CS520 Project 1: Voyage Into The Unknown

Daniel Ying (dty16) Sec 198:520:01, Zachary Tarman (zpt2) Sec 198:520:01

September 28, 2021

Submitter: Daniel Ying

Honor Code:

I abide to the rules laid in the Project 1: Voyage Unknown description and I have not used anyone else's work for the project, and my work is only my own and my group's.

I acknowledge and accept the Honor Code and rules of Project 1.

Signed: Daniel Ying (dty16), Zachary Tarman (zpt2)

Workload:

Daniel Ying: Coded the Program for Voyage Unknown in Python. Formatted the report in LaTeX. Recorded data for questions 8, 9, and BFS. Answered 9, and BFS.

Zachary Tarman: Authored the report for Question 1 through Question 8 and collected corresponding data for those questions.

Together: Brainstormed the Algorithm of Repeated Forward A*. Discussed the variants to the regular Repeated Forward A* algorithm needed for collecting data on Q4-Q9. Discussed problems in the assignment and code.

Problem 1

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

ANSWER:

I think a fairly reasonable explanation is that the agent originally assumes that all cells are unblocked (the "freespace assumption" in the write-up).

Granted, when the agent makes a planned path in the first part of an iteration and then starts to follow that path in the execution phase while discovering cells' status along the way, if the cells are unblocked, it does technically gather knowledge of the environment. However, this is not information that requires the agent to change its planned path since unblocked status was already its assumption.

On the contrary, if it discovers a blocked cell in its path, this will necessarily require that we change our current planned path. The agent cannot continue, and it must replan from its current state.

Problem 2

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

Answer:

No, the agent will not get stuck in a solvable maze.

Though it may be a long time before a successful solution is reached in some cases (i.e. if the agent gets stuck in multiple deadends and has to backtrack), if there is some path from the start to the goal, the replanning process will allow the agent to continually revise its path to the target with each new discovery of a block in its path. For example, if the agent hits a blocked cell, the execution phase of the current iteration is halted, and it will replan its route. With the new knowledge of the block in its path, it will know to avoid that block when planning its future path.

As long as there is some path from start to goal, the agent will discover it by trial and error at the very least (if not also good planning via heuristics or some other method to quicken the search). More generally, most search algorithms such as BFS and DFS would also succeed in not getting stuck because the agent will keep searching unblocked edges until a goal is reached, and only after the entire fringe is exhausted would the algorithms declare that the maze is unsolvable.

Problem 3

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

Answer:

No, the shortest path that the agent has discovered will not necessarily be the most optimal path in the complete gridworld. In other words, it might be, but it cannot be assumed.

The key sentence here from the logical cycle the agent programmatically follows is, “Based on its current knowledge of the environment, it plans a path from its **’current position’** to the goal”. There may very well be a case such that the shortest path in the discovered gridworld is also the optimal path of the complete gridworld, but in general, because replanning means trying to determine what the next best path is from the “current position” as opposed to looking at all discovered cells and deciding the best place to restart the search, this will not always be the case.

The argument above can be seen in the maze run figures 1 and 2, located below. Since the agent encounters many blocked cells in its path, it’s forced to replan multiple times. During these replanning steps, it’s put into a position where it must make the best decision from where it is currently, but it doesn’t know what path on the virtual "fork in the road" to actually take since it has no knowledge of the nebulous cloud ahead. Once it chooses one path, it may be stuck on a less optimal path as compared to if it had knowledge of the complete gridworld.

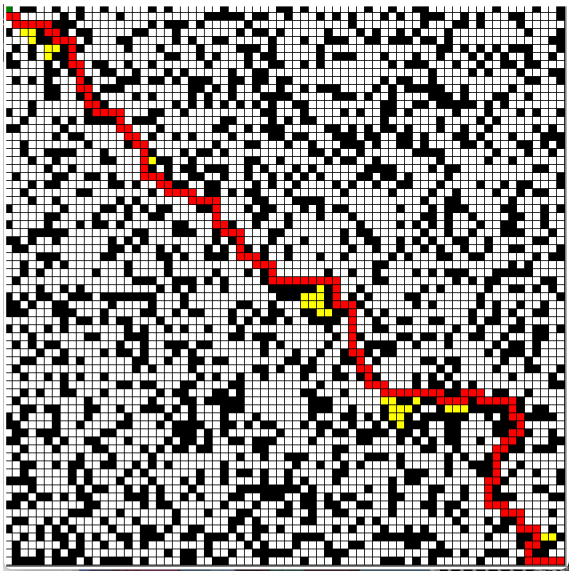

Figure 1: Astar Path Run (Yellow: Path found by agent, Red: shortest path with info found by agent).

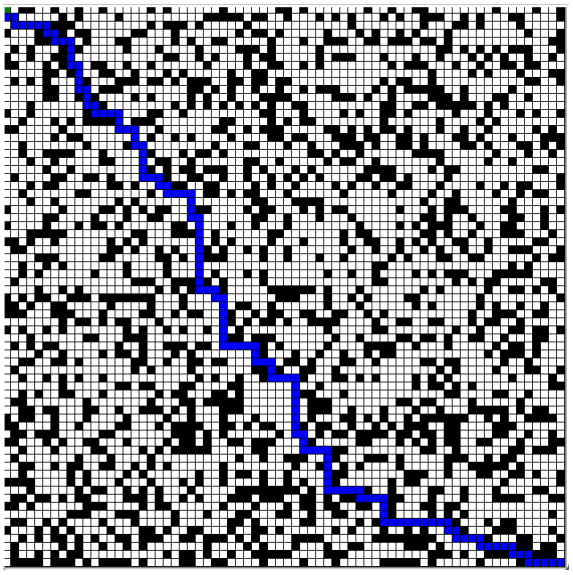

Figure 2: Optimal Path Run (Blue: agent knows location of walls).

Problem 4

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

Answer:

To answer this question, we ran trials for solving the gridworld of density probabilities ranging from 0.0 to 0.4 at an interval of 0.025. For every given density, we ran 30 trials, recorded which ones were solvable and which ones weren't, and then we took the percentages (averages) and plotted density vs. solvability.

As you can see below in the table / graph, there were some interesting results, but overall the downward trend as density increased was what we would have expected. However, it was intriguing to us that the solvability followed that of a quadratic relationship versus a linear one. It becomes obvious when looking at the graph that the solvability percentage decreases more rapidly as density increases. It is logical that the more obstacles are in the way, the harder it will be to make it to the goal, but it's also worth noting that the maze is almost always solvable for densities from 0 to 0.1. I suppose the blocked cells would have to be arranged rather distinctly to be able to block the agent with such a low density. Meanwhile, the more obstacles are in the maze, it doesn't take a special arrangement to completely block the agent from moving forward towards the goal node. For example, if just the cells to the start cell's south and east are blocked, there's nothing the agent can do, and that becomes all the more common the higher the density gets, let alone blocked cells just past those two cells that were just pointed out. And we were having trouble finding any mazes that were actually solvable as we approached 0.35 and 0.4, so this further supports that notion.

Besides that, having applied a quadratic line of best fit, we extrapolate that the p0 value that we were looking for, where for p > p0 most mazes aren't solvable (we're assuming 50 percent of mazes, a simple majority in other words), is equal to 0.286, which was a tad surprising but believable (in line with the prior discussion). So, in conclusion, we found that **p0 = 0.286**.

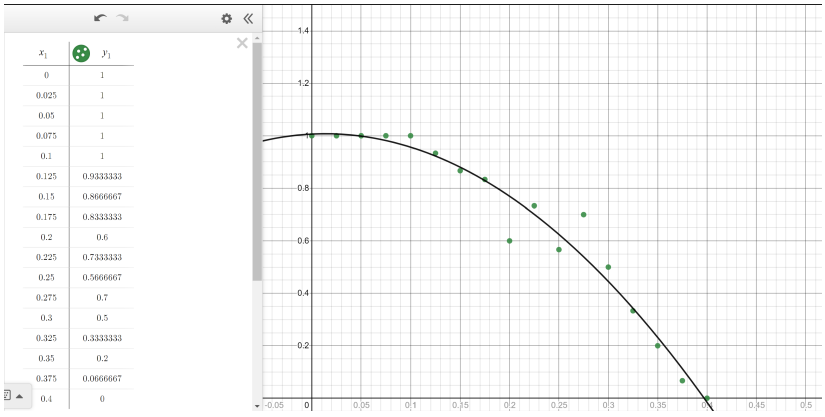

Figure 3: Q4 Density of Blocked Cells vs Solvability of Maze.

Problem 5

Heuristics: Among environments that are solvable, is one heuristic uniformly better than the other for running A*? 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:

We decided that the metric to compare the performance of the heuristics was the runtime of the algorithm itself. Of course, this is specifically for running A* with full knowledge of the gridworld and the given heuristic.

According to the data provided below, we can see the clear winner in terms of quickest runtime is the Manhattan distance heuristic. On average, it performed faster than both the Chebyshev distance heuristic and the Euclidean distance heuristic. The data speaks for itself for the most part, however we'll discuss some of the thought process behind the decision to pick this specific metric.

This is certainly not the only metric that could've been chosen to compare the performance of these heuristics against each other, but I will reference something from class when speaking of performance of the algorithms we've been studying recently. There's a point where the difference between the best and "good enough" / "close enough" is negligible compared to the effort that would need to be put in to find the absolute best outcome. In terms of this situation where we want to find the shortest distance to the goal cell, the difference between one heuristic's distance to goal and the next is not as significant as the time difference (which is displayed by the data collected). Manhattan has a pretty clear dominance over the other two in runtime, and if the output is "good enough" for our purposes in terms of metrics like trajectory length and shortest path in final discovered gridworld in performances of other heuristics, then that's the appropriate heuristic to use from here on out.

With that being said, we find that the **Manhattan distance heuristic** is the best performing, and we choose to use it for all future trials throughout the rest of the report. (Data shown in figure 4)

Heuristic	Euclidean	Manhattan	Chebyshev
Runtime (s)	0.123887	0.049512	0.066961
	0.106638	0.012968	0.117005
	0.083223	0.012969	0.105573
	0.081241	0.039649	0.058868
	0.073404	0.011968	0.055349
	0.088698	0.032519	0.045945
	0.062079	0.060637	0.056876
	0.115619	0.023944	0.088936
	0.092436	0.064559	0.090814
	0.106694	0.039989	0.047882
	0.114308	0.018965	0.097313
	0.059536	0.030912	0.111787
	0.142457	0.026583	0.069893
	0.066025	0.144334	0.090742
	0.073923	0.014959	0.186656
	0.0543148	0.0470876	0.09592247
	0.1121957	0.0329111	0.11246061
	0.0724452	0.0464155	0.06239581
	0.0898513	0.0369081	0.09544301
	0.0520782	0.0169641	0.14927363
	0.0884878	0.0229518	0.06485891
	0.0765033	0.0149607	0.11493683
	0.0764575	0.0219512	0.05476284
	0.0745742	0.0379862	0.0534029
	0.0615096	0.0398924	0.06388879
	0.0624265	0.0339248	0.05138993
	0.0709291	0.0249422	0.10359597
	0.1066019	0.0129695	0.05549765
	0.0779721	0.0199573	0.04987407
	0.1245911	0.0449352	0.07186842
Average Runtime (s)	0.0863702	**0.0346408**	0.08300573

Figure 4: Q5 Runtimes of Trials to Solve Mazes Using Different Heuristics.

Problem 6

Performance: Taking dim = 101, for a range of density p values from 0 to min(p0, 0.33), and the heuristic chosen as best in Q5, repeatedly generate gridworlds and solve them using Repeated Forward A*. Use as the field of view each immediately adjacent cell in the compass directions. Discuss your results. Are they as you expected? Explain.

Answer:

Below, one can observe the various performance metrics of Repeated A* plotted out for a range of densities from 0 to 0.275 (which is just shy of our p0 = 0.286) at an interval of 0.025. For each density, we collected data for 30 trials and took averages as needed from there. As discussed in Q5, we utilized the Manhattan distance heuristic as we found that to be the best performing overall.

A big takeaway is that there is an exponential relationship between density and trajectory length. In a similar way (but not completely the same since it's inverse), trajectory length seems to have the opposite tendency as solvability. As density increases, trajectory will get exponentially longer and solvability will decrease in a quasi-similar fashion.

I think it would be fair to say, based on the data we've collected, that the higher the density, the more obstacles could potentially halt the agent and so the number of possible paths also decreases. As the gridworld becomes more and more restricted, the possible paths decrease in number and the average path lengthens. This is why we see 1) an increase in the average trajectory and 2) a decrease in the solvability.

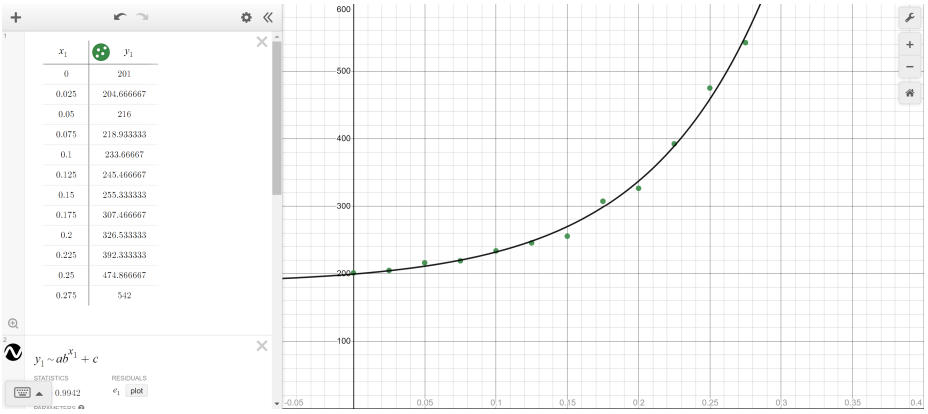

Figure 5: Q6 Density of Blocked Cells vs Average Trajectory Length of Agent

Another interesting observation is that the exponential relationship persists with density versus average trajectory length / length of shortest path in the final discovered gridworld, but we see a linear growth in density versus length of shortest path in the final discovered gridworld / length of shortest path in the complete gridworld. One other thing to note is that throughout all of these trials, the length of shortest path in the complete gridworld stayed very consistent only varying as much as 6 cells and almost universally staying at 201 (this is of course counting the start node as a cell in the path).

The latter graph with the linear line of best fit surprises me though. Why is this the case? I would make an educated guess based on the data presented before me that while there is more work that must be put into the agent travelling through the maze with more and more collisions, the ending performance doesn't grow as rapidly. This is perhaps a great compliment to the performance of the algorithm, that even though the agent is having to travel and collide more, we still end up finding a relatively short path as compared to the most optimal shortest path.

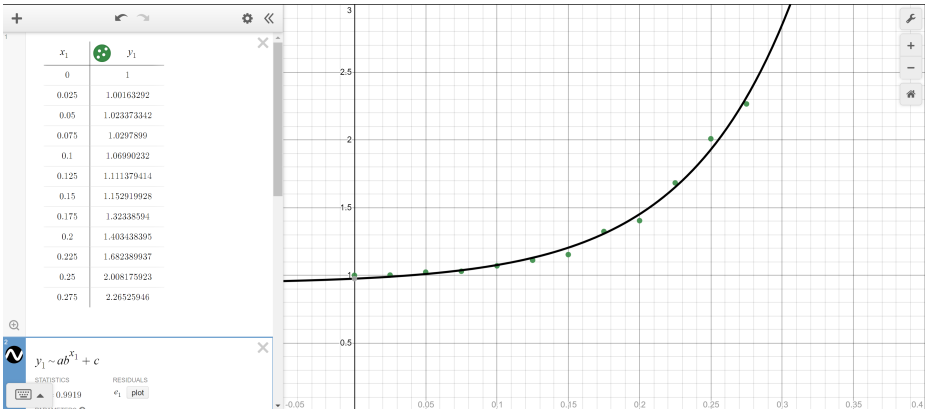

Figure 6: Q6 Density of Blocked Cells vs Average Trajectory Length of Agent Over The Average Length of the Shortest Path in the Final Discovered Gridworld.

Lastly, we can see that there's also an exponential relationship between density and the average number of cells processed. Taking what we've seen in Q4 and the first data set of this question into account, we expected this outcome. One thing of note is that the trend is almost exactly the same as that of trajectory length, but the "constant" values of the best fit equation are just slightly inflated, and this makes sense as well. We're not counting blocked cells in our

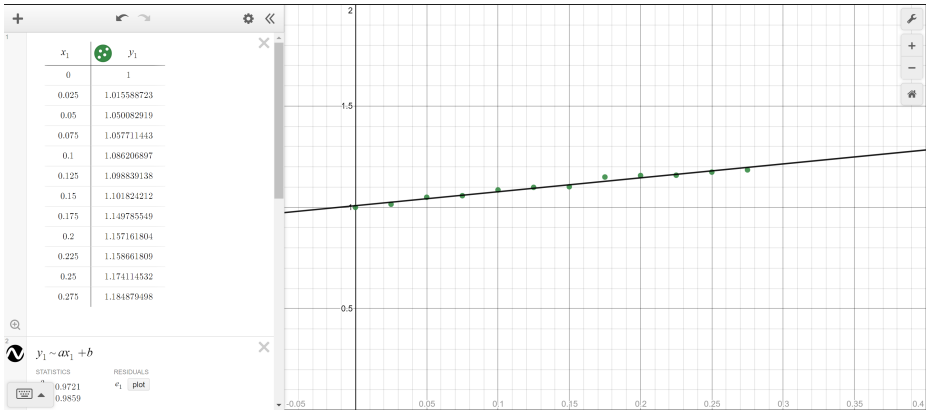

Figure 7: Q6 Density of Blocked Cells vs Average Length of the Shortest Path in the Final Discovered Gridworld Over Average Length of the Shortest Path in the Complete Gridworld.

trajectory, but we have to process them if we run into them. The more collisions we have, the greater the difference between cells processed and trajectory in a fashion that keeps the rate of rate of growth similar, if that makes sense.

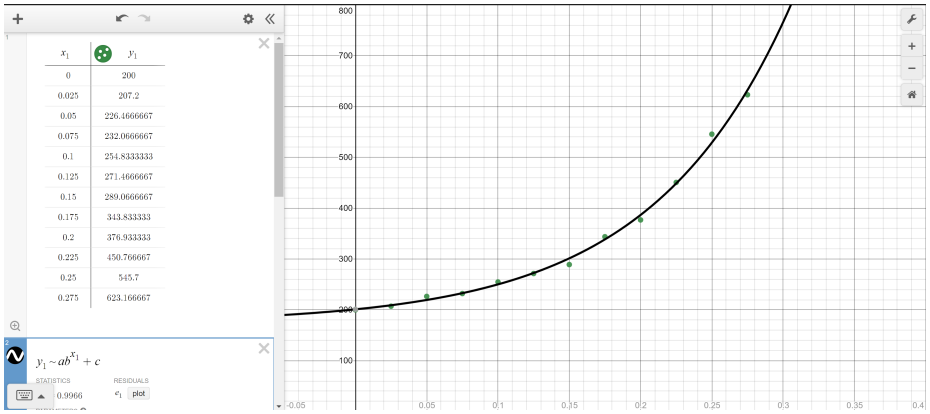

Figure 8: Q6 Density of Blocked Cells vs Average Number of Cells Processed by Agent.

Problem 7

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

Answer:

Below, one can observe the various performance metrics of Repeated A* with a limited line of sight plotted out for a range of densities from 0 to 0.275 (which is just shy of our p0 = 0.286) at an interval of 0.025. These are plotted against the results from Q6 as well to see the relationship between the two environments, so refer to the purple data points and the red line of best fit. For each density, we collected data for 30 trials and took averages as needed from there. As discussed in Q5, we utilized the Manhattan distance heuristic as we found that to be the best performing overall.

Overall, these results surprised our group. While we can see in the plotted graphs below that the trends for no-sight A* are all slightly worse than those of A* that can see in all cardinal directions, we would've expected there to be a greater difference in performance. The handicap doesn't seem to have as much of an effect as it would originally seem.

I wonder if the difference in performance would become even more dramatic the higher the density would become, as the plots and best fit curves would suggest. The reason I say this is because for a lot of lower densities, there aren't too many collisions and in a lot of cases, the agent doesn't need revisit too many areas of the graph twice. For example, at a density like 0.1, the agent might have a block in its path, but it can easily go around that block in the middle of mostly unblocked cells and never have to reroute back to that area. In other words, the forward progress is pretty consistent and unless there's a really tricky situation, the knowledge that the prior agent would have of seeing in all cardinal directions is not as big a benefit as it would first appear.

However, like I said, once the probability of the maze being solvable gets less and less and more collisions pile up and the trajectory length gets higher, perhaps a really significant difference in performance would show itself.

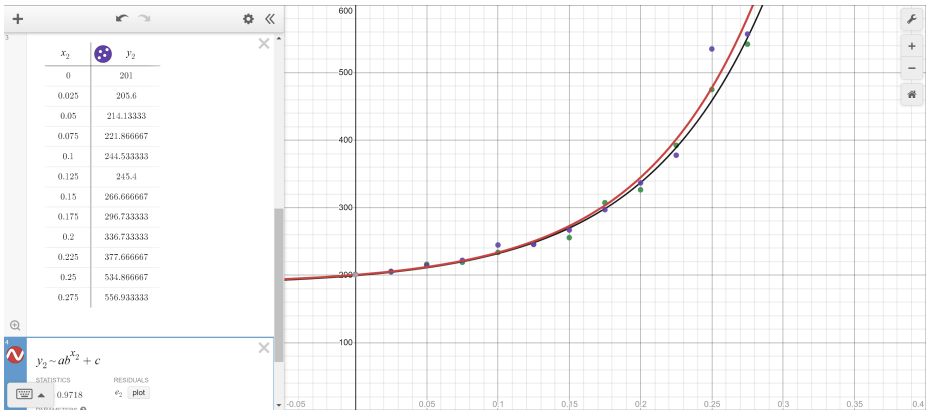

Figure 9: Q7 Density of Blocked Cells vs Average Trajectory Length of Agent (green and black for Q6 data, purple and red for Q7 data).

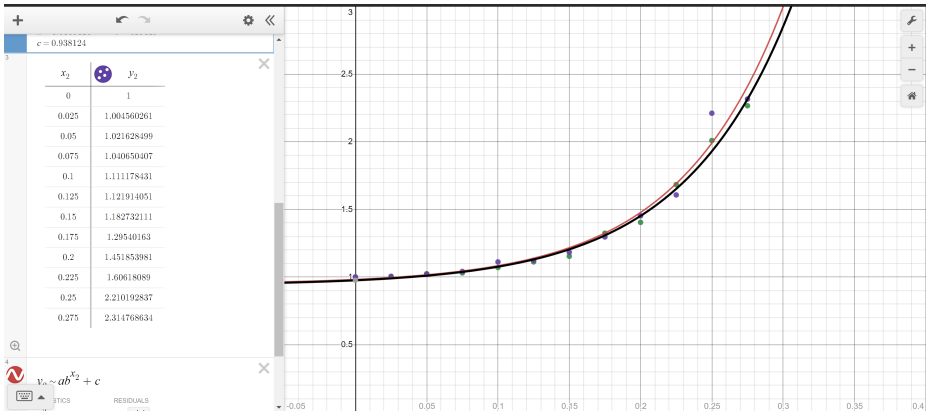

Figure 10: Q7 Density of Blocked Cells vs Average Trajectory Length of Agent Over The Average Length of the Shortest Path in the Final Discovered Gridworld (green and black for Q6 data, purple and red for Q7 data).

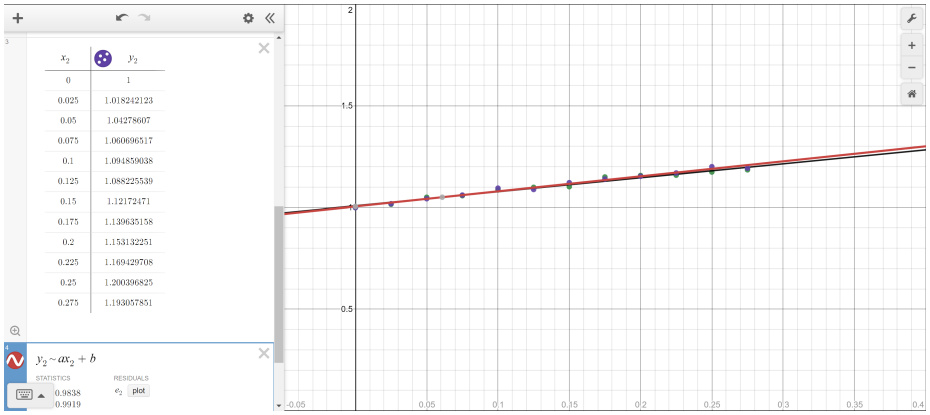

Figure 11: Q7 Density of Blocked Cells vs Average Length of the Shortest Path in the Final Discovered Gridworld Over Average Length of the Shortest Path in the Complete Gridworld (green and black for Q6 data, purple and red for Q7 data).

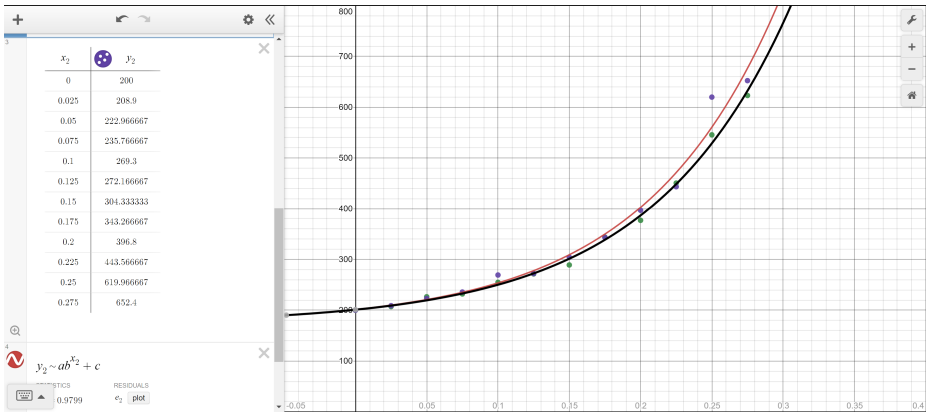

Figure 12: Q7 Density of Blocked Cells vs Average Number of Cells Processed by Agent (green and black for Q6 data, purple and red for Q7 data).

Problem 8

Improvements: Repeated A* may suffer in that the best place to re-start A* from may not be where you currently are - for instance if you are at the dead end of a long hallway, you can

save some effort by backtracking to the end of the hallway (recycling information you already have) before restarting the A* search. By changing where you restart the search process, can you cut down the overall runtime? What effect does this have on the overall trajectory (given that you have to travel between the current position and the new initial search position)?

Answer:

The data collection tactics were the exact same ones as those in Q6 and Q7. For the sake of clarity, once again, averages were computed out of 30 trials for every given density. You can see the results of these trials plotted below with purple coordinates and a red line of best fit as compared to the data from Q6, which is represented by green coordinates and a black line of best fit and it will serve the purpose of being a baseline.

In many of the metrics measured for Q6, the "improved" A* that doesn't necessarily have the agent replanning from the exact spot it's at currently seemed to improve performance, and in some respects quite drastically. For instance, in nearly every plotted graph (with the exception of "Density of Blocked Cells vs Average Length of the Shortest Path in the Final Discovered Gridworld Over Average Length of the Shortest Path in the Complete Gridworld" which we will address in a moment), we saw quite significant improvements as the density p approached p0. The average trajectory length, the average trajectory length over the average length of the shortest path in the final discovered gridworld, and most impressively, the average number of cells processed all decreased in relation to its corresponding data points collected for Q6.

Average number of cells processed improving in performance makes the most sense to me. When we get into a situation like the one described in the lab write-up where the agent gets stuck in a long hallway, we are able to save some computing power by jumping to a position where we are better suited to get to the goal. In fact, in some trials, the number of cells processed was actually slightly lower than the trajectory length which would mean that we're not having to process a whole bunch of unnecessary cells just to get back to a point where we can start "forward progress" again.

I will say I'm slightly surprised by the average trajectory length improving, considering the fact that while the agent restarts the search at another location from the collision cell, we still count the length of getting back to that spot. With that in mind, we would expect average trajectory length to stay more similar, but perhaps this also comes from the fact that the agent might realize it's wasting its time even without a complete deadend and decide to restart somewhere else after a collision. Interesting result there.

Now, we see that the plot, Density of Blocked Cells vs Average Length of the Shortest Path in the Final Discovered Gridworld Over Average Length of the Shortest Path in the Complete Gridworld, which is the one that has had a linear growth instead of an exponential one throughout this whole process, does not have any significant differences. If anything, it's very, very similar to the results from Q6 data collection. I think a conclusion that can be drawn from this is that the algorithm still executes to the same outcome (give or take). In other words, the improvements made to the Repeated A* algorithm don't necessarily get us a shorter path to the goal in the final discovered gridworld as compared to the optimal path in the complete gridworld. Actually, they're pretty much on par with each other. Instead, the differences are seen in the work being done to get to the goal. We see savings in runtime and computations as a result of these improvements (as shown in figures below).

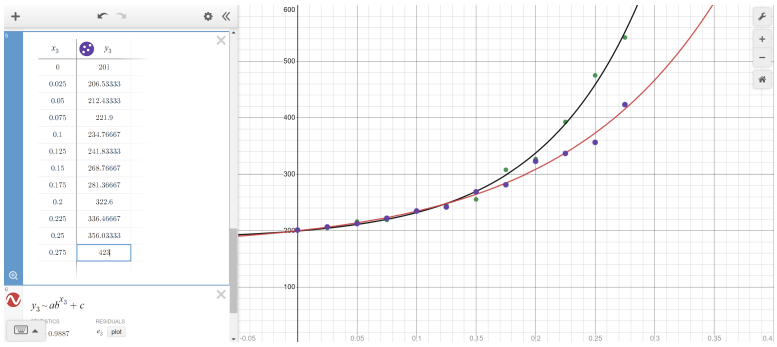

Figure 13: Q8 Density of Blocked Cells vs Average Trajectory Length of Agent (green and black for Q6 data, purple and red for Q8 data).

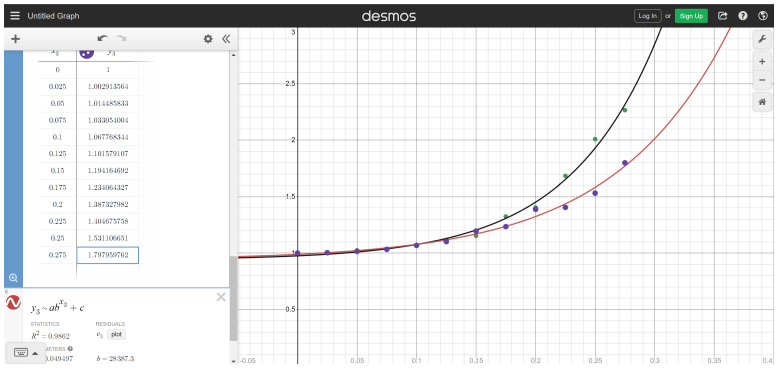

Figure 14: Density of Blocked Cells vs Average Trajectory Length of Agent Over The Average Length of the Shortest Path in the Final Discovered Gridworld (green and black for Q6 data, purple and red for Q8 data).

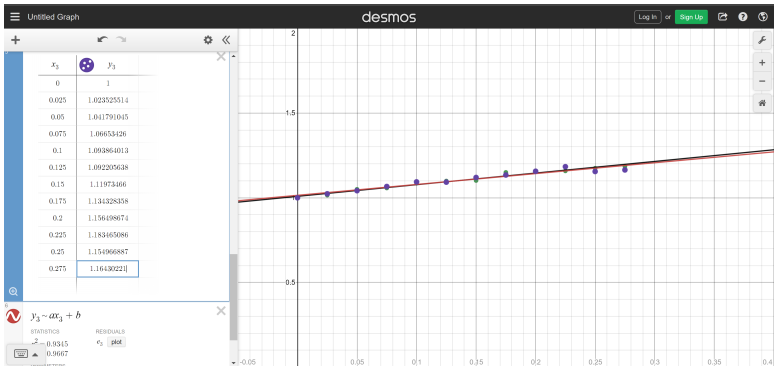

Figure 15: Q8 Density of Blocked Cells vs Average Length of the Shortest Path in the Final Discovered Gridworld Over Average Length of the Shortest Path in the Complete Gridworld (green and black for Q6 data, purple and red for Q8 data).

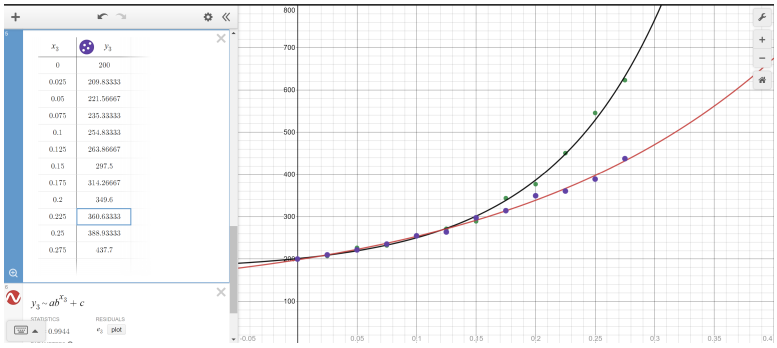

Figure 16: Q8 Density of Blocked Cells vs Average Number of Cells Processed by Agent (green and black for Q6 data, purple and red for Q8 data).

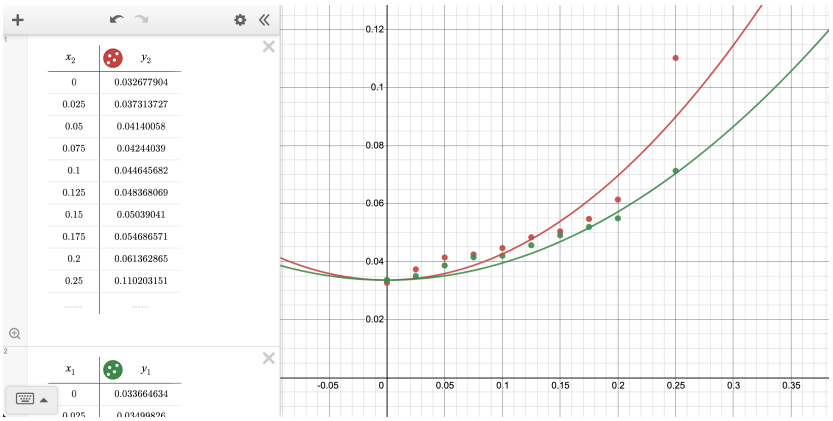

Figure 17: Q8 Density vs Avg Runtime in Repeated Forward Astar (red for Q6 runtime, green for Q8 runtime)).

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.

In the beginning, when the density (p) was below 0.2, the difference between weighted Astar and default Astar was not so obvious. But as the density increase, we observed weighted Astar started to diverge from default Astar. In Figure 18, we can see, weighted Astar though had close to identical grid length average as to that of default Astar, but when density reaches p=0.25, the grid length traversed by weight Astar's agent is more than that of the default Astar's agent.

As according to the given definition of weighted Astar, the algorithm decreases the runtime at the price of increase grid length. Thus the observation and data shown in figure 18, proves that weighted Astar do increase the grid length.

Now, looking at figure 19. The main purpose of weighted Astar is to decrease runtime, the increase of grid length is the price to pay for applying this algorithm. When the density is low around p=0.05, the runtime between default Astar and weighted Astar is quite similar. But when the density increases, the two algorithm's runtime diverge.

We can clearly see that as the density increase, default Astar's runtime increases at a greater rate than weight Astar's runtime. Thus when applied a maze (gridworld) with a high density, the effect of weight Astar become more significant.

When density reaches p=0.25, the default Astar's runtime is significantly greater than weighted Astar.

Thus our data shows that the effects of weighted Astar are true.

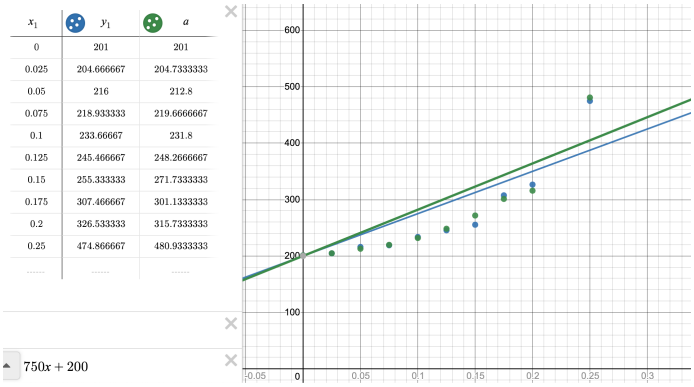

Figure 18: BFS Density vs Average Trajectory Length (blue for Q6 Astar, green for Q9 weighted).

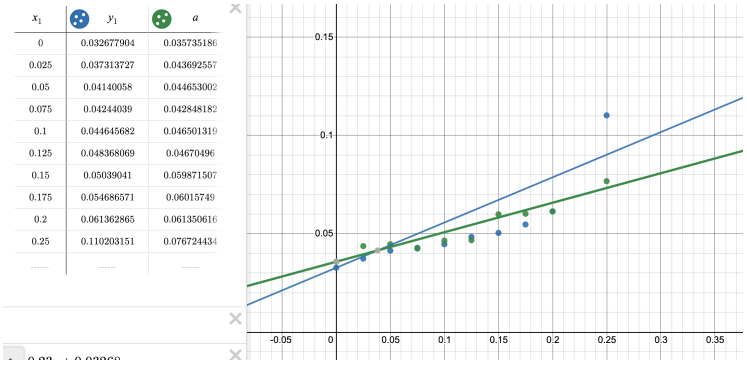

Figure 19: Density vs Avg Runtime in Repeated Forward Astar (blue for Q6 Astar, green for Q9 weighted).

Problem 10

Extra Credit with BFS: 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.

Looking at the figures below, one of the first impressions of the repeating BFS algorithm is its length. The grids traversed in the BFS algorithm is much larger than those traversed by the default repeating Astar algorithm.

This difference of grid trajectory length is very likely caused by the BFS's lack of direction. For default Astar, as stated in the previous problems, employs the heuristics value that helps the agent to plan a possible path that can lead it to the goal. But for BFS, there is no heuristic value for the agent to implement. Without a heuristic direction, we predicted that the agent would take a much longer time looking for a path to the goal, and end up with a much longer trajectory grid length than Astar.

Because the agent has no information in the grid world, and the agent that cannot jump, the agent is designed to prioritize the down direction and right direction before the other two directions (left and up). If a block is found by the agent, similar to Astar, the agent would record and update the information then replan and execute its new path.

As previously predicted, the agent did find the goal of the maze, but it traversed a much longer average length than that of the default Astar agent.

But unexpectedly, BFS agent had a faster runtime (20 40ms) than the default Astar agent. But this phenomenon was only true when the density is low and the maze is much less complicated. When the maze density was increased to 0.175, BFS was no longer faster than Astar. Instead, it takes a longer time for the BFS to reach the goal.

Without a direction calculated by the heuristic value, whenever the BFS agent hits a block, it is essentially wondering in the maze aimlessly when compared to Astar agent, without an effective plan to reach the goal.

But the BFS agent is not without its merits. When the density is relatively low, the BFS agent can reach the goal at a faster runtime than Astar agent. This phenomenon could also be caused by the heuristic value. Because BFS did not have the heuristic value to consider, it was able to move at a faster runtime than Astar.

Thus repeating BFS is suitable for maze with low density. But when faced with a maze with high density of blocks, repeating Astar would be the better algorithm to find the goal grid.

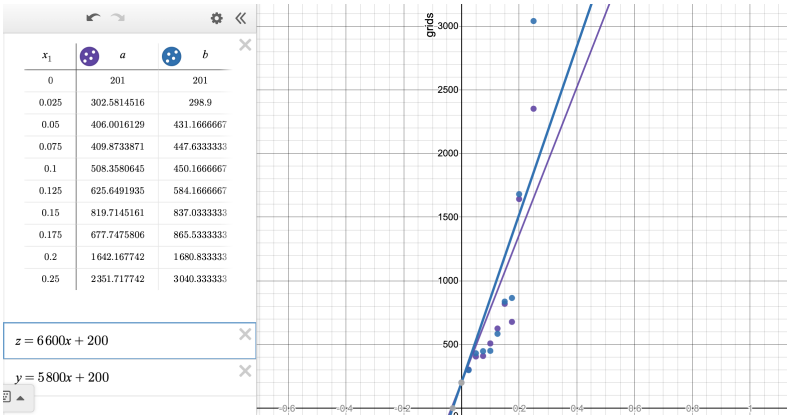

Figure 20: BFS Density vs Avg Trajectory Length (purple for BFS Q6 runtime, blue for BFS Q7 runtime)).

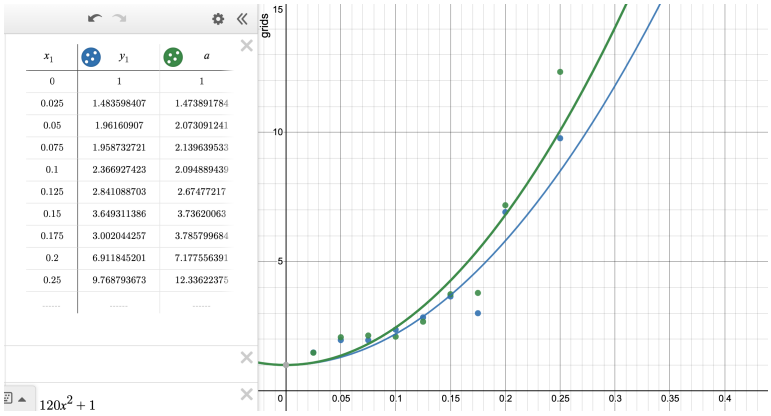

Figure 21: BFS Density vs Avg Trajectory Length Over Length in Final Discovered Gridworld (purple for BFS Q6, blue for BFS Q7)).

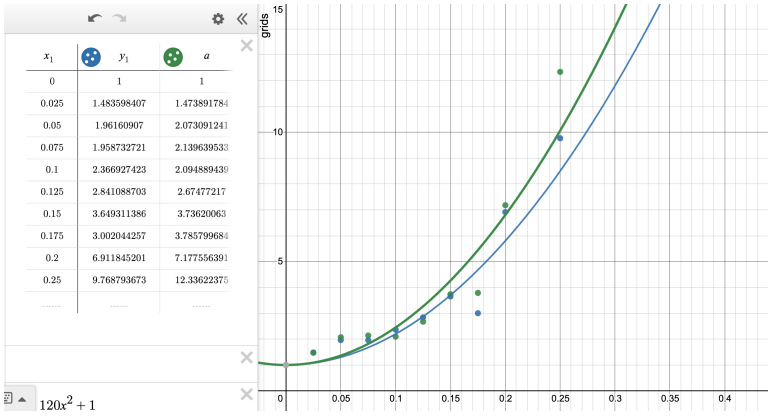

Figure 22: BFS Density vs Avg Trajectory Length Over Length in Final Discovered Gridworld (blue for BFS Q6, green for BFS Q7)).

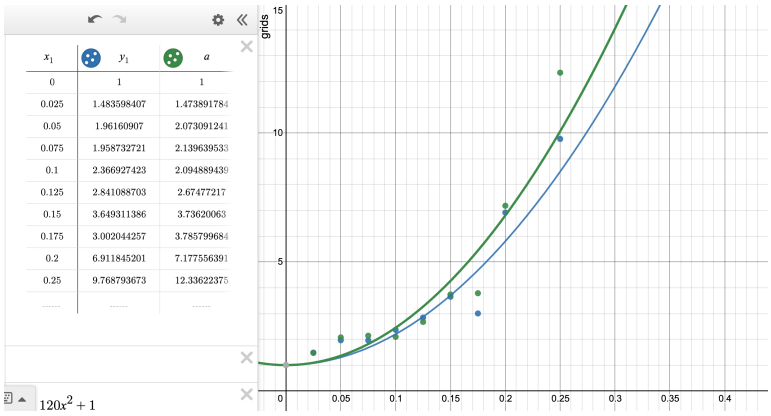

Figure 23: BFS Density vs Avg Trajectory Length Over Length in Final Discovered Gridworld (blue for BFS Q6, green for BFS Q7)).

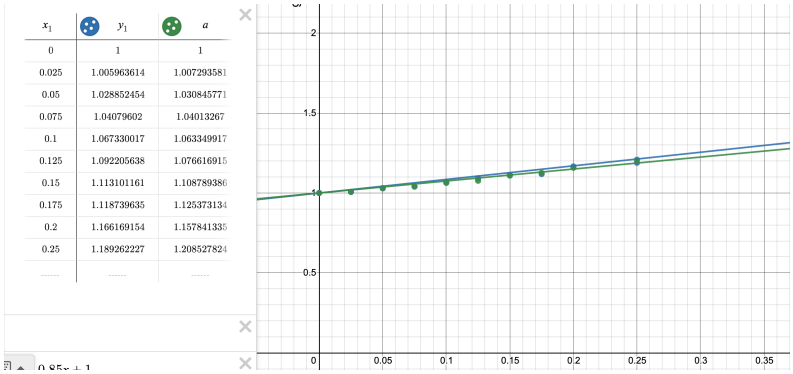

Figure 24: BFS Density vs Average Length of the Shortest Path in the Final Discovered Gridworld Over Average Length of the Shortest Path in the Complete Gridworld (blue for BFS Q6, green for BFS Q7)).

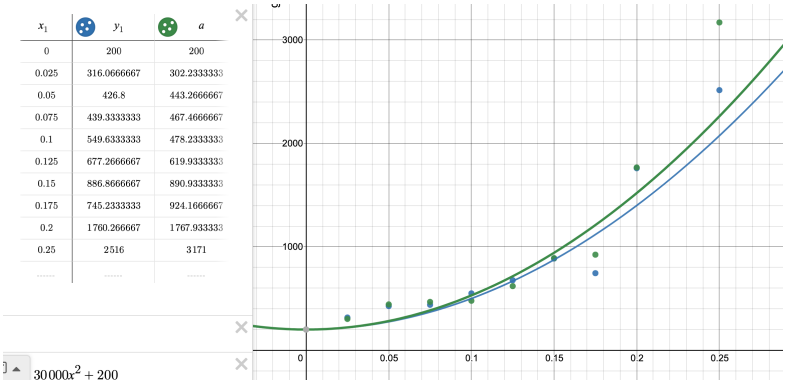

Figure 25: BFS Density vs Avg Number of Cells Processed (blue for BFS Q6, green for BFS Q7)).

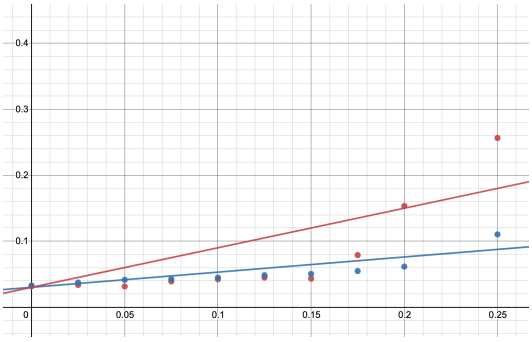

Figure 26: BFS Density vs Avg runtime (red for BFS Q6, blue for Astar Q6)).

Problem 11

APPENDIX

```
from tkinter import *
from random import randint
from matplotlib import pyplot as plot

#size of each grid
grid_size = 10
#dim x dim for maze
size = int(sys.argv[1])
#prob of wall appearance (0~1)
p_wall = getdouble(sys.argv[2])
#MAZE CREATE BEGIN
#set maze of sizeXsize with 'w'
#w: 'path' unvisited
maze= [['w' for _ in range(size)]for _ in range(size)];
def create(ffs, maze1):
    #starting point is green
    color1= 'green'
    for row in range(size):
        for col in range(size):
            if row==0 and col==0 :
                color1 = 'green'
            elif maze1[row][col] == 's':
                color = 'green'
            #white is the accessible grid
            elif maze1[row][col] == 'w':
                color1 = 'White'
            elif maze1[row][col] == 'r':
                color1 = 'red'
            elif maze1[row][col] == 'b':
                color1 = 'blue'
            #P is the wall
            elif maze1[row][col] == 'P':
                color1 = 'black'
            # 'g' is the final path
            elif maze1[row][col] == 'g':
                color1 = 'yellow'
            draw(row, col, color1, ffs)
#assign the maze grids' color
#drawout the maze with assigned color
def draw(row, col, color, ffs):
    x1 = col * grid_size
    x2 = x1 + grid_size
    y1 = row * grid_size
    y2 = y1 + grid_size
    ffs.create_rectangle(x1, y1, x2, y2, fill=color)
#initialize the maze (x,y), make all initial grids
#wall set by p_wall comparison
#set up start and goal
def createMaze():
    for row in range(size):
        for col in range(size):
            s = randint(1,100)
            p = p_wall * 100
            if s<=p:
                maze[row][col] = 'P'
    maze[0][0]='s'
    maze[size-1][size-1] = 'r'
def display_maze(temp_maze):
    window = Tk()
    window.title('Voyage into Unknown')
    canvas_size = grid_size * size
    ffs = Canvas(window, width=canvas_size, height=canvas_size, bg='grey')
    ffs.pack()
    create(ffs, temp_maze)
    return window
#MAZE CREATE END

#A* function begins
# the data structure for A*
import heapq
# for heuristic
import math
#for problem 8
import time
from matplotlib import pyplot as plt
#Node class for the maze grids
class Node:
    #self, current grid
    #x,y coordinates
    #parent, previous grid
    def __init__(self, row, col, parent):
        self.row = row
        self.col = col
        self.parent = parent
        #heuristic
        self.h = 0
        #current cost
        self.cost = 0
    # set parent of node
    def setParent(self, new_parent):
        self.parent = new_parent
    #return parent
    def getParent(self):
        return self.parent
    #return coordinate of node
    def getCord(self):
        return [self.row, self.col]
    #set heuristic in A* algo.
    def setH(self, h):
        self.h = h
    #get heuristic of curr node
    def getH(self):
        return self.h
    # set the actual cost of node
    def setCost(self, cost):
        self.cost = cost
    #return current cost of node
    def getCost(self):
        return self.cost
    # return the comparison of h value between the curr and new node
    def __lt__(self, other):
        return self.h < other.h
# for repeated A* algorithm
# if start_node, ignore the wall beside it and continue running
# if wall_found False, we continue running A*
# if wall_found, we stop A*, backtrack what we have, and start again
def wall_hit(flag1):
    if flag1 == 1:
        # print('wall is hit.')
        return True
#function to check still in maze
#check if shortest path or bad path or visited
#will check if it is wall, if yes, add to temp_walls
# if the agent does hit one of the walls, stop iteration, and add temp walls to w
# 0=True, 1= wall_possible, 2 = out of maze, 3 = visited, 4= wall
def sixMaze(row, col, wall_list, temp_walls, temp_visited):
    #must be >= 0
    if  row< 0 or col < 0:
        #print(row, col, '<0')
        return 2
    #must be < size (not <=)
    if row >= size or col >= size:
        #print(row, col, '>= size')
        return 2
    cord = [row,col]
    # check if it is wall
    if cord == wall_list:
        return 4
    # already in wall_list
    if [row,col] in wall_list:
        #print('found in wall list: ', [row, col])
        return 4
    if [row,col] == temp_walls:
        return 4
    # if already added
    if [row, col] in temp_wall:
        return 4
    #check if the path was visited by current iteration
    if cord == temp_visited:
        print('wall grid found in wall_list, skipping...')
        return 3, temp_walls
    if [row,col] in temp_visited:
        return 3
    # need to be added to temp_wall list
    if maze_copy[row][col] == 'P':
        return 1
    #check if good path as in final grid
    #if false then change to checking grid
    if maze_copy[row][col] != 'g':
        maze_copy[row][col] = 'c'

    return 0
# for finding the shortest path based on information given by repeating forward A
# it would remove inputted coords from a temp path while constructing the heap
# similar to the path_Astar, but used to check for the shortest path for current g
# for f= G + H
def check_path(s, g, type_h, gridworld):
    temp_world = gridworld
    # set goal coor
    g_row = g[0]
    g_col = g[1]
    fringe_heap = []
    visited = []
    start_node = Node(s[0], s[1], Node)
    h_val = h_function(type_h, s[0], g_row, s[1], g_col)
    start_node.setH(h_val)
    heapq.heappush(fringe_heap, start_node)
    while fringe_heap:
        curr_node = heapq.heappop(fringe_heap)
        #print('curr_node: ', curr_node.getCord())
        visited.append(curr_node)
        curr_cord = curr_node.getCord()
```

```
        curr_row = curr_cord[0]
        curr_col = curr_cord[1]
        cost = curr_node.getCost() + 1
        if curr_cord == g:
            return visited
        up = curr_row-1
        down = curr_row + 1
        right = curr_col +1
        left = curr_col - 1
        if [up, curr_col] in temp_world:
            temp_h = h_function(type_h, up, g_row, curr_col, g_col)
            new_node = Node(up, curr_col, curr_node)
            new_node.setCost(cost)
            new_h = temp_h + cost
            new_node.setH(new_h)
            heapq.heappush(fringe_heap, new_node)
            temp_world.remove(new_node.getCord())
        if [down, curr_col] in temp_world:
            temp_h = h_function(type_h, down, g_row, curr_col, g_col)
            new_node = Node(down, curr_col, curr_node)
            new_node.setCost(cost)
            new_h = temp_h + cost
            new_node.setH(new_h)
            heapq.heappush(fringe_heap, new_node)
            temp_world.remove(new_node.getCord())
        if [curr_row, right] in temp_world:
            temp_h = h_function(type_h, curr_row, g_row, right, g_col)
            new_node = Node(curr_row, right, curr_node)
            new_node.setCost(cost)
            new_h = temp_h + cost
            new_node.setH(new_h)
            heapq.heappush(fringe_heap, new_node)
            temp_world.remove(new_node.getCord())
        if [curr_row, left] in temp_world:
            temp_h = h_function(type_h, curr_row, g_row, left, g_col)
            new_node = Node(curr_row, left, curr_node)
            new_node.setCost(cost)
            new_h = temp_h + cost
            new_node.setH(new_h)
            heapq.heappush(fringe_heap, new_node)
            temp_world.remove(new_node.getCord())
    return []
# the heuristics
def euclidean(x_1, x_2, y_1, y_2):
    distance = ( ( ((x_1 - x_2) ** 2) + ((y_1 - y_2) ** 2) )** 0.5)
    return distance
def manhattan(x_1, x_2, y_1, y_2):
    distance = (abs(x_1 - x_2) + abs(y_1 - y_2))
    return distance
def chebyshev(x_1, x_2, y_1, y_2):
    distance = max( abs(x_1 - x_2), abs(y_1 - y_2) )
    return distance
# shortcut for the target heuristic
# h_formula = E, M, C
def h_function(h_formula, curr_row, g_row, curr_col, g_col):
    distance = 0
    if h_formula == 'E':
        distance = euclidean(curr_row, g_row, curr_col, g_col)
    elif h_formula == 'M':
        distance = manhattan(curr_row, g_row, curr_col, g_col)
    elif h_formula == 'C':
        distance = chebyshev(curr_row, g_row, curr_col, g_col)
    return distance
# calcualte g
def create_g(type_h, curr_row, curr_col):
    temp_path = check_path([0,0], [curr_row, curr_col], type_h, visited_list1)
    length = 0
    g = 0
    if len(temp_path) != 0:
        try:
            up = temp_path[len(temp_path)-1].getParent()
            while up.getCord() != [0,0]:
                up = up.getParent()
                temp_path.append(up.getCord())
            g = len(temp_path)
        except:
            return g
    return g
# compare h_val
def compare_h(up, down, right, left):
    h_order = [up, down, right, left]
    #sort by h_value
    h_order.sort()
    h_order[:-1]
    return h_order[0]
# function that runs 1) inMaze() [DO THIS BEFORE FUNCT INSTEAD]
# 2.1) calc cost, store new cost into node
# 2.2) calc heuristic, store h_val into node
# 3) return node
def move_robot(curr_row, g_row, curr_col, g_col, curr_node, h_type):
    new_node = Node(curr_row, curr_col, curr_node)
    g = create_g(h_type, curr_row, curr_col)
    new_h = h_function(h_type, curr_row, g_row, curr_col, g_col)
    new_cost = g + new_h
    new_node.setCost(new_cost)
    return new_node
# try to use this as default
# problem 6 Astar with line of view
# can see if it is block grid as neighbors, no need to hit
# ------------------------------
def six_AStar(path, fringe_heap, s, g, type_h, wall_list, counter):
    g_row = g[0]
    g_col = g[1]
    start_node = Node(s[0], s[1], Node)
    distance = h_function(type_h, s[0], g_row, s[1], g_col)
    start_node.setH(distance)
    heapq.heappush(fringe_heap, start_node)
    wall_met = False
    temp_walls = []
temp_visited = []
    while fringe_heap and wall_met == False:
        curr_node = heapq.heappop(fringe_heap)
        counter = counter + 1
        temp_visited.append(curr_node.getCord())
        # then record current node coordinate
        curr_cord = curr_node.getCord()
        curr_row = curr_cord[0]
        curr_col = curr_cord[1]
        parent_node = curr_node.getParent()

        # if not goal, check if moveable in direction
        # directions: up, down, right, left
        # if applicable create node with move_robot, push node into heap

        # direction cords
        up = curr_row - 1
        down = curr_row + 1
        right = curr_col + 1
        left = curr_col - 1
        #temp_node = Node(s[0], s[1], Node)
        new_node = start_node

        # change wall_found in canmove, not return false
        # instead change the wall_found
        # input init values, we would want the grid closest to goal (h_value)
        up_h = 9999
        down_h = 9999
        right_h = 9999
        left_h = 9999
        #used to count paths available
        no_path = 0
        # 0 is pass, 2 is out of maze, 1 is wall hit (not applied), 3 is visited g
        flag1 = sixMaze(up, curr_col, wall_list, temp_walls, temp_visited)
        if flag1 != 2:
            up_h = h_function(type_h, up, g_row, curr_col, g_col)
        # we do not wait for the agent to hit the wall
        # due to the field of sight, we can extend the wall to temp_walls
        if flag1 == 1:
            temp_walls.append([up, curr_col])
            #flag1 = 4
        if flag1 == 4 or flag1 == 3 or flag1 == 2:
            # wall hit
            up_h = 9999
            no_path = no_path+1

        flag2 = sixMaze(down, curr_col, wall_list, temp_walls, temp_visited)
        if flag2 != 2:
            down_h = h_function(type_h, down, g_row, curr_col, g_col)
        if flag2 == 1:
            temp_walls.append([down, curr_col])
            #flag2 =4
        if flag2 == 4 or flag2 == 3 or flag2 == 2:
            # wall hit
            down_h = 9999
            no_path = no_path + 1

        flag3 = sixMaze(curr_row, right, wall_list, temp_walls, temp_visited)
        if flag3 != 2:
            right_h = h_function(type_h, curr_row, g_row, right, g_col)
        if flag3 == 1:
            temp_walls.append([curr_row, right])
        if flag3 == 4 or flag3 == 3 or flag3 == 2:
            # wall hit
            right_h = 9999
            no_path = no_path +1

        flag4 = sixMaze(curr_row, left, wall_list, temp_walls, temp_visited)
        if flag4 != 2:
            left_h = h_function(type_h, curr_row, g_row, left, g_col)
        if flag4 == 1:
            temp_walls.append([curr_row, left])
        if flag4 == 4 or flag4 == 3 or flag4 == 2:
            # wall hit
            left_h = 9999
            no_path = no_path +1
        #no direction left
        # dead -end
        if no_path == 4:
            if parent_node == []:
                print( 'Maze Unsolvable.')
            try:
                temp_visited.append(parent_node.getCord())
                heapq.heappush(path, parent_node)
                #print('last node: ', temp_visited[-1])
            except:
                print('Maze is not Solvable')
                return -99 , temp_walls, temp_visited, counter
```

```
            temp_walls.append( curr_node.getCord() )
            return len(path), temp_walls, temp_visited, counter
        direction_h = compare_h(up_h, down_h, right_h, left_h)
        # if wall blocking path, re-plan
        # up
        if direction_h == up_h:
            # hit the block on current iteration
            if wall_hit(flag1):
                wall_met = True
                return len(path), temp_walls, temp_visited, counter
            new_node = move_robot(up, g_row, curr_col, g_col, curr_node, type_h)
            temp_visited.append([up, curr_col])
            heapq.heappush(path, curr_node)
            heapq.heappush(fringe_heap, new_node)
        # down
        elif direction_h == down_h:
            if wall_hit(flag2):
                wall_met = True
                return len(path), temp_walls, temp_visited, counter
            new_node = move_robot(down, g_row, curr_col, g_col, curr_node, type_h)
            temp_visited.append([down, curr_col])
            heapq.heappush(path, curr_node)
            heapq.heappush(fringe_heap, new_node)
        # right
        elif direction_h == right_h:
            if wall_hit(flag3):
                wall_met = True
                return len(path), temp_walls, temp_visited, counter
            new_node = move_robot(curr_row, g_row, right, g_col, curr_node, type_h)
            temp_visited.append([curr_row, right])
            heapq.heappush(path, curr_node)
            heapq.heappush(fringe_heap, new_node)
        # left
        elif direction_h == left_h:
            if wall_hit(flag4):
                wall_met = True
                return len(path), temp_walls, temp_visited, counter
            new_node = move_robot(curr_row, g_row, left, g_col, curr_node, type_h)
            temp_visited.append([curr_row, left])
            heapq.heappush(path, curr_node)
            heapq.heappush(fringe_heap, new_node)
        if new_node.getCord() == g:
            heapq.heappush(path, new_node)
            print('goal reached')
            return len(path), temp_walls, temp_visited, counter
            break
        if wall_met == True:
            print('wall coordinates: ', temp_walls)
            break
    return -99, temp_walls, temp_visited, counter

#setup the maze
createMaze()
maze_copy = [x[:] for x in maze]
# set start coor
s_grid = [0, 0]
g_grid = [size-1, size-1 ]
# visited wall coords
wall_list = []
# see if program still runnable
runnable = True
# new additions of path node will be added
while runnable:
    maze_copy = [x[:] for x in maze]
    all_maze_copy = [x[:] for x in maze_copy]
    window = display_maze(maze_copy)
    print('Please choose a heuristic formula: E/M/C')
    print('Euclidean (E), Manhattan (M), Chebyshev (C)')
    type_h = input()
    # do not forget the mainloop at end
    window.destroy()
    # timer
    time_a = time.time()
    # walls found in iteration
    temp_wall = []
    #closed heap list
    path = []
    #all grids visited in iteration
    temp_visited = []
    # total grids visited
    # used for fast cord search
    visited_list= [[0,0]]
    # total grids visited, used for best path info in Astar
    visited_list1 = [[0,0]]
    # opened lsit
    fringe_heap = []
    # flag for pass/fail maze
    passed = 0
    # flag for unsolvable maze
    counter = 0
    # number of grids popped
    grid_counter = 0
    # while goal not found and while maze runnable
    # continue to iterate, until goal reached
    while visited_list[-1] != g_grid and runnable == True:
        path_len, temp_wall, temp_visited, grid_counter =
        six_AStar(path, fringe_heap, visited_list[-1], g_grid,
         type_h, wall_list, grid_counter)
        if path_len < 0:
            runnable = False
                                                    break
        # add visited grids and found walls
        visited_list.extend(temp_visited)
        visited_list1.extend(temp_visited)
        wall_list.extend( temp_wall )
        # if goal grid is added to visited_list
        if visited_list[-1] == g_grid:
            total_time = time.time() - time_a
            # draw map
            final_path = visited_list
            for i in range(len(final_path)):
                curr_location = final_path[i]
                # print('curr_location: ', curr_location)
                maze_copy[curr_location[0]][curr_location[1]] = 'g'
            colored = 0
            for x in range(len(maze_copy)):
                for y in range(len(maze_copy[x])):
                    for i in range(len(final_path)):
                        curr_location = final_path[i]
                        if x == curr_location[0] and
                        y == curr_location[1]:
                            maze_copy[x][y] = 'g'
                            colored = 1
                    if colored == 1:
                        colored = 0
                        continue
                    if maze_copy[x][y] != 'P':
                        maze_copy[x][y] = 'w'
            maze_copy[0][0] = 's'
            maze_copy[size-1][size-1] = 'g'
            print('Grids traveled: ', path_len)
            print('Number of grids popped: ', grid_counter)
            print('Astar: %s seconds used'% (total_time))
            print('\n')
            window2 = display_maze(maze_copy)
            window2.mainloop()
            runnable = False
            passed = 1
            #maze completed, break loop
            break
        counter = counter + 1
    if runnable == False and passed == 0:
        print('No path to Goal.')
```