CS 520 : Project 1 - Voyage Into the Unknown

Team Info:
1. Harish Udhayakumar - hu33
2. Parvathi Mahesh Hedathri - pm850
3. Kavya Kavuri - kk1069

Q1. Why does re-planning only occur when blocks are discovered on the current path? Why not whenever knowledge of the environment is updated?

Answer:

Every time the agent encounters a block, it backtracks to the previous unblocked node to discover alternate paths. We don't need to replan as long as we encounter a block because, until we encounter a block, the algorithm traverses the agent through a path towards the goal node. We also know that if *h* is admissible, then the first time *A** returns a path, it represents the discovery of the optimal path from S to G. So the agent is already moving on the optimal path, it is unnecessary to do replanning of the path unless the block is on the optimal path itself.

Q2. Will the agent ever get stuck in a solvable maze? Why or why not?

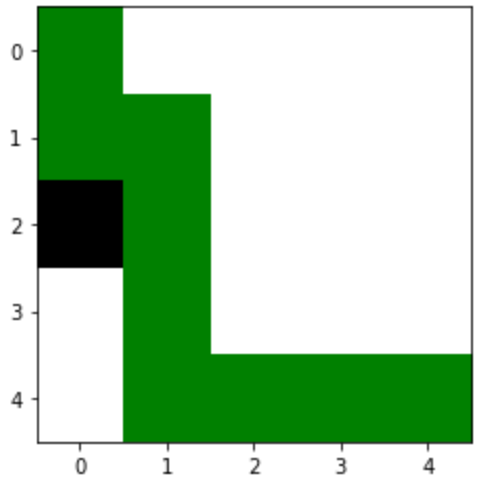
Answer:

In a solvable maze, the agent will not get stuck at any point in time because we explore every possible path on each node that leads to the goal node. When a node is discovered to be blocked, we retrieve the agent to the previous unblocked node and replan alternate paths from there towards the goal node. In the end, we either find the goal node or exhaust all possibilities.
*A** algorithm runs until the fringe of unvisited nodes is empty and continues to discover each node's children on every iteration. Also, it doesn't add any previously visited nodes to the fringe, unless we have a lower cost for that node, in which case a new path is existing and it will be discovered, but at the end, all nodes will be exhausted. This means that *A** explores all the nodes in the maze until it explores the goal node. Hence there is no chance of the agent getting stuck in a solvable maze.

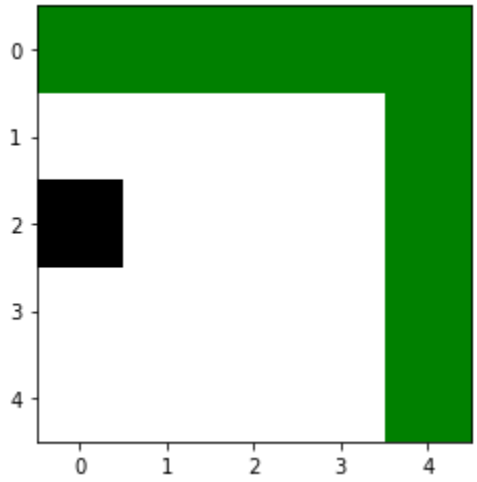
Q3. Once the agent reaches the target, consider re-solving the now discovered gridworld for the shortest path (eliminating any backtracking that may have occurred). Will this be an optimal path in the complete gridworld? Argue for, or give a counterexample.

Answer:

The shortest path in the final discovered gridworld will be an optimal path. It may not be the exact optimal path as *A** returns on the full grid world, but it will be one of the possible optimal paths on the full grid world. In both cases, the length of the shortest paths will be the same.
Ex: *Repeated Forward A** may give this



But *A** may give this:

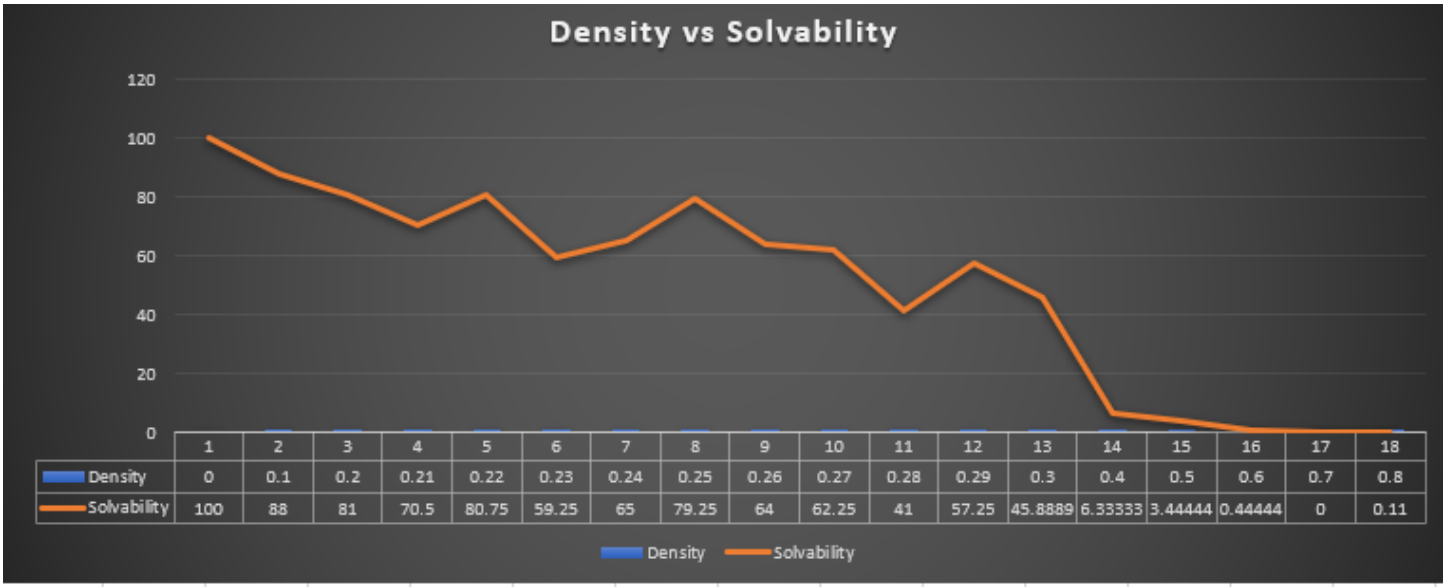


In both cases, the length of the path is 9 and both are optimal paths.

This claim is also consistent with the output received from running both algorithms on different mazes, multiple times. (See figure: bleh in Answer for Q7)

Q4. 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.

Answer:



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Density	0	0.1	0.2	0.21	0.22	0.23	0.24	0.25	0.26	0.27	0.28	0.29	0.3	0.4	0.5	0.6	0.7	0.8
Solvability	100	88	81	70.5	80.75	59.25	65	79.25	64	62.25	41	57.25	45.8889	6.33333	3.44444	0.44444	0	0.11

We have selected 80% to be the solvability limit, in order to select p_0. So the value of p_0 is 0.2 according to the graph, below which solvability is above 80% consistently. This graph is obtained from collating results from running *A** on mazes with dim=101, for 60 times on a range of density values (given in X-axis).

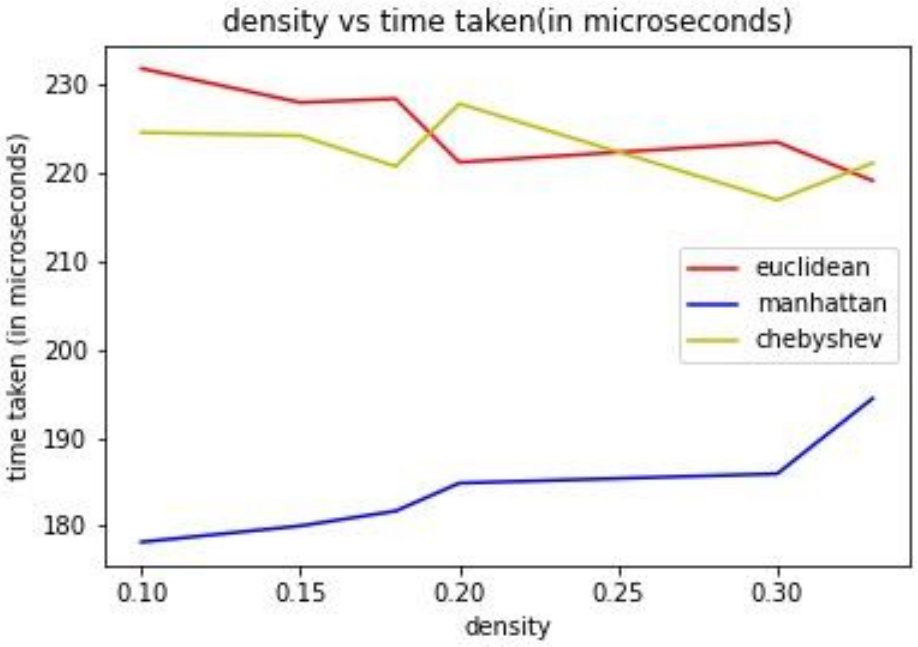
*A** is the best algorithm to use here because, for a consistent heuristic, A* performs the fewest number of necessary expansions required to find the optimal path. BFS, DFS are special cases of A* where heuristic is 0 always.

Q5. Heuristics (a) Among environments that are solvable, is one heuristic uniformly better than the other for running A? Consider the following heuristics: 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.

Answer:

Euclidean distance is the shortest path between source and goal which is a straight line.
Manhattan distance is the sum of all the real distances between source and goal. We know that the distance traveled by the agent from source to goal is greater than its Euclidean distance in a maze problem which means that Euclidean distance is an underestimate of the heuristics. Manhattan distance is the distance which an agent would travel if there were no obstacles. So this is an optimistic estimate but is always less than or equal to the actual distance. Chebyshev distance would be an even more pessimistic estimate because it considers the agent can travel in all directions. For any given configuration, Manhattan>Euclidean>Chebyshev and Manhattan is the closest estimate to the actual distance from the current node to the goal node.

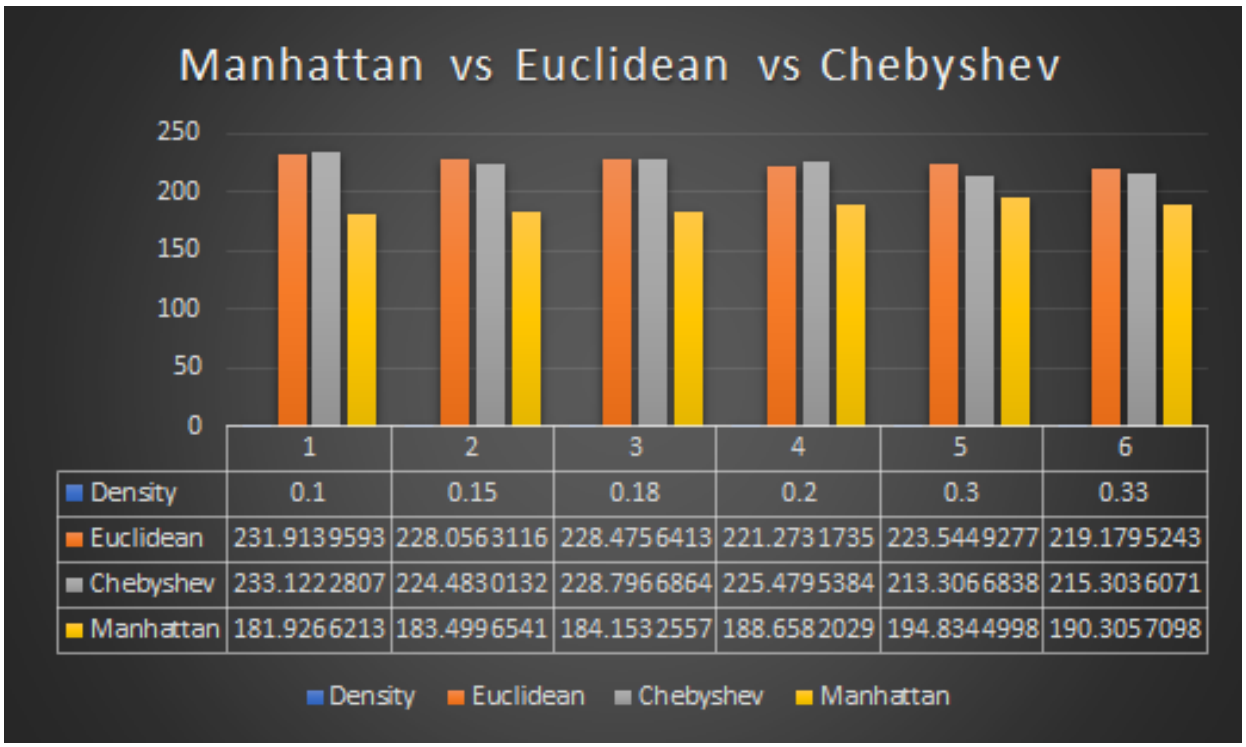
Thus, Manhattan Distance is preferred over other metrics. The following graph demonstrates the same.



Dim = 101

From the graphs, we notice that Manhattan distance takes the least amount of time (microseconds) than that of Euclidean and Chebyshev distances, for a given density.

Figure: 5



	1	2	3	4	5	6
Density	0.1	0.15	0.18	0.2	0.3	0.33
Euclidean	231.9139593	228.0563116	228.4756413	221.2731735	223.5449277	219.1795243
Chebyshev	233.1222807	224.4830132	228.7966864	225.4795384	213.3066838	215.3036071
Manhattan	181.9266213	183.4996541	184.1532557	188.6582029	194.8344998	190.3057098

Figure: 6

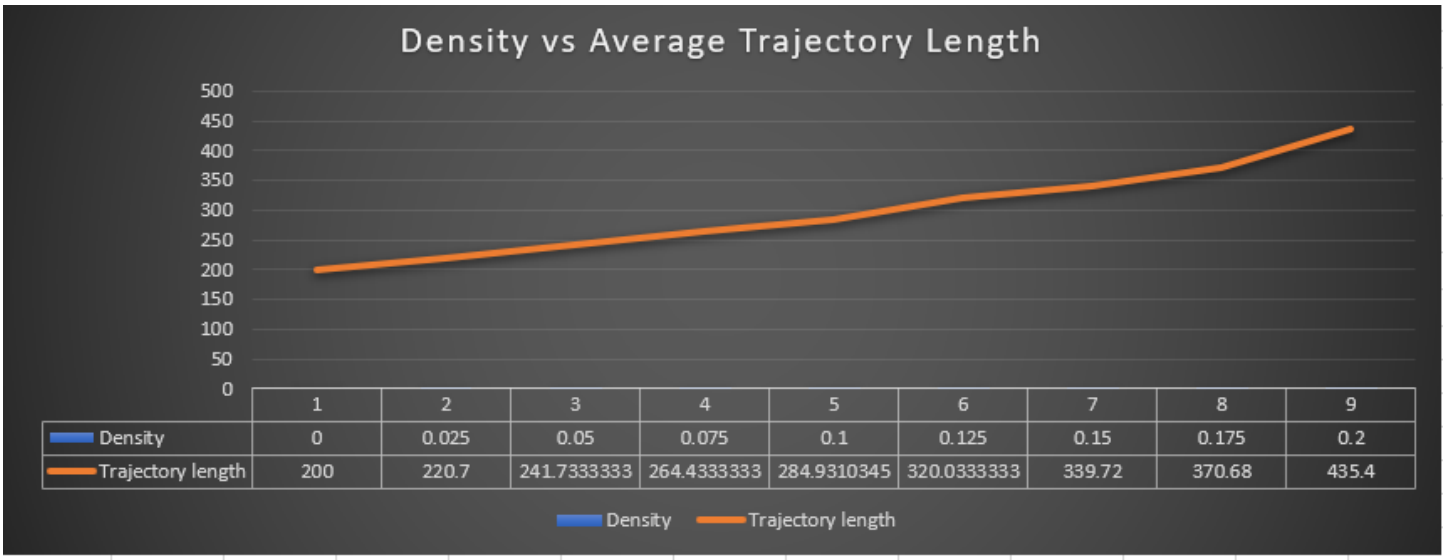
Q6. Performance Taking dim = 101, for a range of density p values from 0 to min(p0; 0:33), and the heuristic chosen as best in Q5, repeatedly generate gridworlds and solve them using Repeated Forward A*. Use as the field of view each immediately adjacent cell in the compass directions. Generate plots 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.

Answer: The chosen threshold p_0 is 0.2. The chosen heuristic is the Manhattan distance.

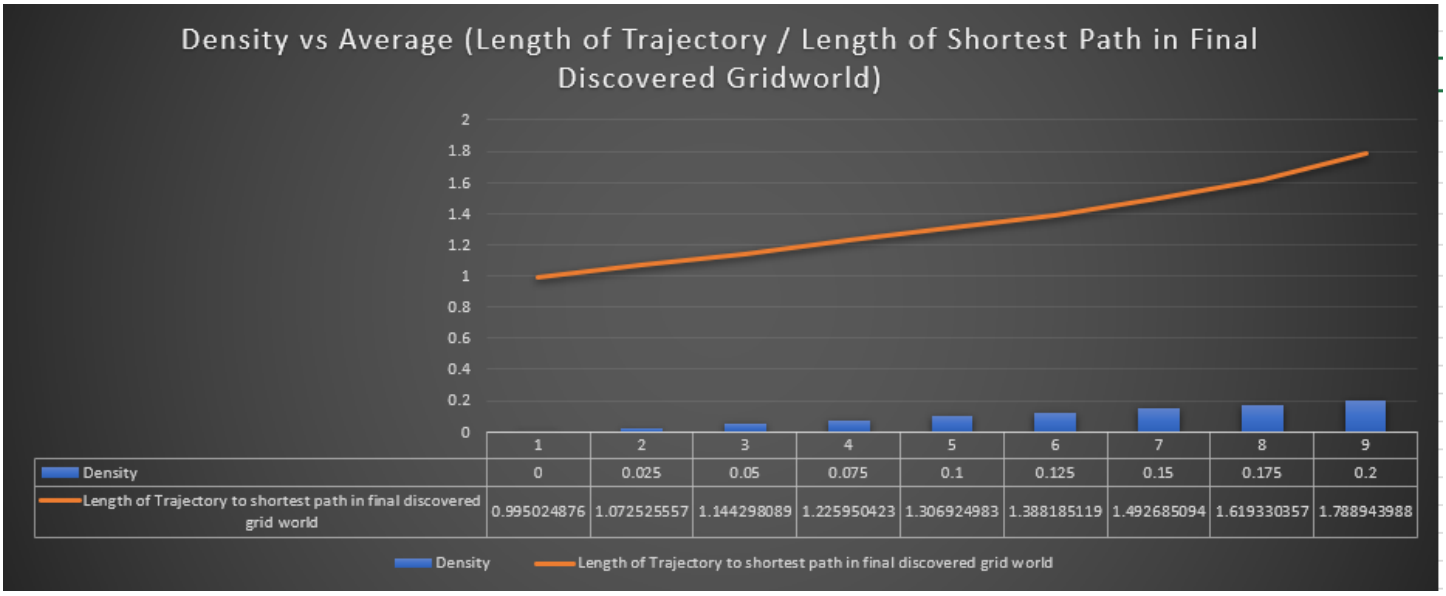
Density vs Average Trajectory Length



	1	2	3	4	5	6	7	8	9
Density	0	0.025	0.05	0.075	0.1	0.125	0.15	0.175	0.2
Trajectory length	200	220.7	241.7333333	264.4333333	284.9310345	320.0333333	339.72	370.68	435.4

We observe that as density increases, the trajectory length increases. The result is as expected because when the maze is dense, *Repeated Forward A** will have to re-plan and explore more alternate paths to find an optimal path as compared to the maze which is less dense.

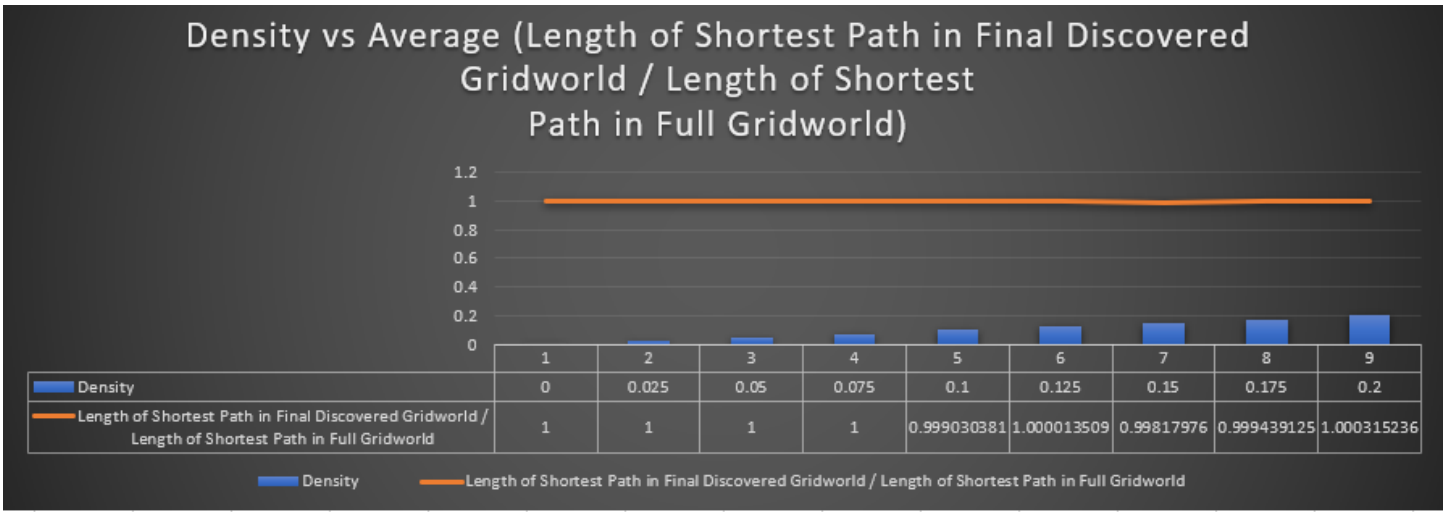
Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Gridworld)



	1	2	3	4	5	6	7	8	9
Density	0	0.025	0.05	0.075	0.1	0.125	0.15	0.175	0.2
Length of Trajectory to shortest path in final discovered grid world	0.995024876	1.072525557	1.144298089	1.225950423	1.306924983	1.388185119	1.492685094	1.619330357	1.788943988

We observe that as density increases, the average value also increases. This result is as expected. If the maze is dense, then trajectory length and value of shortest path length in the final discovered gridworld also increases due to replanning and exploring. So the average increases as density increases.

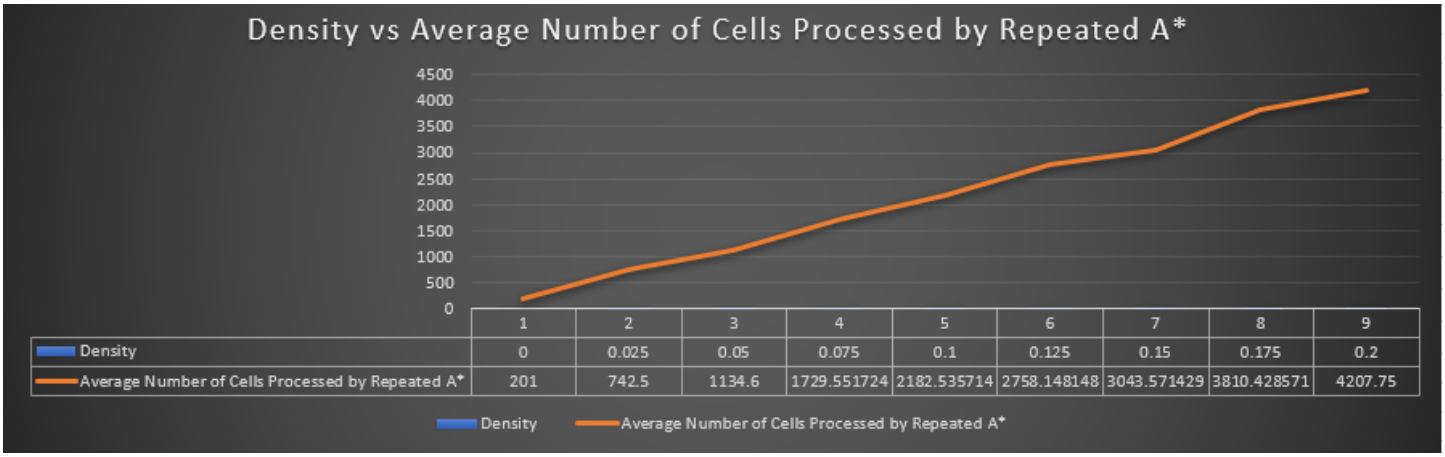
Density vs Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld)



	1	2	3	4	5	6	7	8	9
Density	0	0.025	0.05	0.075	0.1	0.125	0.15	0.175	0.2
Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld	1	1	1	1	0.999030381	1.000013509	0.99817976	0.999439125	1.000315236

We observe that as density increases, the average remains constant. The result is as expected. The average will always be 1 even as the maze gets denser because the Length of Shortest Path in Final Discovered Gridworld and Length of Shortest Path in Full Gridworld will be equal.

Density vs Average Number of Cells Processed by Repeated A*



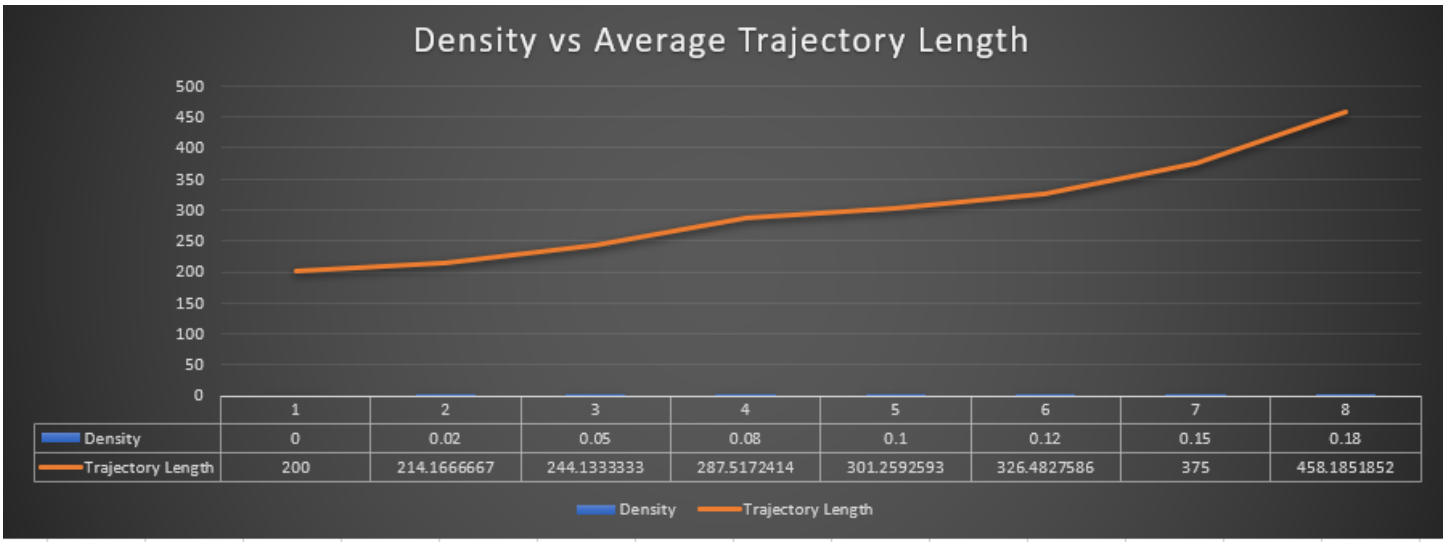
	1	2	3	4	5	6	7	8	9
Density	0	0.025	0.05	0.075	0.1	0.125	0.15	0.175	0.2
Average Number of Cells Processed by Repeated A*	201	742.5	1134.6	1729.551724	2182.535714	2758.148148	3043.571429	3810.428571	4207.75

We observe that as density increases, the number of cells processed increases. The result is as expected because when maze is dense, *Repeated Forward A** will have to re-plan and explore more paths to find an optimal path as compared to maze which is less dense. So, the search involves processing more cells for a denser maze.

Q7. 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?

Answer:

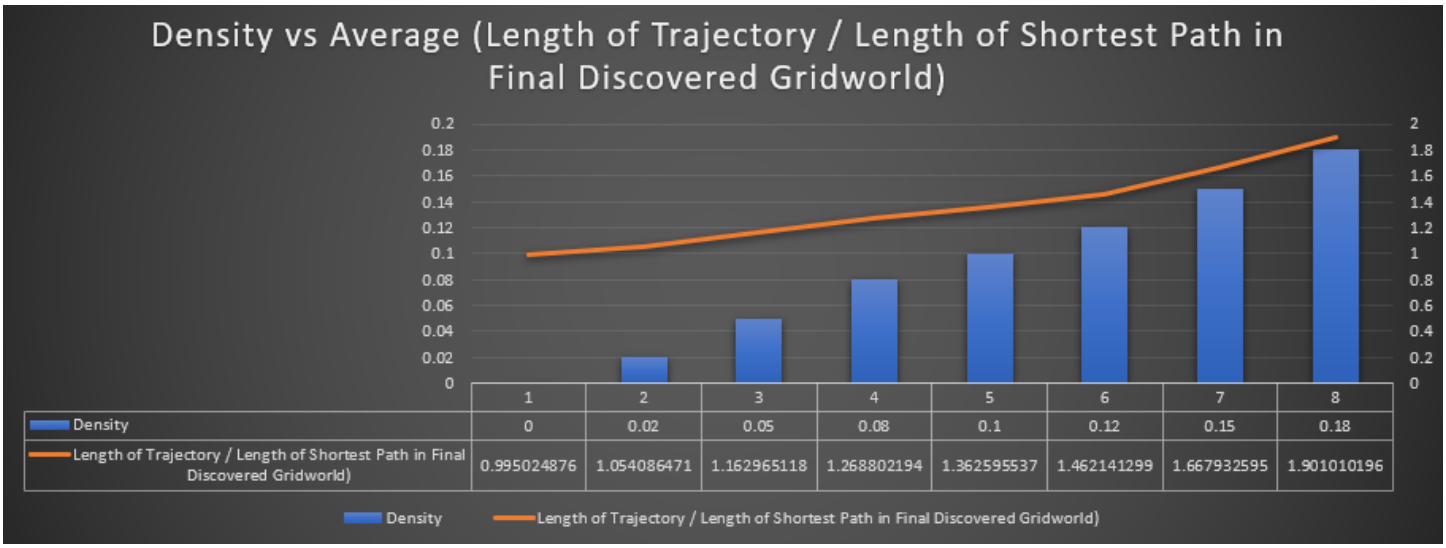
Density vs Average Trajectory Length



	1	2	3	4	5	6	7	8
Density	0	0.02	0.05	0.08	0.1	0.12	0.15	0.18
Trajectory Length	200	214.1666667	244.1333333	287.5172414	301.2592593	326.4827586	375	458.1851852

We observe that as the Density increases, the average trajectory on the final discovered gridworld also increases. This observation is as expected because the more the number of blocked nodes, the more the replanning and alternate paths and hence increase in trajectory length. There is also a 23.7% increase in the avg trajectory length for higher density as compared to Q6 because of the restricted field of view in this case. (comparing for p =0.18).

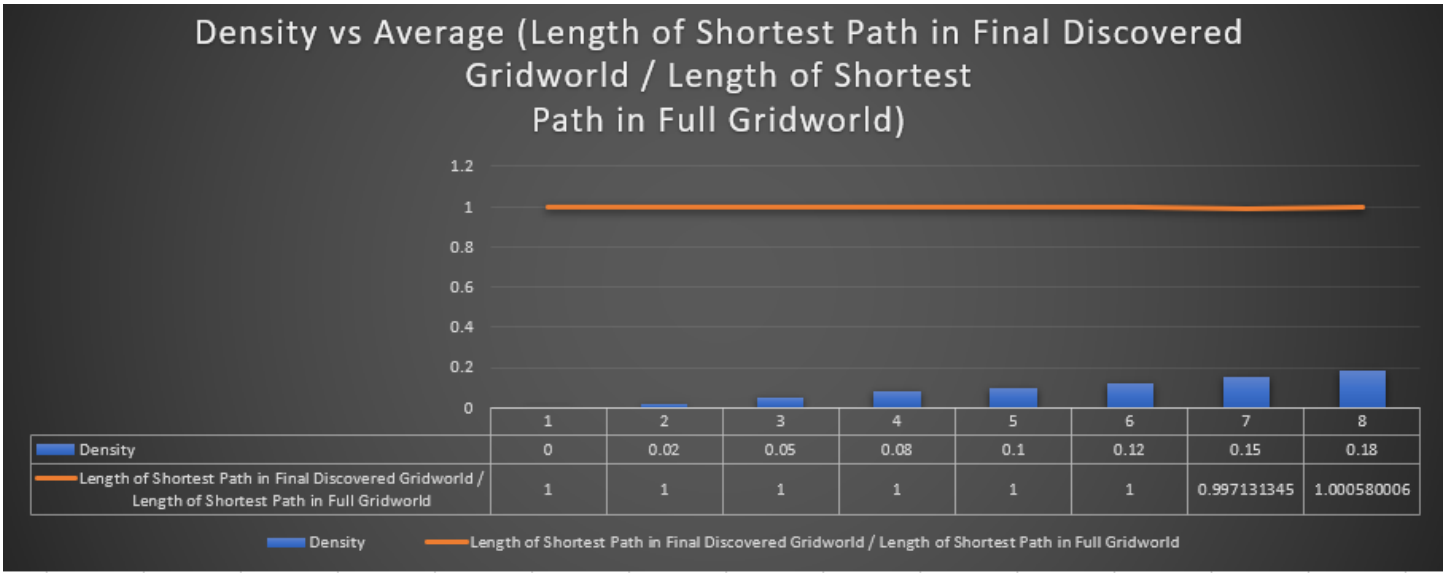
Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Gridworld)



	1	2	3	4	5	6	7	8
Density	0	0.02	0.05	0.08	0.1	0.12	0.15	0.18
Length of Trajectory / Length of Shortest Path in Final Discovered Gridworld)	0.995024876	1.054086471	1.162965118	1.268802194	1.362595537	1.462141299	1.667932595	1.901010196

We observe from the figure that as the Density increases, the average value increases. Because the numerator is increasing at a greater pace than the denominator (the shortest path increases with increasing density), this value is steadily increasing.

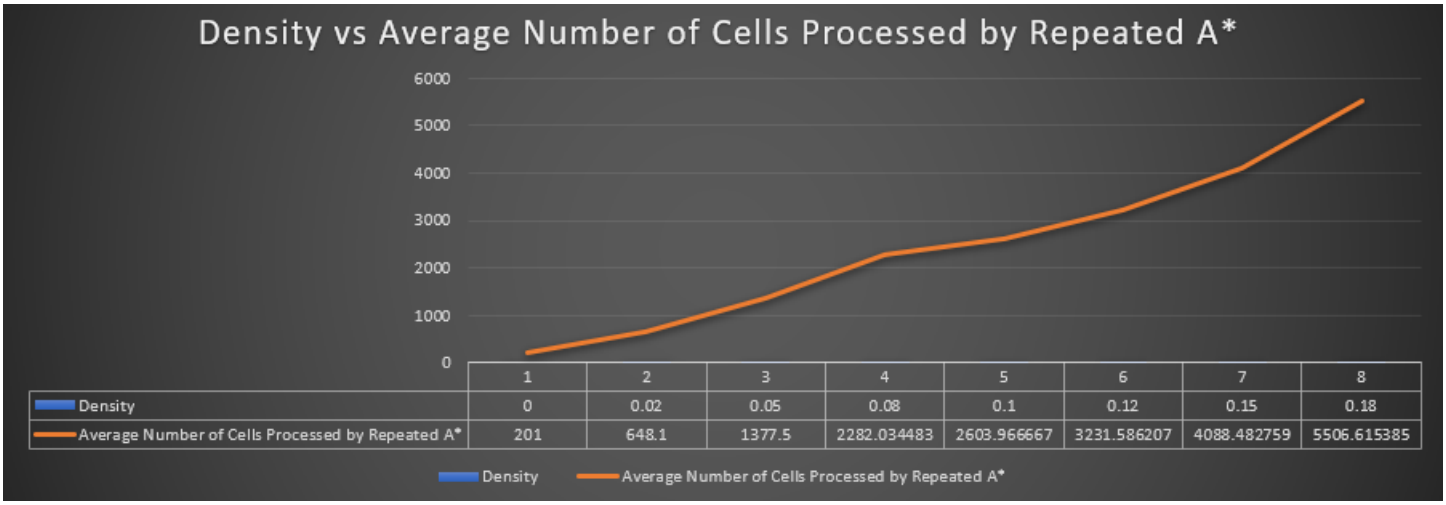
Density vs Average (Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld)



	1	2	3	4	5	6	7	8
Density	0	0.02	0.05	0.08	0.1	0.12	0.15	0.18
Length of Shortest Path in Final Discovered Gridworld / Length of Shortest Path in Full Gridworld	1	1	1	1	1	1	0.997131345	1.000580006

We observe from the figure that as the Density increases, there is no change in the ratio because the shortest path in the final discovered gridworld is one of the optimal paths in the full gridworld. So the lengths are always the same and hence the ratio is 1.

Density vs Average Number of Cells Processed by Repeated A*



	1	2	3	4	5	6	7	8
Density	0	0.02	0.05	0.08	0.1	0.12	0.15	0.18
Average Number of Cells Processed by Repeated A*	201	648.1	1377.5	2282.034483	2603.966667	3231.586207	4088.482759	5506.615385

We observe from the figure that the average number of cells processed increases with the density. This is expected because the increased density implies more blocks and that causes *Repeated Forward A** to call *A** more times. So there is an increased number of replanning phases as density increases.

Q9. 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.

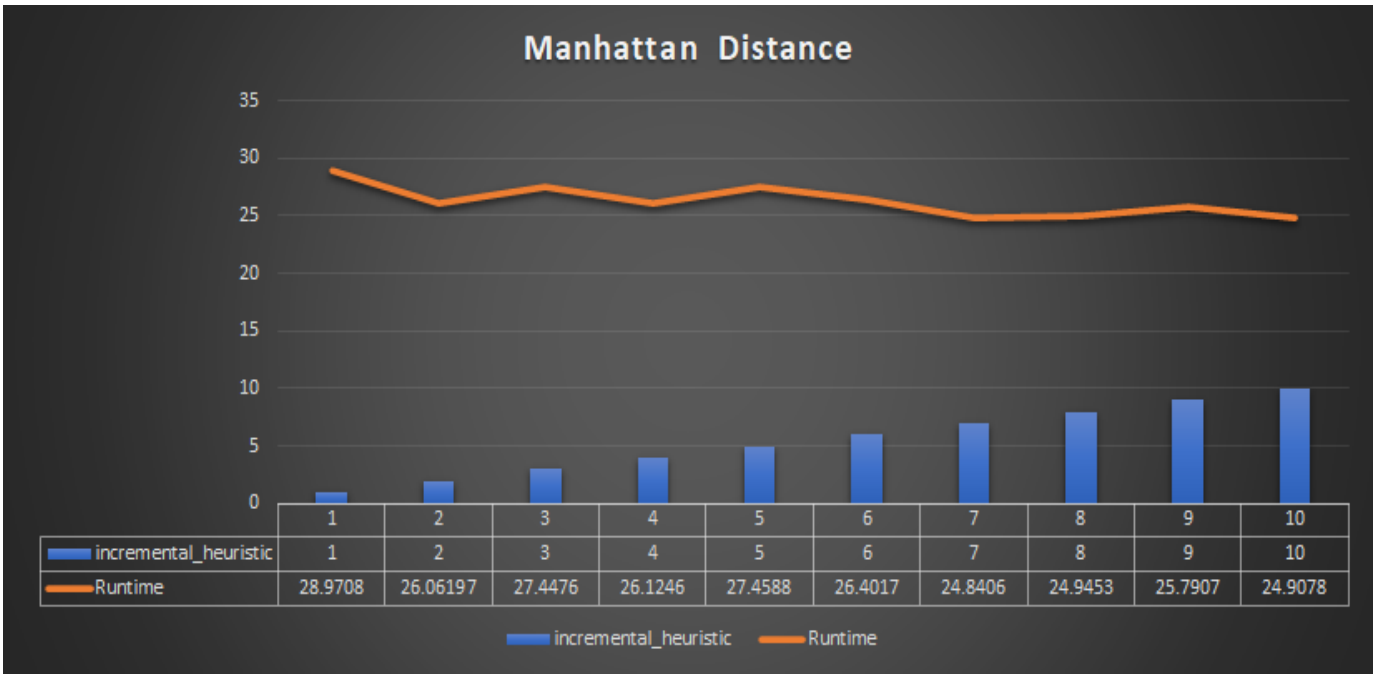
Answer:

Yes, the incremental heuristics/inadmissible heuristics and their combinations can be used in the *A** algorithm. An admissible heuristic is one that approximates a very close distance to the goal node.

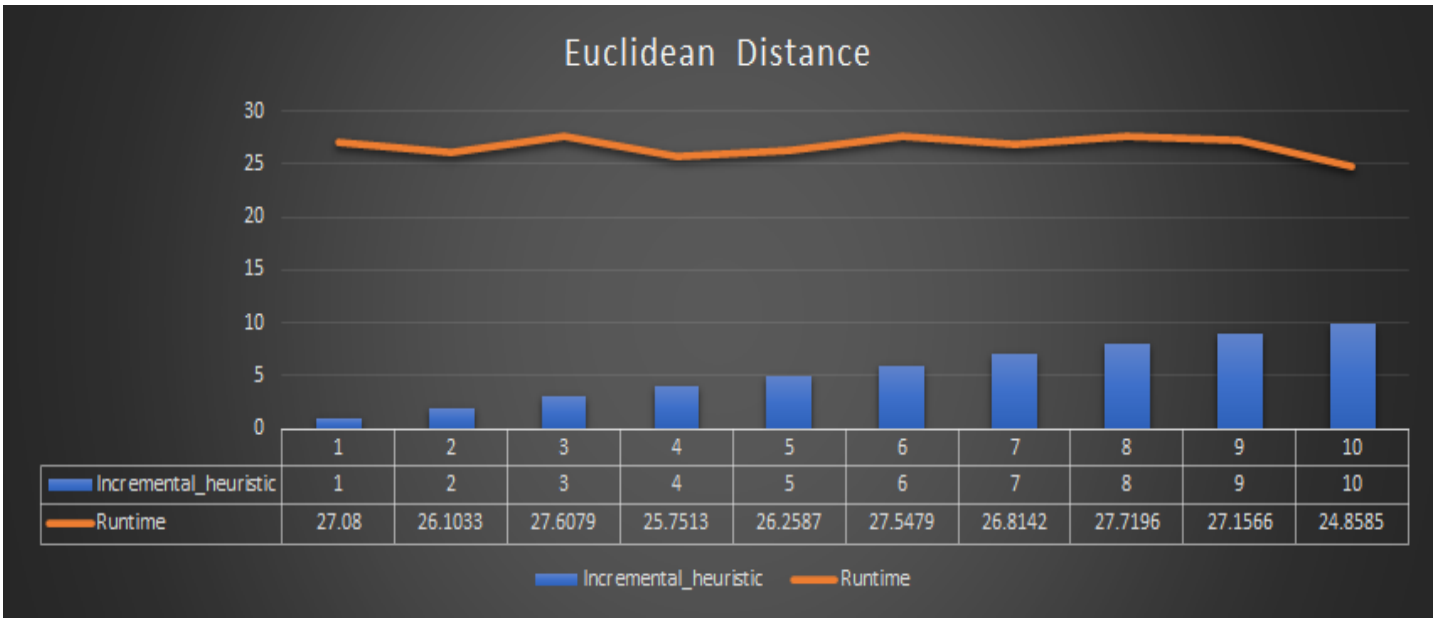
We know that, f(n) = g(n) + h(n) and we explore the nodes with least f(n) on every iteration of the algorithm. However, if we overestimate the h(n), the weights are more centered towards the distance from node to the goal than the distance from the start to the node itself (*g(n)*). Since node selection is based on the *f* cost, an overestimate for the *h* cost diminishes the weight of the *g* cost and the algorithm will explore nodes that have lower *h* values over nodes that have lower *g* values.

Observations on code execution

When *A** search algorithm was executed multiple times it is observed that when we use Manhattan distance as the heuristic and increase the variable by a scalar multiple, we notice a reduction in the runtime of the algorithm. In this subset of observations, we have the graph as shown above. We noticed that the algorithm increased its efficiency in reducing execution time by 14%. (Figure below)

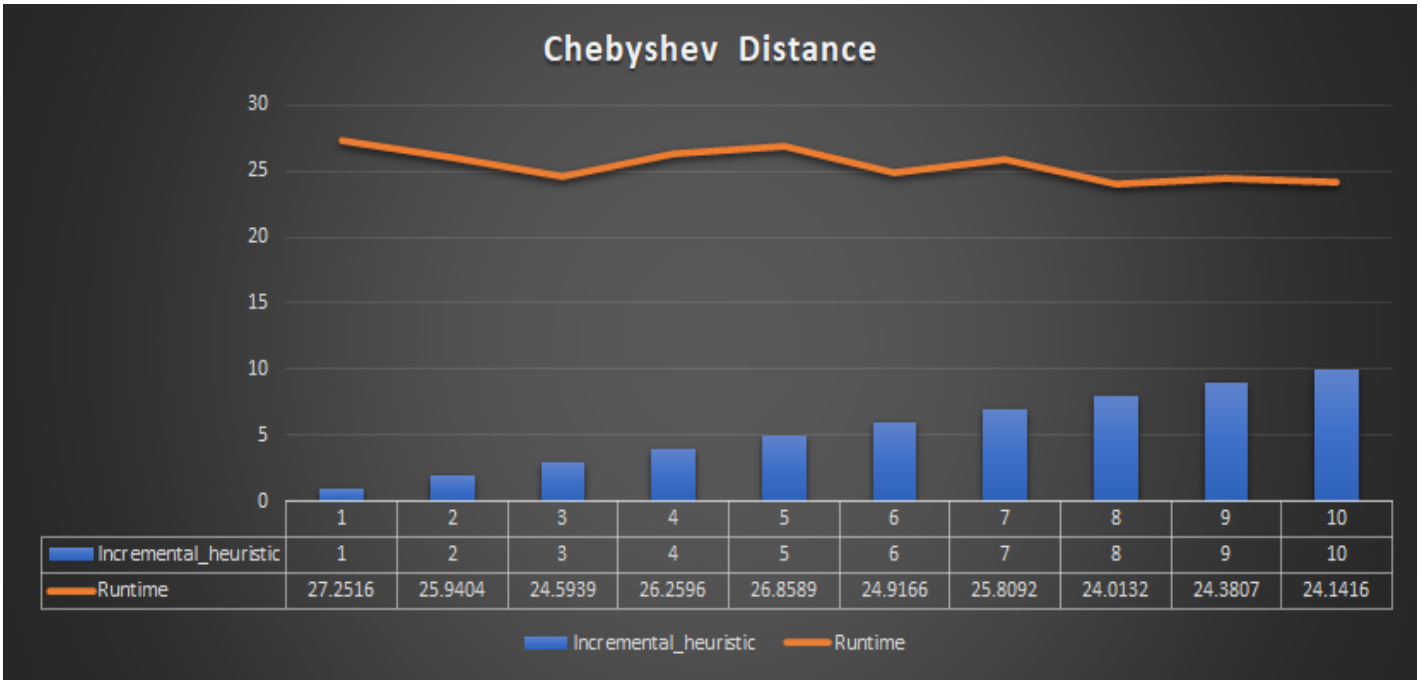


When *A** search algorithm was executed multiple times, it is observed that when we use Euclidean distance as the heuristic and increase the variable by a scalar multiple, we notice a reduction in the runtime of the algorithm. In this subset of observations, we have the graph as shown above. We noticed that the algorithm increased its efficiency in reducing execution time by 8%.(Figure below)

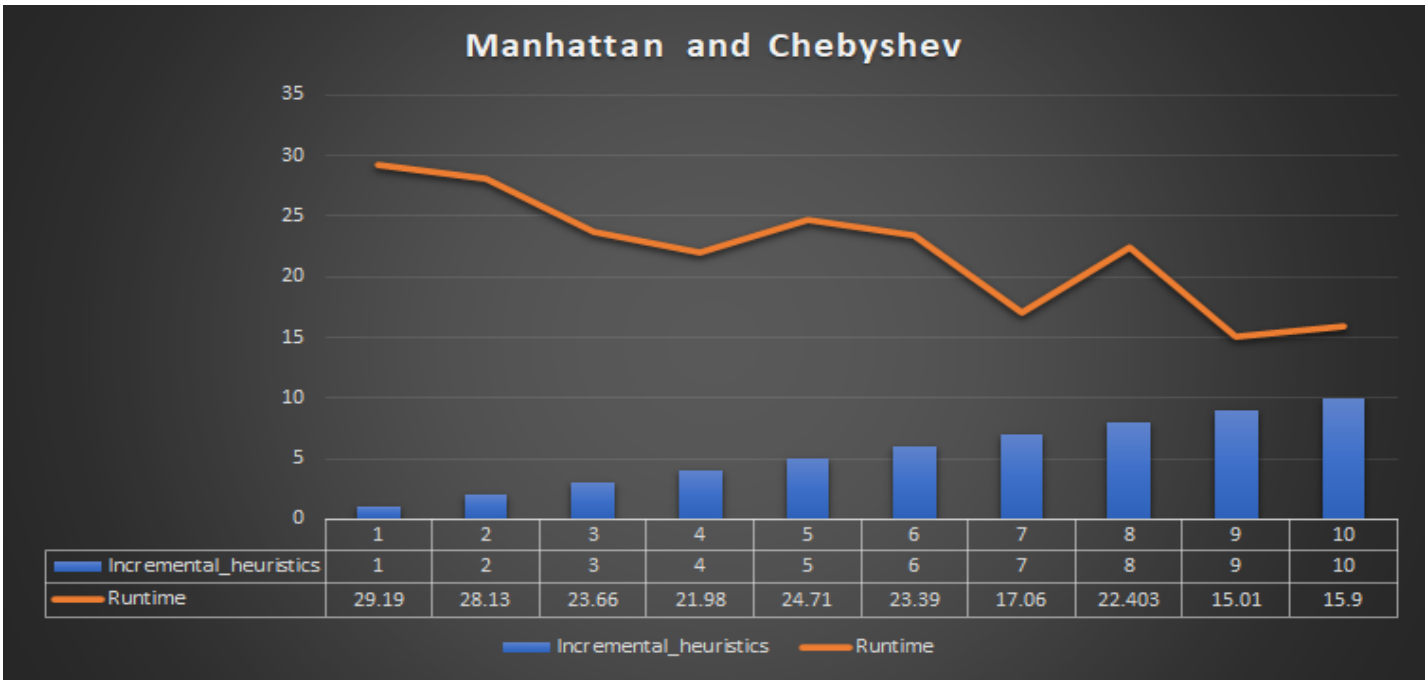


	1	2	3	4	5	6	7	8	9	10
Incremental_heuristic	1	2	3	4	5	6	7	8	9	10
Runtime	27.08	26.1033	27.6079	25.7513	26.2587	27.5479	26.8142	27.7196	27.1566	24.8585

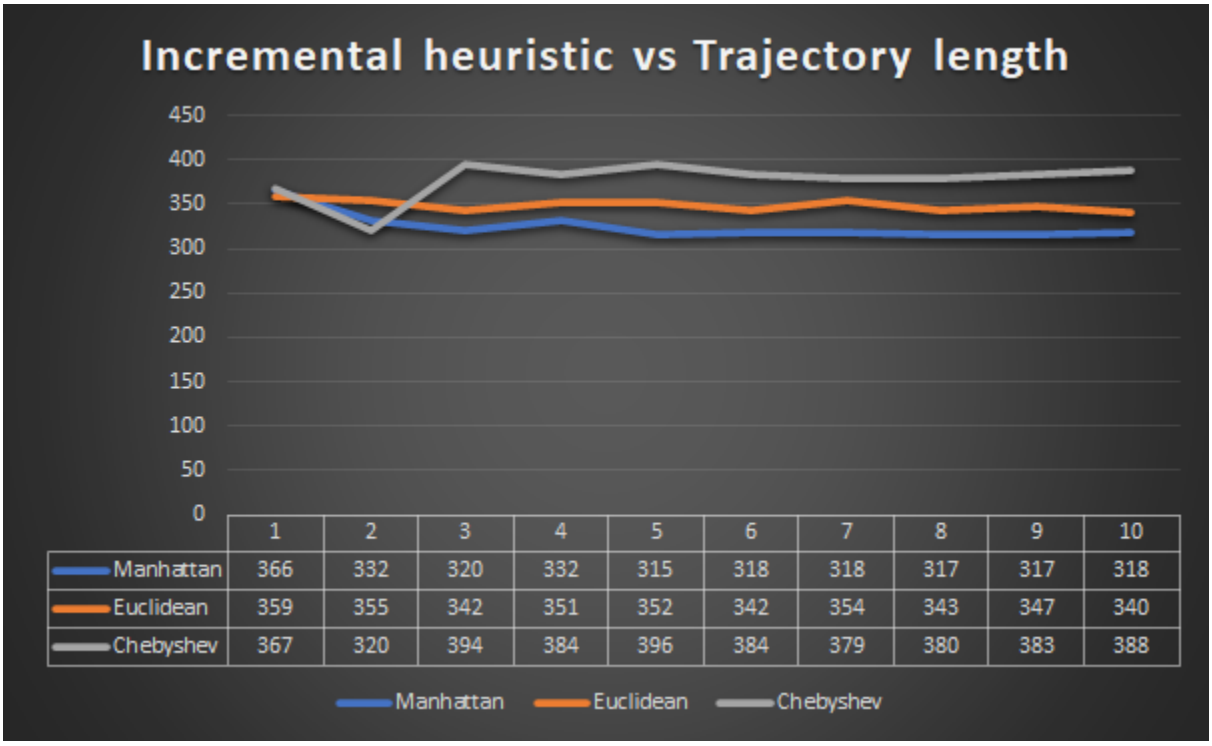
When *A** search algorithm was executed multiple times it is observed that when we use Chebyshev distance as a heuristic and increase the variable by a scalar multiple, we notice a reduction in the runtime of the algorithm. In this subset of observations, we have the graph as shown above. We noticed that the algorithm increased its efficiency in reducing execution time by 11%. (Figure below)



When *A** search algorithm was executed multiple times it is observed that when we use a combination of Chebyshev distance and Manhattan distance as an Incremental heuristic, we notice a reduction in the runtime of the algorithm. In this subset of observations, we have the graph as shown above. We noticed that the algorithm increased its efficiency in reducing execution time by 46%. (Figure below)



It was observed that if the A* takes X seconds of time to complete its execution, a combination of incremental heuristics reduced the runtime by approximately 46%, and the resulting time taken was 0.6X.



	1	2	3	4	5	6	7	8	9	10
Manhattan	366	332	320	332	315	318	318	317	317	318
Euclidean	359	355	342	351	352	342	354	343	347	340
Chebyshev	367	320	394	384	396	384	379	380	383	388

Executing the code with incremental heuristic we observe that trajectory length varies with a small value on each increment of the heuristic. We also notice that using Manhattan distance as a heuristic the trajectory length is least compared to the other two heuristics. In this case, Manhattan distance significantly decreased with an increase in heuristic increments. However, as the heuristic increases, f(n) is more focused towards h(n) which results in finding the goal node faster. Therefore, as the heuristic increases, the trajectory length decreases.

Individual Contribution to the project:

- Except for the heap library this program was coded completely by the team. The team split up the work equally amongst each other.

- Each individual solved a couple of questions along with individual input to the code build. Answering the question involved analysis of the code output, data collection, plotting graph and adding the answer to the report.

Harish: Worked on A-star algorithm, Worked on Q1 and Q9.
Kavya: Worked on enhancing to repeated A-star, Worked on Q7, Q3, Q4
Parvathi: Worked on enhancing to repeated A-star, worked on Q5, Q6, Q2

Link to the code: https://github.com/harish-udhayakumar/A-star

Appendix

A*

```
def astar(start, agent_matrix):
    """
    This function is used to discover the path from start to the goal
    agent_matrix: The matrix which store information about discovered maze
    """
    visited_list = []
    open_list = []
    trajectory_path = []
    trajectory_plus_gofn = []
    path_block = 0
    children_dict = {}
    i = 0
    goal = (n-1, n-1)
    num_of_cells_processed = 0

    open_list.append(start)
    gofn_matrix = compute_gofn(n)
    hofn_matrix = compute_hofn(n, "manhattan")

    while open_list:
        child_list = []
        current_node = open_list.pop(0)
        num_of_cells_processed += 1

        # adding current node to visited list of nodes if not added before
        if current_node[:2] not in visited_list:
            visited_list.append(current_node[:2])

        # adding current node to list of final path of nodes if not added before
            if current_node[:2] not in trajectory_path:
                trajectory_path.append(current_node[:2])
                trajectory_plus_gofn.append(current_node)

            # Condition to have reached the goal node
            if agent_matrix.item(current_node[0], current_node[1]) == 0:
                if current_node[0] == n-1 and current_node[1] == n-1:
#                     print("Reached Goal!!!!!!!!!!!!!!!!!!!!!!")
                    trajectory_plus_gofn.append(current_node)
                    buildParentDict(children_dict)
                    shortestPath = getShortestPath(start)
```

```
                    return
(trajectory_path,trajectory_plus_gofn,num_of_cells_processed,shortestPath)

                # Compute the list of children nodes for the current node
                children = get_children(agent_matrix, current_node[0],
current_node[1], n, current_node[2], visited_list)

                # Remove the third parameter (g(n)) from each of the child tuple
                for x in children:
                    child_list.append(x[:2])
                children_dict[current_node[:2]] = child_list

                # Get the next node of least weight/cost
                for node in children:
                    if node not in open_list:
                        open_list.append(node)
                open_list = priority_queue(open_list, hofn_matrix)

                # Remove the node if it is blocked to retract from the previous
unblocked node
                node_removal_list = node_block_check(children_dict, agent_matrix,
current_node[:2])
                for node in node_removal_list:
                    if node in trajectory_path:
                        trajectory_path.remove(node)
                        trajectory_plus_gofn = [i for i in trajectory_plus_gofn
if i[:2] != node]

            else:
                if open_list == []:
                    print("There is no path available to goal Node")
                    return ([],[], 0,[])
                else:
                    if current_node[:2] in trajectory_path:
                        trajectory_path.remove(current_node[:2])
                        trajectory_plus_gofn.remove(current_node)
        else:
            if open_list == []:
                    print("There is no path available to goal Node")
                    return ([],[], 0,[])
```

Repeated A*

```
def repeated_astar():
"""
    This function is used to call a_star repeatedly to get the shortest path from
start to the goal node
    """

      global agent_matrix

      goal_reached = False
      goal = (n-1,n-1)
      path = []
      final_path = [(0,0,0)]
      trajectory_length = 0
      num_of_cells =0

      while not goal_reached:
      #planning phase
      (path1,path2,astar_num_cells,shortest_path) =
astar((final_path[-1][0],final_path[-1][1],0),agent_matrix)
      num_of_cells += astar_num_cells
      if(path1 == []):
            return (0,0,[])
      # Execution Phase
      for node in path2:
            if block_unblock_matrix[node[:2]] == 0:
#               print("node: ",node[:2]," is unblocked")
                  if node not in final_path:
                  final_path.append(node)
                  trajectory_length += 1
                  if node[:2] == goal:
                  goal_reached = True
                  break
            else:
                  agent_matrix[node[:2]] = 1
                  trajectory_length += 2
                  Break
      # Running A* on final discovered gridworld for the shortest path
      trajectory_path ,_,_,shortest_path_final_discovered_gridworld=
astar((0,0,0),agent_matrix)
      return (trajectory_length, num_of_cells,
shortest_path_final_discovered_gridworld)
```