Appendix

Voyage Into The Known is a maze problem that an agent needs to solve by reaching the maze's exit or the goal node. We will try to help the agent find its path from source to goal using Repeated A* algorithm.

Initially there will be a random maze and the agent only knows about its entry point and what the exit point or the goal looks like. In this case, source will always be the top left cell of the maze or (0,0) and goal will always be the bottom right cell of the maze or (dim_x – 1, dim_y – 1).
1 -> clear nodes; -1 -> blocked nodes

```
maze = np.random.choice([1, -1], p=[1-p(d), p(d)], size=(dim_x, dim_y))
```

After generating the maze, we would need to make sure that source and goal nodes are not blocked.

```
if maze[0,0] != 1:
        maze[0,0] = 1
if maze[dim_x-1,dim_y-1] != 1:
        maze[dim_x-1,dim_y-1] = 1
```

We will set the initial knowledge of the agent to source (0,0) and plan a path for execution with the help of f(n).
F(n) – Total cost from source to goal via node n
g(n) – Cost of node n from source
h(n) – heuristic value for node n to reach goal node.

$$F(n) = g(n) + h(n)$$

```
fringe = PriorityQueue()

#function to calculate manhattan distance which is our h(n)
def manhattan_distance(ag_X, ag_Y):
   dist = abs(ag_X - (dim_x-1)) + abs(ag_Y - (dim_y-1))
   return dist

heuristic = manhattan_distance(source_x, source_y)

#initailly source will have f(n) = h(n), as its g(n) = 0
#fringe will take 3 values: f(n), x, y
fringe.put((heuristic, 0, 0))

#initialize maze knowledge or the closed set for agent from source, no parent for source
closed_knowledge = {'0,0':{ 'g': 0, 'h': heuristic, 'p': (None, None) } }
#discovered grid will have knowledge of source, no parent for source
discovered_grid = {'0,0':{ 'g': 0, 'h': heuristic, 'p': (None, None)}}
```

Now agent will proceed towards one planned path depending on the calculation of f(n) which is based on heuristics.

```
def maze_search(closed_knowledge, discovered_grid, fringe):
        while not fringe.empty():
        # take first node on the fringe and start generating its valid child to follow a path
                current = fringe.get()
                current_x = current[1]
                current_y = current[2]
```

```
            if agent has reached goal node (dim_x-1,dim_y-1)
            return a backtracked final path with get_discovered_path(current_x, current_y)

            # if blocked cell ecountered for first time, set last unblocked cell as new source
            # keep updating discovered grid for any new information
            # empty fringe, reset closed set for new source, update discovered grid
            # Repeat maze_search for new source

            else :
              #generate and add valid children of current node to fringe, process surrounding blocks
              valid_child(child_x , child_y, current)
              blocked_child(child_x, child_y, current)

        # if all the nodes are exhausted from fringe, no path is possible from source to node
        return "No Path!!"
```

Now, let's look at the functions which do the heavy processing for agent.
valid_child(child_x , child_y, current) and *blocked_child(child_x, child_y, current)* perform the calculations to add the generated child on node and update the closed_knowledge as well as discovered_grid.

```
def valid_child(child_x , child_y, current):
  heuristic = manhattan_distance(child_x, child_y)
  child_cost = closed_knowledge[_format(current[1], current[2])]['g'] + 1 #maze[child_x, child_y]
  dist = child_cost + heuristic

  #if not present in closed set, generate child perform appropriate calculations and put on fringe
  if _format(child_x, child_y) not in closed_knowledge:
    closed_knowledge[_format(child_x, child_y)] = {'g': child_cost, 'h': heuristic, 'p': (current[1], current[2])}
    fringe.put((dist, child_x, child_y))

  #if encountered cost of child is less than what we have in closed set update the child information and
#put on fringe
  elif _format(child_x, child_y) in closed_knowledge:
    if closed_knowledge[_format(child_x, child_y)]['g'] > child_cost:
      closed_knowledge.update({_format(child_x, child_y):{ 'g': child_cost, 'h': heuristic, 'p': (current[1],
current[2]) }})
      fringe.put((dist, child_x, child_y))

  #discovered grid, keep updating as cells are encountered
  if _format(child_x, child_y) not in discovered_grid:
    if discovered_grid[_format(current[1], current[2])]['g'] < closed_knowledge[_format(current[1],
current[2])]['g']:
      discovered_grid.update({_format(child_x, child_y):{ 'g': discovered_grid[_format(current[1],
current[2])]['g'] + 1, 'h': manhattan_distance(child_x, child_y), 'p': (current[1], current[2]) }})
    else:
      discovered_grid[_format(child_x, child_y)] = {
        'g': closed_knowledge[_format(current[1], current[2])]['g'] + 1,
        'h': manhattan_distance(child_x, child_y),
        'p': (current[1], current[2])
      }
    elif _format(child_x, child_y) in discovered_grid:
     if discovered_grid[_format(child_x, child_y)]['g'] > child_cost:
```

```
        discovered_grid.update({_format(child_x, child_y):{ 'g': child_cost, 'h': heuristic, 'p': (current[1], current[2]) }})
```

```
#define blocked cells, include in knowledge set and fringe on first occurence
def blocked_child(child_x, child_y, current):
   heuristic = manhattan_distance(child_x, child_y)
   child_cost = closed_knowledge[_format(current[1], current[2])]['g'] + 1
   dist = child_cost + heuristic
   if _format(child_x, child_y) not in discovered_grid:
      discovered_grid[_format(child_x, child_y)] = { 'g': sys.maxsize,'h': heuristic,'p': (current[1], current[2])}
      fringe.put((dist, child_x, child_y))
```

Finally, when all the processings are completed and agent has reached the goal node, we can backtrack in our discovered_grid to get the shortest path in final discovered grid.

```
def get_discovered_path(goal_X, goal_Y):
   discovered_path = [(goal_X, goal_Y)]
   while discovered_grid[_format(discovered_path[-1][0], discovered_path[-1][1])]['p'] != (None, None):
      discovered_path.append(discovered_grid[_format(discovered_path[-1][0],discovered_path[-1][1])]['p'])
   discovered_path.reverse()
   return discovered_path
```

Thus, this repeated A* algorithm helps agent to find a local minima in this unknown gridworld.

1

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

For any cell in a maze its surrounding environment can have only 4 cells in each compass direction when agent moves. Initially there is no knowledge of the grid and when agent moves it assumes that there is no blocked cell. A path to goal is found in planning phase and its knowledge is updated in execution phase. When knowledge is being updated about the surrounding cells , agent can move in any of the direction. It chooses the best cost path based on f(n) (where f(n) = g(n) + h(n)). When agent keeps on following one path , based on cost and priority in queue it may lead to goal with the most optimal path.

We can't replan when environment is updated , as we don't have knowledge of full grid world at that point. Replanning at this point can only make the algorithm worse and can divert agent from most optimal path. Suppose replanning occurs at the time when environment is updated, agent might get stuck badly in the new path discovered as we might not know what is ahead as some of the new nodes which are yet to be discovered could block the way. This increases the cost of the algorithm and agent may get stuck in the solvable maze.

When the agent bump onto a block in current node, irrespective of the environment, agent cannot move in that direction . It might appear that whole path should have been planned before as agent can't move forward. However, at this moment A* is again applied and source is set as parent of the blocked path. Discovered Grid is updated for the blocked path.

Now in new iteration agent moves from new source to a planned path based on f(n). For every node that we traverse, information is updated on the discovered grid. With each iteration we will have more knowledge of the grid but still we will not have entire knowledge of grid world. So we will not replan unless we stumble upon a block.

Let's take a 3*3 matrix below and discuss the possibilities here:

	0	1	2
0	1	1	1
1	−1	1	−1
2	−1	1	1

This is a 3*3 matrix, where 1 represents the unblocked cell and -1 represents blocked cells. We will see step by step depiction of fringe and discovered grid.

Planning Phase 1:
Fringe - [4,(0,0)].
Fringe - [(4,(0,1)),(4,(1,0))]
Fringe - [(4,(0,2)),(4,(1,0))(4,(1,1))]
Fringe - [(4,(1,2)),(4,(1,0))(4,(1,1))]
Fringe -[(4,(2,2),(4,(1,0))(4,(1,1)),(5,(1,1)]

Path in planning phase - [(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)].

Execution phase 1:

1. Node (0,0) - Knowledge grid - {m[0][0]=1,m[0][1] = 1,m[1][0] = -1}. At this point we have 2 children which we update in knowledge grid along with parent. We don't have full knowledge of grid world. There are several other nodes that hasn't been discovered and if we replan here , it could give us wrong path.

2. Node(0,1) - Knowledge grid - {m[0][0]=1,m[0][1] = 1,m[1][0] = -1, m[0][2] = 1, m[1,1] = 1}

3. Node (0,2) - Knowledge grid - {m[0][0]=1,m[0][1] = 1,m[1][0] = -1, m[0][2] = 1, m[1,1] = 1}

4. Next path (1,2) is blocked cell. We discover now that this is blocked and previously unknown in knowledge grid .We will add the details in knowledge grid and replan everything. This is the point where we replan and set initial point as (0,2)

We will repeat the same process in this iteration and we will eventually get the path to goal node (2,2) as final path [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2)].

We have proved from above example that its best that we replan only when we see a blocked cell.

2

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

Agent can never get stuck in a solvable maze. Whenever there is a path from source to destination, agent will reach there. Agent will make sure it takes the shortest path to the goal. The reason agent cannot get stuck in the solvable maze is because of the backtracking and fringe as priority queue.

Since in a fringe a path with the lowest total cost (cost from source to node + heuristic) will have highest priority, it will be traced first before any other path. Even if heuristic couldn't predict blocked path in middle, fringe will make sure next path with next highest priority is chosen to trace upon. This will always ensure that right path is taken to reach the goal in an efficient way. If in the worst case there is only one path to the goal and initially it looks to have worst heuristic and total cost, backtracking and priority queue will make sure that all other paths with blockers are not followed.

Whenever we start solving a maze we are first planning a feasible shortest path and then updating the surroundings of that path in execution phase in our closed set and discovered grid. While we are updating the path in execution phase , if we find a blocker , we will perform repeated A* and take initial point as parent of the blocked node. We will however keep the cost of this new initial point same as its original cost in the previous iteration. We will maintain a new fringe for new A*. This repeated A* will make sure that eventually we get the optimal solvable path to the final destination. We can use discovered grid to find the path length and shortest path by backtracking from goal to initial point.

It may appear initially that a certain path may have less cost than the shortest path and agent might stuck on that path, because that would not lead to the goal and the node leading to the correct path is behind in priority queue. This is simply not true due to several reasons.

First of all we are using heuristic and initial cost to put a node in priority queue. This itself puts the node of such complexity behind in the priority queue. Even if such path has less cost initially , continuous traversing will ensure that overall cost will get higher later pushing it behind the original path in the queue. All these reasoning proves that agent will never get stuck in a solvable maze.

We will further support this by looking at an example below.

	0	1	2	3
0	1	1	1	−1
1	1	−1	1	1
2	−1	1	1	1
3	−1	1	−1	1

To solve the above maze using the code we have assumed below:

- In this example we are taking 4*4 matrix where indexing is starting from 0 to length-1
- If a node or maze cell has a value of 1 , that means its solvable and unblocked with a cost of 1.
- If a node or maze has value of -1, its a blocked cell and no further path or children is from that maze. We are putting a cost of maximum integer to that maze.
- We are taking fringe as priority queue of cells with each element of fringe having priority and the coordinates . Its represented as (cost,(x, y)) where x, y are coordinates of maze cell and cost determines the priority.
- Total cost is sum of initial cost and heuristic. Heuristic is calculated using Manhattan distance.

H(x,y) = |Xgoal - x|+|Ygoal - y|, where h is heuristic and x,y are coordinates

- We are assuming that when 2 priorities are same we choose one in horizontal direction.

Let's start tracing the best path by calculating fringe and closed set at each step.

1. Planning phase - We don't have knowledge of the environment initially and we are adding it in fringe based on heuristic and initial cost. We don't have knowledge of blockers at the start.
    1. Starting point - (0,0). Fringe -[6,(0,0)]
    2. Popping (0,0) and adding its children, Fringe - [(6,(0,1)),(6,(1,0))]
    3. Popping (0,1) and adding its children, Fringe - [(6,(0,2)),(6,(1,1)),(6,(1,0))]
    4. Popping (0,2) and adding its children, Fringe - [(6,(0,3)),(6,(1,2)),(6,(1,1)),(6,(1,0))]
    5. Finally after few more steps on same side we will reach goal destination with Fringe - [(6,(3,3)),(6,(1,2)),(6,(1,1)),(7,(2,1)),(7,(2,2)),(7,(2,3))]
    6. Path found to goal is (0,0) (0,1) (0,2) (0,3) (1,3) (2,3) (3,3)

Execution phase - We will check each node of the path found with full grid world if it is blocked or unblocked and update the environment in knowledge grid.We will also update its children Based on it we might apply repeated A* algorithm.

1. (0,0) - its an unblocked path , We will update its children. (0,1)- unblocked path. (1,0) - unblocked path.
2. (0,1) - unblocked path,We will update its children. (1,1) - blocked path, (0,2) - unblocked path.
3. (0,2) - unblocked path, children - (0,3) - blocked, (1,2) - unblocked.
4. (0,3) - blocked.This is where the path found in planning phase is wrong as there is a blocked node. We need to run A* again with (0,2) as initial point. It seems that agent might get stuck at this point as we have found a blocker.However repeated A* makes sure that it doesn't happen and we will eventually find path.

2. Planning phase - This is the 2nd planning phase

1. Starting point - (0,2). Fringe -[6,(0,2)]
2. Destination will be reached eventually in this iteration having following path[(0,2),(1,2),(2,2),(2,3),(3,3)]

Execution phase - Path is correctly found and checked.

We reach final destination with cost 6 at (3,3). Despite some blockers in middle we are able to reach the final destination. This will be true for worst cases as well and it will always find the path. Hence agent will never get stuck ink solvable maze.

Final Path: [(0, 0), (0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 3)]

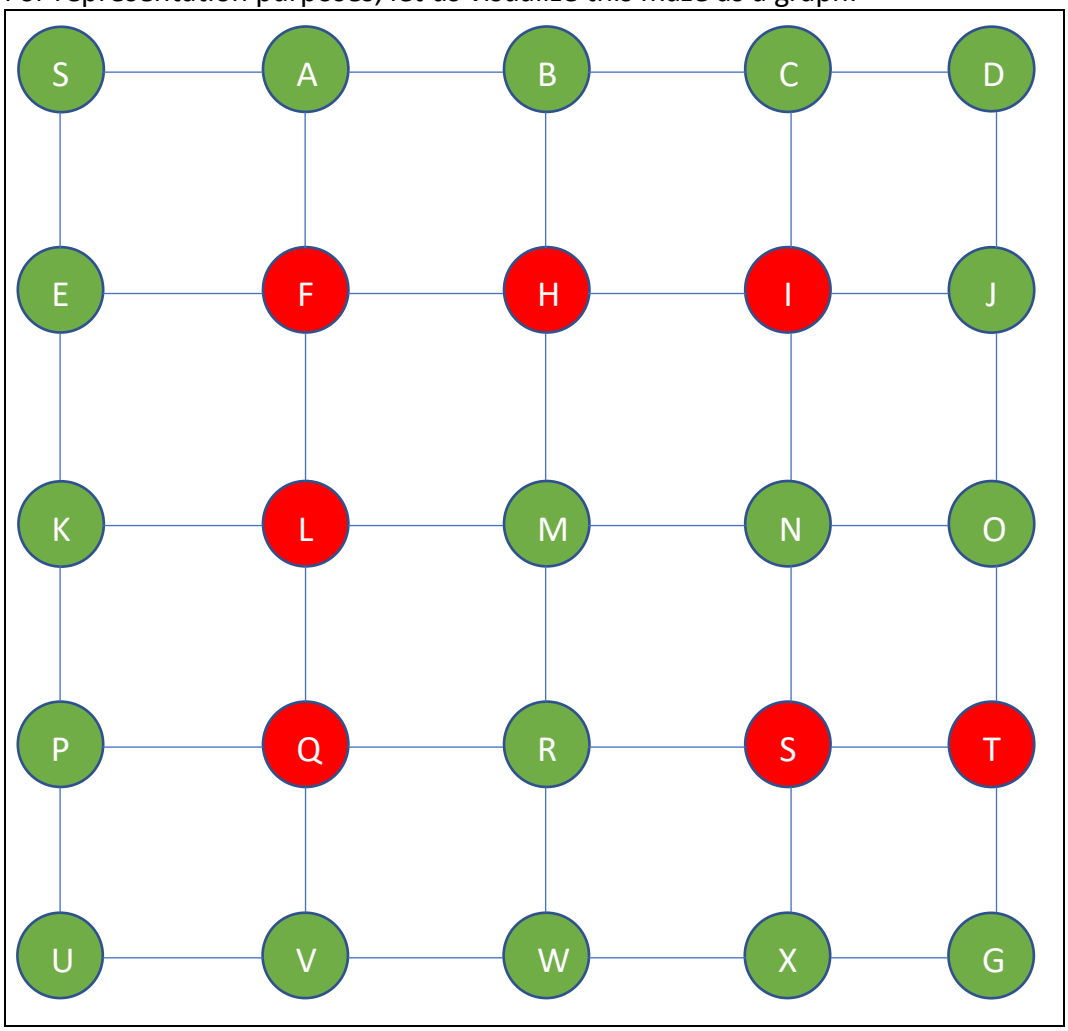
3

Let us consider a gridworld of 5 x 5 square matrix.
(Note: All the gridworlds that are taken as example will have the first cell (0,0) as their start node and last cell (x, y) as their goal node, x – number of rows; y – number of columns)

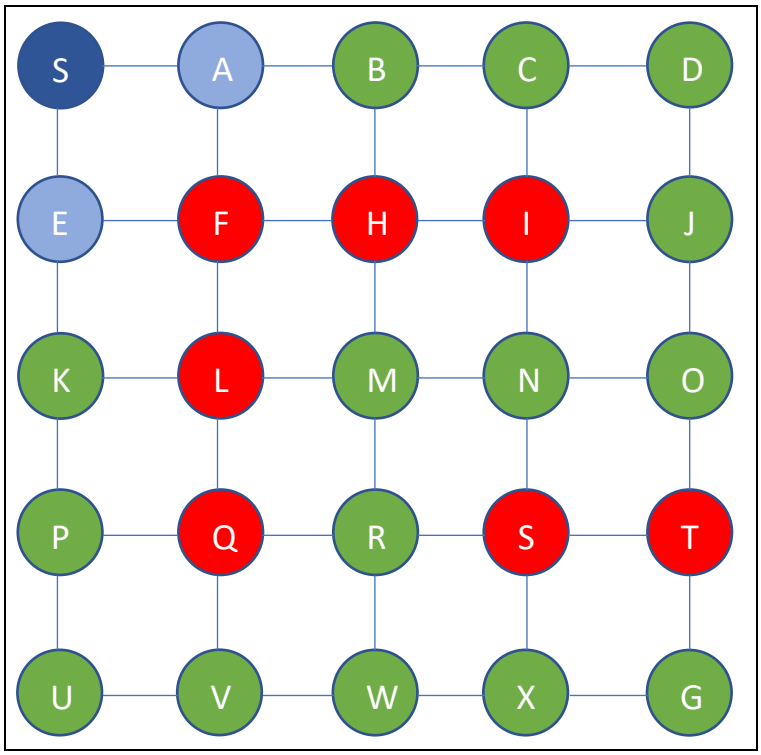
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)

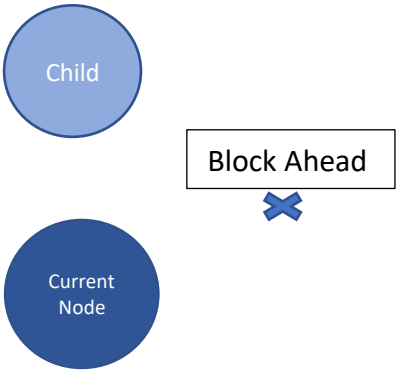
Cost of parent node, i.e., (0,0) will always be 0.
Cost of blocked cells ■ will be taken as -1 and traversal through blocked cell is not permitted.
All other cells ■ (unblocked) will have a cost of 1.

For representation purposes, let us visualize this maze as a graph:

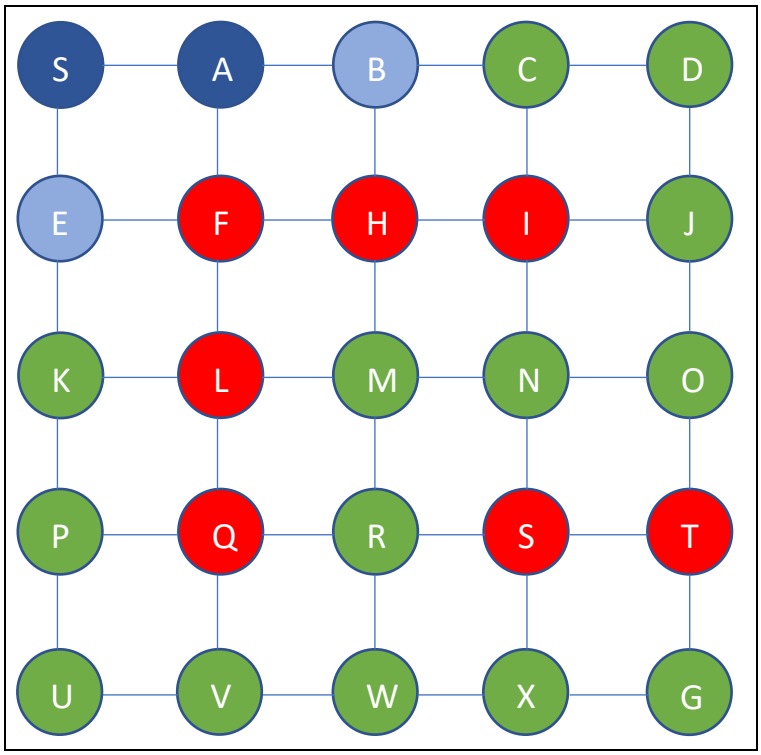


Let us try to navigate through the maze for a planned path from S -> G.

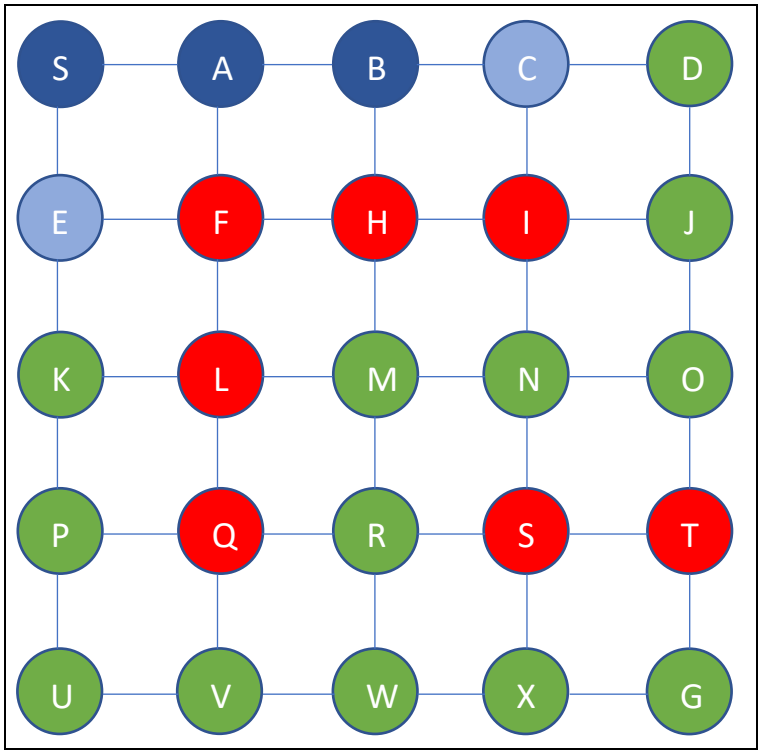




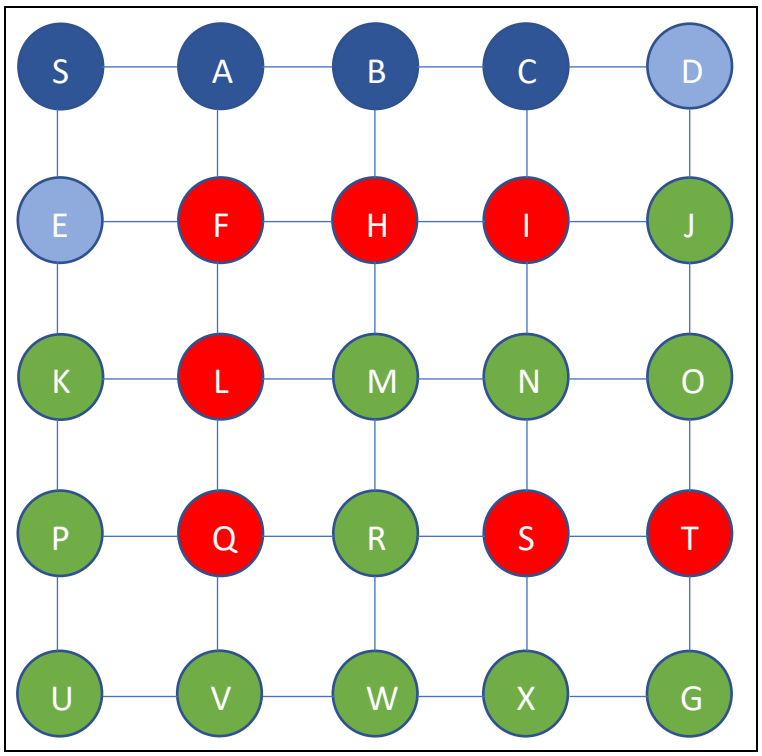
Let's say that agent chooses to follow path through S -> A -> ….. -> G and start traversing it based on f(n) -> heuristics + cost of child from source, agent will continue to traverse this path until it encounters a block.



Agent continues to travel along path S -> A -> B -> ….. -> G

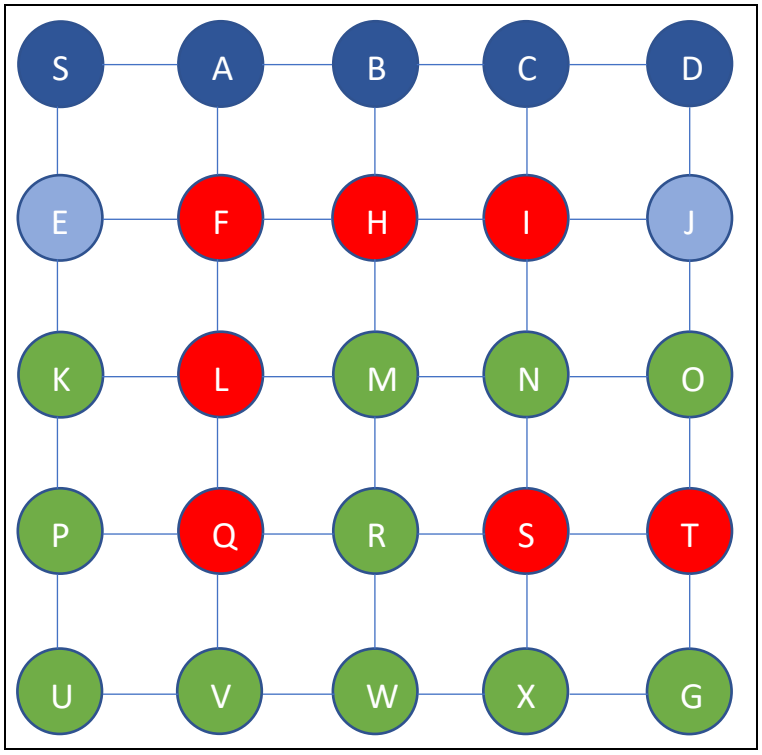


S -> A -> B -> C -> …. -> G



S -> A -> B -> C -> D -> …… -> G

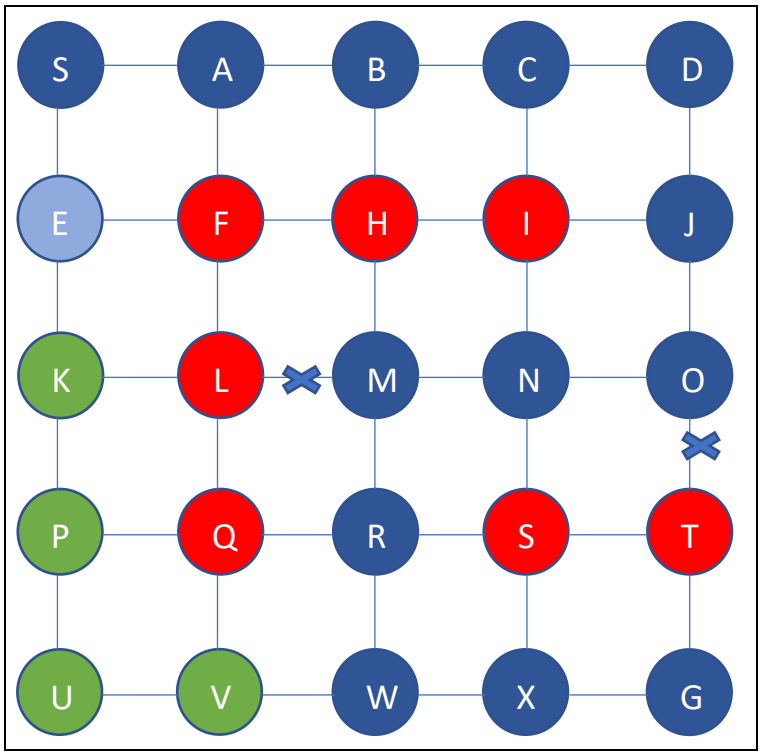
As the agent traverses through its planned path, it continues to store information about its surrounding, so that it can use the same in the future.



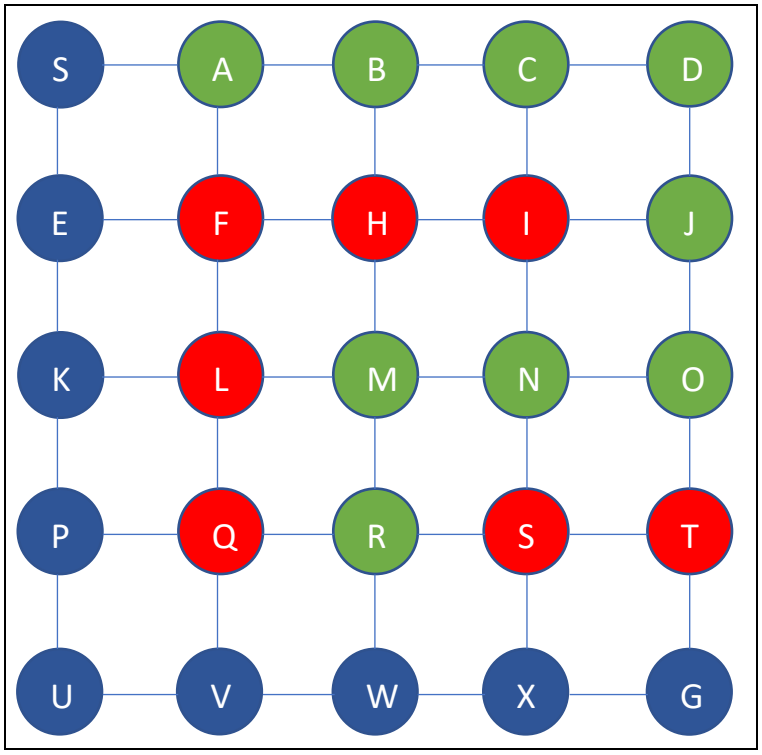
Agent will continue to follow this path and encounter a block at node T. At this point repeated A* comes into the picture and agent again starts at node O, taking Node O as source. It continues to traverse through O -> N -> M, again hits a block and restart at Node M, continues same path and reaches goal Node.

Now, since it has not traversed the path along S -> E, because of resetting the child information on the fringe every time it comes across a block. It is not aware that there is a path available through that node as well and it finalizes the path:

S->A->B->C->D->J->O->N->M->R->W->X->G



In this example, we can clearly see that although agent has found a path from source to goal node, it is not the optimal path, which would have been:
S -> E -> K -> P -> U -> V -> W -> X -> G



Let us try to understand all the actions that were taken:

Closed Set keeps track of all the nodes that we can observe in the line of path including blocked nodes. In this case Closed Set is a Dictionary.
The closed set will hold information like g: cost of node from source, h: heuristic value from node to the goal, p: parent of node and refreshes with change of source.

Closed Set - {node: {g, h, p}}

Fringe is basically a data structure that we will use to navigate through the maze and try to find a shortest path. In this case fringe is a Priority Queue.
Priority Queue will have the total cost (f) from source to goal via that node and the node itself.
[Note: f = g + h]

Fringe – (f, node)

Discovered Grid is the knowledge of the original grid acquired by the agent at any given point. Discovered Grid keeps on updating until agent has reached the goal, at that point it backtracks its path to source.

Discovered Grid - {node: {g, h, p}}

Stage	Closed Set	Fringe
At Source	{S: {g: 0, h: 8, p: None}}	(8, S)
Initial	{S: {g: 0, h: 6, p: None}}, {A: {g: 1, h: 7, p: S}}, {E: {g: 1, h: 7, p: S}}	(8, A), (8, E)
Transition 1	{S: {g: 0, h: 6, p: None}}, {A: {g: 1, h: 7, p: S}}, {E: {g: 1, h: 7, p: S}}, {B: {g: 2, h: 6, p: A}}, {F: {g: sys.maxsize, h: 6, p: A}},	(8, B), (8, F), (8, E)
Transition 2	{S: {g: 0, h: 6, p: None}}, {A: {g: 1, h: 7, p: S}}, {E: {g: 1, h: 7, p: S}}, {B: {g: 2, h: 6, p: A}}, {F: {g: sys.maxsize, h: 6, p: A}}, {H: {g: sys.maxsize, h: 5, p: B}}, {C: {g: 3, h: 5, p: B}}	(8, C), (8, H), (8, F), (8, E)
Transition 3	{S: {g: 0, h: 6, p: None}}, {A: {g: 1, h: 7, p: S}}, {E: {g: 1, h: 7, p: S}}, {B: {g: 2, h: 6, p: A}}, {F: {g: sys.maxsize, h: 6, p: A}}, {H: {g: sys.maxsize, h: 5, p: B}}, {C: {g: 3, h: 5, p: B}}, {D: {g: 4, h: 4, p: C}}, {I: {g: sys.maxsize, h: 5, p: C}}	(8, D), (8, E), (8, I), (8, H), (8, F)
Transition 4	{S: {g: 0, h: 6, p: None}}, {A: {g: 1, h: 7, p: S}}, {E: {g: 1, h: 7, p: S}}, {B: {g: 2, h: 6, p: A}}, {F: {g: sys.maxsize, h: 6, p: A}}, {H: {g: sys.maxsize, h: 5, p: B}}, {C: {g: 3, h: 5, p: B}}, {D: {g: 4, h: 4, p: C}}, {I: {g: sys.maxsize, h: 5, p: C}}, {J: {g: 5, h: 3, p: D}},	(8, J), (8, E), (8, I), (8, H), (8, F)
Transition 5	{S: {g: 0, h: 6, p: None}}, {A: {g: 1, h: 7, p: S}}, {E: {g: 1, h: 7, p: S}}, {B: {g: 2, h: 6, p: A}}, {F: {g: sys.maxsize, h: 6, p: A}}, {H: {g: sys.maxsize, h: 5, p: B}}, {C: {g: 3, h: 5, p: B}}, {D: {g: 4, h: 4, p: C}}, {I: {g: sys.maxsize, h: 5, p: C}}, {J: {g: 5, h: 3, p: D}}, {O: {g: 6, h: 2, p: J}}	(8, O), (8, E), (8, I), (8, H), (8, F)
Transition 6	{S: {g: 0, h: 6, p: None}}, {A: {g: 1, h: 7, p: S}}, {E: {g: 1, h: 7, p: S}}, {B: {g: 2, h: 6, p: A}}, {F: {g: sys.maxsize, h: 6, p: A}}, {H: {g: sys.maxsize, h: 5, p: B}}, {C: {g: 3, h: 5, p: B}}, {D: {g: 4, h: 4, p: C}}, {I: {g: sys.maxsize, h: 5, p: C}}, {J: {g: 5, h: 3, p: D}}, {O: {g: 6, h: 2, p: J}}, {T: {g: sys.maxsize, h: 1, p: O}}, {N: {g: 6, h: 2, p: O}}	(8, T), (8, N), (8, E), (8, I), (8, H), (8, F)
Transition 7 (Source Resets)	{O: {g: 6, h: 2, p: None}}	(8, O)
Transition n	**(Similarly, it continues till it reaches Goal Node is reached)**	Goal Node pops out of Fringe

At this point, agent has found the Goal node, and there will be no other nodes in the Priority Queue that has less cost or priority as compared to Goal node. Therefore, the Goal Node now pops out of the Priority Queue indicating that the path traversal is completed.

3) Thus, when agent re-calculates the path using closed set once it reaches the target, it excludes all the backtracking and repetitions, but it is not an optimal path in the complete gridworld.

Path Found by agent using repeated A*:
S->A->B->C->D->J->O->N->M->R->W->X->G

Optimal Path:
S -> E -> K -> P -> U -> V -> W -> X -> G

Thus, there will be few cases when path returned by agent is **not optimal**.

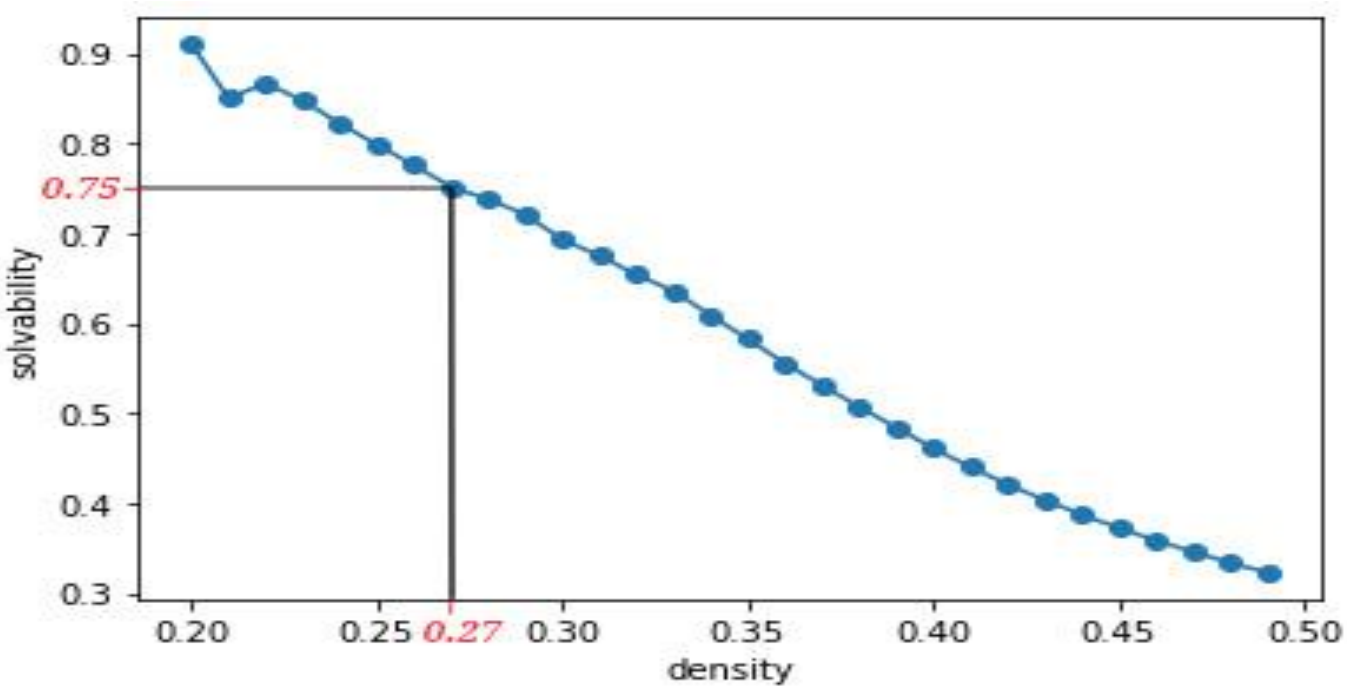
4. Solvability

A gridworld is solvable if it has a clear path from start to goal nodes. Given dim= 101, we will now observe how does solvability depend on p? For a range of p values, we estimated the probability that a maze will be solvable by generating multiple environments and checking them for solvability. We will now plot density vs solvability, and try to identify as accurately 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 we assume that the entire gridworld is known, and hence only needs to be searched once each)

As can be clearly seen from the graph, solvability does depend on p. On increasing the value of p we can see that solvability decreases. We ran the code to check for solvability on a maze of dimensions 101 x 101, with probability of a cell being a blocked one ranging from 0.2 to 0.5, taking 100 iterations for each density. As we can see the solvability starts from close to 1 at density 0.2 and it decreases to as less as 0.3 at density 0.5.

We have chosen 0.27 as our threshold p0 as we can see that for p < 0.27, more than 75% of the mazes are solvable. As we go for p higher than 0.27 then the percent of solvable mazes start decreasing almost linearly.



As it is given in the question that the entire gridworld is known and we only want to check for solvability and not actually find the shortest path, we feel that A* algorithm might not be our best choice for this.

A* is best when we don't actually know the whole maze and therefore we have to try different paths with backtracking, but if we know the maze then we can actually use a more greedy approach because we can accept any path in this case. We can try to move in the direction of the goal node and if we want to make a decision between multiple cells then we can try the one having most unblocked children.

5. Heuristics

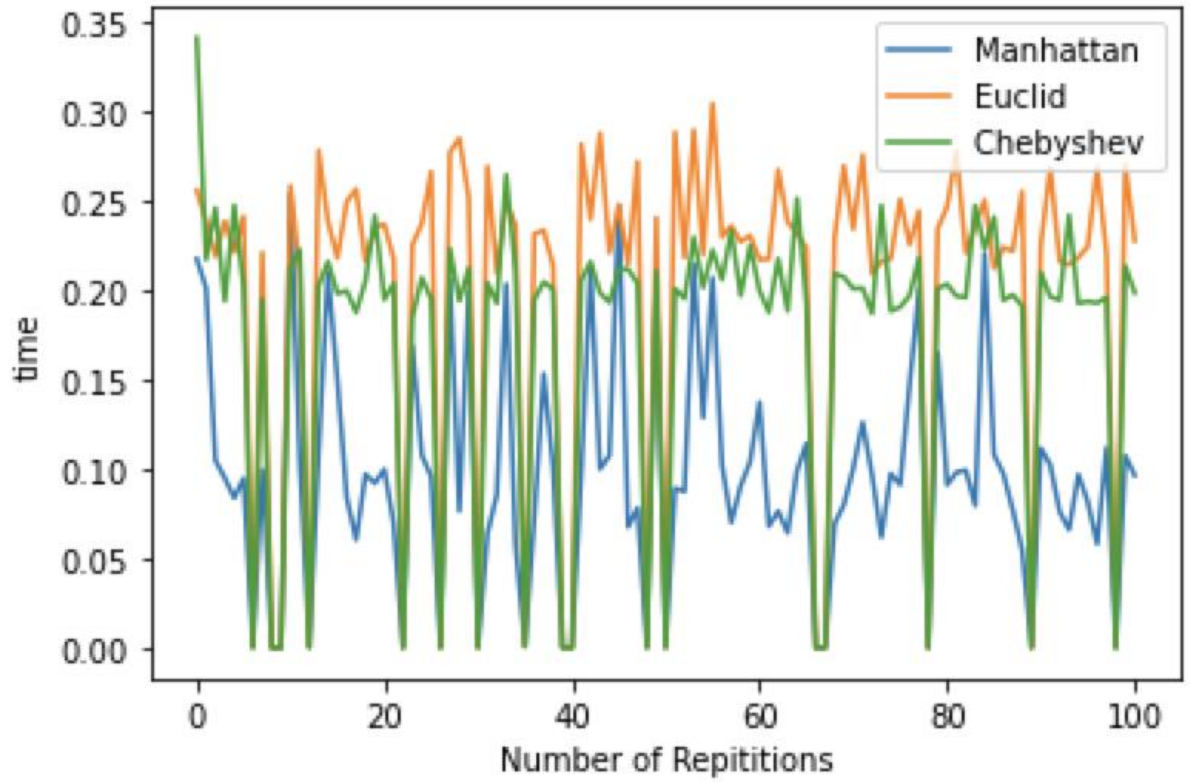
(a) Among environments that are solvable, is one heuristic uniformly better than the other for running A∗?

Let us Consider the following heuristics:

Euclidean Distanced - ((x1, y1),(x2, y2)) =√(x1−x2)2+ (y1−y2)
Manhattan Distanced - ((x1, y1),(x2, y2)) =|x1−x2|+|y1−y2|.
Chebyshev Distanced - ((x1, y1),(x2, y2)) = max(|x1−x2|,|y1−y2|).

How can they be compared?

We will plot the relevant data and justify your conclusions.
We assume to take each gridworld as known, and thus only search once.



- To solve the above problem we have plotted running time of different heuristics with number of repetitions .
- We took 101 dimensions square maze and performed 101 repetitions to get the plot.
- We have plotted comparison line graph, where Euclid is represented by Orange, Chebyshev by green and Manhattan by blue.

Result

- We can see that Manhattan has best run time , followed by Chebyshev and Euclid.
- Total Runtime for: Manhattan ~ 10.2 sec; Chebyshev ~ 17.3 sec; ~ 19.7 sec

We can conclude from this that Manhattan will run better than Chebyshev followed by Euclid. There may be some cases for maze search where Manhattan may be worse than others, but overall in a long run it is much better than other two.

Reason for Manhattan to be better than others is because it is more realistic to grid world . Its more closer to actual cost than other two heuristics. Manhattan is calculated as sum of difference X and Y coordinates to goal coordinates. This is actually equal to number of steps taken by agent from that point to reach goal , is there is no block in the middle. In a maze , agent can move either horizontal or in vertical directions and can't move diagonally. Even if there is a block in the path. After replanning ,next path to be followed will have the best heuristic as Manhattan is more realistic than any other .

Compared to Manhattan, Euclid will have worse run time. Euclidean heuristic is more about computing displacement between 2 paths. It will be lesser than Manhattan and lesser accurate. Euclid is useful when a path is diagonal and there is diagonal movement . But in Gridworld we are moving upwards , backward, left and right, so Manhattan is complementing it properly. There may be possibility that 2 node has same overall costs with Manhattan but it may come different for Euclid and which may not be right. Because of the more overall costs, that node might be chosen later in fringe due to high cost and low overall priority. Now if agent traverses through 1st path and finds a block but most optimal path is actually through 2nd node. In that case we can backtrack to 2nd path more faster in Manhattan than Euclid. When priorities are same, fringe chooses random node, so there is a probability that 2nd node is chosen before 1st node and make algorithm even more faster in Manhattan. This possibility is definitely not there in Euclid. So Manhattan is better than Euclid

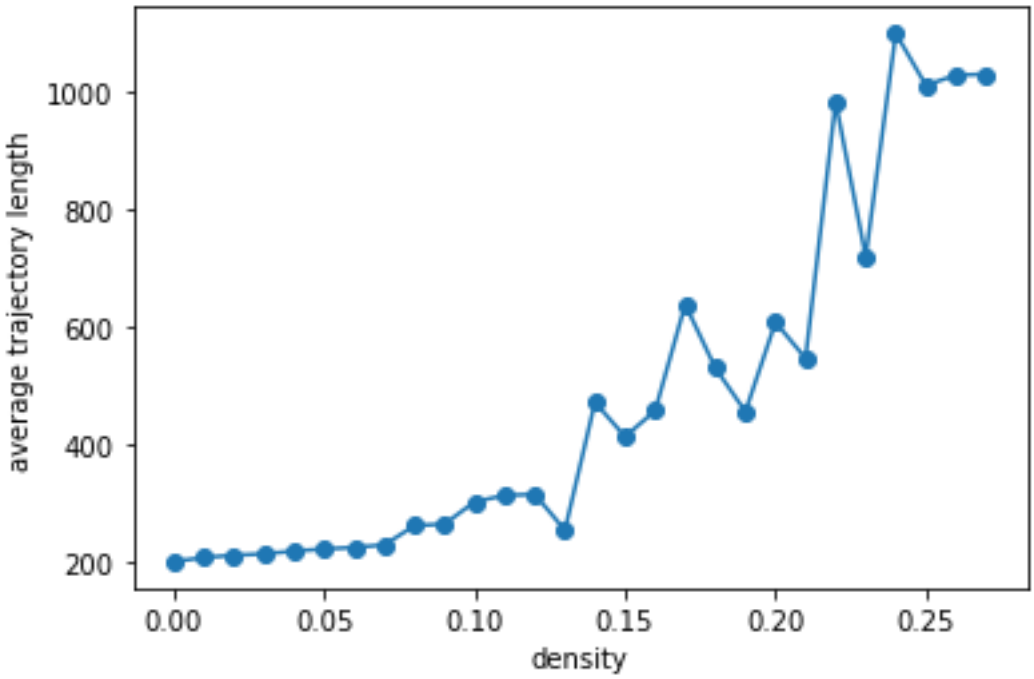
Whenever Heuristics and initial cost sum is closer to overall cost , heuristics is often considered better. For Chebyshev and Manhattan, following relation always hold true.
Hc + initial Cost <= Hm + initial Cost<= overall cost. Where 'Hc' is Chebyshev heuristic and 'Hm' is Manhattan heuristic.

It can be seen that Manhattan heuristics is more closer to overall cost and greater than equal to Chebyshev. Chebyshev is just the maximum of difference of X or Y coordinates of goal and current node. Consider 4 * 4 matrix and points (2,0) and (0,1). 1st point has overall cost as 6 with Manhattan and 5 with Chebyshev and 2nd point has 6 and 4 costs with Manhattan and Chebyshev heuristics. We are considering that these nodes are visited first time and first run is done on maze. In Chebyshev , overall costs keeps on fluctuating a lot due to its heuristics. For ex - agent moves from (1,0) to (2,0) and then (2,1).Heuristics are 3,3 and 2 again respectively for Chebyshev, while its 5,4 and 3 for Manhattan. Manhattan seems to be more logical here as there is a unit change when agent moves ahead, while in Chebyshev heuristics didn't have change from (1,0) to (2,0) which doesn't seems to be apt. Because of this when Chebyshev heuristic is chosen , agent might go on right path very late as compared to Manhattan . Thus Manhattan is better than both Chebyshev and Euclid.

6. Performance

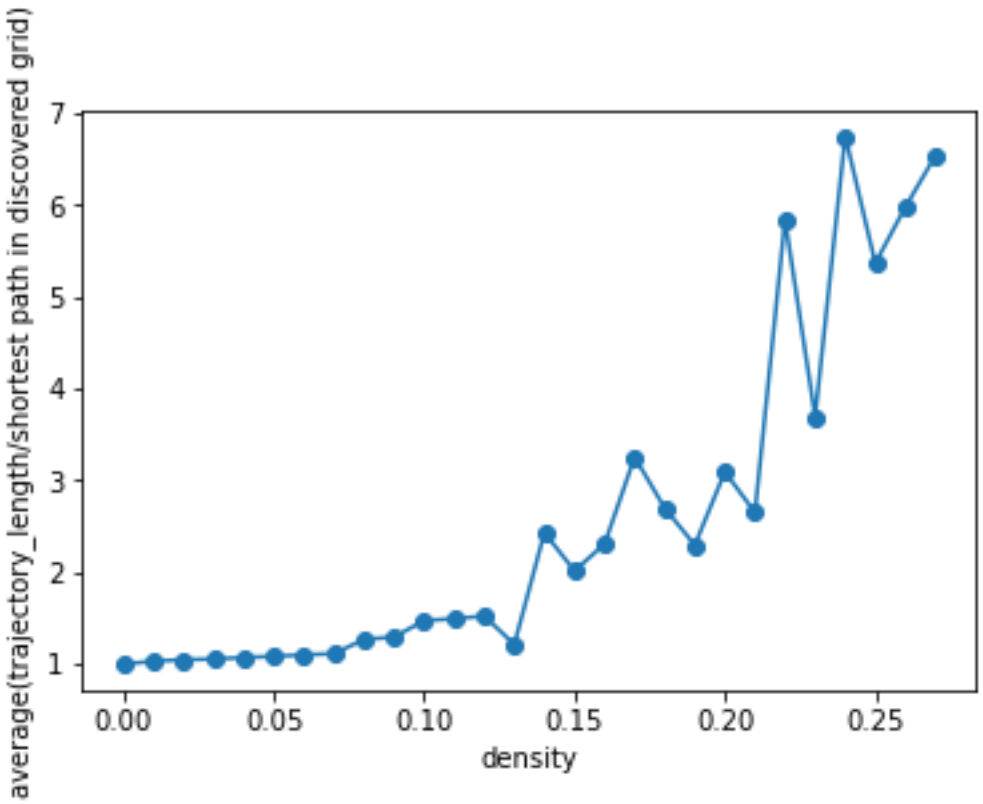
We generate random grid of dim 101 x 101 and run repeated A* for range of density p values from 0 to min(p0, 0.33). In this case our p0 is 0.27. And thus, we run it for 100 repetitions at each density p values from (0, 0.27) and take our observations for performance metrics. At each iteration, we generate new random gridworld of dim 101x101.

1) Density vs. Average Trajectory Length



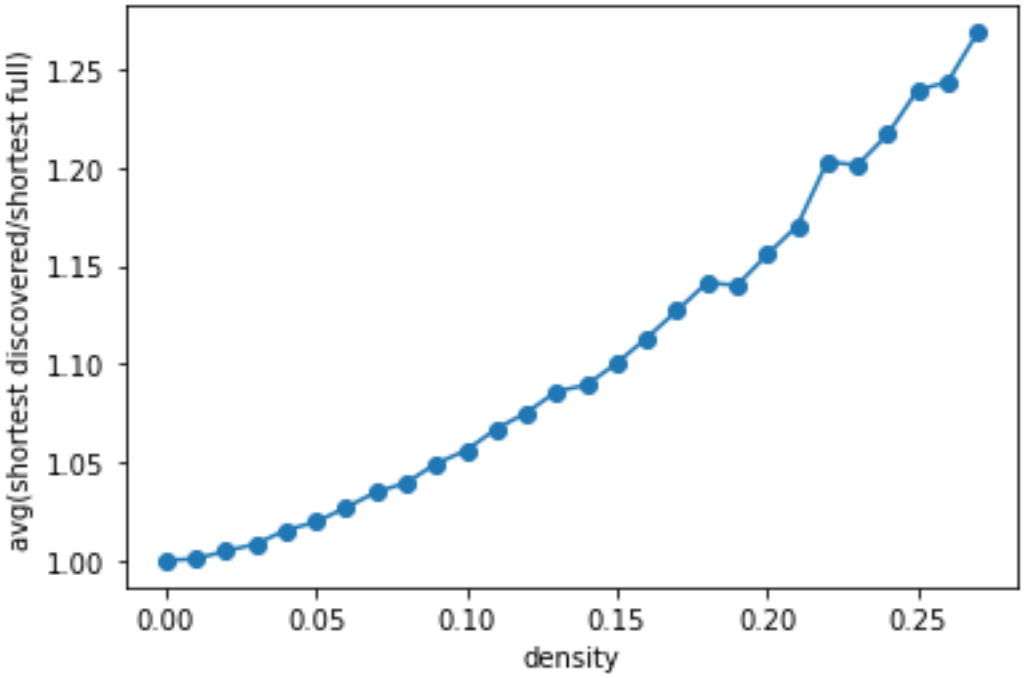
We clearly observe here that as the density p value for blocked cells increases, the trajectory length starts to increase, as agent must traverse through multiple possible paths now and replan its path whenever it finds a block.

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



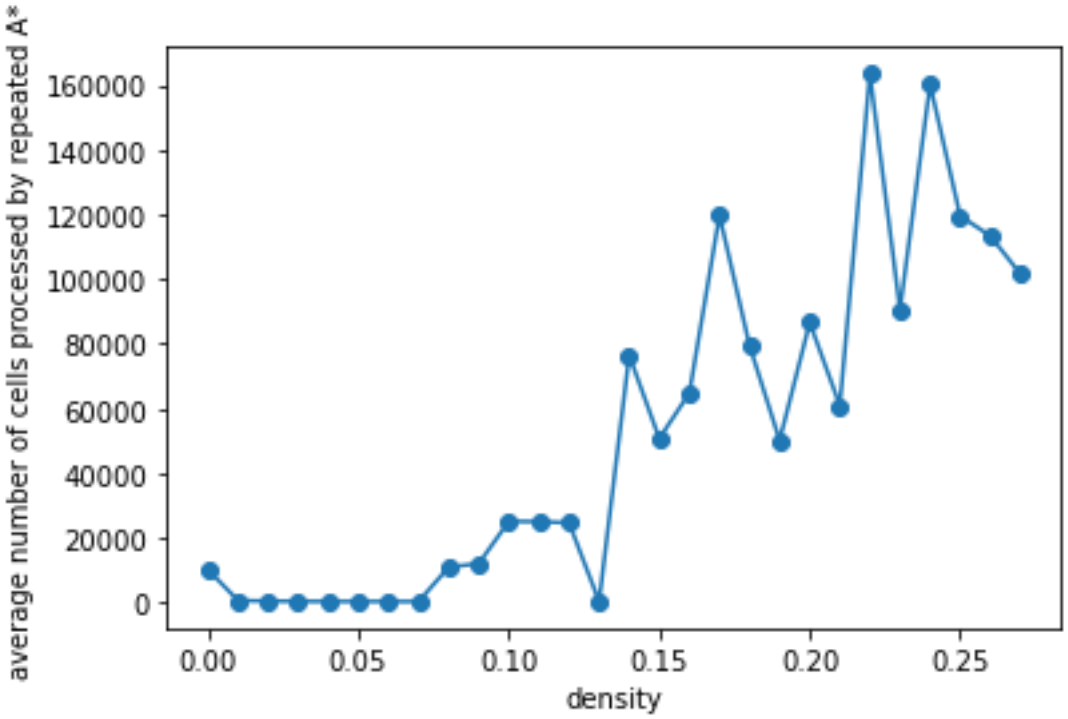
Here, we will observe that the plots are gradually increasing with increase in density p values. This is also expected as the number of blocks increases, trajectory also increases. On the other hand, with increase in density, the agent is unable to reach the goal node and hence providing no path to the Goal.

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



As we witnessed with one example in question 3, our observation comes true with this plot. First, we do observe that average length of shortest path is smaller in full gridworld as compared to final discovered gridworld by the agent. And we are somewhat aware why this happens. The agent plans one path and tries executing that particular path until it encounters any block, at which point it replans its path from the previous unblocked position. This might lead to some paths which are never traversed by the agent. And as we have seen, those paths can be the optimal shortest paths. Hence, this plot of density vs average(Length of shortest path in Final Discovered Gridworld / Length of shortest path in full gridworld) is somewhat expected behavior, where shortest path in full gridworld turns out to be smaller than shortest path in final discovered gridworld.

4) Density vs Average Number of Cells Processed by Repeated A*



This plot is again an expected rise. We have witnessed in earlier plots as well that with increase in density p value or in other words, with increase in number of blocks, the trajectory increases. With trajectory, we mean that with more number of blocks coming into picture, agent has to plan and execute multiple paths in order to reach the goal node. And thus, we can also say that with number of increased trajectories, number of processed cells are bound to increase.

7. Performance (Part – 2)

Generate and analyze the same data as inQ6, 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?

Due to some minor errors which we weren't able to solve in the given time, we weren't able to plot the graphs for this question. However, we'd like to discuss what we think they might have looked like and what we can infer from them.

We feel that if the agent is only allowed to move in one direction (i.e, it only knows it's surroundings once it bumps into them), this would have an effect on the average trajectory length. Since the agent now only knows the information about it's current cell as compared to 5 cells from Question no 6 (current + 4 children), we feel that it would have a negative effect on the trajectory length. This means that the agent will now spend much more time in finding the path from start to goal as compared to the previous problem, bumping into blocks and backtracking a much more number of times than before. So the trajectory length should come out to be more than before. Therefore, Density vs Average Trajectory Length plot should have a much steeper incline than the previous problem.

However, we do feel that the shortest path in the final gridworld will still be the same. We think that the agent will now take much more running time to reach the goal node but once it's there, it should return the smallest path which should be of the same length as returned by the agent in the previous problem. Therefore, the plot for Density vs Average (Length of Shortest Path in final discovered gridworld vs Length of Shortest path in full discovered gridworld) should come out to be the same as in the previous problem.

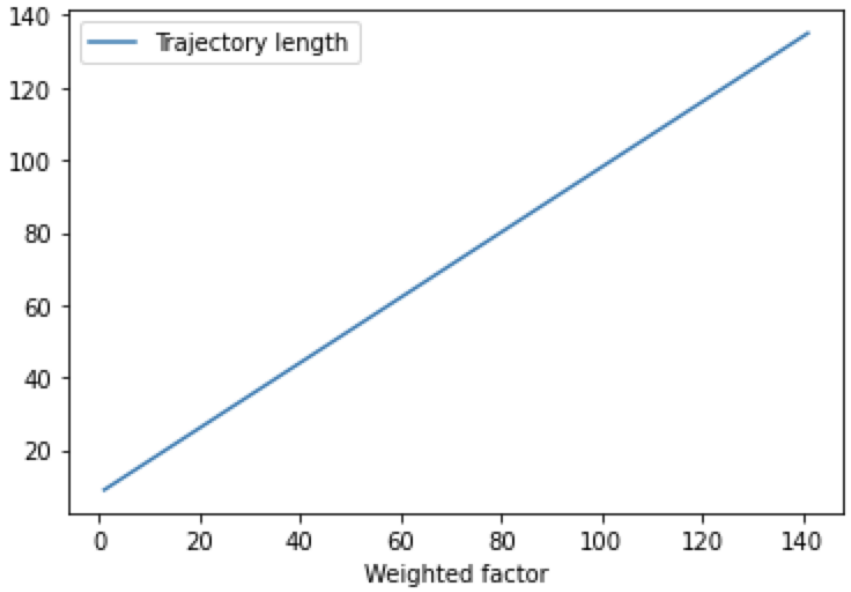
We feel that the average number of cells processed by our repeated A* algorithm will be much more for this problem as compared to the previous problem. Since our field of view has been decreased, the agent will bump into much more blockages than before, leading for it to backtrack much more times as well. This will eventually lead to an increase in the number of cells processed. Therefore, we feel that the plot for Density vs Average number of cells processed by repeated A* will have a much steeper incline than the previous problem.

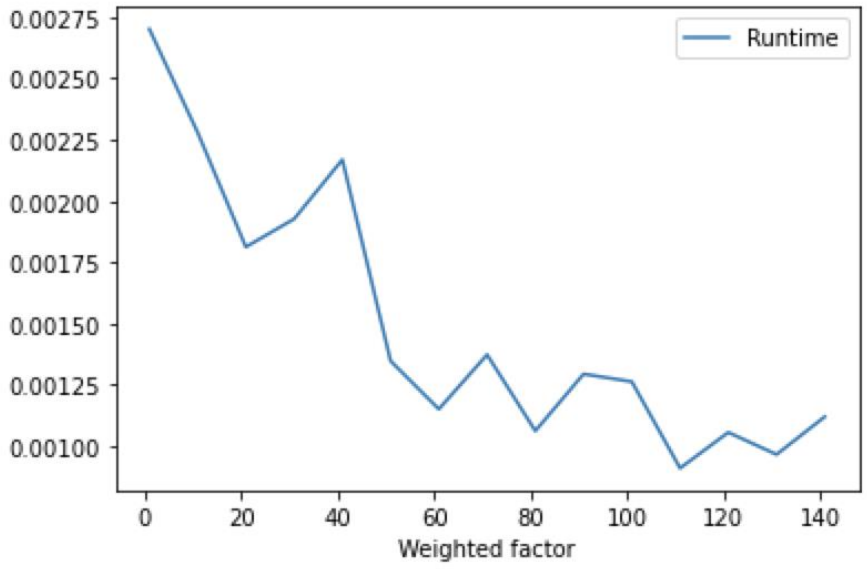
Since, we feel that the length of trajectory will be more for this problem as compared to the previous problem but feel that the length of shortest path in final discovered gridworld will be the same, therefore the plot of Density vs Average (Length of trajectory/ Length of shortest path in Final Discovered Gridworld) should come out to be a little more inclined than the previous problem.

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.





We have plotted Trajectory length and Runtime against weighted factor, where weighted factor is the variable which multiplies the heuristic. We have taken weighted factor till 150

- Trajectory Length is increasing linearly with almost constant slope.
- Runtime is decreasing , but with uneven slope until weighted factor of 110. After that It seems to be increasing.
- By increasing the weighted factor we are cutting down the runtime significantly . However trajectory length increases.

Decrease in runtime is because of the fact that overall cost for most of the blocks initially will appear different unlike when weighted factor is 1. Most of the nodes had same cost in the maze. This changes when heuristic becomes much larger and now priority of each block will be different. Now path will appear more clear and thus traversing it will take less time.

When we decide heuristics its better if we choose lower bound. But now we are multiplying it with a big factor, moving it towards upper bound. Agent won't be touching some of the nodes because of very high estimated heuristic, which will increase overall cost. Now since agent is visiting lesser no of nodes, runtime decreases.

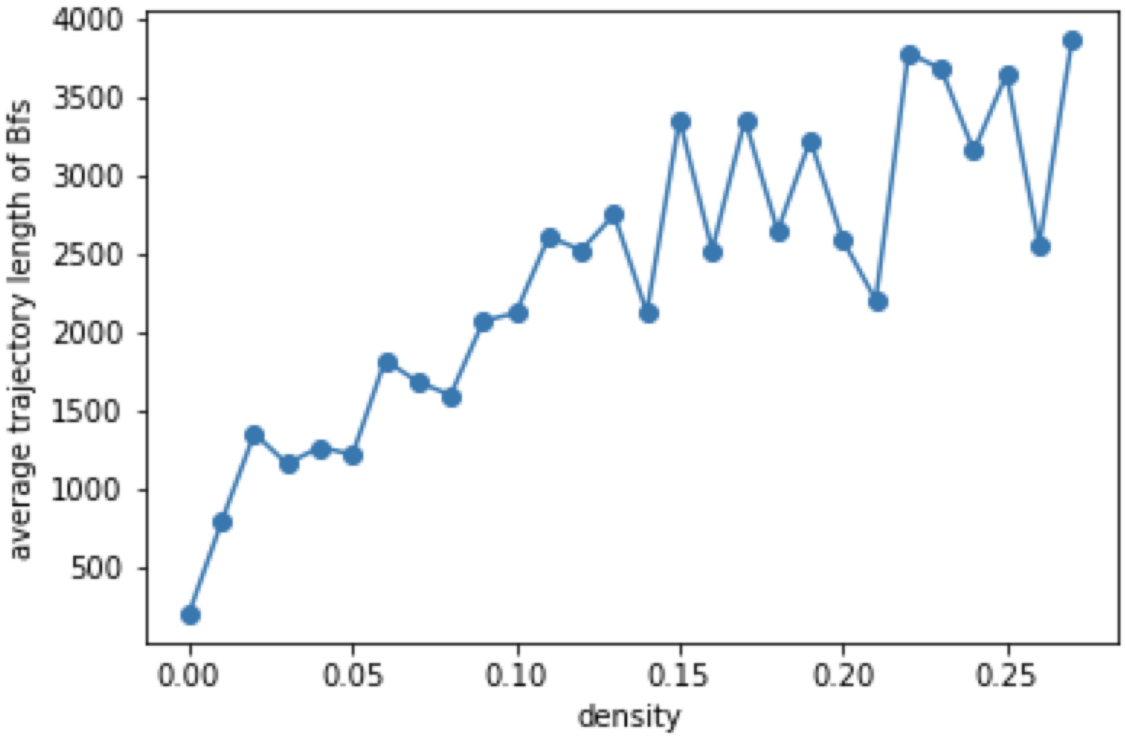
On the contrary , trajectory length increases because every time we are visiting lesser nodes. Nodes might have the best and shortest path. Agent chooses larger trajectory as it loses the shortest path.

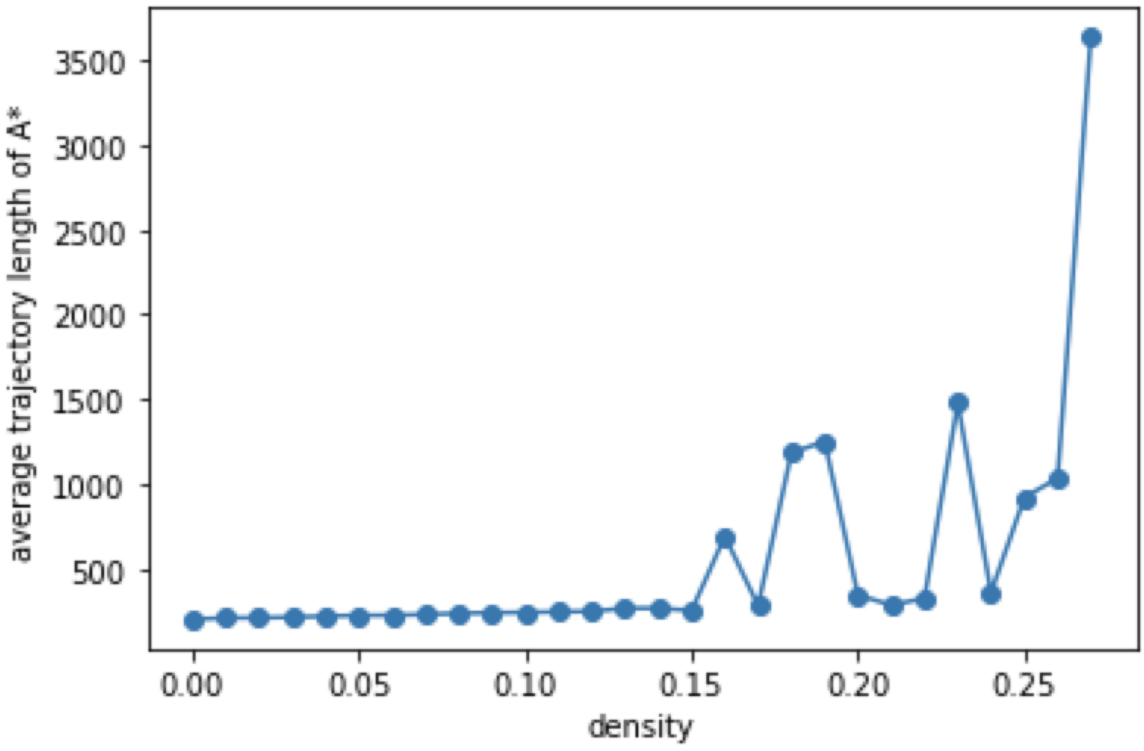
So, Runtime will reduce and trajectory length will increase.

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.

Density vs Average Trajectory Length:

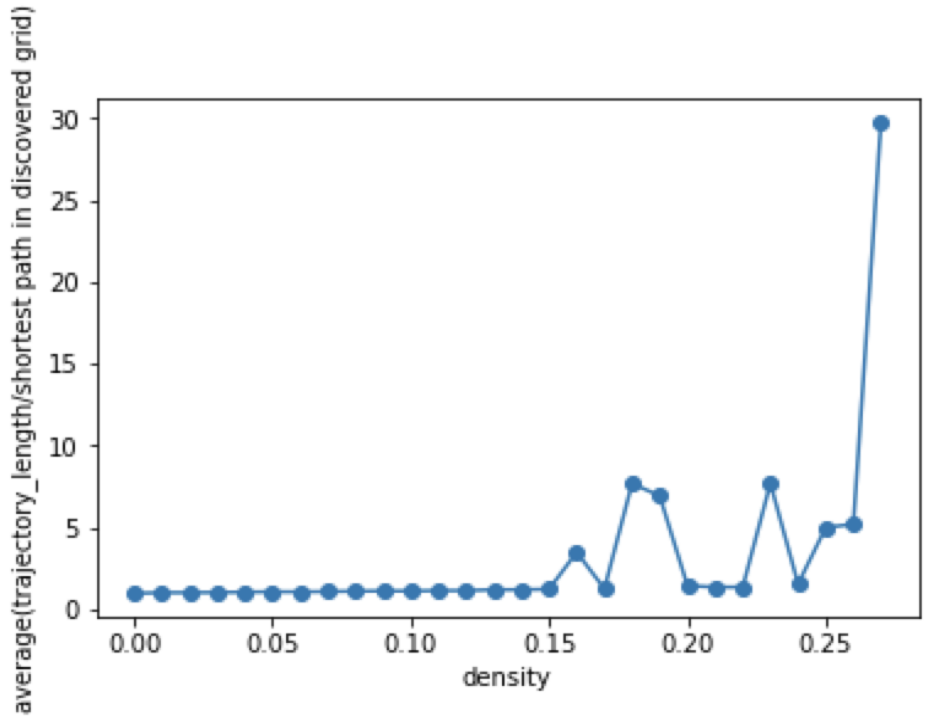




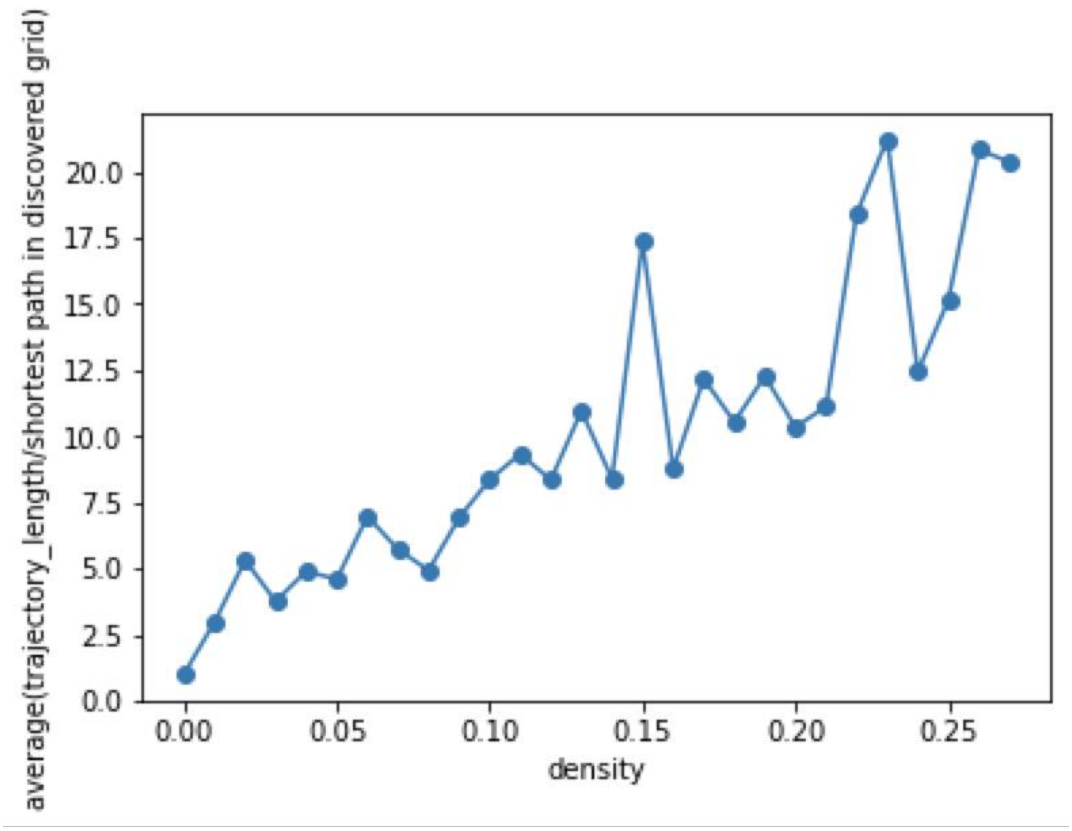
Above graph is plotted for 101 Dimensions for both repeated BFS and A*
Plot is for Average Trajectory length against density.

- We can see average trajectory length of BFS is greater than A* when density is lesser and most part.
- Average Trajectory length growth for BFS is more regular than A*. In A* initially growth is very lessened slope is much lesser compared to BFS But as soon as the density increases average trajectory length shoots up and is comparable to BFS.
- BFS checks all the elements in a level order traversal and there is more chance of it facing blocked cell, which will lead to continuous planning of the maze and it may have high number of fringe allocation. This increases trajectory path significantly even for lesser no of blocks.
- While A* checks a path based on the priority of the cost. We are calculating heuristics which helps in determining the most optimal path. Since it doesn't checks all the elements like BFS , chances of it facing blocked cell may be less as its only traverses along priority node unlike BFS where traversal is done for all the elements. However when density increases , number of blocked cell increases, solvability reduces , chances of replanning increases considerably. Advantage of priority reduces since it's also decided on estimation. Sometime estimation may go wrong and a path leading to goal is missed. In this case BFS might turn out to be better as it takes all nodes into consideration. However on an average A* is much better than BFS.

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



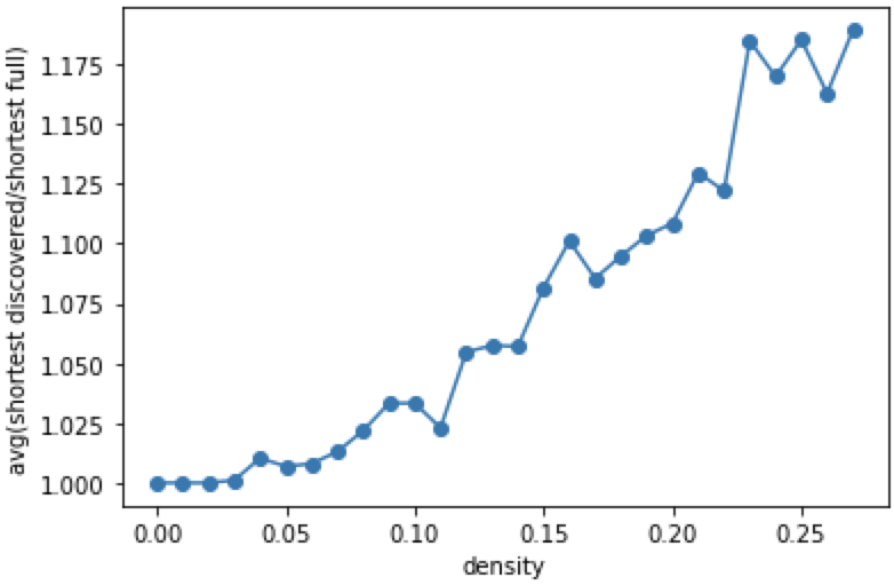
A* Graph



BFS Graph

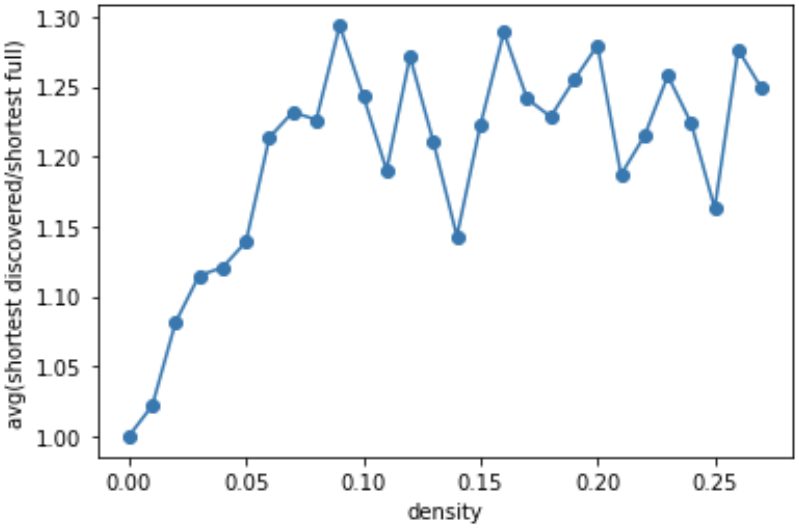
Above graph is for Density vs Average (Length of Trajectory / Length of Shortest Path in Final Discovered Gridworld).

- We can see from the above graphs that A* grows negligible when density is less while its growth shoots up when density increases. BFS grows regularly and its Trajectory against shortest is mostly more than A*. However when density increases the difference between both algorithms reduces.
- Worst case of A* is worse than BFS. This is the case when there are many blocked cells . A* might select a path which is estimated better but turns out to be much worse due to many blocked cells . Since Bfs is level order traversal, it will take all the paths in consideration and it might find goal with the best path much before A*. A* might have to replan several times and eventually the path found may not be shortest of full grid, which Bfs could find . In that not only trajectory length increases but shortest path length also be more . So total factor becomes worse. This happens when heuristic estimation doesn't work that effectively. That will exactly happen when there are many blocks.



\

A*

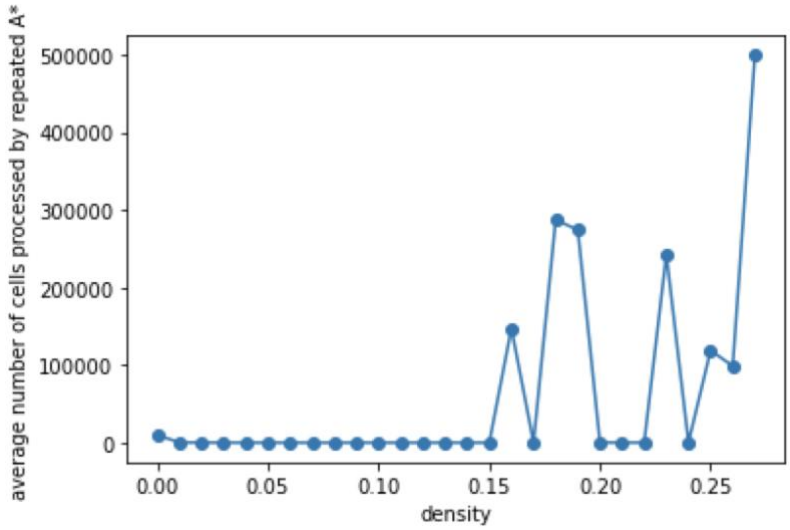


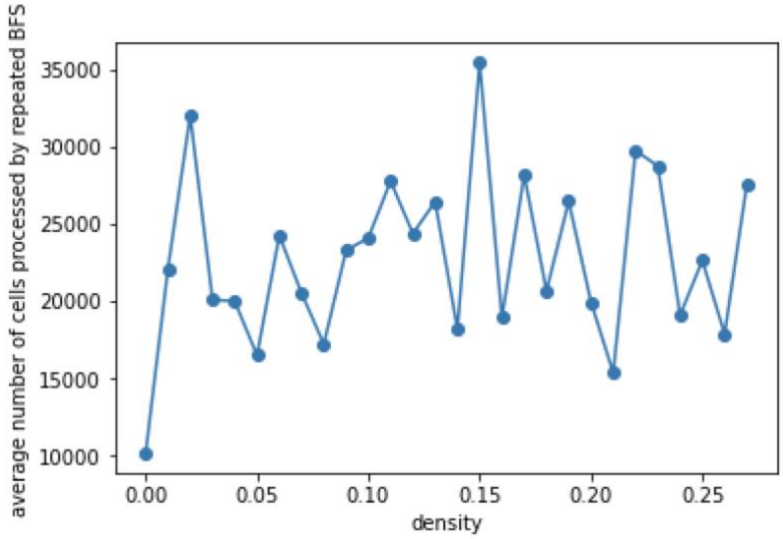
BFS

With the above plots we can see that repeated A* works properly and better than repeated BFS. A* takes into consideration priority queue with better costs. It also has a heuristic for more accurate result. Even if there is a replanning , heuristic approach will always give the best path. As we can see from the plot shortest discovered is almost equal to shortest full path. However the ratio shoots up when density increases as heuristic becomes inefficient.

In case of BFS best ratio is at the start and after that ratio increase remains steady even if density increases. Unlike A* it doesn't depend on density to that extent here. It's worse than A* for lesser density and ratio of shortest discovered is higher. This happens because Bfs is level order traversal and takes all the nodes into account. It is more accurate for full gridworld, but when replanning occurs, due to its level order traversal approach it will find the path which might not be accurate as it might reach goal before exploring some of the best paths.

Density vs Average Number of Cells Processed





- From above we can see that increasing density hardly affects number of cells processed in repeated BFS, while for A* when density is less , hardly any cells are being processed, but when density increases number of cells increases astronomically and in the end it even crosses BFS.
- When Density increases heuristics become ineffective causing A* to get much worse. The reason are same as mentioned before.

Contributions

Parth Goel:

- Created the second version of the code, in which the first version was bettered, fixing all the errors and also improving the complexity of some functions.
- A* algorithm was completed in this version, contributed in Appendix.
- Apart from this worked on solutions of the problem numbers 1, 4, 7 from the assignment.

Prashant Prasad Kanth:

- Created the third and final version of the code, in which the second version was converted from A* to repeated A* algorithm.
- Some minor bugs and logical errors were fixed as well, contributed in Appendix
- Apart from this worked on solutions of the problem numbers 3, 6, extra credit from the assignment.

Utkarsh Jha:

- Created the initial first version of the code which had a few errors but layed a strong backbone for the entire project.
- Apart from this worked on solutions of the problem numbers 2, 5, 9 from the assignment.
- Contributed in Appendix