RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

School Of Graduate Studies

VOYAGE INTO THE UNKNOWN

Authors: Jal Ashishkumar Shah Net Id: js2985

Professor: Wes Cowan
CS520: Introduction to Artificial Intelligence

September 27, 2021

Table of Contents

List Of Figures

Introduction

This project focuses on building an algorithm for ***finding*** a path in a grid world and then *walking* on it. A grid world is a terrain of squared cells which are either blocked or unblocked. An agent is at the start node of the grid world and intends to reach the target node as efficiently as possible. The agent can move in any direction (north, south, east, west) if the cell is unblocked and is in the part of the grid world. The agent does not initially know if the cells are blocked or unblocked. It updates its environment by observing and remembering its surrounding for future use. Here we use A* search algorithm for path planning and a variant of A* search algorithm called the Repeated Forward A* which uses A* search algorithm at each recent unblocked step whenever the agent finds a blocked cell on its path, to find the optimal path to the target node.

Question 1

- Re-planning the path means considering the node on which the agent is, as to be the current node and from there calculating the shortest unblocked path(presumed) to target node on which the agent will move. The path is a ***presumed*** shortest path because initially, the agent comes up with this path with no knowledge of the grid world considering all nodes to be unblocked.

- When the agent is moving, the knowledge of the environment is updated while following the shortest (presumed) path which it planned earlier. But when the agent discovers a blocked cell in its path, it needs to re-plan its path as the path planned before will no longer serve the purpose of reaching the goal node.

- Re-planning the path at each step whenever the knowledge of the environment is updated would be ***redundant and time consuming*** as until or unless the agent reaches a blocked cell, it is moving on the correct shortest (presumed) unblocked path. Re-planning on each cell will only lead to an i***ncrease in the total time*** taken by the agent to reach the goal node (runtime of the algorithm) as it will calculate the best shortest path for each cell which is not required as we already have a best shortest path.

Question 2

- If the agent discovers a block node or a more convoluted series of block nodes, if there exists a way out of it, it will find it and never get stuck in it, in a solvable maze.

- The grid world has the start node at the top left corner and the target node at the bottom right corner. As the agent starts moving from the start node towards the target node, if there exists a clear path from start to goal it will be able to find it (using Repeated A* algorithm).

- The only hindrance for the agent to reach the target node in a solvable maze is a blocked cell or series of blocked cells. But the agent is programmed to re-plan its path whenever it finds a blocked node on its current path. The agent, while traversing a path, keeps on *updating* its environment and ***remembers*** the path it has traversed on. The agent re-plans its path from the last unblocked cell it has discovered.

- In worst case if it again encounters 'n' number of cells having all the child states of the cells to be blocked at any given point in the maze, it will *backtrack* on the path to the state 'n' times and goes on to the path to previously ***un-explored*** unblocked node.

- So as the agent moves over a solvable maze it will certainly find a path from the start node to the goal node.

Question 3

- After the agent has successfully reached the goal node, he will have discovered some part of the maze but *not entirely*. Now, when we try to solve this now discovered grid world again, we still wouldn't have complete information of the grid world. So, we will get a path from start to goal node, but there is no 100 percent surety that we would've discovered the shortest path.

- The below example proves this whole concept:

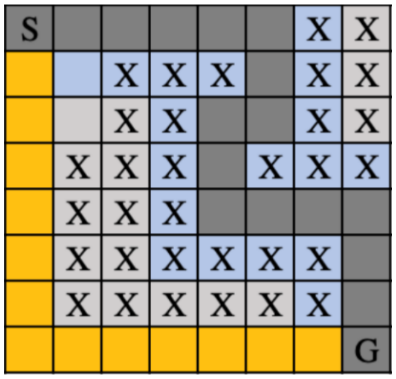


Figure 3. 1 Counter Example Maze

- Consider this grid world of 8x8 cells. In attempt to reach the goal node, the agent decides to move right as both the cells i.e., cell at location (0,1) and cell at location (1,0) are equally favorable to reach the goal. While walking down the path planned by it, the agent must backtrack 3 times because of the discovery of the blocked cells.

- While traversing, the agent discovers the nodes along its path and these nodes are marked in the given figure with a shade of light blue. Now resolving the discovered grid world for the shortest path, we eliminate any backtrackings this time that may have occurred before. So, we attempt to search for the optimal path in now discovered grid world and we again get the same path that we got the previous time**. *(This newly found path is the same as the path that we found with backtracks, but this will not happen every time. This pertains to the structure of blocked and unblocked cells in the grid world.)*** Now the newly discovered path in the discovered grid world is optimal with the limited information that we have for the maze.

- But we can still clearly see that the path followed by the agent while resolving for the shortest path over the discovered grid world is not optimal as the shortest path for the full grid world with all the information about the cells is marked by the yellow cells. Hence, shortest path in discovered grid world will not be the same as the optimal path in the whole grid world.

IMPLEMENTATION:

A Algorithm:*

For the implementation of the A* algorithm, the most import data structure is the fringe for storing all the nodes that will be processed and this will be used most extensively in the code so the runtime of all the function on fringe should be as low as possible. Keeping this in mind we have used Sorted Set from Sorted Containers which is a library written in python and works as fast C extensions. All the operations on Sorted Set that we have used namely ***add()*** and ***discard()*** works in ***log(n***) ***time complexity.*** Initially when a call id made to the function of A* named astar(), it will be provided with the maze to work on, the dimensions of the maze and the starting cell from where the path must be planned. Then inside the function, initially the Sorted Set is initialized by giving a name ***open_list*** and an entry of the starting cell is made. Each entry for a given cell in the Sorted Set will contain a tuple consisting of more tuples. The first tuple in the whole tuple for the cell contains f(n) and h(n) value. The second tuple contains the indices or the location of the cell in the maze in (x, y) coordinates. And the final tuple will contain the location of the parent from where the current cell has been reached. The function also creates 2 empty sets for keeping the track of the location of nodes in (x, y) coordinates which are in Sorted Set which are not yet explored- ***open_set*** and the set closed_set which contains the coordinates of the cells which have been processed. ***All the operations of sets take place in O(1) time and so this is the wisest choice when it comes to checking for a cell if it has been processed or not.***

Now after initialization of all the required data structures***, we will now loop until there is no node left to process in open_list*** *(in which path with [-1] is returned)* ***or we have reached the goal cell.*** Inside the loop, we initially pop the first element from the open_list. Now in open_set as it is a Sorted Set, all the elements will be added in such a way that the ***elements of lower f(n) value come first*** and then the ones with higher f(n) value. This is done automatically, and all the cells will be sorted in this way. So, when we perform pop(0) operation on open_set, we will receive the cell with lowest f(n) value which should occur. If 2 cells have same f(n) value, then ***Sorted Set will give us the cell with lower h(n) value*** and if this is also ***same, then the cell will be given according to the direction, right, down, left, up***. This has been formulated because the agent will have in mind that to find the shortest past, it will go first right all the way to dead-end and then go down. So, when a choice comes where 2 cells have same f(n) and h(n) value, the agent will choose according to this strategy and no randomness occurs.

Now, coming back to the process, after getting the cell by popping from open_set fringe, a counter is incremented which will tell us the number of cells which has been processed yet. Then the (x,y) coordinates are stored in a variable. First the condition of goal is checked and if the current node is the goal node, then, a new list is created which will contain the final shortest path from start to goal. The current cell is appended, and a loop is created where we append the parent of the current cell until start cell is reached. Then along with this path, a list of blocked cells discovered, and number of cells processed are also returned. If current cell is not goal then, the neighbors in 4 directions, right, down, left, and up are looked up. For each neighbor, the following process takes place to assess it:
If the neighbor is in open_set and the current g(n) value(+1 to g(n) value of current cell under process) is less than the old g(n) value, then parent is set to the cell under process and the cell's g(n) and f(n) values are updated and if not so then nothing is done. If neighbor is in closed_set and old g(n) is not optimal then changes to the attributes are made and the cell is removed from ***closed_set*** and added to open_list and open_set and if none of the cases applies, then the cell is newly discovered and is added to open_list and open_set.

After all the neighbors are analyzed the node currently under process is added to closed_set. This is continued until no nodes are left to process or the goal is reached.

Repeated Forward A* Algorithm:

When a call to the function for Repeated Forward A* Algorithm named repeated_forward_astar(), the functions receive maze with full information, dimension of the maze and the starting position of the maze (0,0). A ***new maze*** is also created with no information and all nodes are set to be unblocked. This is the maze, the agent will use and update according to the field of view and the cell it is currently in. A ***new_path*** list is created which will keep tracks of the agent's movement and will be returned by the function of Repeated Forward A*.

Initially, a call is made to A* with (0,0) as the starting node to plan a path. The path returned will be based on the new maze with no initial information of blocked cells. Then after receiving shortest path from A*, the agent will start to move and do the needful.

A loop will start with ***2 end conditions***: goal cell reached, and no path returned by A* during planning phase is empty which means no path exists. Inside the loop the following operations occur for each value in the path list given by A*. The values will be the coordinates of the cells:

First it is checked if path is [-1] or not which denotes if some path is returned by A* or no path exists. If there is no path, then we will not move ahead and end the loop by returning no path and the number of cells processed till now which we would've updates every time A* is called for path planning. If this is not the case then path contains some cells to process, then the condition for goal cell is checked. If we have reached the goal, then we append our current cell under process that is the goal node to our new_path list and return it along with number of cells processed till now. If this condition is not met, then we go on to process the neighbors of the current cell after appending the current cell to the new_path list. The environment is updated while checking for each neighbor in the field of view. This is done by comparing each neighbor's state to the one in the original maze we received from caller. After updating the field of view in the new maze, it is checked if the next incoming cell in the path list given by A* is blocked or not, this is done by retrieving the index of that next incoming cell and then checking it with the newly updated environment. If it is blocked, then the path we got earlier no longer serves its purpose and A* is again called with the current cell as the start position and a new path is returned. Also, whenever A* is called, the variable total_cells_popped(this variable keeps track of all the cells processed till now) is updated. It is added with the number of new cells processed by A*.

This continues, unless we find a goal node or there exists no cells to process meaning no path is present.

Question 4

- A grid world is solvable if there exists a clear path from the start node to the target node.

- Consider each cell to be blocked with a probability ‘p’ and empty with the probability of ‘1-p’. This is clearly evident that, as the number of blocked nodes increases in the grid world, the probability of reaching the target node successfully decreases.

- Therefore, solvability is inversely proportional to the value of ‘p’ (‘p’ denotes the probability of the node being blocked). As the probability of the nodes being in a blocked state increase, the number of paths from start node to goal node also decreases. This automatically decreases the probability of the grid world being solvable.

- For a grid world of dimension 101, the same above stated thing applies. As the value of ‘p’ increases in the maze, we will get more and more blocked cells. Thus, our solvability will decrease with respect to probability ‘p’

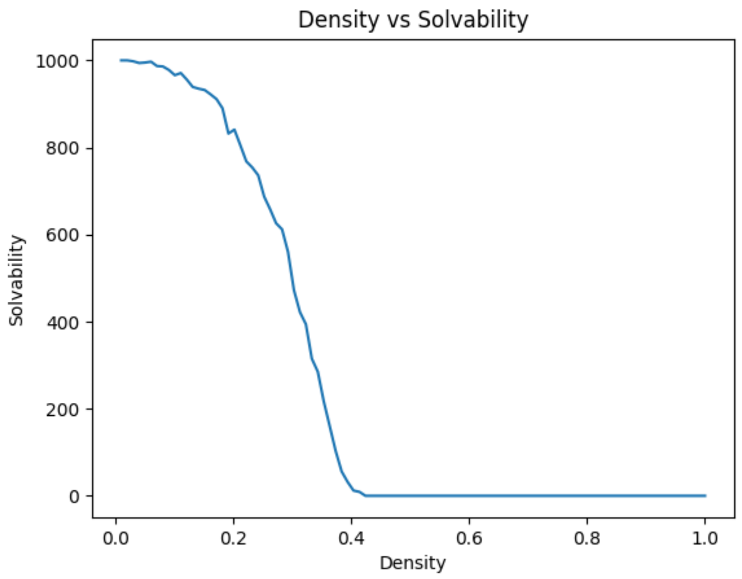


Figure 4. 1 Density vs Solvability

- Here we have plotted the density versus solvability graph which corroborates our above-mentioned theory in practical terms.

- We define threshold p_0 such that for $p<p_0$ most mazes are solvable and for $p>p_0$ most mazes are not solvable. Here to find the value of p_0 we ran an A* algorithm for the whole range of probability (0,1] with step size of 0.01 and 1000 iterations of different generated grid worlds according to the density per step. Thus, we get a very precise graph showing the adverse effect of density on solvability.

- We got the value of p_o as 0.23 where 75% of mazes are solvable when p is in range (0,0.23].

- We can see from the graph that when probability is close to 0, most grid worlds are solvable. This number of solvable grid world keeps on decreasing as density of blocked nodes in maze increase.

- Number of solvable mazes drops to 0 when density of blocked nodes is approximately 0.4 (0.41 to be precise which we got from the output data). After 0.4, on an average no more mazes are solvable.

- When comparing mazes for the test of solvability, A* algorithm is the best choice as it is bound to get the path from start to goal node of there exists one. Other algorithms like DFS, BFS, etc., will also find the path but it is worth mentioning that A* will ***outperform*** other algorithms because the path found by A* will be the shortest and optimal one while other algorithms doesn't provide such guarantee with extreme confidence. Also, A* also takes the least amount of time to reach the goal node so this is the other criteria which shows the A* algorithm's prowess in finding the shortest path ***optimally***.

Question 5

- Heuristics are formulated techniques to ***speed*** up the process of an algorithm by giving *rough* ***estimate*** about the distance from current position to desired output. Here considering the grid world, a heuristic is such a thing which gives us the estimate of how far a particular node is from the goal node.

- We took average cells traversed as the measure to compare each heuristic and **plotted *graphs of density vs average cells traversed*** to get a more insights. We took the range of probability: (0, 0.41) and iterated for 1000 times for each step of probability for different 101* 101-dimension grid worlds generated according to the density. We choose the upper limit as 0.41 because we can see from the data, we get from Q4 graph that for mazes with density lesser or equal to 0.41, mazes are solvable (even if its 1 in 1000). So, by plotting this graph we get the following result:

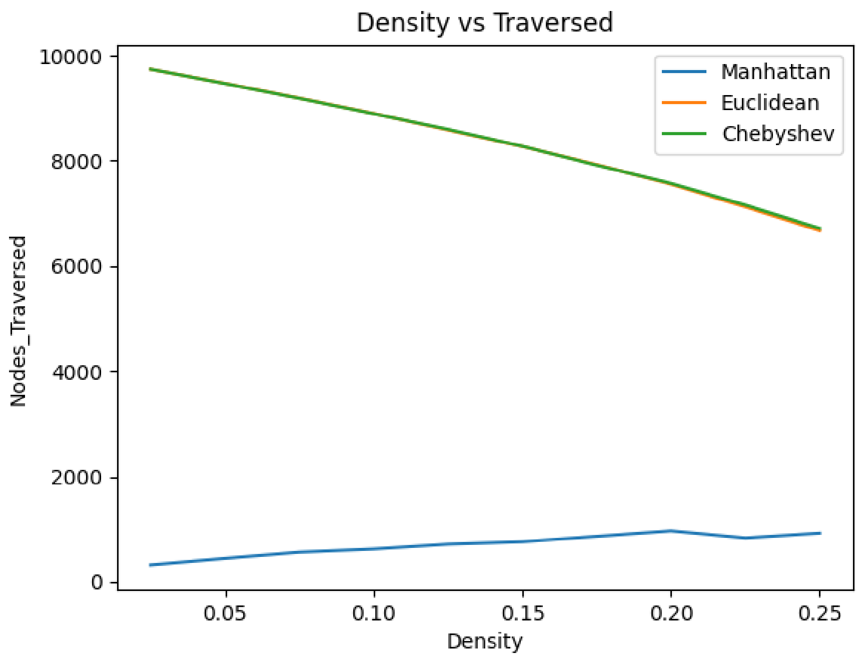


Figure 5. 1 Density vs Nodes_Traversed

- For Manhattan heuristics, the number of cells traversed increases starting from 221 for probability 0.004. As the density increases the number of cells traversed to reach the target node increases as we will have to follow more convoluted path to reach the target node. We can see the above trend because Manhattan is the ***most optimistic heuristic*** and for each node in the cell, it will give us the precise ***h(n)*** value to reach the goal node. So, while selecting Manhattan distance as our heuristic, the A* algorithm will only process and traverse the nodes which are absolutely necessary ignoring the useless ones.

- For Euclidean heuristics, the number of cells traversed starts from 9958 for a mere density of 0.004 and then decreases as and when probability of cells getting blocked increases. Same happens for Chebyshev too.

- This is because, both Euclidean and Chebyshev distance underestimates the cost of reaching goal node from a current node. Because of this we tend to *stray more away* from the nodes which we must directly choose to explore that leads to the goal node more effectively. By using Euclidean and Chebyshev distance, we ignore this kind of useful cells and in-turn reach those cells only after selecting the cells which don't provide any useless information because those cells will have ***lower h(n) value hence lower f(n)*** than the cells we should travel to get to the goal node quicker.

- But for increasing probability, we see a decrement in the number of nodes getting traversed as we ignore the blocked states more effectively. The on-hand information now becomes better because we ignore cells which should have been explored (which occurs while using Manhattan distance) and visit smaller number of blocked cells. In other words, bumping into useless nodes (more like blocked nodes), helps in ***eliminating the path choices from a certain node***. The underestimation of the cost now leads to decrease in the number of cells being traversed.

- Though there is a decrease in number of nodes getting traversed with increase in density for Chebyshev and Euclidean, it never matches the exactness of Manhattan to estimate the cost. In grids, to find optimal path, we will either go first right till edge then downwards or first downwards till edge and then right. This information is being utilized by Manhattan and so it proves to be a true optimistic heuristic and is ***uniformly better than*** the other too.

- Time can also be a measurement for comparison but is not calculated because time taken by an algorithm is directly proportional to the nodes traversed so it becomes redundant to choose time as measurement to compare the same algorithm but for only different heuristics.

Question 6

- *A brief about Repeated Forward A* Implementation:*

Initially, the agent has no information about the grid world and is open for exploration for it, so he ***assumes that all the nodes are unblocked.*** The agent is at the start node of the maze. At the start node it plans a path to reach the goal node (done by a call to A* algorithm). Now it starts to move along this path to reach the target node. While moving along this path, it updates the knowledge of current node as well as of the surrounding neighbors.

The agent might discover a blocked node on its pre-planned path to the target node. So, when an agent discovers a block node it considers planning a new path (again done by a call to A*)

This repeated use of A* search algorithm whenever a block node is discovered in the planned path is called Repeated Forward A* search.

- For question 6, we are using the Repeated Forward A* algorithm with surrounding field of view to solve generated grid worlds with the best heuristic obtained from the above question which is the ***Manhattan Distance.***

- Here we solve the grid world with the value of probability $p_o = 0.23$ found from question 4. We select the range of density values from 0 to min(po,0.33) which equals to (0,0.23) for.' We took 101*101-dimension grid worlds for the computations too.

1.) *Density vs Average Trajectory Length:*

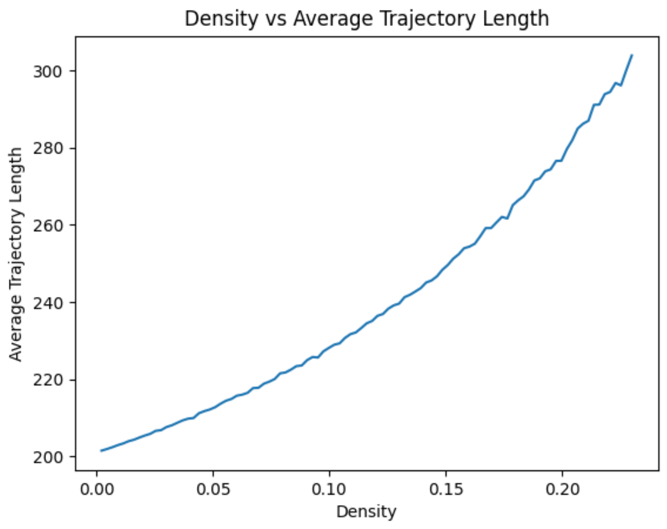


Figure 6. 1 Density vs Average Trajectory Length

- Trajectory length is the total sum of the number of cells that the agent has walked on while traversing from the start node to the goal node. This also includes any backtracking that might have occurred when a block node has been observed and a new path has been planned.

- To find the average trajectory length in the range of probability po (0,0.23), we took step size of 0.01 in (0,0.23) and iterated for 1000 different grid worlds for each step of probability.

- We observed that as the density of blocked nodes in a cell increases the average trajectory length also increases. This is bound to happen and was expected too because of the influx of a block nodes with increasing density. Due to increase in block nodes the agent at times might have to backtrack to an unblocked node and plan another path considering this unblocked node as start node i.e., running the A* algorithm on that node, which in-turn increases the overall trajectory length. At p_o 0.23 the trajectory length is at 303 as we find more blockages along the way.

2.) *Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Grid world)*

- When the agent reaches the target node successfully with or without backtracking, it has discovered some part or whole of the grid world by updating the environment of the cells it has traversed on and its corresponding neighbors. Most of the times, it will not have discovered every bit of the grid world. This final grid world with all the information which is not complete that has been stored by the agent during traversing phase is called the final discovered grid world and in 99 percent of cases will be very different from the actual grid world with full information- ***full grid world.***

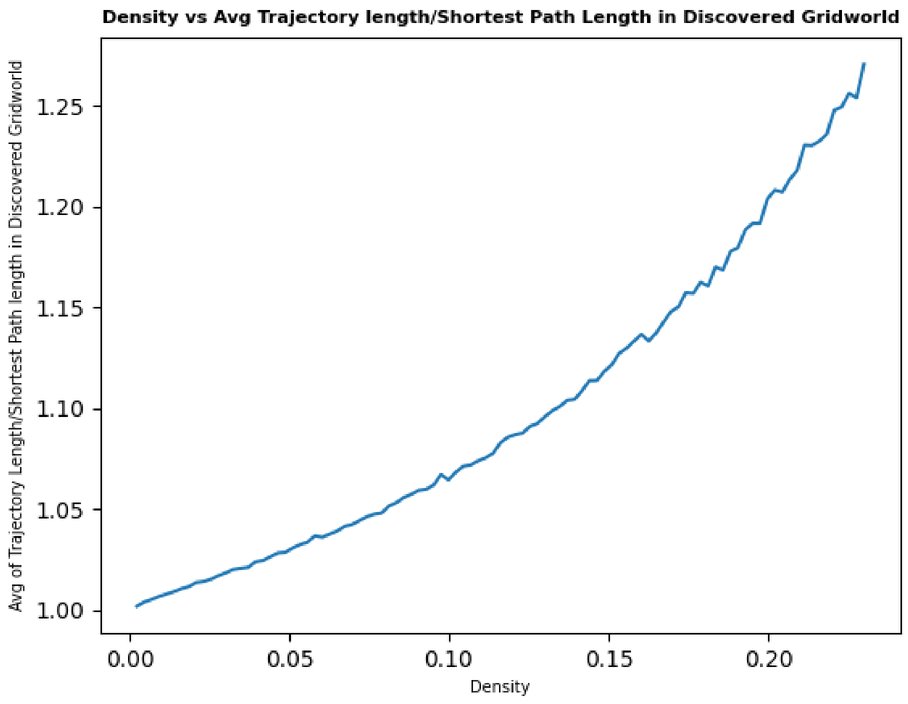


Figure 6. 2 Density vs Average(Trajectory length/Shortest path in final discovered grid world)

- The length of the shortest path in the final discovered grid world will be the length of the path obtained by the agent when it again travels on the final discovered grid world without any backtracking from the start node to the goal node.

- The ratio of length of trajectory and the length of the shortest path in the full grid world will always be more than or equal to 1. In most of the cases it will be more than one which is exactly what the graph demonstrates. This is because, length of trajectory will contain repeated nodes because of backtracking and shortest path in the final discovered grid world will not contain any backtracking (because of the use of A* algorithm on the final discovered grid world).

- The graph also shows that the ***ratio increases as density increases*** and this was expected too because when density of blocked nodes increases, ***more backtracking*** will occur which results in more repetition of unblocked nodes. But at the same time, length of the shortest path in the final discovered grid world won't change that much on an average. The ratio starts from 1 as low density means less backtracking and goes up to 1.27 which attest the stated explanation.

3.) Density vs Average (Length of Shortest Path in Final Discovered Grid world / Length of Shortest Path in Full Grid world.)

- Length of shortest path in the final discovered grid world is the value that we extracted in the above graph. A full grid world is the original maze where we have the exact knowledge of the blocked and unblocked cells in the grid world. Hence the length of the shortest path in the full grid world will be the shortest path in the grid world and ***should be shorter*** than the length of shortest path in the final discovered grid world in most cases.

- This should follow because as presented in Q3 with a counter example, the final discovered grid world contains only limited information about of the actual full grid world. So, in most of the cases, traversing again on the now discovered grid world will not give us the optimal path. Shortest path will be obtained but will only be shortest for the final discovered only and will be greater than the shortest path in full grid world as that will be the optimal path.

- The ratio of the shortest path in the final discovered grid world and the full discovered grid world will be more than or equal to 1. ***In most cases it will be more than 1.***

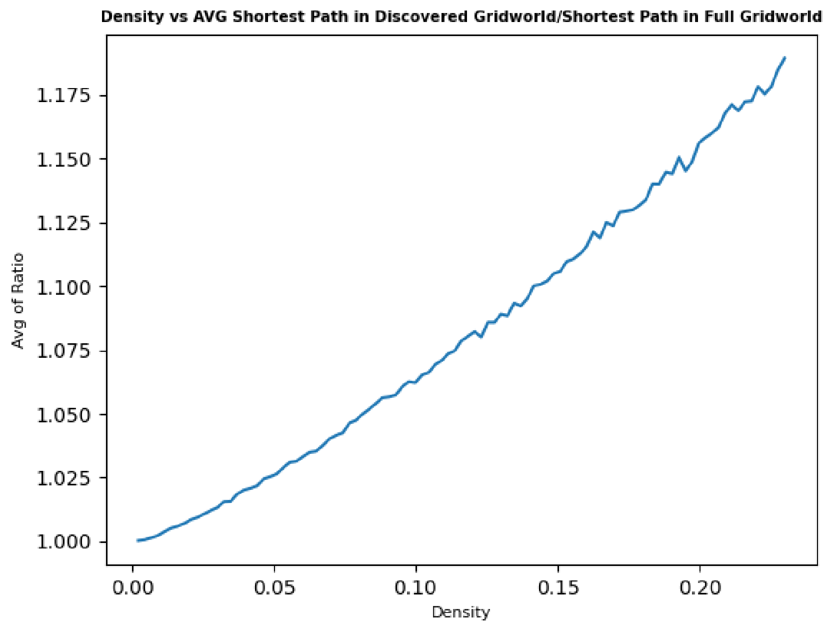


Figure 6. 3 Density vs Average(Shortest path in discovered gridworld/shortest path in full gridworld)

- Also, the ratio increases as the density of the blocked nodes increases. This is because, when density of blocked nodes increases, the shortest path in the discovered grid world will become more and more ***convoluted and crooked***. It will not be possible to reach the goal nodes with 0 or smaller number of backtracks. The graph also explains this as when density is near to 0, the ratio is near to 1 and when density increases, the ratio also gradually increases and reaches to when density becomes.

*4.) Density vs Average number of cells processed by Repeated A**

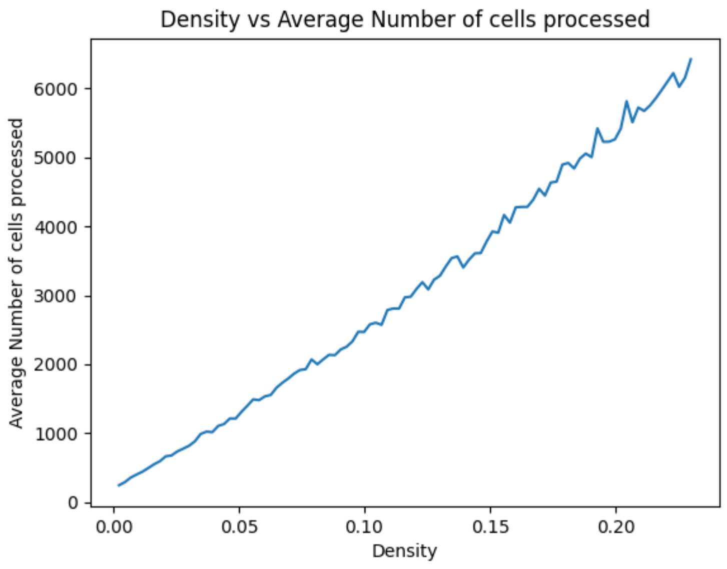


Figure 6. 4 Density vs Average number of cells processed

- Total number of cells processed can be defined as the total number of cells agent processes in his every planning phase. This means the total of the summation of every node popped off from the fringe during each call to A* whenever a blockage is discovered.

- We see an increasing trend in the graph for density vs Average number of cells processed by Repeated A*. This is purely because when density increases, a ***greater number of path planning phases will occur.***

- For low density we see that the number of cells starts from approximately 200 and then reaches to the highest of approximately 6200 for density 0.23.

Question 7

- Here the agent's field of view is crippled as he can only see the direction of the attempted motion and using this only, he moves ahead to solve the grid world. Because of this, the information that was available when the agent reaches a particular cell in the grid world i.e., the information of the surrounding cells is ***decreased by 75 percent*** and so the agent receives the information of a cell being blocked when it bumps the block. The consequence of this will be that agent will have to check more for the blocked nodes and this decreases the overall performance of the algorithm.

- We have plotted the same graphs as in the question 6 but have run the algorithm for 500 grid worlds (101*101 dimension) per 0.01 step of probability in the range of (0, 0.23).

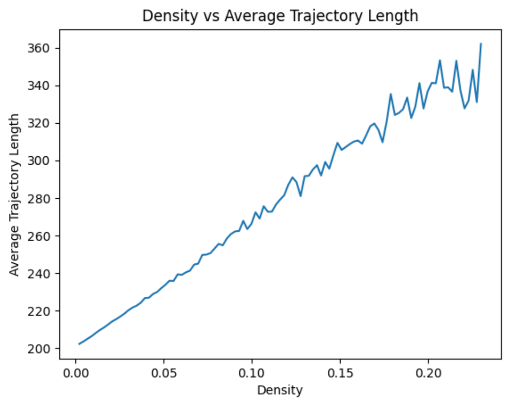


Figure 7. 1 Density vs Average Trajectory Length

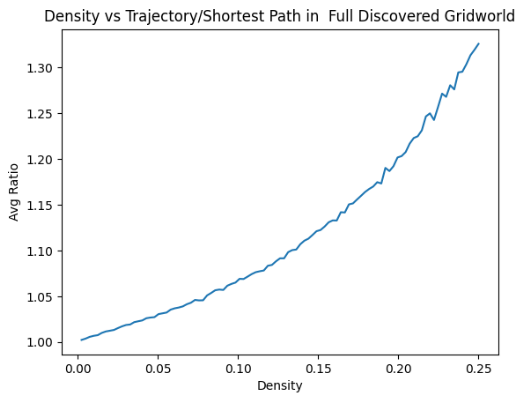


Figure 7. 2 Density vs Avg. Trajectory/Shortest path in full gridworld

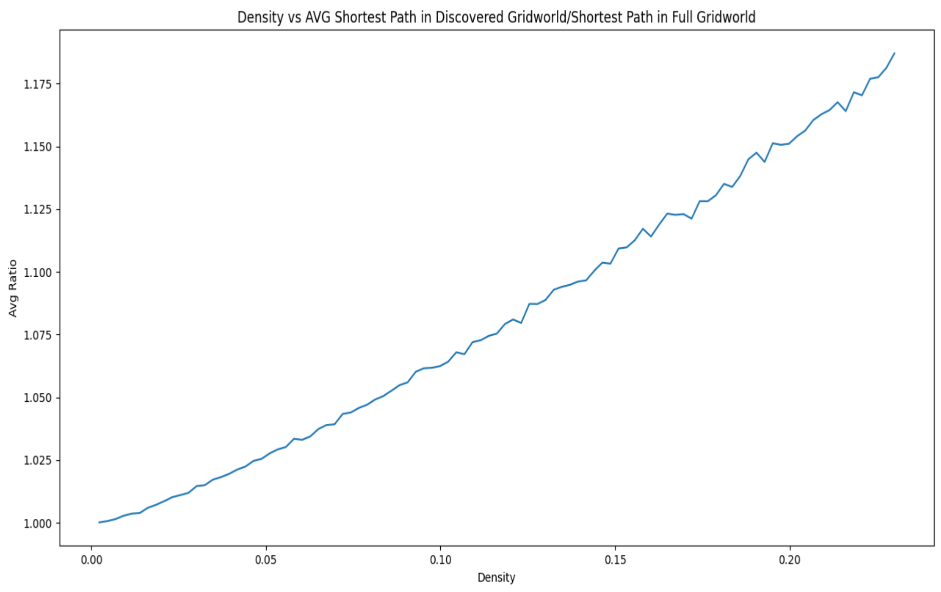


Figure 7. 3 Density vs Avg shortest path in discovered gridworld/shortest path in full gridworld

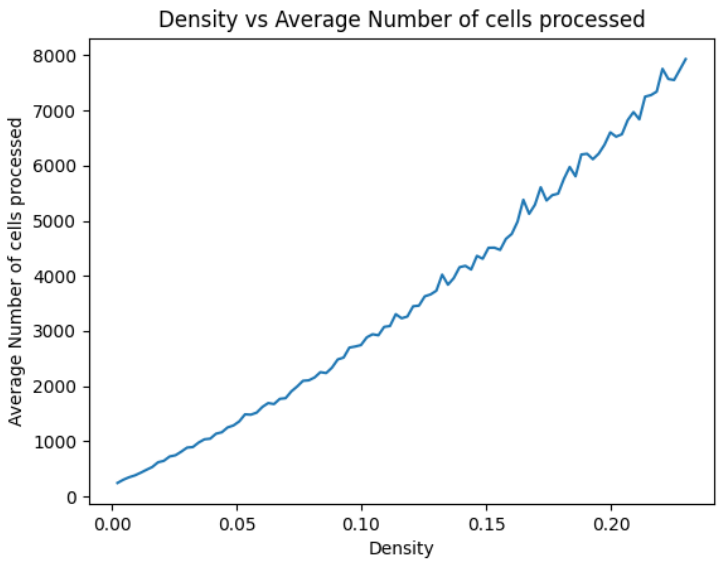


Figure 7. 4 Density vs Average number of cells processed

- The trajectory length of the agent slightly increases in these cases given the agent uses the direction of the attempted motion as the field of view.

1.) <u>*Density vs Trajectory Length:*</u>

- Because of the decrement in the field of view by 75 percent, the trajectory length is expected to increase when density increases, and we got similar results too. Initially when ***density is low, the trajectory length matches*** with the one in Q6 with full field of view but as and when ***density decreases, we can that this similarity between trajectory lengths for Q7 and Q6 decreases***. Trajectory length later increases by *roughly 15 percent* when density is at its peak i.e., at 0.23.

2.) <u>*Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Grid world)*</u>

- Comparing the data that we get from the plotted graph of Q7 and Q6 for the Density vs Average (Length of Trajectory / Length of Shortest Path in the Final Discovered Grid world) we don't see that much of a change when density is less because there will be lesser number of blocked nodes, so no backtracking is needed. But when density increases, we see that the given ratio for Q7 becomes more than the ratio for Q6. This is expected because for question Q7, trajectory length increases on a larger scale than Q6. The graph is almost same with one difference that for density 0.23, the ratio for Q7 reaches to 1.30 which.

3.) *Density vs Average (Length of Shortest Path in Final Discovered Grid world / Length of Shortest Path in Full Grid world.).*

- We don't see any change in the ratios of Q6 and Q7 throughout the range of density because even with a greater number of backtrackings in Q7 with limited field of view, the agent reaches the goal node and will have discovered the same amount of grid world discovered in Q6. So, the final ratio won't change that much.

- We see the graphs for Q6 and Q7 for the density vs given ratio is same throughout.

4.) *Density vs Average number of cells processed by Repeated A* with limited field of view:*

- This is ***the proper metric*** for comparing the efficiency of agent with field of view and with limited field of view. For Q6, we saw that average number of cells processed starts from 200 for extremely low density and increases up to 6000 for high density (0.23). For Q7, the graph starts with the same number of cells processed but when density increases to 0.23, we see that the number goes to 8000. This was bound to happen because for Q7, the agent doesn't have full field of view and bumps into more block cells. This leads to greater planning steps than for Q6 and so number of processed nodes would increase. We can see this being demonstrated by the clear difference between the data received from graphs of Q6 and Q7.

- We can also state that computational time increases in Q7 with limited field of view. This is because, computational time is directly proportional to the number of cells processed. So, Q6 version of Repeated A* will be slightly faster (in milliseconds) than Q7 version.

Question 9:

Below we can see the mazes with h(n) values for all the cells. These mazes for different heuristics, helps us in understanding the actual functioning of the Repeated A* algorithms when different heuristics both admissible and inadmissible are use

8	7	6	5	4
7	6	5	4	3
6	5	4	3	2
5	4	3	2	1
4	3	2	1	0

Figure 9. 1 Manhattan Heuristics.

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 9. 2 Weighted Manhattan Heuristics

5.65	6.0	6.47	7.1	8.0
6.0	6.24	6.60	7.16	8.0
6.47	6.60	6.82	7.23	8.0
7.12	7.16	7.23	7.41	8.0
8.0	8.0	8.0	8.0	8.0

Figure 9. 3 Euclidean Heuristics

11.31	11.0	10.94	11.24	12.0
11.0	10.48	10.21	10.32	11.0
10.94	10.21	9.65	9.47	10.0
11.24	10.32	9.47	8.82	9.0
12.0	11.0	10.0	9.0	8.0

Figure 9. 4 Weighted Euclidean Heuristics

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 9. 5 Chebyshev Heuristics.

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 9. 6 Weighted Chebyshev Heuristics

Comparison between Manhattan and Weighted Manhattan (Inadmissible) Heuristics:

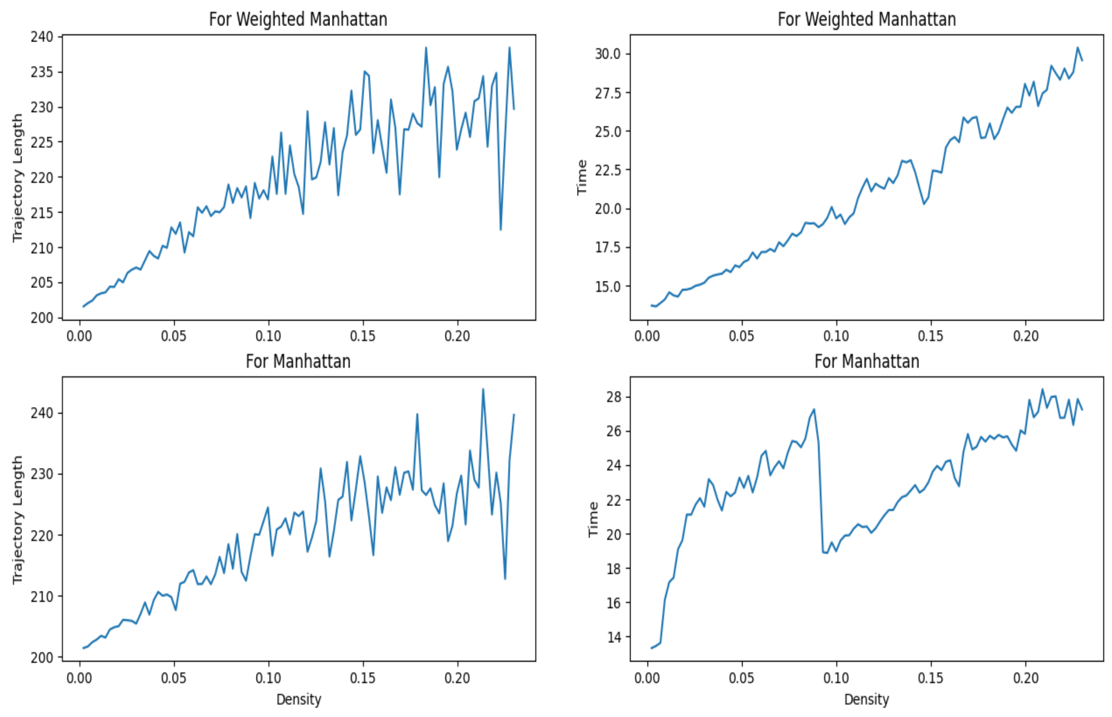


Figure 9. 7 Manhattan vs Weighted Manhattan(Inadmissible) Heuristics

The above graph distinguishes between the runtime and average trajectory length for Admissible Manhattan and for Inadmissible Manhattan. We have multiplied the Manhattan distance by 2 so that we get a new Weighted Manhattan which is inadmissible. Then running Repeated A* for using both variants of heuristics for density range (0,0.23) for 200 different grid worlds for each 0.01 step size in density range, we can observe the following results:

- We can' see much of a difference in the trajectory length. Also, Manhattan distance is the most optimistic heuristic, and no heuristic can outperform it because of its ability to calculate h(n) with utmost precision.

- We can see that inadmissible weighted Manhattan Heuristic, performs poor when it comes to computational time. When inadmissible heuristic is used, the time taken is more right from the beginning. We see that the graph for density vs time starts at 14.5 seconds and ends at 30 seconds at high density. But for Manhattan, it starts at around 13.5 and end at 28 seconds. This happens because when applying inadmissible heuristic more cells are processed.

So, no inadmissible heuristic can outperform Manhattan no matter how much we try to change the heuristic and by how big or small of a scale we make it admissible.

Comparisons between Normal Chebyshev and Euclidean Heuristics and their inadmissible weighted counterparts

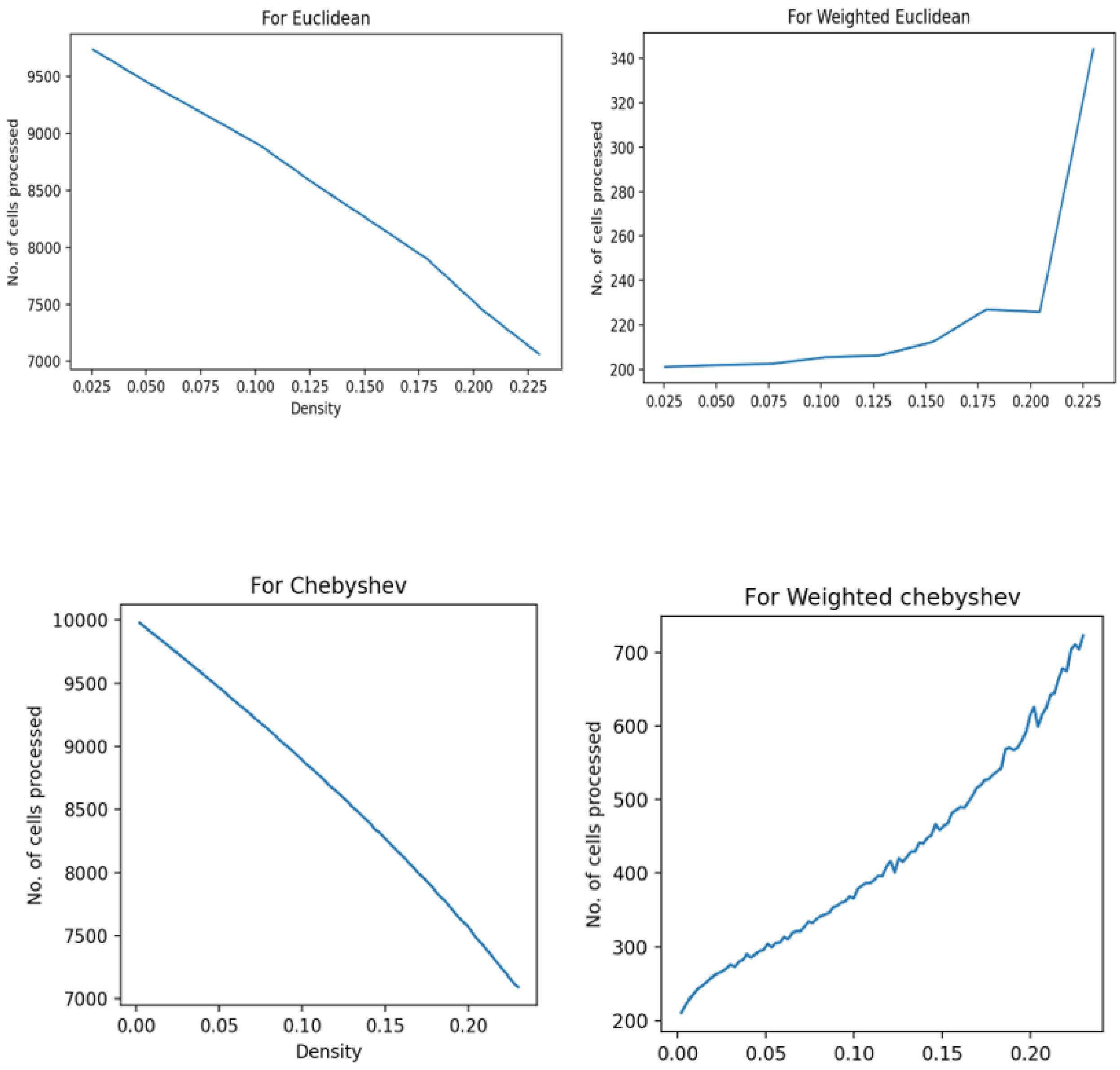


Figure 9. 8 Chebyshev vs Weighted Chebyshev(Inadmissible) & Euclidean vs Weighted Euclidean(Inadmissible)

In normal Chebyshev and Euclidean Distance Heuristics, the agent starts by processing almost all the cells in the maze for probability 0. Gradually as the probability of blocked cells increases the number of cells processed decreases as the agent starts moving along a more specified path to reach the goal node. But in weighted heuristics the number of cells processed are around 200 initially when running on A*, which is about very less compared to normal heuristics. This is because the in weighted heuristics the value of H(n) is now estimated with much more precision. Thus, the agent will discard the nodes that are towards the edges as it will have more value of F(n). Agent moves around the diagonal path in the weighted heuristics to reach the goal node.

As shown the average trajectory length #for density 0.23.

In weighted Chebyshev and Euclidean Heuristics, the value of H(n) is considerably more with respect to normal heuristics value for a given maze. We are actually ***doubling*** the value of Chebyshev and Euclidean Distance. Thus, the value of H(n) which was previously underestimated for each cell now becomes more precise. This can be seen from the maze consisting of h(n) values for normal variants and weighted variants for both heuristics.

Trajectory Length and Time Comparison when Repeated A* is used to solve mazes: (graphs here)

Trajectory Length:

For these weighted heuristics the average trajectory length for 0.0 probability almost starts near to 200 and then as the number of blocked nodes increases the trajectory length also increases. The same thing happens for the normal variants of the heuristics. So, we don't see a change in the trends and numbers for trajectory lengths of admissible and inadmissible heuristics for Chebyshev and Euclidean.

Time Comparison:

In weighted Chebyshev and Euclidean Heuristics, the time taken for solving 200 mazes using Repeated A* for probability in range of (0,0.23] increases gradually from # seconds to around # seconds. This occurs because for weighted heuristics agent traverses through the lesser number of nodes initially, but with increases in block nodes in the maze the overall runtime increases. Initially for weighted Heuristics the agent estimates the value of H(n) to be more than as compared to normal heuristics.

In normal heuristics the time taken for solving 200 mazes using Repeated * for probability in range (0,0.23] decreases gradually starting from 93 seconds from start and reduces to # for density as high as 0.23. Here we can see the actual difference between inadmissible and admissible heuristics. The time by admissible heuristics is very large compared to the inadmissible heuristics. By adjusting the underestimation of each cells h(n) value in the admissible ones, we can see a drastic decrease in solving mazes through Repeated A*

Extra Credit: Using Repeated BFS instead of Repeated A*

Brief intro to BFS and Repeated BFS:

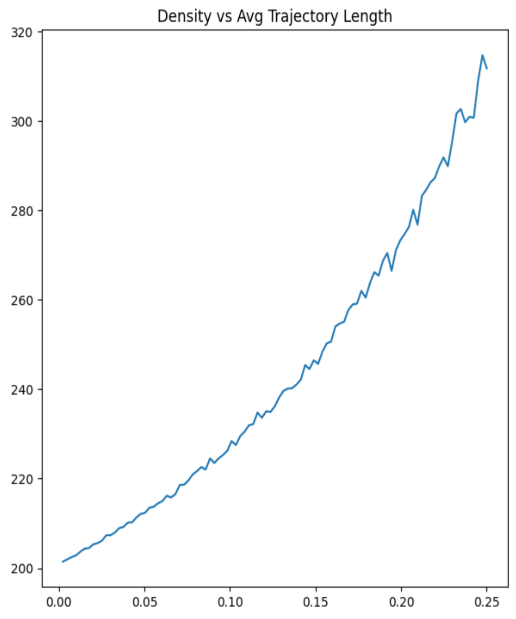
- Breadth First Search algorithm as the name implies, works by analyzing all the nodes that are at the same level or have same probability of reaching a goal node. It will analyze all such nodes at same level and generate their children (neighbors in grid world) and later analyze these newly discovered children in the same manner as their parents were analyzed.

- Considering our problem of finding the goal node in the grid world consisting of blocks, the concept of Repeated BFS is the same as the concept of Repeated A*. The only difference is that when planning phase occurs, the path that the agent plans its path using the BFS algorithm. So, for each discovery of block cells, BFS is called instead of A*, hence the name Repeated BFS.

Question 6

For the calculations and formulation of different graphs we have used density range of (0, 0.25) and ran our algorithm for 200 differently generated 101*101-dimension grid worlds for each step size of 0.01 in density range.

1.) Density vs Trajectory Length:



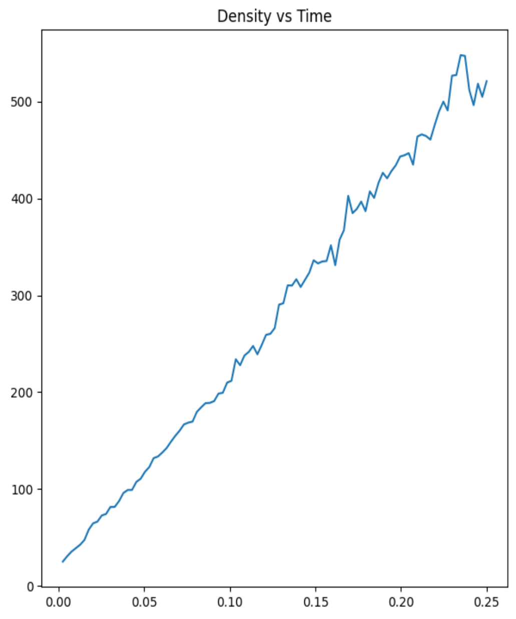


Figure 10. 1 Density vs Trajectory Length

Figure 10. 2 Density vs Time

- Trajectory length is supposed to be dependent on the number of blocked cells discovered on the path and not on the algorithm. This is further proved by our graph for Density vs

Trajectory. For BFS the graph starts from 200 and then moves up with increasing density and reaches up to approximately 300 for density 0.23. This same exact thing happens with the use of Repeated A*. So, there is no observed change in the average trajectory length for Repeated BFS and Repeated A*.

- The most important thing to notice where the 2 algorithms differ in extremely large scale is the ***computational time***. Computational time for Repeated BFS is extremely more than the Repeated A* because of the working mechanism of the BFS algorithm. With A*, it is more devoted to a single route until it finds a blockage, but for BFS, every cell in the grid worlds (except block cells) are more likely to be processed.

2.) <u>*Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Grid world)*</u>

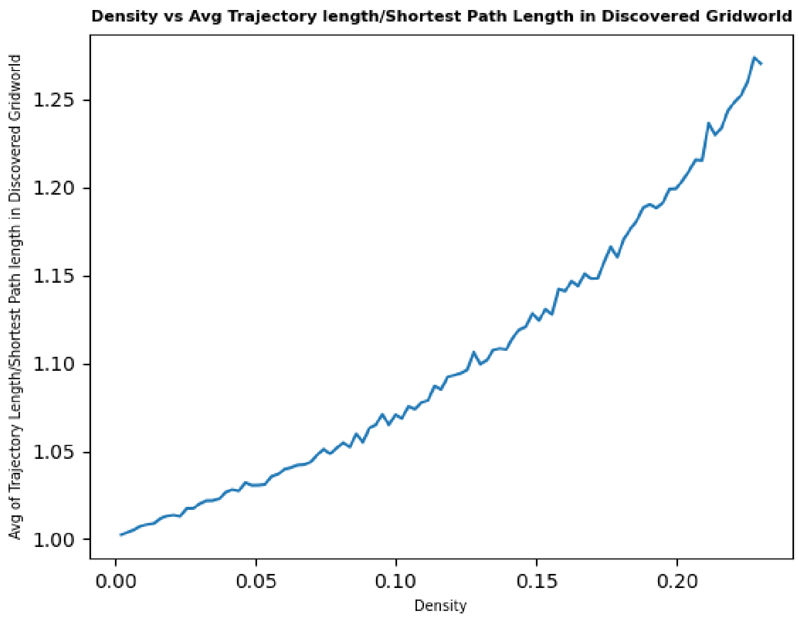


Figure 10. 3 Density vs Avg of Trajectory Length/shortest path length in discovered grid world

- As mentioned before, length of trajectory is dependent on the blocked nodes in the algorithms, so here also we don't see much of a change in graphs for Repeated A* and Repeated BFS. The graph starts from closer but more than 1 for low density which it should because here the trajectory found in the first try will most likely be the shorted path in the final discovered grid world. But we see that ratio steadily increases as density increases because the trajectory length will increase but at the same time, we won't see much change in the shortest path in the final discovered grid world. The ratio reaches to an all-time maximum of 1.25 for density 0.23 which is the same as of Repeated A*.

3.) *Density vs Average (Length of Shortest Path in Final Discovered Grid world / Length of Shortest Path in Full Grid world.).*

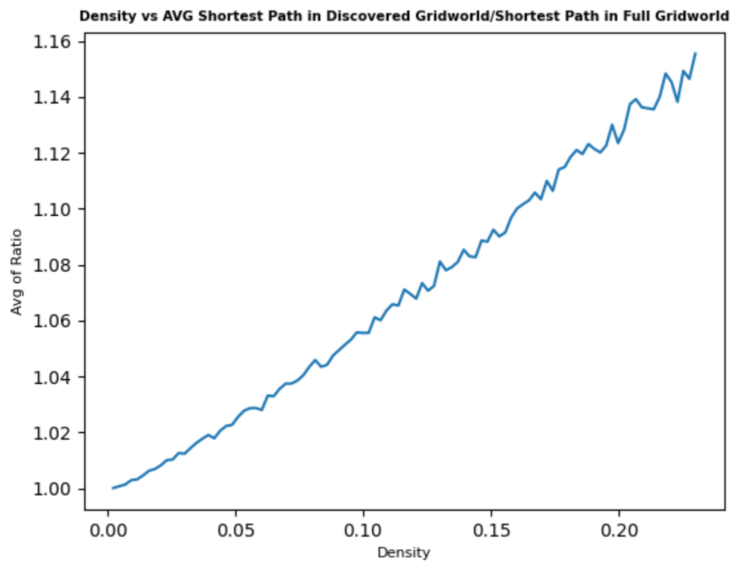


Figure 10. 4 Density vs Avg shortest path in discovered grid world/shortest path in full gridworld

- Here also we see the same trend as observed in the graph of Repeated A* Algorithm. As density increases the ratio gradually and steadily increases from near to but more than 1 to the peak of 1.16 for density 0.23. This is a similar result that we got from Repeated A* too.

4.) *Density vs Average number of cells processed by Repeated BFS:*

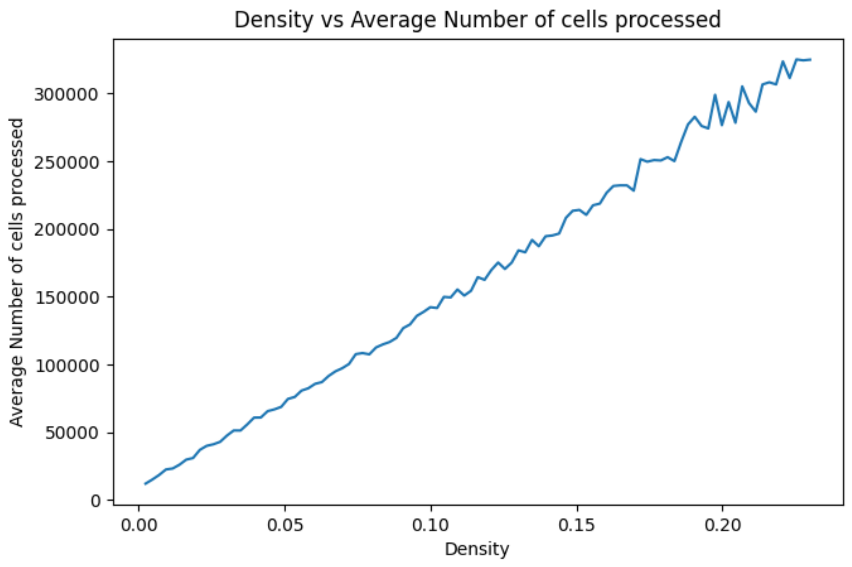


Figure 10. 5 Density vs Average number of cells processed

- This is where we can see the true nature of Repeated BFS. The computation of the cells processed by the Repeated BFS and the graph can help us differentiate between Repeated A* and Repeated BFS- their workings and the efficiency in reaching a goal node.

- We can see a clear increase in the number of cells processed by Repeated BFS than the cells processed by Repeated A*. The number increases on a magnitude of more than times 50. For Repeated A*, the number of cells processed starts from 200 with low density and reaches to a max of 6000 for density as high as 0.23.

- But for Repeated BFS, we can see that the graph itself starts from 12000 (number got from the output data). This is more than the highest number of cells processed in Repeated A* for a high density of 0.23. And as density increases, the number of cells processed increases steadily and quickly and crosses the mark of 300,000 for density of 0.23.

- This shows how bad Repeated BFS is in comparison to A*.

- This graph can also shed light on the computational time for Repeated BFS. As computational time is directly proportional to the number of cells processed, the time taken by Repeated BFS is also extremely large than the time taken by Repeated A* to reach goal node.

- So, in conclusion, BFS can reach goal node, but it is an extremely bad choice considering the high computational time.

Question 7

Here is this version of Repeated BFS we are crippling the field of view of agent as we did in Question 7 for Repeated Forward BFS. For the calculations and formulation of different graphs we have used density range of (0, 0.25) and ran our algorithm for 200 differently generated 101*101-dimension grid worlds for each step size of 0.01 in density range.

1.) Density vs Trajectory Length:

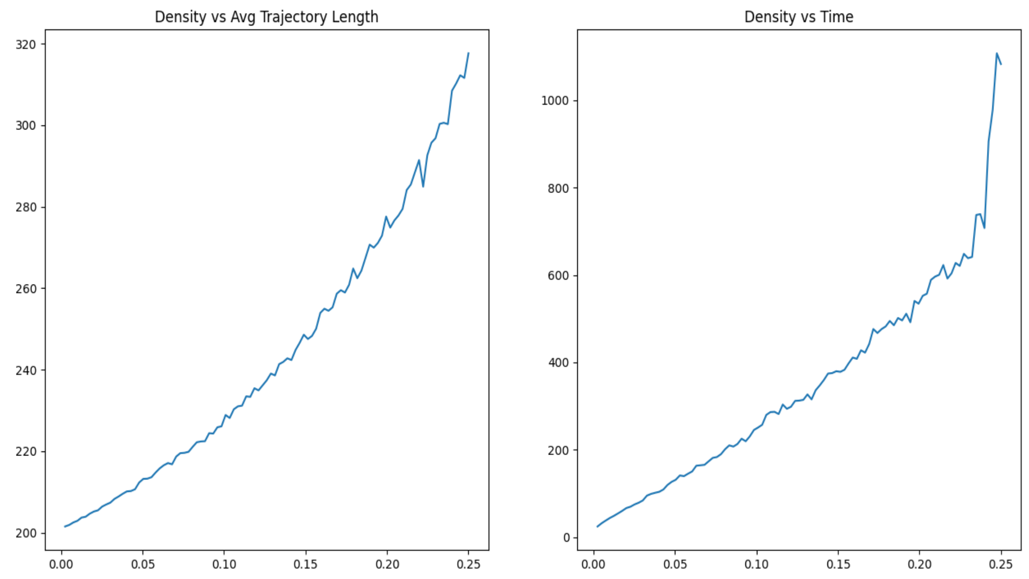


Figure 10.2 1 Density vs Trajectory Length and Time

- By crippling the agent's 75 percent field of view, we are making agent very inefficient in handling blocked nodes. The consequence of this reduction in field of view is that agent will perform more path planning steps (more calls to BFS) and because of this as we previously saw in Q6 of Repeated BFS, the overall computation time increases drastically(almost double). The trajectory length remains same on an average.

- The trajectory length starts from 280 for lowest density of 0.023 and goes up to 320 for density as high as 0.23. This was also expected because of the increase in the blocked nodes. We can see that for limited field of view Repeated BFS, the trajectory length doesn't increase that much in comparison with Q6 of Repeated BFS.

2.) *Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Grid world)*

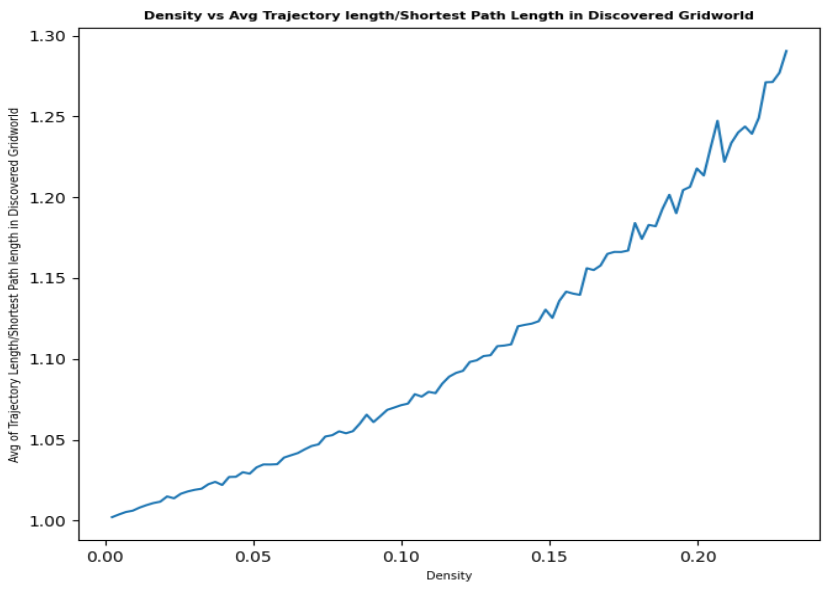


Figure 10.2 2 Density vs Avg Trajectory length / Shortest path length in discovered grid world

- The rule that applied for Q7 of Repeated A*, the same rule applies here too. With reduced field of view, the ratio of (length of trajectory) / (length of shortest path in final discovered Grid world) first starts from 1 when density is extremely low because there would be no or very a smaller number of backtracking and as the density goes up, backtracking increases which increases the trajectory length but there would be no significant amount of difference observed in the length of shortest path in final discovered grid world. So, in turn, this increases the ratio to 1.3 for the highest density of 0.23 which is a little bit more than what we observed for Q6.

3.) *<u>Density vs Average (Length of Shortest Path in Final Discovered Grid world / Length of Shortest Path in Full Grid world.).</u>*

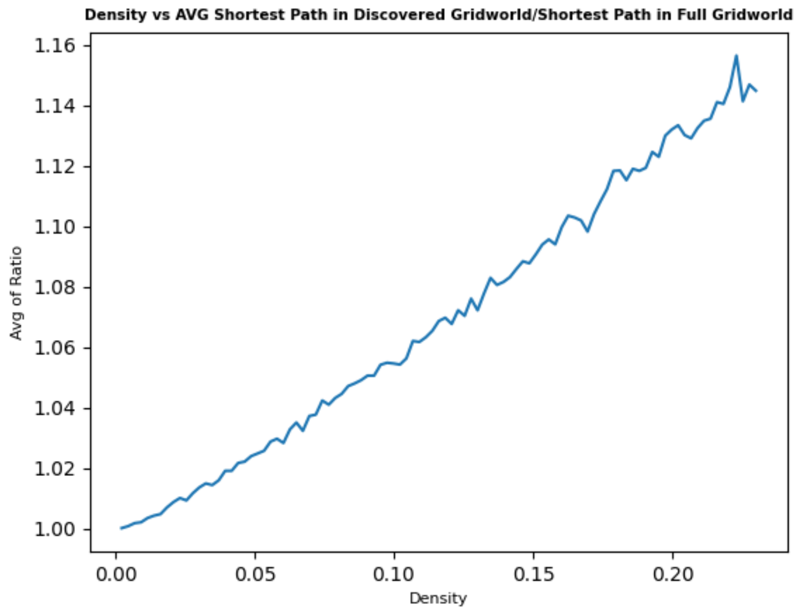


Figure 10.2 3 Density vs Average shortest path in discovered grid world / shortest path in full grid world

- For this ratio on an average, the initial computations of shortest path in final discovered grid world would be the very similar if not same as the length of the shortest path in full grid world for very low density because of 0 or low number cell bumping. The path in both grid world with different knowledge would be different, but for low density, the length of both these paths would be similar. And as the density increases, the ratio increases and goes up to 1.15 for the highest density of 0.23. This is almost same as what we observed in Q6 which was totally expected for the reason mentioned above in the case of Q7 for Repeated A*.

4.) *Density vs Average number of cells processed by Repeated BFS:*

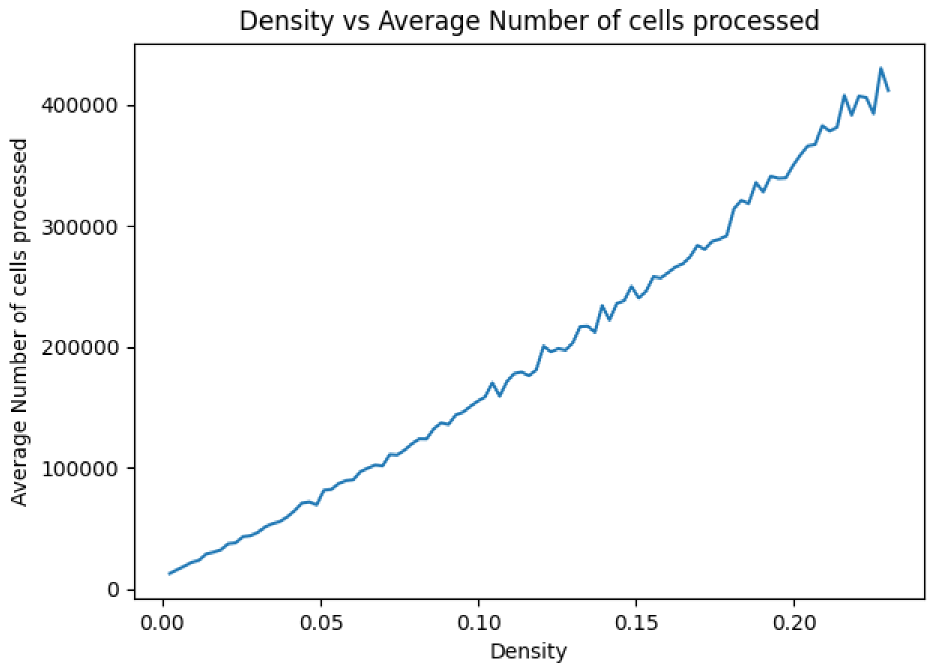


Figure 10.2 4 Density vs Average number of cells processed

- This is computational metric which really captivated us but was also expected as it was the same result that we got when we compared the 2 variants of Repeated A*. As we can see in the graph, the number of nodes processed for the current BFS starts somewhat in the same manner as we saw for the Repeated BFS with full field of view. But as and when density increases, we could see that for the current BFS, the number now doesn't coincide with the Q6 variant of Repeated BFS. It is more than the Repeated BFS and is also supposed to happen too because as density increases, more number of blocked nodes are met by agent and so more number of path planning steps occurs which makes the number of cells processed shoot up. For density as high as 0.23, the number comes around 400000 which is almost 1.3 times the result that we got from the Q6 of Repeated BFS.

- Thus, crippling the field of view creates a lot of adverse effect on the total runtime (because of a greater number of average cells processed) as well as the efficiency of the Repeated BFS algorithm.

Question 8: Extra Credit

In this question, instead of calling the Repeated A* from last discovered unblocked cell, we are backtracking more to see if we can reduce the overall computational time and trajectory length. This is an attempt to see if we benefit from this or not. So for this, we have formulated Repeated A* such that when agent observes that the next cell is blocked, instead of calling from the location he is at, it will move more steps back to see if it has some more favorable position from where it can plan the path. This is how it reuses the information that he already has.

We got the following results when we computed both the current Repeated A* and the Q6 Repeated A*(NORMAL ONE). Each one is made to run for 200 different grid worlds for each step size of 0.01 in density range (0,0.23):

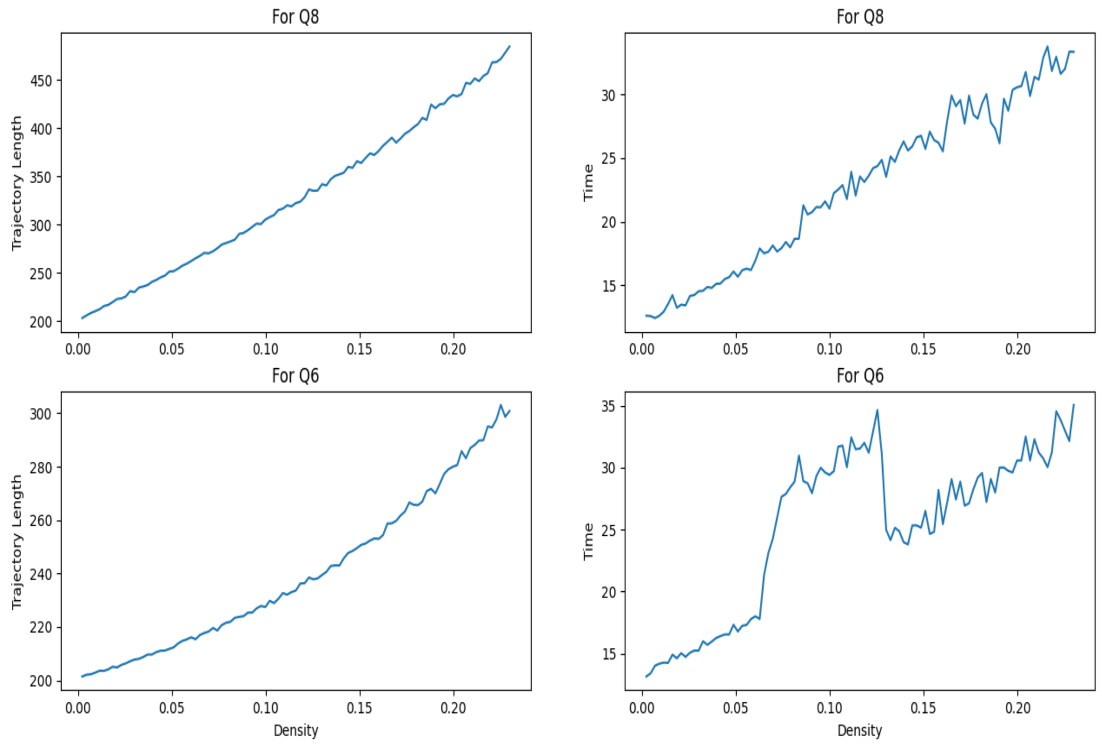


Figure 8.1

In the graph, the upper 2 graphs are for the present version of Repeated A* with a greater number of backtracking steps. Below 2 graphs are for normal variant of Repeated A*.

Comparing the 2 graphs of average trajectory lengths for Question 8 and Question 6, we see that the average trajectory starts at 200 for both versions of Repeated A* when density is very low. But as and when the density increases, we don't see a drastic increase in normal Repeated A* but when it comes to Repeated A* for Q8, the increase takes place on a larger scale. This is because the added cost of backtracking the same path for 'n' number of times until we reach a better node for path planning which is 'n' cells away from agent's current position. The average trajectory length becomes all time highest of

300 for density 0.23 in case of normal Repeated A* but for the variant of Q8, it goes as high as 450 when density is near 0.23. So, this is the adverse effect of backtracking more.

But when we compare the graphs again for computational time, we see contrasting results than what we got for trajectory lengths. For normal Repeated A*, the runtime goes on increasing from reaches approx. 35 seconds to solve 200 mazes with density 0.23. But for the Q8 Repeated A* variant, we see that it goes only as high as 30 seconds for highest density. Thus, we see the benefit of backtracking more over here. This decrement in the runtime is because the agent has avoided a greater number of path planning steps when density is high. With the normal Repeated A*, the agent will plan the path more times using A* algorithm for each blocked nodes which raises overall runtime. With more backtracking, we are avoiding this as we neglect more cells when we backtrack to a farther node. This makes the path choices less. If this were not to happen, then for each cell that is not backtracked, for high density, the agent might've ended up making more path planning steps i.e., calling the A* Algorithm. We don't see a drastic reduction in time complexity, but there is a decrement for sure.

Thus, in certain cases, backtracking more on the path that has already been travelled leads to decrement in time with the cost of added trajectory length.

APPENDIX:

Function for Repeated A*:

```
# Import necessary requirements

from src. create_maze import create_maze
from src.set_attributes.set_attr_for_new_maze import set_attr_new_maze
from src.AStar import astar   # importing A* to use during planning phase


def repeated_forward_astar(MAZE, DIM, x, y):
    """
    This is the function of Repeated A*. The agent uses this to plan the path and  actually traverse on it.
    :param MAZE: original maze with all full information used by agent when it walks down the path.
    :param DIM: the dimension of maze.
    :param x: i value in (i,j) denoting start cell (0,0)
    :param y: j value in (i,j) denoting start cell (0,0)
    :return: path planned, blocked-set list, nodes processed
    """

    dim1 = DIM
    # maze with full information.
    og_maze = MAZE
    # create a new maze and update it when agent gets knowledge of the cells.
    new_maze = create_maze(dim1)
    # setting attributes of the newly created maze with all cells unblocked.
    set_attr_new_maze(new_maze, dim1)
    # new_path is the list which contains the final trajectory of agent.
    new_path = list()
    # fist call to A* to plan the initial path.
    path, blocked_cell, cells_popped = astar(new_maze, dim1, x, y)
    # total_cells_popped keeps track of all the cells popped from the fringe by A*.
    total_cells_popped = cells_popped
    # loop until path returned by A* is [-1] which means no path is available, till goal is reached, or path list
    # becomes empty denoting no node from start to goal.
    while path:
        # z contains the indices of the path at 0 position.
        for z in path:
            # checking if path returned by A* is [-1] or not
            if path == [-1]:
                return [-1], total_cells_popped
            # reaching here means path has values in it which the agent will now process and traverse accordingly.
            (i1, j1) = z
            # check if goal is reached. If condition met, then append the goal node indices to the new_path and assess
            # the neighbouring cells in FOV (field of view) and update the environment. Also return the trajectory
            # nodes popped off(nodes processed)
            if (i1, j1) == (DIM - 1, DIM - 1):
                new_path.append((i1, j1))
                neighbours = [(0, 1), (1, 0), (0, -1), (-1, 0)]
                for (X, Y) in neighbours:
                    a = i1 + X
                    b = j1 + Y
                    if a + b > 0 and 0 <= a < dim1 and 0 <= b < dim1:
                        new_maze[a][b].state = og_maze[a][b].state
                return new_path, total_cells_popped
            # if not goal then append it to new_path as agent walks on it. Also update the FOV during walking phase.
            else:
                new_path.append((i1, j1))

                neighbours = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # for updating field of view
                for (X, Y) in neighbours:
                    a = i1 + X
                    b = j1 + Y
                    if a + b > 0 and 0 <= a < dim1 and 0 <= b < dim1:
                        new_maze[a][b].state = og_maze[a][b].state
                # check if next cell in the path returned by A* is blocked or not. If it's blocked, then run the
                # planning phase by making a call to astar with the current cell the agent has walked on.
                index = path.index((i1, j1)) + 1
                (p, q) = path[index]

                if new_maze[p][q].state == 1:
                    path.clear()
                    path, blocked_cell, cells_popped = astar(new_maze, dim1, i1, j1)
                    total_cells_popped = total_cells_popped + cells_popped
                    # popping to avoid 2 entries of the same current cell the agent is on because we have already
                    # appended it and the path from A* will also have the same start cell which will again append in
                    # next iteration
                    new_path.pop()
    # return [-1] as trajectory because no path exits as all cells have been processed and goal is not met.
    return [-1], total_cells_popped
```