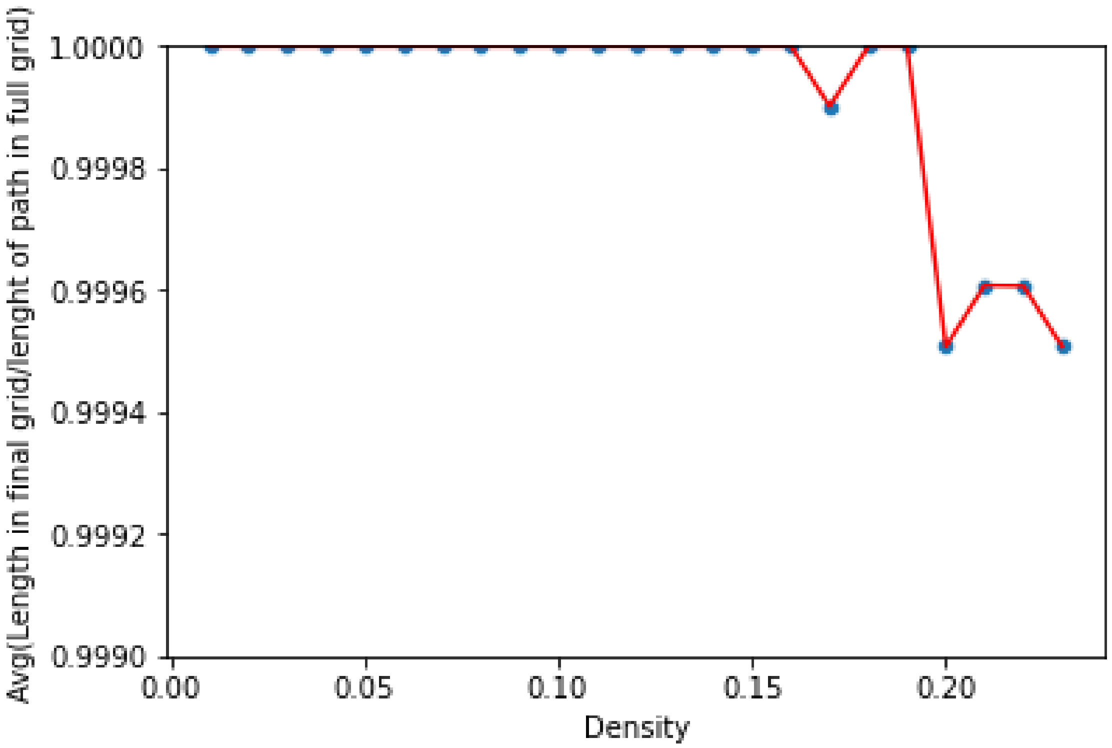

Figure 19: Density v/s Average(Length in final grid/length of the path in full grid)

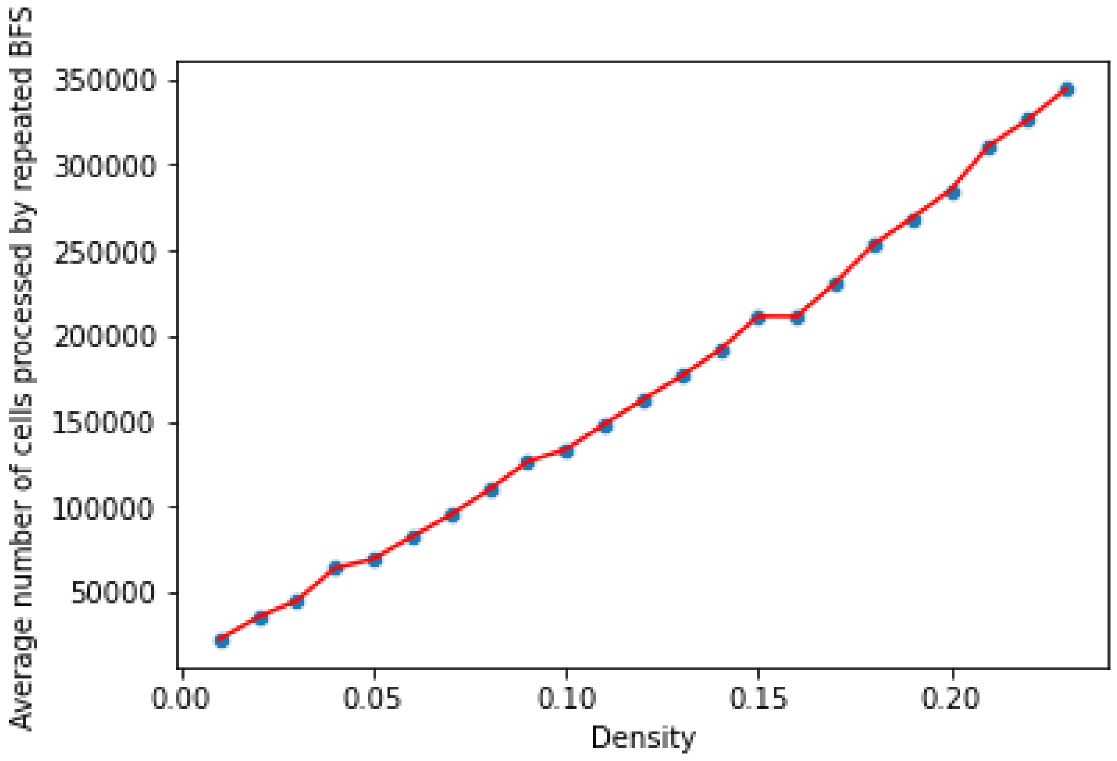

Figure 20: Density v/s No. of nodes processed by repeated BFS

9.2 Repeated BFS - Q7

Figure 21 depicts the relationship between the density and the average trajectory length. From the graph we can make out that as the density increases the average trajectory length also increases with some points where a trough can be observed because the field of view is reduced.

The figure 22 depicts the relation between the density and the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld). As the density increases, the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) also increases with some points where anomaly can be seen.

Figure 23 depicts the relation between the density and Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) where in as the density increases, the Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld) is steady for the initial values of density and then has a sharp drop.

Figure 24 depicts the relation between density and Average Number of Cells Processed by Repeated BFS. As the density increases, a general upward trend is observed.

A* is more preferrable than repeated BFS. The reason being that the length of the trajectory is less for repeated A* as compared to repeated BFS when the value of density increases. Moreover, the average number of nodes processed by A* is just 6000 where as its 350000 for repeated BFS giving a solid reason as to why A* is more preferable.

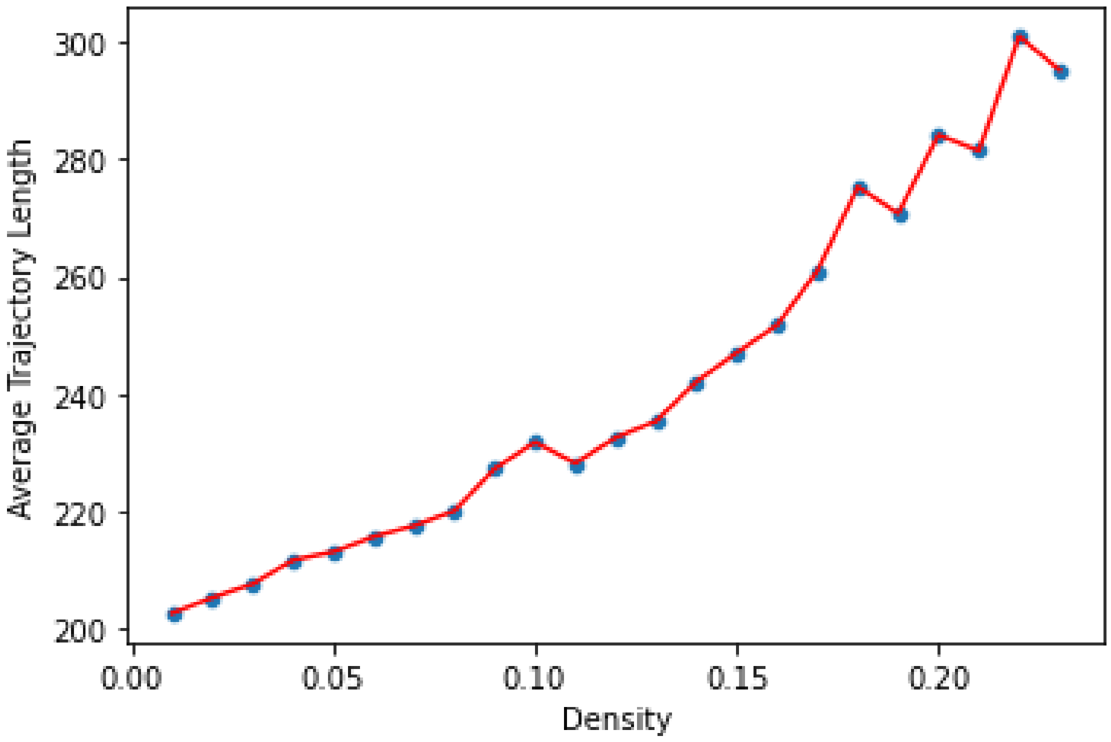

Figure 21: Density v/s Average trajectory length

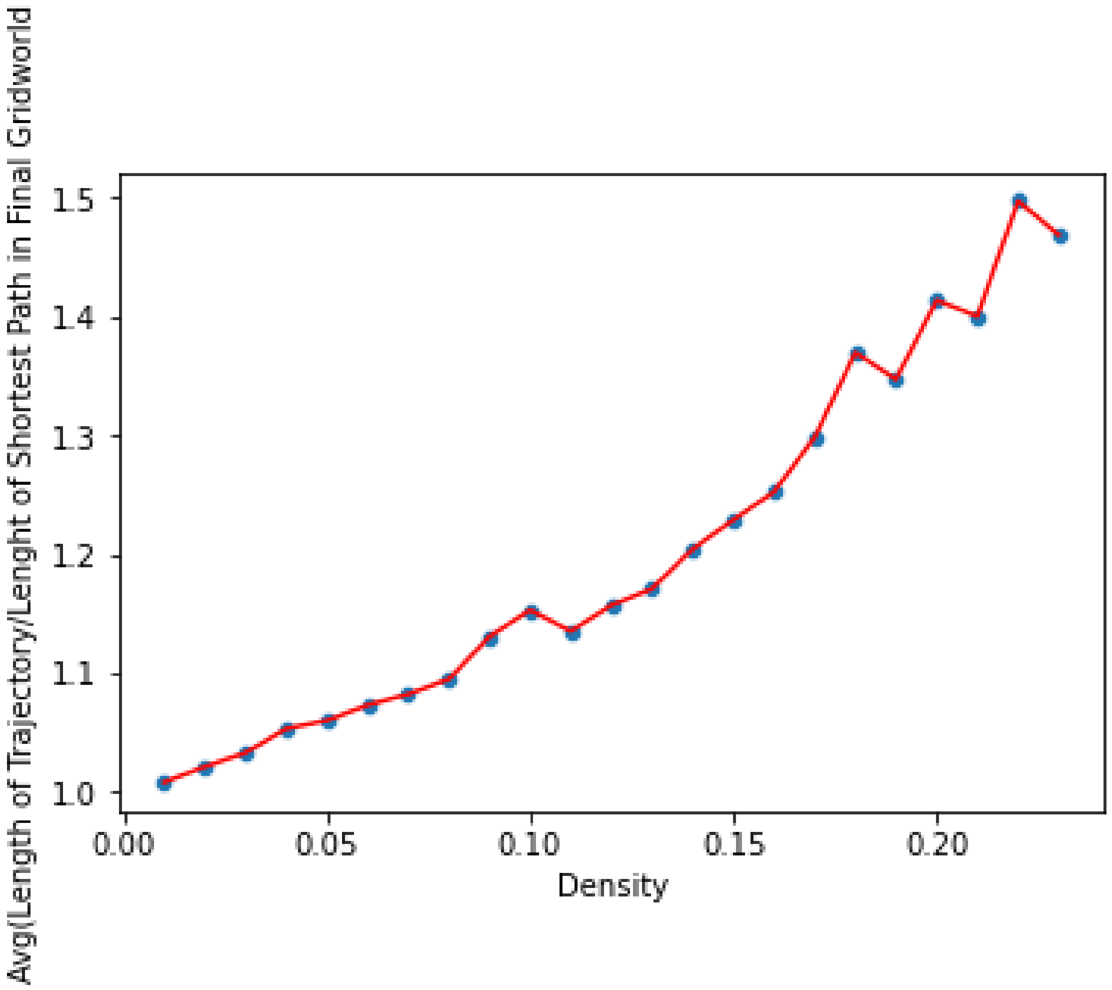

Figure 22: Density v/s Average(Length of trajectory/Length of shortest path in final gridworld)

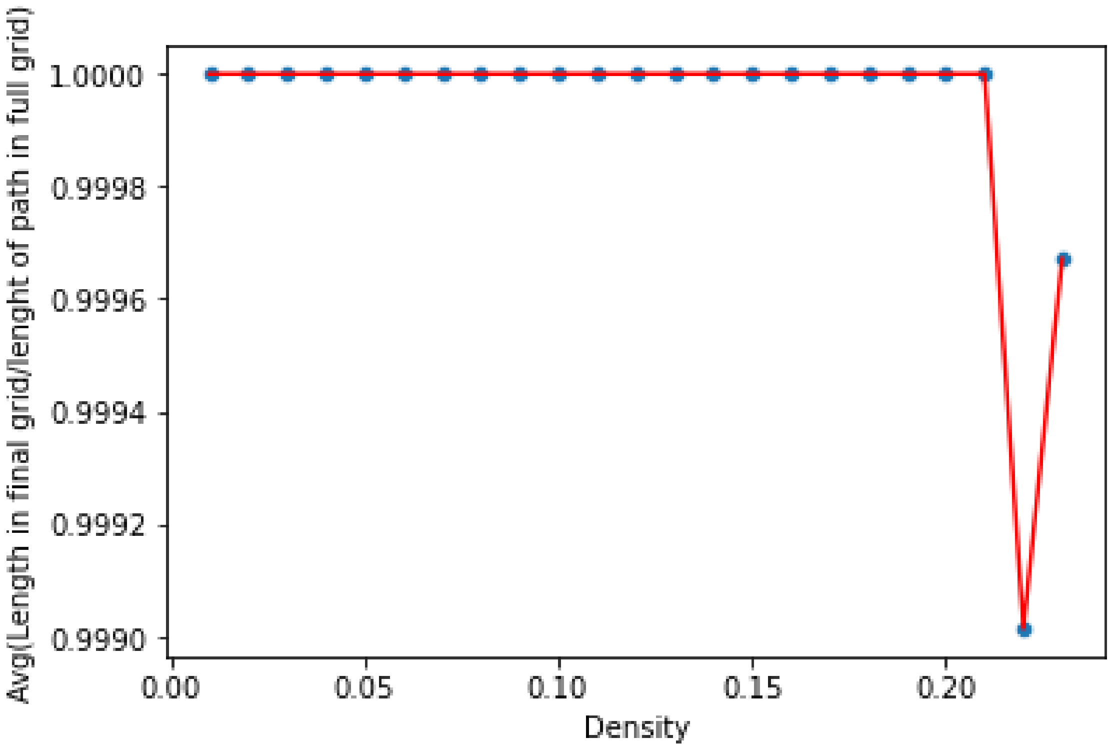

Figure 23: Density v/s Average(Length in final grid/length of the path in full grid)

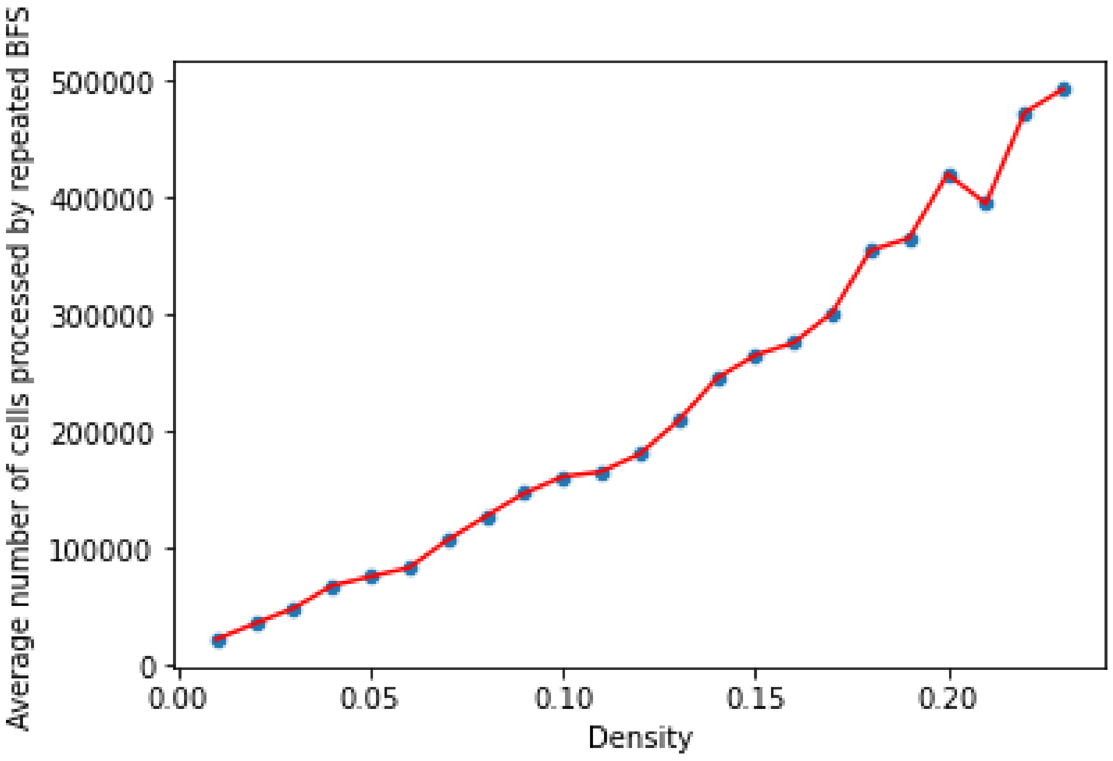

Figure 24: Density v/s No. of nodes processed by repeated BFS

10 Appendix

```
def main(initial_grid , agent_grid):
    '''
    this main function is where the execution process of the algorithm begins
    Parameters
    ----------
    initial_grid : TYPE: 2D numpy array
        DESCRIPTION: it is the initial gridworld that is unknown to the agent in
    agent_grid : TYPE: 2D numpy array
        DESCRIPTION: it is the gridworld that is updated as the agent traverses
    Returns
    -------
    TYPE: 2D numpy array, nested function, integer, 1D list
        DESCRIPTION: updated agent grid, a call to simple A*, no. of nodes proce
    '''
    #initial_grid , agent_grid = get_grid()
    print(initial_grid)

    def check_bounds(x, y, initial_grid):
        '''
        this function checks if the cell under consideration is within the limit
        Parameters
        ----------
        x : TYPE: integer
            DESCRIPTION: the x coordinate of the cell
        y : TYPE: integer
            DESCRIPTION: the y coordinate of the cell
        initial_grid : TYPE: 2D numpy array
            DESCRIPTION: the initial grid
        Returns
        -------
        bool
            DESCRIPTION: returns True is cell is in bounds and False otherwise
        '''
        if x>=0 and y>=0 and x<=initial_grid.shape[0]-1 and y<=initial_grid.shap
            return True
        return False

    def get_heuristic(goal, x1, y1, heu=1):
        '''
        this function calculates the heuristic from the node under consideration
        Parameters
        ----------
        goal : TYPE: tuple
```

```
            DESCRIPTION: a tuple containing the coordinates of the goal node
        x1 : TYPE: integer
            DESCRIPTION: the x coordinate of the current cell under consideratio
        y1 : TYPE: integer
            DESCRIPTION: the y coordinate of the current cell under consideratio
        heu : TYPE, optional: integer
            DESCRIPTION. The default is 1. 1 represents Manhattan, 2— Euclidean
        Returns
        -------
        heuristic : TYPE: integer
            DESCRIPTION. returns the heuristic value after computation
        '''
        x2 = goal[0]
        y2 = goal[1]

        #Manhattan Distance
        if heu==1:
            heuristic = abs(x1-x2) + abs(y1-y2)
        #Euclidean Distance
        elif heu==2:
            heuristic = math.sqrt((x1-x2)**2 + (y1-y2)**2)
        #Chebyshev Distance
        elif heu==3:
            heuristic = max(abs(x1-x2), abs(y1-y2))

        return heuristic

    def get_coordinates(node):
        '''
        this function extracts the coordinates given a tuple of the form (x,y)
        Parameters
        ----------
        node : TYPE: tuple
            DESCRIPTION: a tuple containing the coordinates of the cell
        Returns
        -------
        x : TYPE: integer
            DESCRIPTION. The x coordinate of the cell
        y : TYPE: integer
            DESCRIPTION. The y coordinate of the cell
        '''
        x = node[0]
        y = node[1]
        return x, y

    global nodes_processed
```

```
nodes_processed = 0
def a_star(agent_grid, start, goal):
    '''
    this function carries out the planning phase by running the A* algorithm
    Parameters
    ----------
    agent_grid : TYPE: 2D numpy array
        DESCRIPTION. The grid that is updated by the agent as it traverses t
    start : TYPE: tuple
        DESCRIPTION. Coordinates of the start node
    goal : TYPE: tuple
        DESCRIPTION. Coordinates of the goal node
    Returns
    -------
    TYPE: dictionary
        DESCRIPTION. Returns the parent dictionary to the execution phase so
    '''
    q = queue.PriorityQueue(maxsize=0)
    global nodes_processed
    parent = {}
    visited = []
    g = {}
    x1, y1 = get_coordinates(start)
    h = get_heuristic(goal, x1, y1)
    g[start] = 0
    f = g[start] + h
    q.put((f, start))
    visited.append(start)
    while not q.empty():
        current = q.get()[1]
        nodes_processed += 1
        x1, y1 = get_coordinates(current)

        if current == goal:
            return parent

        else:
            if check_bounds(x1+1, y1, agent_grid) and not math.isinf(agent_g
                parent[(x1+1, y1)] = current
                visited.append((x1+1, y1))
                g[(x1+1, y1)] = g[current] + 1
                h = get_heuristic(goal, x1+1, y1)
                f = g[(x1+1, y1)] + h
                q.put((f, (x1+1, y1)))

            if check_bounds(x1, y1+1, agent_grid) and not math.isinf(agent_g
                parent[(x1, y1+1)] = current
                visited.append((x1, y1+1))
                g[(x1, y1+1)] = g[current] + 1
                h = get_heuristic(goal, x1, y1+1)
                f = g[(x1, y1+1)] + h
                q.put((f, (x1, y1+1)))

            if check_bounds(x1-1, y1, agent_grid) and not math.isinf(agent_g
                parent[(x1-1, y1)] = current
                visited.append((x1-1, y1))
                g[(x1-1, y1)] = g[current] + 1
                h = get_heuristic(goal, x1-1, y1)
                f = g[(x1-1, y1)] + h
                q.put((f, (x1-1, y1)))

            if check_bounds(x1, y1-1, agent_grid) and not math.isinf(agent_g
                parent[(x1, y1-1)] = current
                visited.append((x1, y1-1))
                g[(x1, y1-1)] = g[current] + 1
                h = get_heuristic(goal, x1, y1-1)
                f = g[(x1, y1-1)] + h
                q.put((f, (x1, y1-1)))

    return {}

def execution(initial_grid, agent_grid, start, goal, path=[]):
    '''
    this function carries out the execution phase using repeated A*
    Parameters
    ----------
    initial_grid : TYPE: 2D numpy array
        DESCRIPTION. This is the initial grid to be traversed
    agent_grid : TYPE: 2D numpy array
        DESCRIPTION. The grid that is updated by the agent as it traverses t
    start : TYPE: tuple
        DESCRIPTION. Coordinates of the start node
    goal : TYPE: tuple
        DESCRIPTION. Coordinates of the goal node
    path : TYPE, optional: list
        DESCRIPTION. The default is []. The path from the start node to the
    Returns
    -------
    TYPE: 2D numpy array
        DESCRIPTION. Returns the agent grid with updated knowledge
    TYPE: list
        DESCRIPTION. Returns the path from the start node to the goal node
    TYPE: integer
        DESCRIPTION. Returns the number of nodes processed during the execut
    '''
    parent = a_star(agent_grid, start, goal)
    global nodes_processed
    new_path = []
    if len(parent) == 0:
        print('No path')
        return [], [], 0
    else:
        new_path.append(goal)
        value = parent[goal]
        new_path.append(value)
        while value != start:
            key = value
            value = parent[key]
            new_path.append(value)
        new_path.reverse()
        for i in new_path:
            path.append(i)
        for j, i in enumerate(path):
            x, y = get_coordinates(i)
            if not math.isinf(initial_grid[x][y]):
                if check_bounds(x+1, y, initial_grid) and math.isinf(initial
                    agent_grid[x+1][y] = math.inf
                if check_bounds(x, y+1, initial_grid) and math.isinf(initial
                    agent_grid[x][y+1] = math.inf
                if check_bounds(x-1, y, initial_grid) and math.isinf(initial
                    agent_grid[x-1][y] = math.inf
                if check_bounds(x, y-1, initial_grid) and math.isinf(initial
                    agent_grid[x][y-1] = math.inf
            else:
                agent_grid[x][y] = math.inf
                for k in range(j, len(path)):
                    path.pop()

                return execution(initial_grid, agent_grid, path[j-1], goal,
    print(path)
    return agent_grid, path, nodes_processed



new_agent_grid, path, nodes_processed = execution(initial_grid, agent_grid,
return new_agent_grid, a_star, nodes_processed, path
```