AI - CS520

Project1
Maze Runner

Group members:
Wenhao Luo, Bo Mao, Runfeng Xu

1. Abstraction:

This project is intended as an exploration of various search algorithms, both in the traditional application of path planning, and more abstractly in the construction and design of complex objects.

2. Introduction:

This project is divided into 5 different parts with the fifth part is a bonus. Due the time issue, we left the part 5 to be blank, and focusing much more on the previous four parts. In this report, we generate many figures to support some of our assumptions and to show the functions of our coding part. We copy and paste our original code into this report at the ending part instead of generating an URL.

——**2.1**) The first part is about implementing four algorithms which are BFS, DFS, and A*(Manhattan & Euclidean) to find the path from start to goal. In this part, we assume the start and goal node are at left top and right bottom relatively. (page3)

——**2.2**) In this part, it's more about using the code from 2.1) to analyze the data we have generated. (page3)

——**2.3**) Making a harder maze is the requirement of this part. We choose to use beam search algorithm as our local search algorithm to implement this part. Randomly choosing two points on the maze and keeping generating the harder maze after comparing the total of fringe, history(node expanded), and the path in different weight. (Total = 0.5*path + 0.1*history + 0.4*fringe) (page13)

——**2.4**) The last part we did is about Thinning A*. We generated lots of data to support the comparison required. (page14)

——**2.1**) Environments and Algorithms

The code will be shown at the end of this report.

To explain the code, we write different code separately. For Bo Mao and Runfeng Xu, they considering the wall as value 1, empty as value 0; however, for Wenhao Luo, he considering empty as value 1, wall as value 0 since using python to draw the figures, the value 0 represents color black, which is much easier for visualization after.

——**2.2**) Analysis and Comparison

2.2.1\\

The conditions we considered have four parts: the resources we have, the cost of time, maze solvability and good visualization

i) The first restriction we want to consider is the nature of the algorithm and the resources we have. Basically, DFS may frequently run out of memory when dim = 1000, p = 0.1 after our initial test. Hence, this is a upper bound for our later discussion, we will not expand the dimension of maze to that large. And we will try not use low probability since this may consume large resources and the maze can be easily solved (which is not interesting).

ii) The second part is the cost of time, since we run the test usually larger than 100 times, in this way, for every process, we hope the running time can be restricted to the level of 1 second. In this part we use A star algorithm to run the test. After several test, when dim = 500, p=0.2, the generated maze can be created and searched very close to 1 second.

Table 1. Test of A star Manhattan in maze with dim = 500, probability = 0.2

Searched nodes	12292	115215	125037	133926	120324
shortest path	999	999	999	999	999
time consumed/s	0.8936	0.9174	0.97247	1.0445	0.9016

iii) The third part we considered is the solvability. We hope the maze has enough walls, so it will not be too easily and cost to much time on our algorithm. But, as the number of blocks goes high, the path we need to search become less, which may reduce the difficulty drop down. More blocks will also easily trigger unsolvable mazes. Initially, to balance the number of blocks and solvability, we choice p = 0.2 to run the test.

iv) The final part we consider is visualization. Since we print the maze and paths with matplotlib package. When we want to draw a maze with dim>75, the pixels of our computer screen can hardly print the grids. So, for a good visualization, we may use a maze with dim = 75, and maze with dim = 500 for some pressure test.

for A star, we think that 500 is a proper dim. Because it takes 188 secs to test p =[0,1,0.05] and 100 tests for each value of P. Less than 100 ,like 50 or 10, the maze become relatively too easy, but increase dim to 500 or more. It takes averagely 5 secs to solve a maze which is too long for multiple repetition.

```
C:\Users\Runfe\AppData\Local\Programs\Python\Python37-32\python.exe C:/Users/Runfe/Desktop/rutgers/ai520/pj11/maze.py
finish  0.0
29.828125
finish  0.05
55.703125
finish  0.1
78.484375
finish  0.15
98.40625
finish  0.2
115.546875
finish  0.25
128.9375
finish  0.3
139.90625
finish  0.35
148.34375
finish  0.4
153.921875
finish  0.45
157.984375
finish  0.5
161.140625
finish  0.55
164.34375
finish  0.6
167.4375
finish  0.65
170.546875
finish  0.7
173.671875
finish  0.75
176.75
finish  0.8
180.1875
finish  0.85
183.015625
finish  0.9
185.921875
finish  0.95
188.953125

Process finished with exit code 0
```

2.2.2\\

The visualized mazes with dim = 40, p =0.3 are showed below with different algorithm:

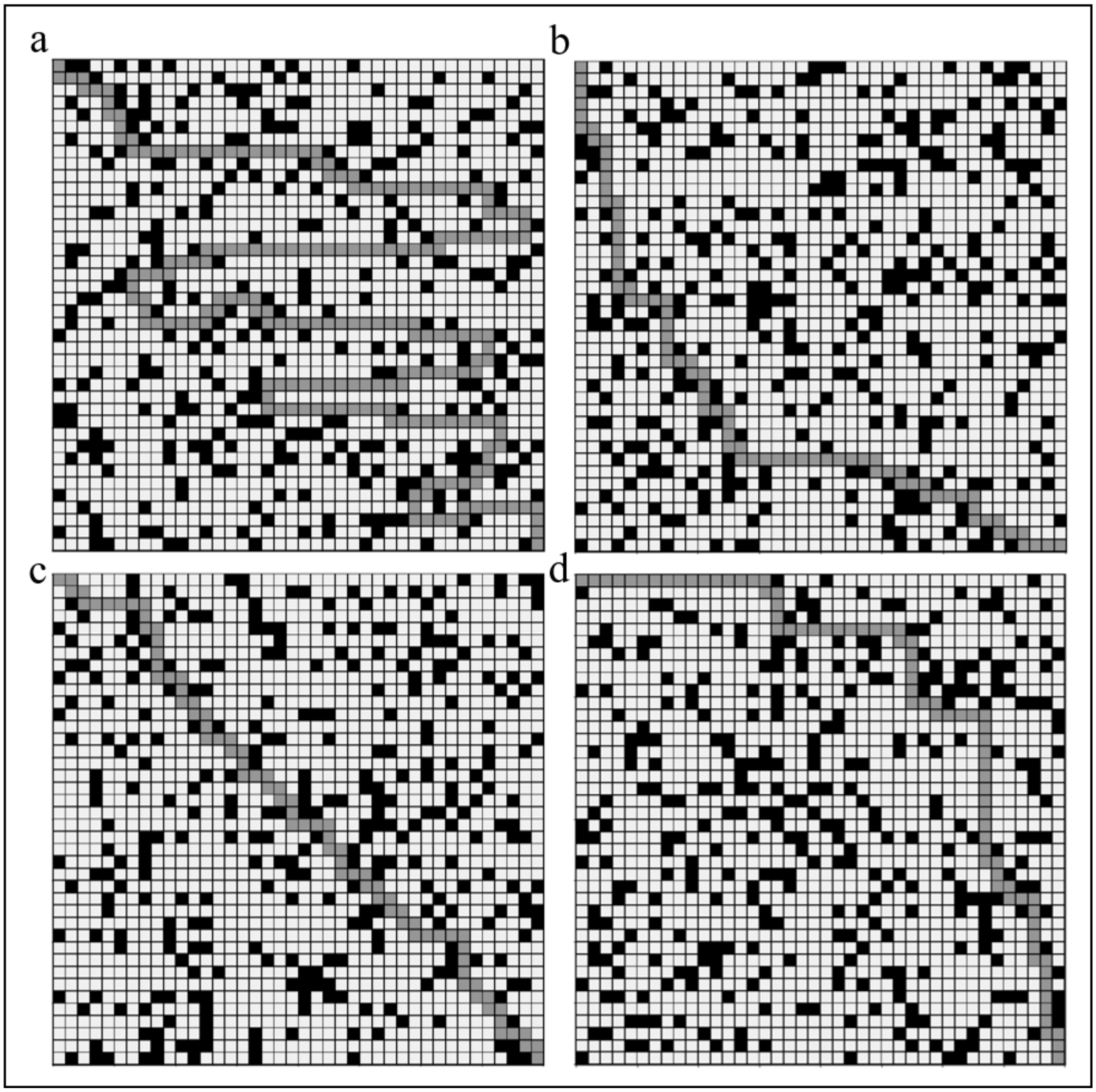
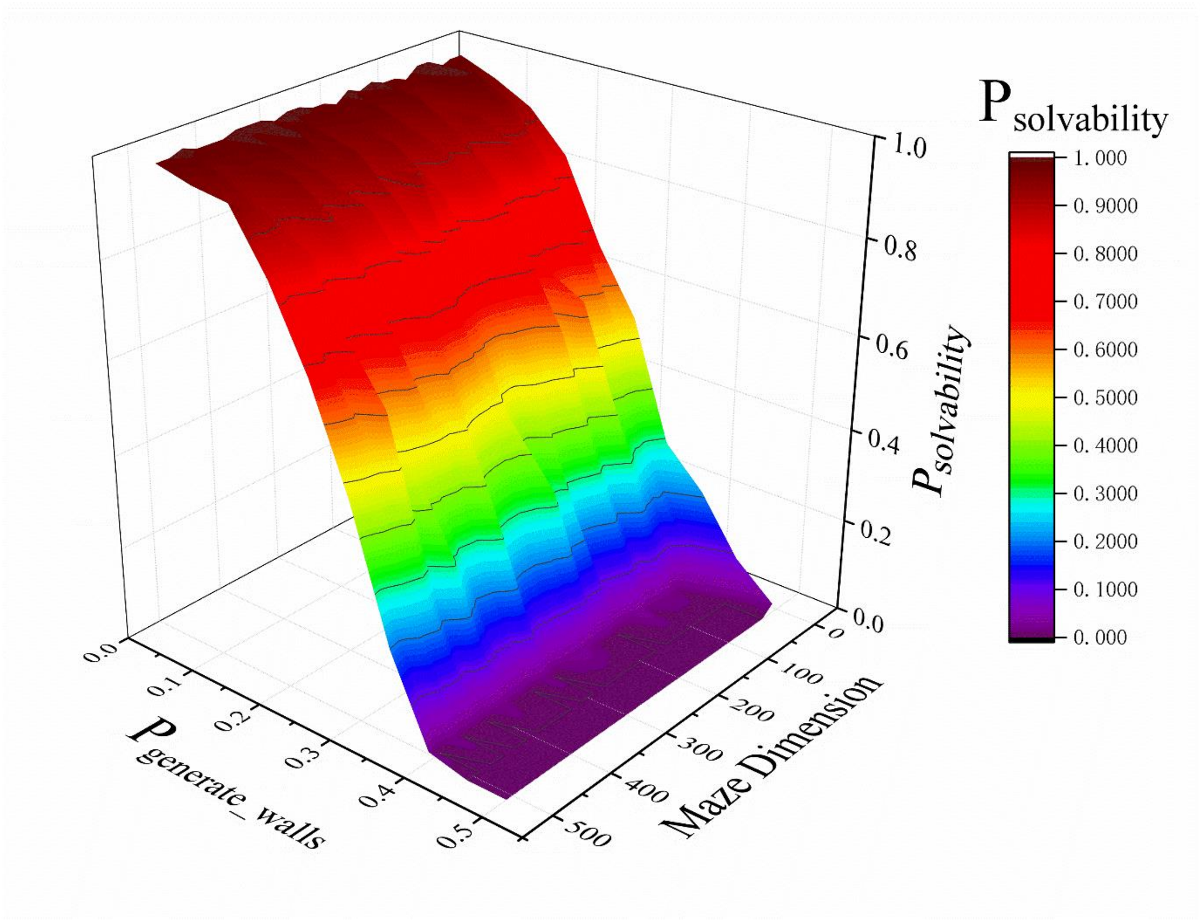


Figure 1. This is the visualized mazes and found paths with a) DFS algorithm (choices for available neighbors are always in the same direction); b) BFS algorithm; c) A star algorithm with Euclidean heuristic; d) A star algorithm with Manhattan heuristic. All the algorithm except DFS worked pretty well. And DFS has natural weakness, we will modify it at later issues.

2.2.3\\

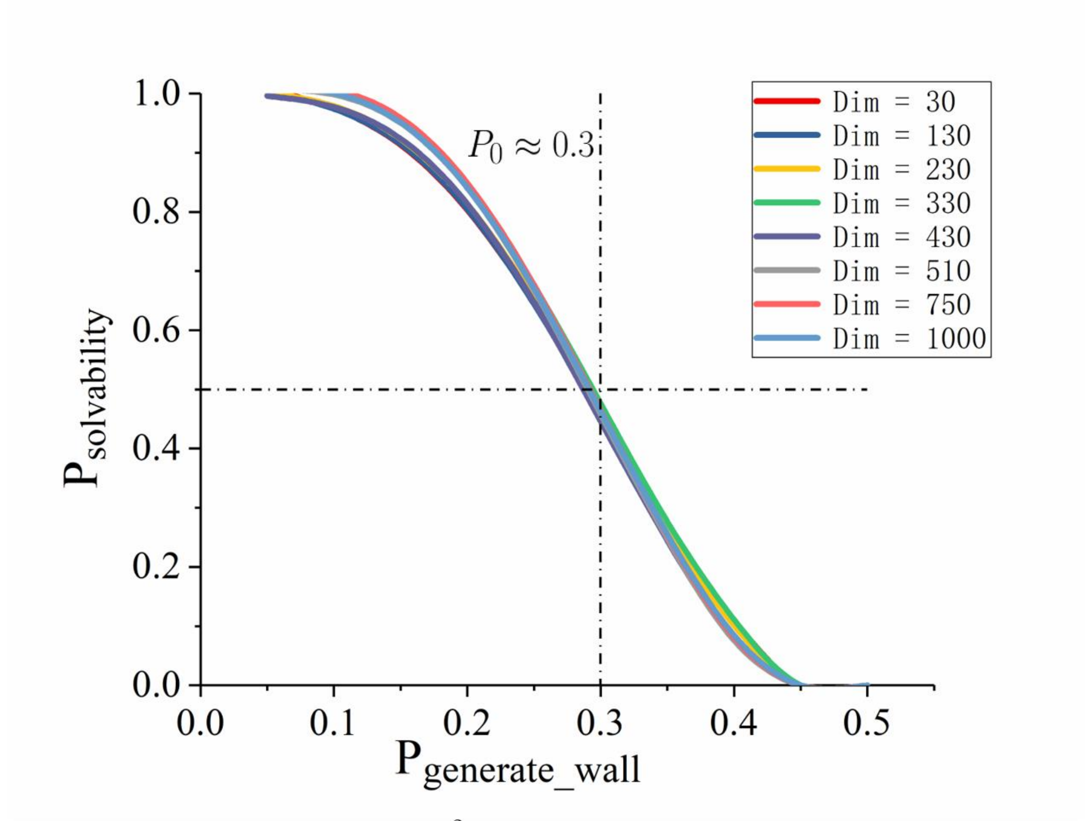
i) For this issue, we generated several mazes with different dimensions (28 different dimensions) at different probability of generating walls (p is from 0.05 to 0.5, for $p > 0.5$, almost all mazes generated are unsolvable). For each pair of metrics (dimension and probability), we processed 100 times randomly and tested their probability of solvability.

Figure 2. The 3D plot for the probability of solvable mazes with different maze dimension (from Dim =10 to Dim = 510) and different probability of generating walls (from 0 to 0.5). It is noticeable that when $P_{generate\_walls} > 0.4$, almost all the mazes are unsolvable (randomly



generating 100 times). And $P_{solvability}$ almost retain on the same value when dimension expanded.

ii) The first conclusion is that when $P_{generate\_wall} > 0.4$, mazes can hardly be solvable. Now let us check whether different dimensions have some influences on the $P_{solvability}$. In figure 3, we picked some points and show the curves of $P_{generate\_wall}$- $P_{solvability}$ at different dimensions.



$$P_{solvability} = -1.666P_{generate\_wall}^2 - 1.775P_{generate\_wall} + 1.169$$

Figure 3. We only show some points in the graph above, we can see that from Dim =30 to Dim=1000, almost all the curves overlapped. We can see that it is very close to 0.3. We then calculated all the data we have, get their averages and fitted them with 2nd order polynomial fitting. The final fitting function is showed on the bottom of the figure. Upon this function (actually we also fitted the function of $P_{solvability}$ versus $P_{generate\_wall}$), we calculate the P_0 is approximately 0.3. When $P_{generate\_wall}$>0.3, the $P_{solvability}$ will below 0.5 and vise versa.

iii) We also plot the figure of simulated P_0 value at different dimensions (see figure 4). From dim = 10 to dim = 1000, the value of P_0 almost remain the same.

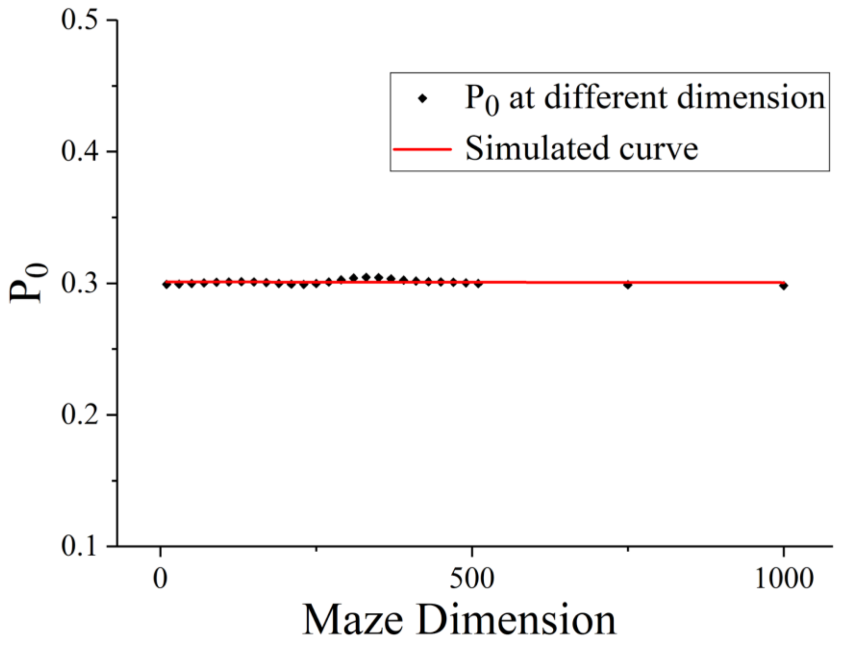


Figure 4. The value of P_0 at different Dimensions with simulated curve. We can see that the value of P_0 almost remain the same. So, as this dimension scale (10-1000), we can say that P_0 is independent with maze dimension.

iv) Reversely, we can also simulate the value of P_0 with expected value of $P_{solvability}$. The simulated formula is below:

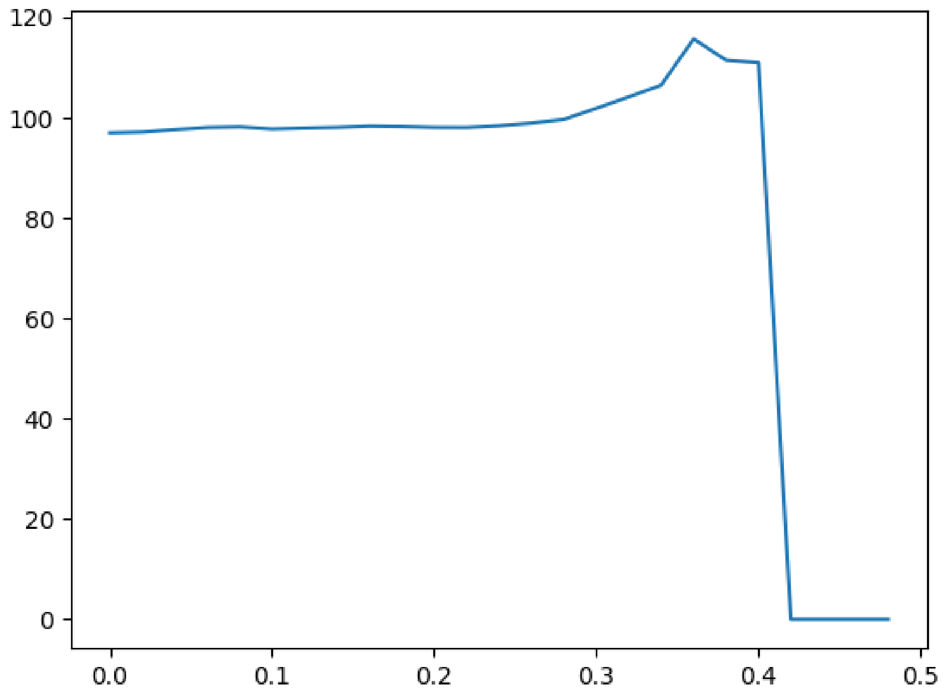
$$P_0 = -0.1528P_{solvability}^2 - 0.2024P_{solvability} + 0.4458$$

Table 2. Different P_0 with different expected $P_{solvability}$

$P_{solvability}$	0.2	0.25	0.3	0.35	0.4	0.45	0.5
P_0	0.4	0.3855	0.371	0.3561	0.34	0.3237	0.3

2.2.4\\

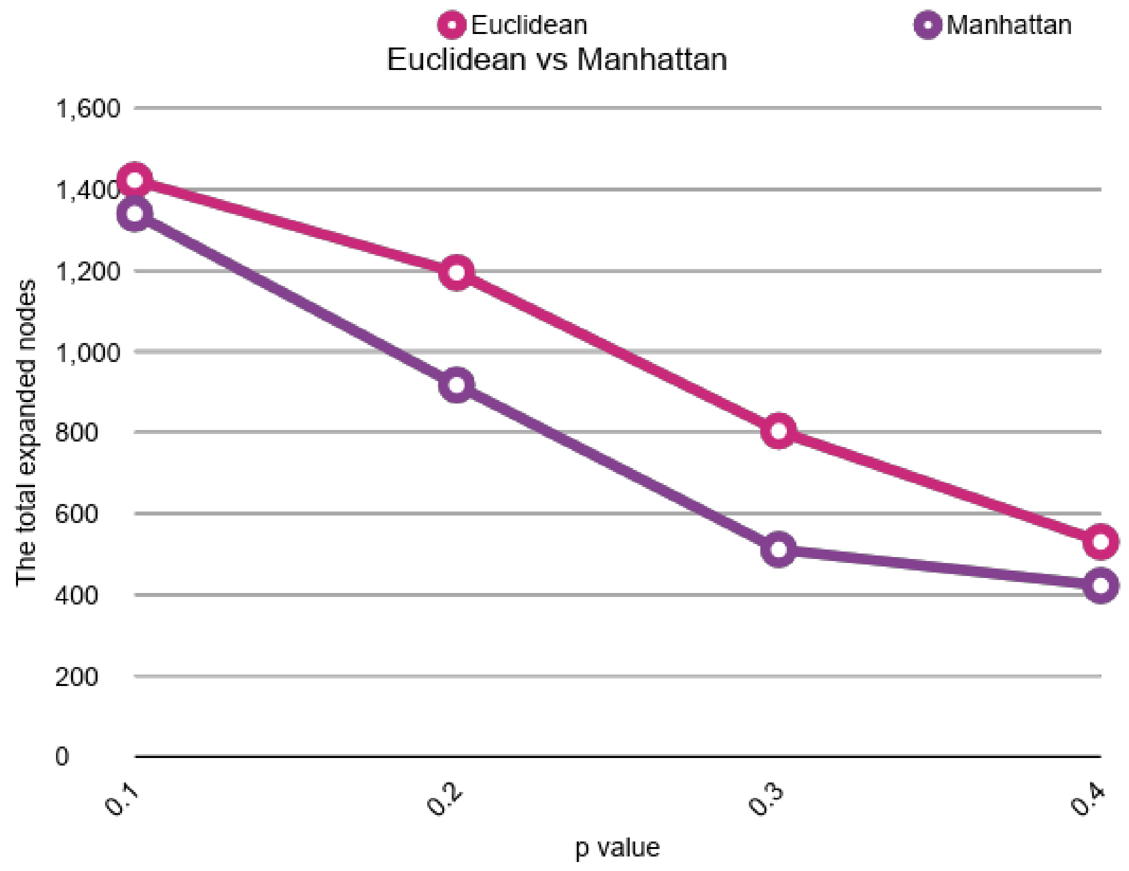
p and shortest path length



2.2.5\\

In the following figure, we compare the Euclidean and Manhattan algorithm with their total expanded nodes with different p value (0.1-0.4). And we initialize the maze as 40*40 for a better visualization.

Figure 5, for this figure, we use 40 as the dim and p from 0.1-0.4 with 0.1 difference for each step data collection. As we can see, the Euclidean is always expanded more nodes than



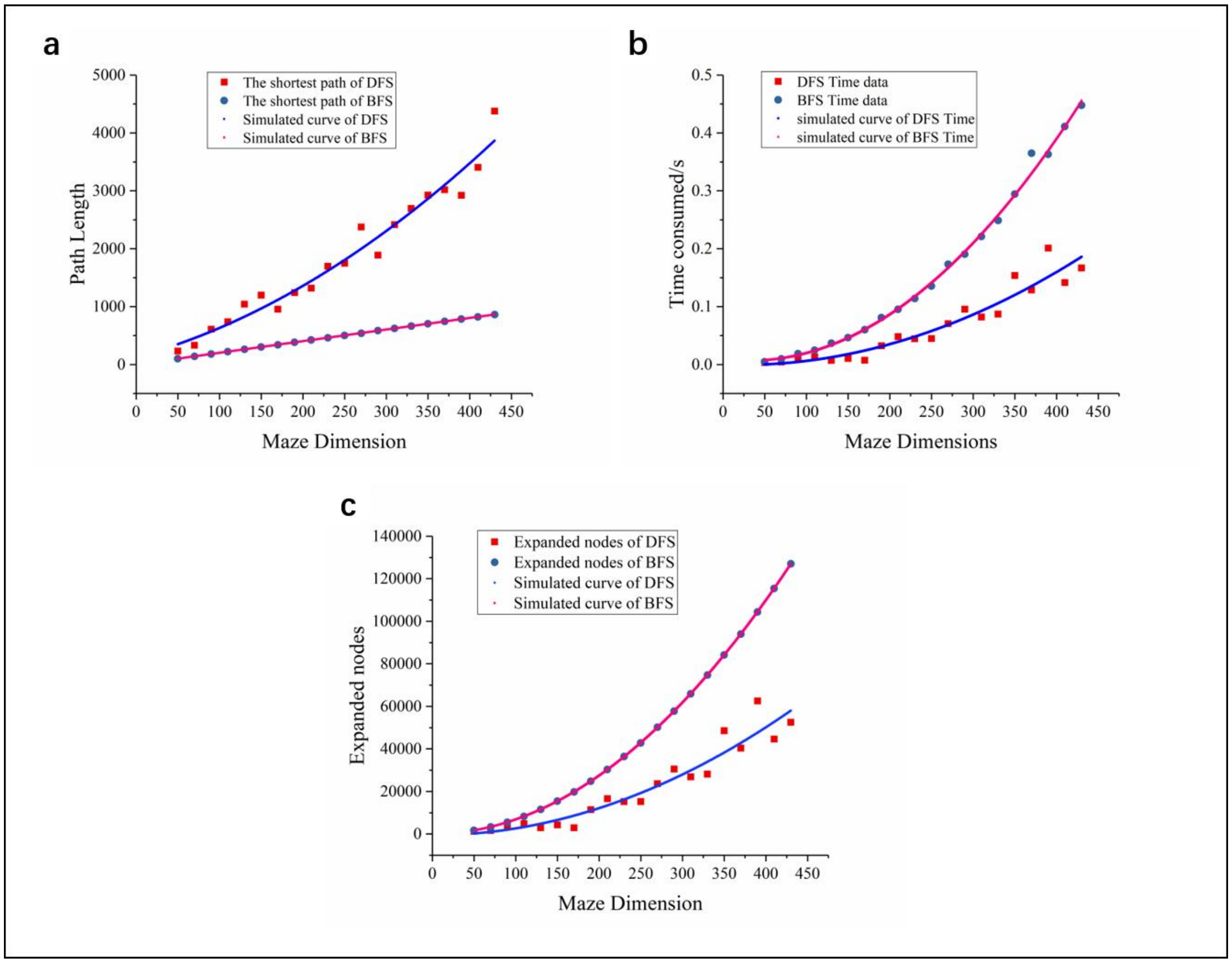
Manhattan from p(0.1-0.4). Additionally, there is a trend when p>0.3 that Euclidean and Manhattan going to intersect with each other in some future(p nearly 0.43). It is very important to mention that when the dim becomes larger like 500 or 1000, the data collection will show the distinction accurately.

2.2.6\\

The answer for this question is yes. To compare them, we ran a same list of mazes with DFS and BFS, and compared their cost time, shortest paths and the number of expanded nodes.

At first, we generated a list of mazes, the dimension scope from Dim =50 to Dim = 430. For each dimension, we generated 5 solvable mazes for the test and calculated their average data corresponding to the maze dimension.

Based on the Figure 6. We can see that the time cost both for DFS and BFS will expand exponentially with the maze dimension. But the time consumed by DFS increases much



faster than that for DFS.

Figure 6. We compared the property of DFS and BFS for **a)** shortest paths, **b)** cost time, **c)** total expanded nodes. We can see that the shortest path founded by BFS is much shorter than paths found by DFS. And the shortest paths found by BFS only increases linearly, which is much better than that of DFS, which increases exponentially. But DFS reached to the end much earlier than that of BFS and the nodes expanded by the DFS is much smaller, so when we trying to find a path, maybe not the shortest, using DFS can be a good trial.

Also, the expanded nodes in DFS algorithm is much less than that of BFS. This is because using DFS, we always expand the nodes that is much deeper. So, when we run it on a randomly generated maze, it is highly possible that DFS can find the way in less trials.

However, for the length of shortest path, DFS is much longer than that of BFS. We can see in the Figure 5a that when maze's dimension goes high, the shortest path found by DFS will explode exponentially, but that for BFS only growth proportionally. Since BFS always expand all the fringes step by step. So, the path found by BFS must be the shortest path in the maze.

In sum, the shortest path found by BFS is much better than DFS, but it wastes much time and expands much more nodes than the DFS.

2.2.7\\

Yes, of course.

However, As we know, in BFS queue is been used and in DFS the stack strategy is been used. Therefore, DFS suppose to be expanded less nodes than the BFS. Additionally, the DFS will run faster than BFS in most of cases with different value of dim.

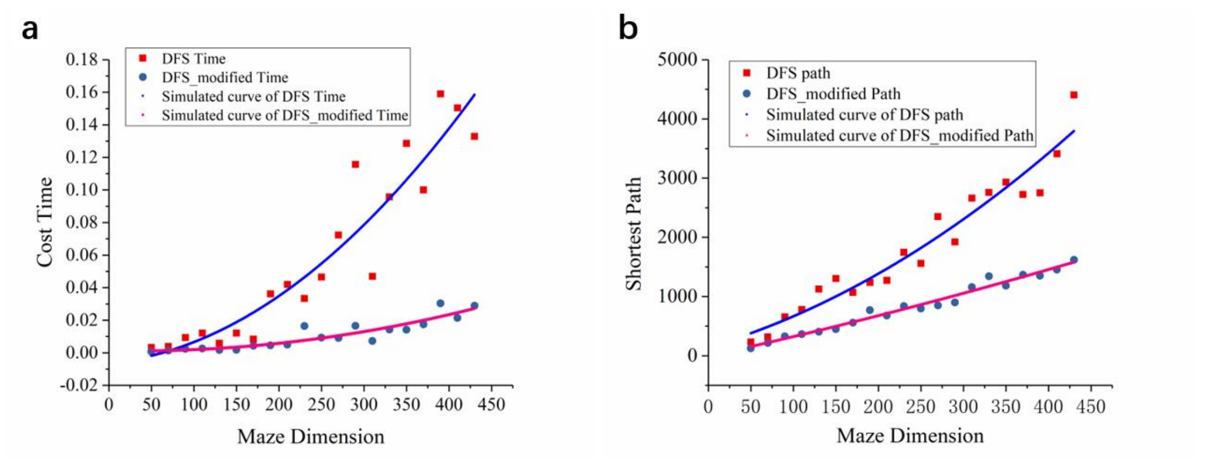
Similarly, the BFS should get relatively more running time than the A* algorithm. However, something surprised us is that A* has more running time than BFS does. This situation is not supposed to be occurred. After our discussion, maybe the reason is insertion of each element into the fringe will take log of n times to sort, and this manipulation is in the deepest loop, so, this is why the A* algorithm in our report is a little bit lower speed than the BFS does.

2.2.8\\

In our original DFS algorithm, at the beginning, we pushed the start point into the stack, then, expanded the start node and see if there are neighbors next to it which are not wall and not already visited; if so, go there and change it status to visited. In this algorithm, it will fetch all the unvisited neighbors. Therefore, in some ways, we can change the algorithm to force it make some trends we like to expand or fetch the neighbors. For example, considering an 10*10 chess board, we have King at left upper corner, and Queen at right lower corner, and some other chess randomly put on the board. We must let the King to find a path to meet the Queen without stepping on other chess. Now, considering another scenario, if the King is not on the left upper corner, then, in this algorithm, the King maybe will fetch the cell of the board on the lefthand side. Therefore, add a condition of choosing the path for the DFS will optimize the path running time. In the new algorithm, I choose the chess to load the neighbors firstly to the righthand side, then if meet any walls or the boarder of the board, go straight down for one step, then select two ways to fetch new neighbor.
1.)If the current node has a wall beside of it on the right, then keep going straight down;

2.)on the other hand, if there is no wall or boarder on the righthand side next to it, going right side
3.)continuing 1) and 2) until the goal node has been found or there is no path to the goal.

In this way, we can easily find that we give the algorithm a priority direction to force it make a decision before fetch to neighbors. It will expand much more less nodes for sure.



——2.3)

In this part, we choose beam search algorithm to be our local search algorithm. In our algorithm, eventually, we first generate 5 different mazes with 50 dim, and calculated the weight=path/dim + history/dim^2 + fringe/dim. The reason why we didn't use the previous formula discussed in Introduction part is because we find that the value of history is much more larger than fringe and path. Therefore, divided history by dim^2 will get a number <1 always. Whatever, the history element in this situation will have small affect on the weight.

Furthermore, after generating the mazes, we copy every initialized mazes for the further works. Then, we compare each weight for one iteration, and print out the maximum weight for the final comparison. Once we get the maximum over 5 different mazes, then let each other mazes do the same manipulations than the one with maximum weight.

Eventually, we can generate the hardest mazes for the given algorithm.

The results agree with our intuition.

Conclusion:

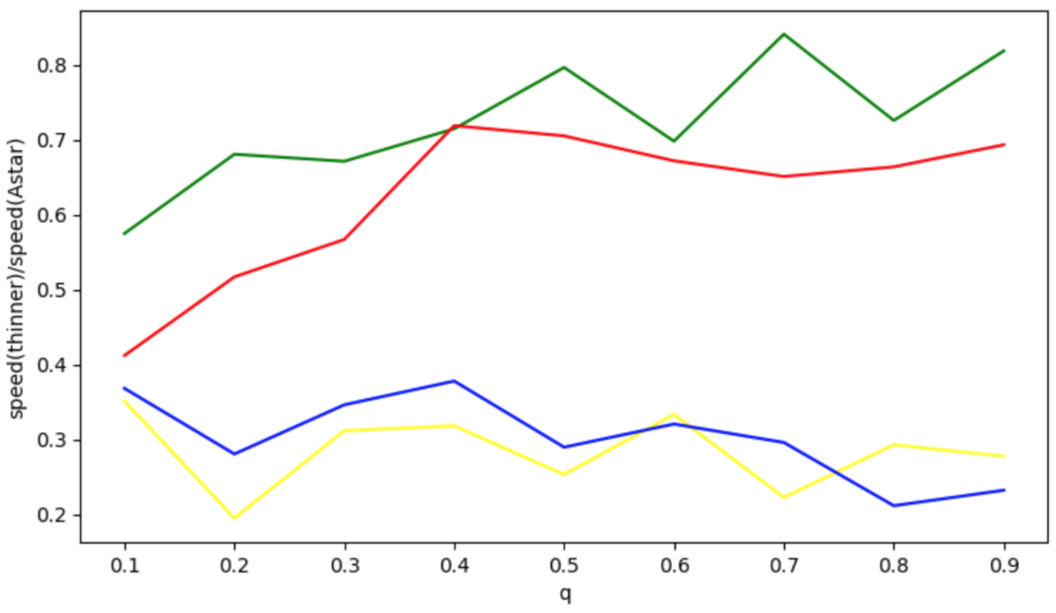
1) In this test, the number of trials for every maze need to be large, up to 1000 is better. If not the map may remain unchanged.

2) In our test, using fringe as the heuristic can usually generate more good result, shortest path for DFS and expanded nodes for A star is not that good.

3) we tested the maze with dim = 50, p = 3.6. 5 mazes and 1000 trials for each trail

4) The largest heuristic can increase from 2.145 up to 3.6

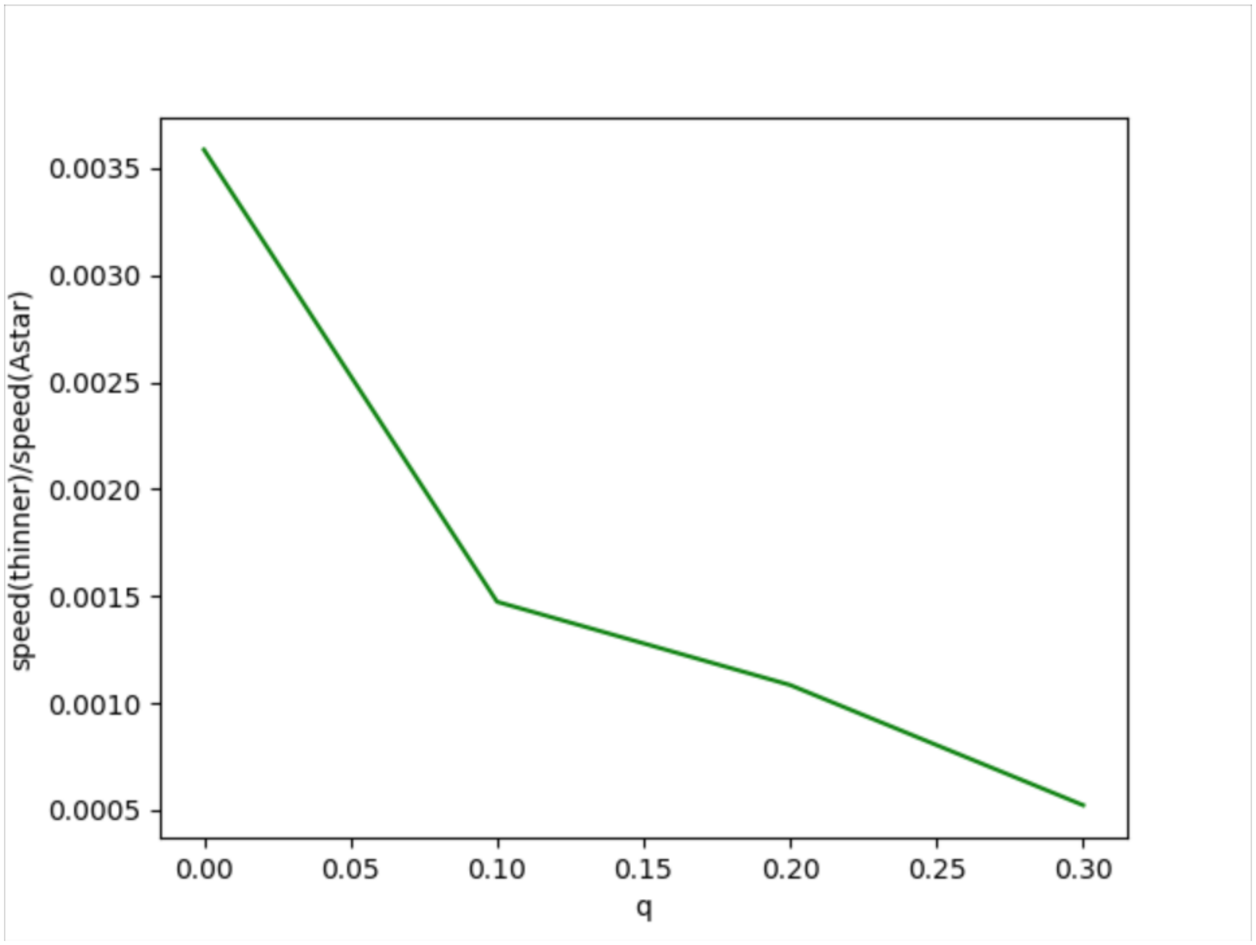
——2.4)



This figure represents dim= 100, 200, 500, 1000 relatively. And using the new A*/ old A*, we can find the fraction is always less than one, therefore, we can get a conclusion that new A* is always slower than the old A*.

Algorithm:

1.) new h() is the shortest distance from node to thinner maze, whenever, the new A* expanded one node, pop() a shortest path point from thinner maze, and its neighbor's h().
2.) for every node in the maze, find the shortest path length from this point to goal in the thinner maze as its h(), if there is no path, then not count this point to queue.



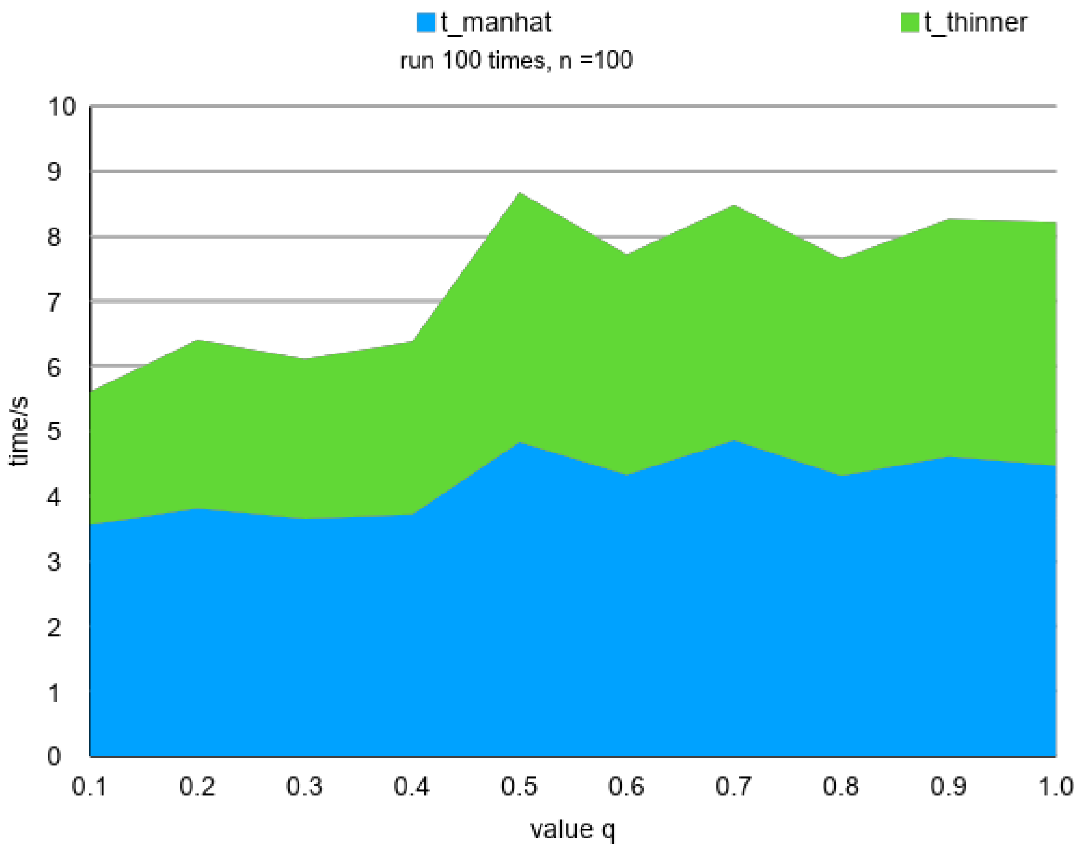
The above figure is when dim = 100. Since we found that when q>=0.4, the new A* will have a long running time, it's hard for us to make data analysis and visualization as well. Therefore, this figure only represent the q in (0 -- 0.3)

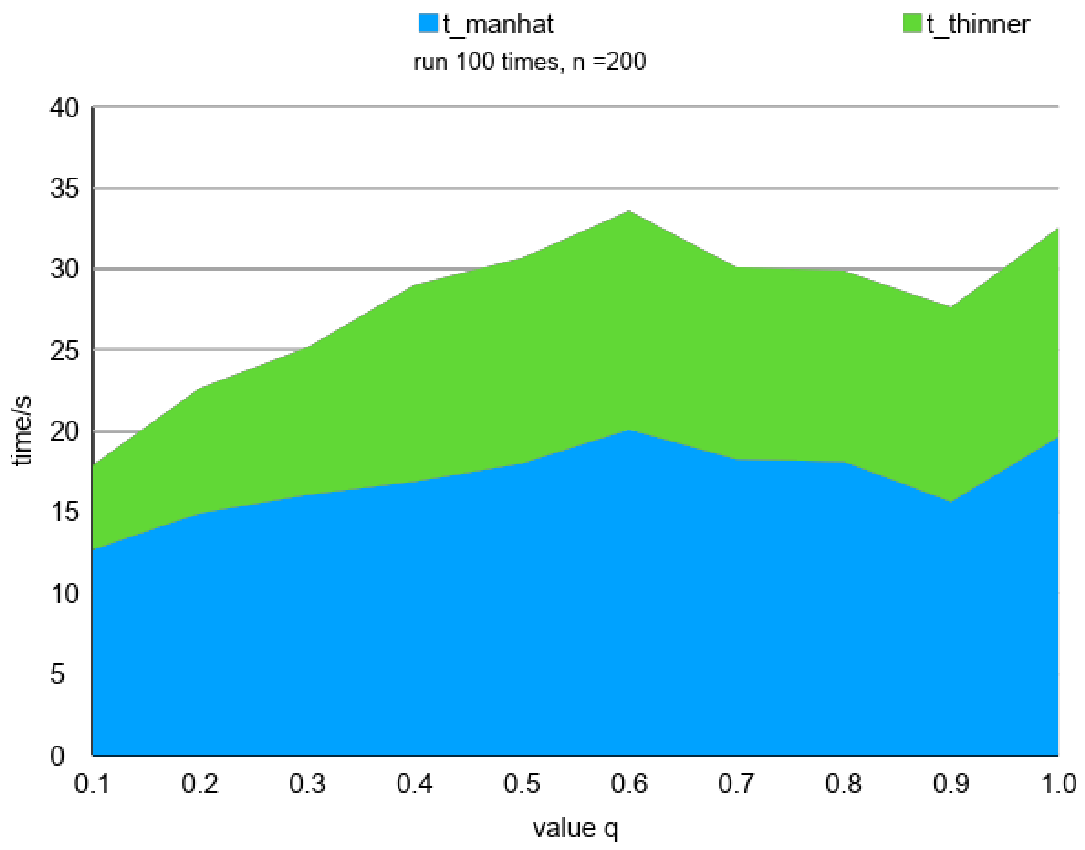
code as below:

```
import numpy as np
import matplotlib.cm as cm
from matplotlib import pyplot as plt
import sys
import heapq
import time

sys.setrecursionlimit(1000000)



# Generalize a maze with dimension and probability
class Maze:
    def __init__(self, num, p=0):
        self.dim = num
        self.M = np.random.rand(self.dim, self.dim)
        blank = self.M >= p
        blocked = self.M < p
        self.M[blank] = 1
        self.M[blocked] = 0
        self.M[0, 0] = 1
        self.M[self.dim - 1, self.dim - 1] = 1
    def simplify_maze(self,q):
        N = np.random.rand(self.dim, self.dim)
        blank = self.M == 1
        # blocked = maze.M == 0
        N[blank] = 1
        disappear = N <= q
        self.M[disappear] = 1

    def readMaze(self, m):
        self.dim = len(m)
        self.M = m

    # Return the available
    def get_available_neighbors(self, node):
        rows, cols = node
        adj = [(rows - 1, cols), (rows + 1, cols), (rows, cols - 1),
(rows, cols + 1)]  # left, right, down, up
        choice = []
        if adj[0][0] >= 0 and self.M[adj[0]] == 1:
            choice.append(adj[0])
        if adj[1][0] < self.dim and self.M[adj[1]] == 1:
            choice.append(adj[1])
        if adj[2][1] >= 0 and self.M[adj[2]] == 1:
            choice.append(adj[2])
        if adj[3][1] < self.dim and self.M[adj[3]] == 1:
            choice.append(adj[3])
        return choice

    def show_figure(self, route=[(0, 0)]):
        if self.dim <= 40:
            self.showMaze_route_grid(route)
        else:
            self.showMaze_route(route)

    # Generalize a image with some grids
    def showMaze_route_grid(self, route):
        image = np.zeros((self.dim * 10 + 1, self.dim * 10 + 1),
dtype=np.uint8)
        for row in range(self.dim):
            for col in range(self.dim):
                for i in range(10 * row + 1, 10 * row + 10):
                    if (row, col) in route and self.M[row, col] == 0:
                        print("Route conflicted with blocks")
                    elif (row, col) in route:
                        image[i, range(10 * col + 1, 10 * col + 10)] =
np.uint8(128)
                    elif self.M[row, col] == 0:
                        image[i, range(10 * col + 1, 10 * col + 10)] =
np.uint8(0)
                    else:
                        image[i, range(10 * col + 1, 10 * col + 10)] =
np.uint8(225)
        plt.imshow(image, cmap=cm.Greys_r, vmin=0, vmax=255,
interpolation='none')
        plt.show()

    def showMaze_route(self, route):
        image = self.M.astype(np.uint8).copy()
        image[image == np.uint8(1)] = np.uint8(255)
        for row, col in route:
            image[row, col] = np.uint8(128)
        plt.imshow(image, cmap=cm.Greys_r, vmin=0, vmax=255,
interpolation='none')
        plt.show()

def simpler_maze(maze, q):
    N = np.random.rand(maze.dim, maze.dim)
    blank = maze.M == 1
    #blocked = maze.M == 0
    N[blank] = 1
    disappear = N <= q
    maze.M[disappear] = 1
    return maze

def Manhattan_distance(node, maze):
    row, col = node
    distance = np.abs(maze.dim - row) + np.abs(maze.dim - col)
    return distance

def thinner_astar_heuristic(node,point):
    distance = abs(node[0]-point[0])+abs(node[1]-point[1])
    return distance
```

```
def A_star_thinner(node, maze, previous_path):
    find = False
    path = []
    level = {node: 0}
    parent = {node: None}
    fringe = []
    heapq.heapify(fringe)
    heapq.heappush(fringe,(level[node], node))
    while fringe and not path:
        currentNode = heapq.heappop(fringe)[1]
        if len(previous_path):
            point = previous_path.pop()
        for choice in maze.get_available_neighbors(currentNode):
            if choice not in parent:
                parent[choice] = currentNode
                level[choice] = level[currentNode] + 1
                if choice == (maze.dim - 1, maze.dim - 1):
                    temp = choice
                    find = True
                    while temp:
                        path.insert(0, temp)
                        temp = parent[temp]

                heapq.heappush(fringe,
(thinner_astar_heuristic(choice,point) + level[choice], choice))


    history = list(parent.keys())
    return path, history,find



def A_star_Manhattan(node, maze):
    find = False
    path = []
    level = {node: 0}
    parent = {node: None}
    fringe = []
    heapq.heapify(fringe)
    heapq.heappush(fringe, (Manhattan_distance(node, maze) +
level[node], node))
    while fringe and not path:
        currentNode = heapq.heappop(fringe)[1]
        for choice in maze.get_available_neighbors(currentNode):
            if choice not in parent:
                parent[choice] = currentNode
                level[choice] = level[currentNode] + 1
                if choice == (maze.dim - 1, maze.dim - 1):
                    temp = choice
                    find = True
                    while temp:
                        path.insert(0, temp)
                        temp = parent[temp]
                heapq.heappush(fringe, (Manhattan_distance(choice,
maze) + level[choice], choice))


    history = list(parent.keys())
    return path, history,find

if __name__=='__main__':
    t_thin=0
    t_man=0
    for i in range(1):

        test = Maze(1000, 0.3)
        test_copy = Maze(1000,0.3)
        test_copy.M=test.M.copy()
        begin = (0, 0)
        test.simplify_maze(0.9)
        # print(test.M)
        # print(test_copy.M)
        start=time.process_time()
        path0,history0,find = A_star_Manhattan(begin,test)
        end=time.process_time()
        print(end-start)
        print(find)
        if find:
            t1=time.process_time()
            path,history,find = A_star_thinner(begin,test_copy,path0)
            print(find)
            t2 = time.process_time()
            t_thin=t_thin+(t2-t1)
            # if not path:
            #     print("There is no route to solve the problem")
            #     print("Search history length is: ", len(history))
            #     test.show_figure(history)
            # else:
            #     print("There is a path exist from start to end")
            #     print("Path record length is: ", len(path))
            #     test.show_figure(path)
            t3=time.process_time()
            path2,history2,find2=A_star_Manhattan(begin,test_copy)
            print(find)

            t4=time.process_time()
            t_man=t_man+(t4-t3)
            # if not path:
            #     print("There is no route to solve the problem")
            #     print("Search history length is: ", len(history2))
            #     test.show_figure(history)
            # else:
```

```
        #     print("There is a path exist from start to end")
        #     print("Path record length is: ", len(path2))
        #     test.show_figure(path)



print(t_thin)
print(t_man)

# test = Maze(1000, 0.3)
# begin = (0, 0)
# time_start = time.process_time()
#
# path, history,find = A_star_Manhattan(begin, test)
# path_find = time.process_time()
#
# print(path_find-time_start)
#
# if not path:
#     print("There is no route to solve the problem")
#     print("Search history length is: ", len(history))
#     #test.show_figure(history)
# else:
#     print("There is a path exist from start to end")
#     print("Path record length is: ", len(path))
#     #test.show_figure(history)
```


t_manhat
t_thinner
run 100 times, n =100
time/s
value q
0
1
2
3
4
5
6
7
8
9
10
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0



Ending part:

During this project, we have a better knowledge about the local search algorithm. Furthermore, we know there is no specific one perfect solution, the best solution is by the large data analysis and the solution to just a little bit better for each step. Then, we can get the much more optimal solution eventually.

Additionally, below is our code for Part 1(BFS, DFS, A*):

DFS:

```
import time

start = time.time()

class Stack:

    def __init__(self):

            self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        return self.items.pop()
    def peek_X(self):
        return self.items[len(self.items) - 1]
    def peek_Y(self):
        return self.items[len(self.items) - 2]
    def size(self):
        return len(self.items)



s = Stack()


## initial the size of the maze.
row = 50
column = 50
import numpy as np
import random
def MazeRunner(row, column, prob):
    maze = [[np.random.choice(np.arange(0, 2), p = [1-prob, prob]) for i in range(row)]
for j in range(column)]


    ## maze[x][y] is the start node.
    maze[0][0] = 0
    #maze[row-1][column-1] = 2
    return maze
```

```
initmase = MazeRunner(row, column, 0.2)
##print initmase

edge = [[(row*column,row*column) for i in range(row)] for j in range(column)]

## initial the start node.
s.push(0);
s.push(0);
while(s.size() != 0):
    x = s.peek_X();
    y = s.peek_Y();
    mylist = []
    if x < (len(initmase) - 1):
            mylist.append(x+1)
            mylist.append(y)
    if x > 0:
            mylist.append(x-1)
            mylist.append(y)
    if y < (len(initmase) - 1):
            mylist.append(x)
            mylist.append(y+1)
    if y > 0:
            mylist.append(x)
            mylist.append(y-1)
    #print (mylist)

    i = 0;
```

```
totalnodeexpand = 0;
length = len(mylist)
while(i < len(mylist)):
        x_prime = mylist[i];
        y_prime = mylist[i+1];
        i = i+2;
        ## there is a wall here, can't be there.
        if (initmase[x_prime][y_prime] == 1):
                totalnodeexpand = totalnodeexpand + 1
        ## this node is visited, not a optimal node to be.
        elif (initmase[x_prime][y_prime] == 3):
                totalnodeexpand = totalnodeexpand + 1
        ## the node is not visited and no wall there, go there an print as visited.
        elif (initmase[x_prime][y_prime] == 0):
                ## print visited.
                initmase[x_prime][y_prime] = 3;
                ## need push y' first then push x' (LIFO)
                s.push(y_prime);
                s.push(x_prime);
                edge[x_prime][y_prime] = (x,y)
                break
        ## reached!
        ##elif (initmase[x_prime][y_prime] == 2):
                ##print 'reached, success.'
                ##break
#print maxsteps
if totalnodeexpand == (length/2):
        s.pop()
```

```
            s.pop()
goal_x = 49
goal_y = 49
path = []
while (edge[goal_x][goal_y] != (0,0)):
    if (edge[goal_x][goal_y] == (row*column, row*column)):
            print "fail, can't make it."
     break
    path_node = edge[goal_x][goal_y]
    path.append(path_node)
    goal_x = path_node[0]
    goal_y = path_node[1]

### Use matplot to draw the path.
for item in path[::-1]:
   print item
end = time.time()
t = end-start
print t
```

BFS:

```
def BFS(node, maze):
   i=1
   level = {node : 0}
   parent = {node: None}
   fringe = [node]
   path = []
   while fringe and not path:
        next = []
        for currentNode in fringe:
            for choice in maze.get_available_neighbors(currentNode):
                if choice not in level:
                    level[choice] = i
                    parent[choice] = currentNode
                    next.append(choice)
                    if choice == (maze.dim-1, maze.dim-1):
                        temp = choice
                        while temp:
                            path.insert(0, temp)
                            temp = parent[temp]
        fringe = next
        i+=1
    history = list(parent.keys())
    return path, history, fringe
```

DFS:(using class as maze)

```
def DFS_oriented(node, maze):
    path=[]
    parent = {node: None}
    fringe = [node]
    currentNode = node
    while fringe and not path:
        for choice in maze.get_nearest_neighbors(currentNode):
            if choice not in parent:
                parent[choice] = currentNode
                fringe.append(choice)
                if choice == (maze.dim - 1, maze.dim - 1):
                    temp = choice
                    while temp:
                        path.insert(0, temp)
                        temp = parent[temp]
        currentNode = fringe.pop()
    history = list(parent.keys())
    return path, history, fringe
```

A*(Euclidean):

```
def Euclidean_distance(node, maze):
    row, col = node
    distance = np.sqrt((maze.dim-row)**2+(maze.dim-col)**2)
    return distance

#Different A_star function
#A_star function with Euclidean heuristic
#A_star function with Manhattan heuristic
def A_star_Euclidean(node, maze):
    path = []
    level = {node : 0}
    parent = {node : None}
    fringe = []
    heapq.heapify(fringe)
    heapq.heappush(fringe, (Euclidean_distance(node, maze)+level[node], node))
    while fringe and not path:
        currentNode = heapq.heappop(fringe)[1]
        for choice in maze.get_available_neighbors(currentNode):
            if choice not in parent:
                parent[choice] = currentNode
                level[choice] = level[currentNode] + 1
                if choice == (maze.dim - 1, maze.dim - 1):
                    temp = choice
                    while temp:
                        path.insert(0, temp)
                        temp = parent[temp]
                heapq.heappush(fringe, (Euclidean_distance(choice, maze) + level[choice],
choice))
    history = list(parent.keys())
    return path, history
```

A*(Manhattan):

```
def Manhattan_distance(node, maze):
    row, col = node
    distance = np.abs(maze.dim-row) + np.abs(maze.dim-col)
    return distance


def A_star_Manhattan(node, maze):
    path = []
    level = {node : 0}
    parent = {node : None}
    fringe = []
    heapq.heapify(fringe)
    heapq.heappush(fringe, (Manhattan_distance(node, maze)+level[node], node))
    while fringe and not path:
        currentNode = heapq.heappop(fringe)[1]
        for choice in maze.get_available_neighbors(currentNode):
            if choice not in parent:
                parent[choice] = currentNode
                level[choice] = level[currentNode] + 1
                if choice == (maze.dim - 1, maze.dim - 1):
                    temp = choice
                    while temp:
                        path.insert(0, temp)
                        temp = parent[temp]
                heapq.heappush(fringe, (Manhattan_distance(choice, maze) + level[choice], choice))
    history = list(parent.keys())
    return path, history
```

Thinner A*:

```
import numpy as np
import matplotlib.cm as cm
from matplotlib import pyplot as plt
import sys
import heapq
import time

sys.setrecursionlimit(1000000)




# Generalize a maze with dimension and probability
class Maze:
    def __init__(self, num, p=0):
        self.dim = num
        self.M = np.random.rand(self.dim, self.dim)
        blank = self.M >= p
        blocked = self.M < p
        self.M[blank] = 1
        self.M[blocked] = 0
        self.M[0, 0] = 1
        self.M[self.dim - 1, self.dim - 1] = 1
    def simplify_maze(self,q):
        N = np.random.rand(self.dim, self.dim)
        blank = self.M == 1
        # blocked = maze.M == 0
        N[blank] = 1
        disappear = N <= q
        self.M[disappear] = 1


    def readMaze(self, m):
        self.dim = len(m)
        self.M = m


    # Return the available
    def get_available_neighbors(self, node):
        rows, cols = node
        adj = [(rows - 1, cols), (rows + 1, cols), (rows, cols - 1), (rows, cols + 1)]  # left, right,
down, up
        choice = []
        if adj[0][0] >= 0 and self.M[adj[0]] == 1:
            choice.append(adj[0])
        if adj[1][0] < self.dim and self.M[adj[1]] == 1:
            choice.append(adj[1])
        if adj[2][1] >= 0 and self.M[adj[2]] == 1:
            choice.append(adj[2])
        if adj[3][1] < self.dim and self.M[adj[3]] == 1:
            choice.append(adj[3])
        return choice

    def show_figure(self, route=[(0, 0)]):
        if self.dim <= 40:
            self.showMaze_route_grid(route)
        else:
            self.showMaze_route(route)

    # Generalize a image with some grids
    def showMaze_route_grid(self, route):
        image = np.zeros((self.dim * 10 + 1, self.dim * 10 + 1), dtype=np.uint8)
        for row in range(self.dim):
            for col in range(self.dim):
                for i in range(10 * row + 1, 10 * row + 10):
                    if (row, col) in route and self.M[row, col] == 0:
                        print("Route conflicted with blocks")
                    elif (row, col) in route:
                        image[i, range(10 * col + 1, 10 * col + 10)] = np.uint8(128)
                    elif self.M[row, col] == 0:
                        image[i, range(10 * col + 1, 10 * col + 10)] = np.uint8(0)
                    else:
                        image[i, range(10 * col + 1, 10 * col + 10)] = np.uint8(225)
        plt.imshow(image, cmap=cm.Greys_r, vmin=0, vmax=255, interpolation='none')
        plt.show()

    def showMaze_route(self, route):
        image = self.M.astype(np.uint8).copy()
        image[image == np.uint8(1)] = np.uint8(255)
        for row, col in route:
            image[row, col] = np.uint8(128)
        plt.imshow(image, cmap=cm.Greys_r, vmin=0, vmax=255, interpolation='none')
        plt.show()

def simpler_maze(maze, q):
    N = np.random.rand(maze.dim, maze.dim)
    blank = maze.M == 1
    #blocked = maze.M == 0
    N[blank] = 1
    disappear = N <= q
    maze.M[disappear] = 1
    return maze

def Manhattan_distance(node, maze):
    row, col = node
    distance = np.abs(maze.dim - row) + np.abs(maze.dim - col)
    return distance

def thinner_astar_heuristic(node,point):
    distance = abs(node[0]-point[0])+abs(node[1]-point[1])
    return distance
```

```
def A_star_thinner(node, maze, previous_path):
    find = False
    path = []
    level = {node: 0}
    parent = {node: None}
    fringe = []
    heapq.heapify(fringe)
    heapq.heappush(fringe,(level[node], node))
    while fringe and not path:
        currentNode = heapq.heappop(fringe)[1]
        if len(previous_path):
            point = previous_path.pop()
        for choice in maze.get_available_neighbors(currentNode):
            if choice not in parent:
                parent[choice] = currentNode
                level[choice] = level[currentNode] + 1
                if choice == (maze.dim - 1, maze.dim - 1):
                    temp = choice
                    find = True
                    while temp:
                        path.insert(0, temp)
                        temp = parent[temp]


                heapq.heappush(fringe, (thinner_astar_heuristic(choice,point) + level[choice],
choice))



    history = list(parent.keys())
    return path, history,find




def A_star_Manhattan(node, maze):
    find = False
    path = []
    level = {node: 0}
    parent = {node: None}
    fringe = []
    heapq.heapify(fringe)
    heapq.heappush(fringe, (Manhattan_distance(node, maze) + level[node], node))
    while fringe and not path:
        currentNode = heapq.heappop(fringe)[1]
        for choice inf
```

Trying to solve 5th problem:

5 What if the maze were on fire? (Hard, Bonus)

Thoughts:

First seeing this question, this is kind of similar to the scenario that two node make action at the same time, and see if there is an intersection occuring. If so, the robot will die, otherwise, it survives.

Furthermore, the condition mentions the free cell has a probability p' that will catch a fire, so, 0.5 < p' < 0.9375.

Similarly, if the fire point on the upper right corner do not have a path to goal, it won't affect the robot moving since the robot won't have path to the fire zone.

In this time-evolving environment, counting the real running time in seconds is not required. Each node expanded will count (int time = 0; time = time +1) and with n node expanded.

To mentioning that, it is possible the robot and the fire reach the goal node at the same time. (this may count robot destroy as well)