Maze Runner Report

CS 520 Assignment-1

Haotian Xu(hx105), Chaoji Zuo(cz296), Xuenan Wang(xw336)

Spring 2019

Maze Runner Report

CS 520 Assignment-1

1. Environments and Algorithms

Generate a maze of 10×10, probability p is 0.2 and try to find escape way by following algorithms:

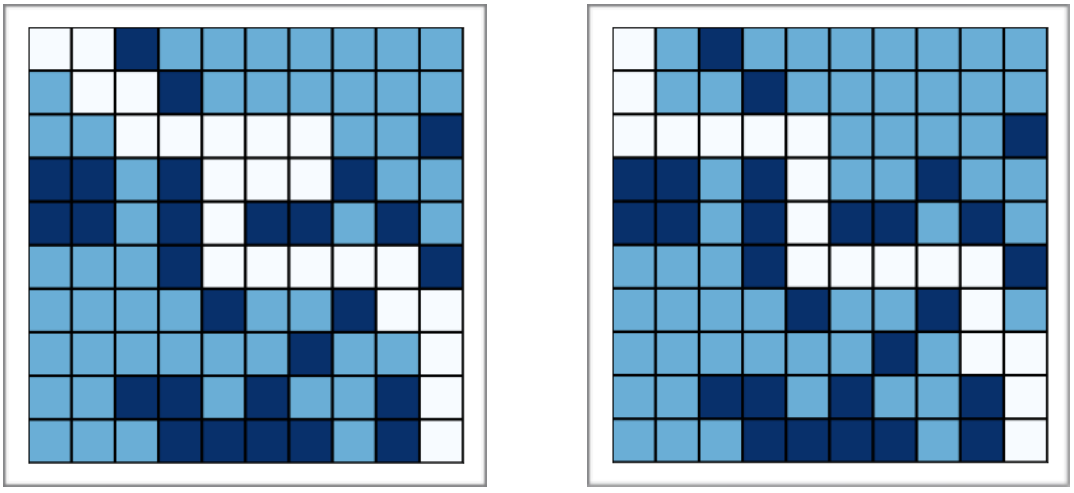

Figure 1.1 Deep-First Search Algorithm & Breadth-First Search Algorithm

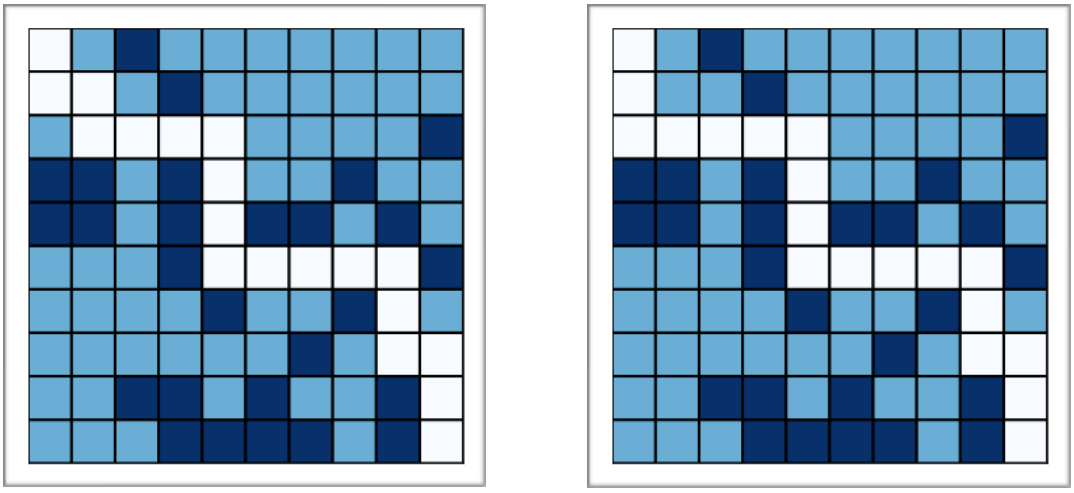

Figure 1.2 A-star Algorithm(Euclidean Distance) & A-star Algorithm(Manhattan Distance)

	Running Time(unit: second)	Operations
DFS	0.060016	47
BFS	0.083329	858
A-star Euclidean	0.059519	292
A-star Manhattan	0.059880	247

Table 1.1 Algorithms Comparison

2. Analysis and Comparison

1) We tested different map sizes for 2000 rounds and found that the map size would hardly influence the solvability. The result is in Table 2.1. However, the solving time would increase a lot as the map size enlarge. Therefore, we pic 10 as our map size. It is big enough and would not take mach time to solve.

	Running time(unit: second)	Solvability
Dim = 5	0.248	0.909
Dim = 10	1.131	0.898
Dim = 20	4.289	0.8965
Dim = 50	24.074	0.882

Table 2.1 Map sizes comparison

2) Our paths are showed in part 1. The white blocks are the path, the light blue blocks are free blocks, the dark blue blocks are wall.
3) We set the p from range 0 to 0.7, and run each p for 2000 times. Then got a curve graph of density vs solvability. Figure 2.1 shows our result

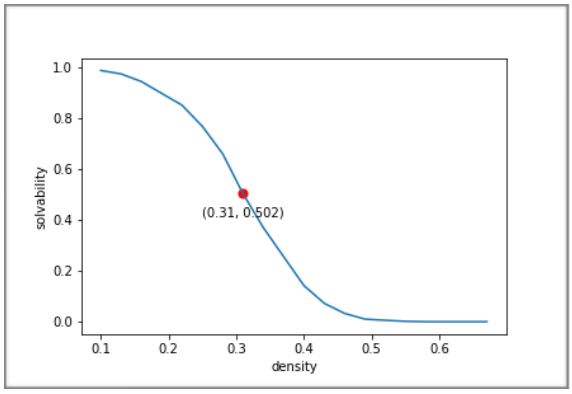

Figure 2.1 density vs solvability

As you can see, when p larger than 0.3, most mazes would be unsolvable. When p less than 0.3, most mazes would be solvable. So we thought the p0 should be 0.3. To find p0, we used DFS algorithm because it runs fast.

4) We used A* and BFS to plot density vs expected shortest path length. Because these algorithms can always find the shortest path. The results are showed in picture 2.2

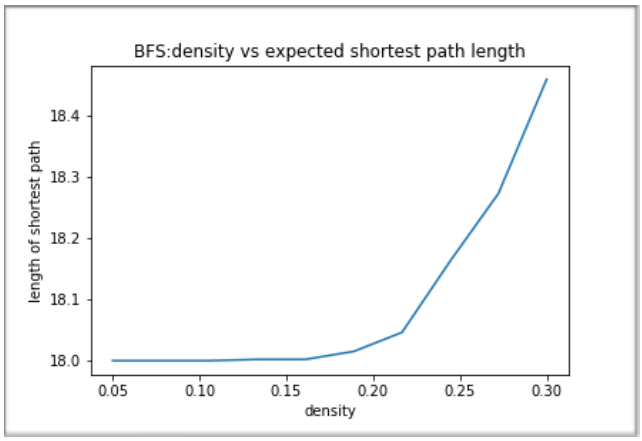

Figure 2.2 density vs expected shortest path length (BFS)

5) We thought the A*-Manhattan would perform better than A*-Euclidean in maze solving. Because A*-Manhattan would expand less nodes, so it would take less time and space. And we think the reason is the Manhattan distance can precisely calculate the smallest distance between two nodes in our maze. Figure 2.3 shows our comparison.

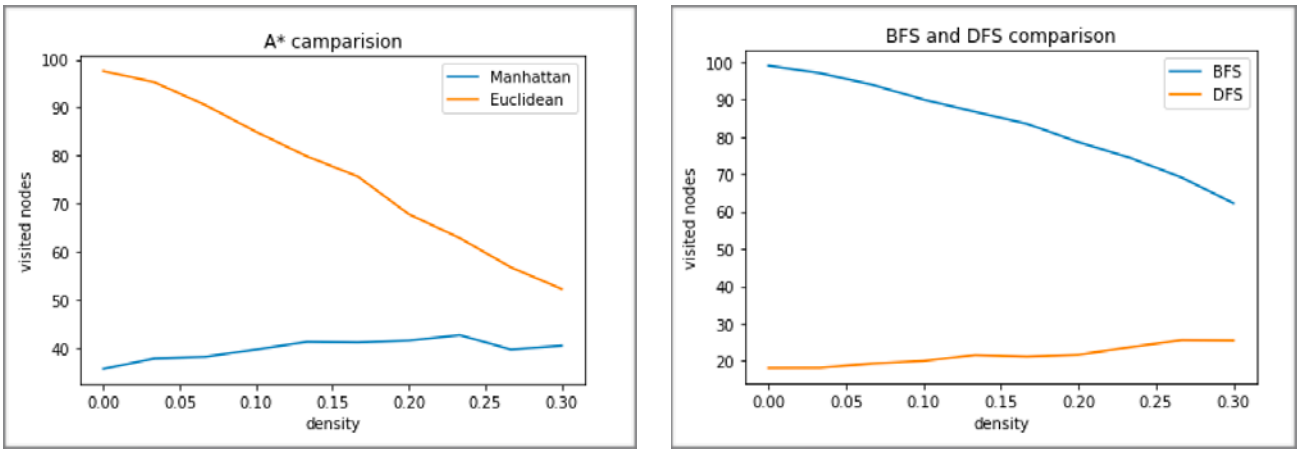

Figure 2.3 average visited nodes vs density of 4 algorithms

6) BFS would always generate an optimal shortest path in this case. But it has another shortcoming, like A*-Euclidean, it would always visit much more nodes than DFS. The result is showed in Figure 2.3.

7) These algorithms behave as they should.

8) The direction counts a lot for DFS to find a shorter path. The right and down directions should have a higher priority than left and up directions. We built two DFS algorithms and calculate their average path length. The result is showed in Figure 2.4. As you can see, the good DFS's path would be much shorter than bad DFS, they have a huge difference.
9) bonus: we think with the n increase, the p0 would increase, but just increase a little.

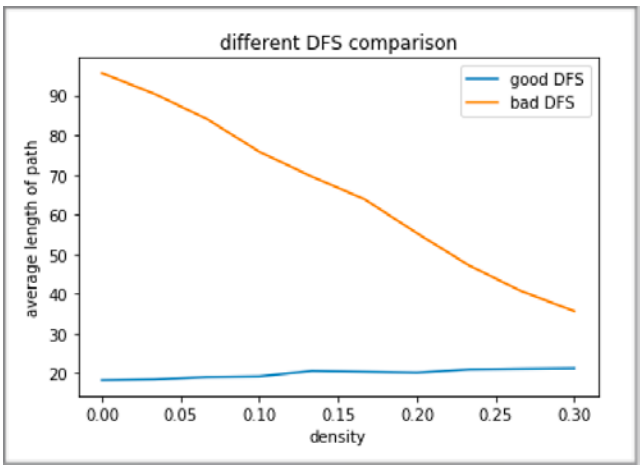

Figure 2.4 different DFS comparison

3. Generating Hard Mazes

In this problem, we pick Genetic algorithm to make a normal maze as hard as possible. And our goal state of each local search is a much longer shortest path.

Here is the main processes of our hard maze generating:

I. Generate a normal maze. Put the maze into a modified A*-Manhattan search algorithm.
II. Every 5 steps, stop, and save the nodes that had been visited, include free blocks and walls. This process can be regarded as inherit.
III. Shuffle the other blocks randomly, and use a normal A*-Manhattan to calculate the new shortest path. If the path is longer than the original one, save the new state of path. This process can be regarded as mutation.
IV. Repeat process II and III until get the goal.

Our algorithm got a quite great result. Figure 3.1 shows an original maze compare to its final evaluated maze.

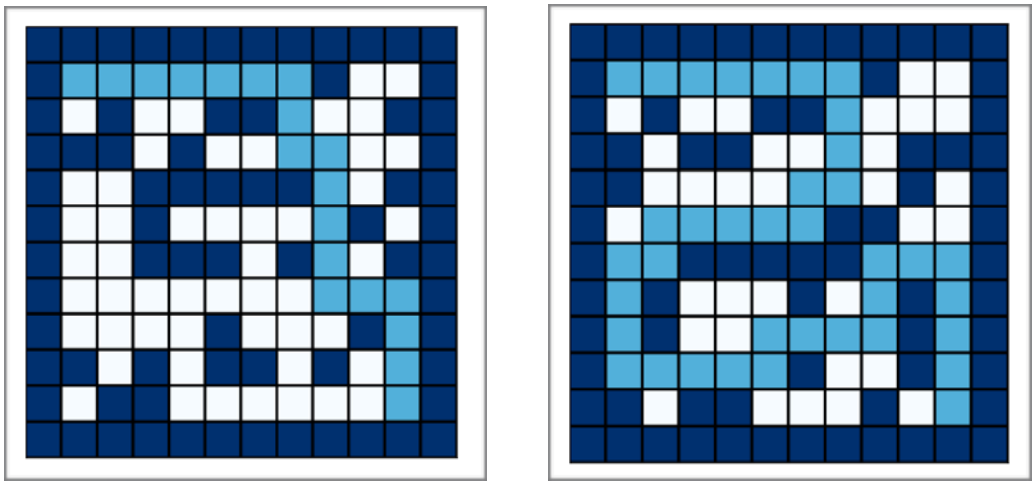

Figure 3.1 original maze compare to its final evaluated maze

Our algorithm also can use other paired metric, and the result would not have a big difference. We thought our results agree with our intuition.

4. Thinning A*

- For this problem, we want to have the a more optimal heuristic for A*; therefore, we need to calculate the kind of actual distance for each state reaching the goal state in a different version of the maze.
- For a given maze (10x10, with probability 0.3 for generating walls), we will strip out a fraction of blocks (in this case, strip 30% walls). Then we run BFS or DFS or even another A* search on this simplified maze. We will get a better heuristic by running the simplified maze. Left is original maze, and right one is simplified version.

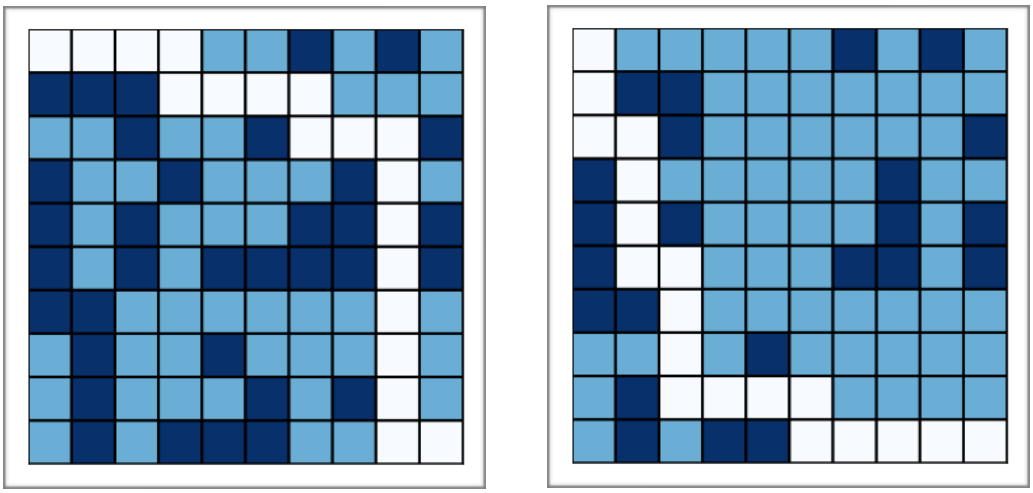

Figure 4.1 origin image vs simplified image

- More thought on problem 4:
  - Using DFS to provide new heuristics: DFS has a better space complexity than BFS, which will be easy to compute the heuristics. However, DFS cannot guarantee to provide the optimal solution; in this case, DFS may generate some inadmissible heuristics.
  - Using BFS to provide new heuristics: BFS will guarantee to have optimal solution, but its space complexity is worse than DFS, which means it will not be efficient to calculate the new heuristics.
  - Using A* to provide new heuristics: A* takes advantages from both BFS and DFS, but it is a little bit weird, because we using A* to find the heuristics, and heuristics comes from the result we run A*. I just feel a little bit awkward for using the solution to answer the same problem once more.
  - Using bi-direction A*: Run an A* function which has two start nodes (start states and goal states), start -> goal and goal -> start. There will be two fringes. When we pop those fringes, if we ends with the same state, return the path list.
  - Using hashMap with A*: when we run the A* function, we use a recursion inside the A*, then we can be bottom-to-root, memorizing the optimal heuristics for each state. In that way, we might reduce the node expansions.

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

Idea 1:

- Using a expect-max tree to find out the path that makes the robot survive and escape from the fire maze.
- First, generating the probability of turning to be fire for each state (x, y). If there is only one fire spot near a point, the point will have 1 - (1/2) ^ (n) probability of being fire in the next round, n represent the number of fire around the point.
- The probability will distribute the fire spot and decrease its value as we leave from the original fire place.
- We consider this fire maze as a Game, which will have a penalty of -100 when the robot reach a fire spot and a award of 500(it may vary because of the size of the maze) for escaping from the maze. Moreover, for each decision the robot makes, there will be a constant cost -1 for moving.
- Based on those information, if we run the expected-max tree, the algorithm should give us the optimal solution, which is the shortest path for escaping from the fire and getting out from the maze.

- The expected-max tree will be huge, if we have a large maze. Therefore, we make some change about the algorithm. When we reach a certain state, we use BFS to find out the short path for the state's neighbors for reaching a fire or reaching the goal. If there is one neighbor which reaches goal state before getting into a fire, the we will choose that neighbor as the next state. Otherwise, we find out the neighbor with the minimum utility and take it as the optimal choice (utility: penalty * (state == fire)[[Probability(state)) .
- This method is not guarantee for optimal solution. Here are three situations:
    1. Escaping the maze
    2. the maze itself has no way to escape
    3. there is no better choice for robot

  For 1 and 2, the maze is kind of under control, because those are result we might expect.

  For 3, there fire may go out of we expected, which makes robot have to choose to stay in the same place, or back track. However, fire is still going to spread; going backward will eventually reach a fire spot.

Idea 2.

• Use A*-Manhattan search algorithm as the basic maze solving problem. Update the maze map each step using the fire condition.

• Modified the A*-Manhattan search algorithm, before push a new node to the heap, find whether there is a fire nearby, if yes, add a penalization to the heuristic, if not, subtract a reward to the heuristic.

• Make the A* expand as less nodes as possible to make it run faster.

• But we still find that the performance would not be too great since the start point of fire is nearer to the maze goal than maze start. So the most important thing in escaping the fire is speed. Figure 5.2.1 shows the comparison between modified A* algorithm and DFS algorithm.

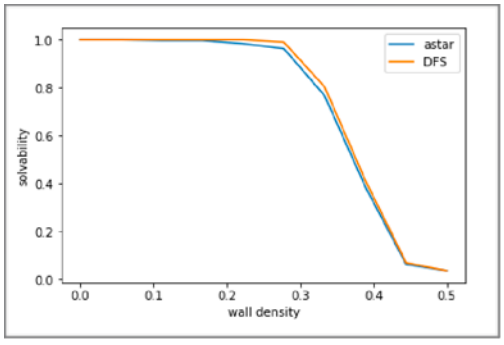

Figure 5.2.1 solvability modified A* vs DFS in fire maze problem

6. Appendix

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
import structure
import numpy as np
import matplotlib.pyplot as plt
import random
import math

from datetime import datetime

e = math.e

ctr = 0

def manhattanDistance(state, goal):
        (x1, y1), (x2, y2) = state, goal
        return abs(x2 - x1) + abs(y2 - y1)

def euclideanDistance(state, goal):
        (x1, y1), (x2, y2) = state, goal
        return ((x2 - x1) ** 2 + (y2 - y1) ** 2)**0.5

class Maze:
    def __init__(self, length, width, initialP):
        self.length = length
        self.width = width
        self.maze = {}
        self.initialP = initialP
        self.numerator = -(self.length + self.width)
        self.denominator = length * width -2
        self.probability = initialP * (e)**(self.numerator/self.denominator)
        for i in range(length):
            for j in range(width):
                self.maze[(i, j)] = -1
        self.start = (0, 0)
        self.goal = (length - 1, width - 1)

    def generateMaze(self):
        '''
            (x, y) -> 0 : empty
            (x, y) -> 1 : obstruction
        '''
        for coordinates in self.maze.keys():
            if coordinates != self.start and coordinates != self.goal:
                p = random.random()
                if p < self.probability:
                    self.maze[coordinates] = 1
                    self.denominator -= 1
                else:
                    self.maze[coordinates] = 0
                    self.numerator += 1
            self.probability = self.initialP *
```

```
(e)**(self.numerator/self.denominator)


    def printMaze(self, Search, heuristic = None):
        path = self.getPath(Search, heuristic)
        mazeMap = np.zeros((self.length, self.width), dtype = int)
        for (x, y) in self.maze.keys():
            mazeMap[x, y] = self.maze[(x, y)]
        for (x, y) in path:
            mazeMap[x, y] = -1
        plt.figure(figsize=(5,5))
        plt.pcolor(mazeMap[::-1],edgecolors='black',cmap='Blues',
linewidths=2)
        plt.xticks([]), plt.yticks([])
        plt.tight_layout()
        plt.show()

    def isWall(self, state):
        return self.maze[state] == 1

    def getSuccessor(self, state):
        '''input:
            - state: a tuple stands for coordinates
           output:
            - a list of successor,
            - a successor contains the neighbor's cooridinates and the action
              for current state to go there, and the cost(distance) between
        '''
        successors = []
        (x, y) = state
        if x - 1 >= 0 and not self.isWall((x - 1, y)):
            successors.append(((x - 1, y), "west", 1))
        if y - 1 >= 0 and not self.isWall((x, y - 1)):
            successors.append(((x, y - 1), "north", 1))
        if x + 1 < self.length and not self.isWall((x + 1, y)):
            successors.append(((x + 1, y), "east", 1))
        if y + 1 < self.width and not self.isWall((x, y + 1)):
            successors.append(((x, y + 1), "south", 1))
        return successors

    def isGoalState(self, state):
        '''
            check if the current state is the goal state
        '''
        return state == self.goal


    def getPath(self, Search, heuristic = None):
        '''
           Input: a search function, which may have heuristic function

           Turning the series of actions into coordinates

           Output: a list of states
        '''
        path = []
        if heuristic == None:
```

```
            path = Search(self)
        else:
            path = Search(self, heuristic)
        states = [self.start]
        for action in path:
            (x, y) = states[-1]
            if action == "west":
                states.append((x - 1, y))
            elif action == "north":
                states.append((x, y - 1))
            elif action == "east":
                states.append((x + 1, y))
            elif action == "south":
                states.append((x, y + 1))
        return states

    def simplify(self, fraction):
        '''
           input:
            fraction: a number ∈ [0,1]
            fraction == 1: empty maze
            fraction == 0: no changes
           output:
            a copy of the maze, which strip out a fraction of the
obstructions
        '''

        simple = Maze(self.length, self.width, 0)
        simple.maze = self.maze.copy()
        wallState = []
        numberOfWall = 0
        for state in self.maze.keys():
            if self.maze[state] == 1:
                wallState.append(state)
                numberOfWall += 1
        random.shuffle(wallState)
        stripNumber = math.ceil(fraction * numberOfWall)
        for state in wallState:
            simple.maze[state] = 0
            stripNumber -= 1
            if stripNumber == 0:
                break
        return simple

class FireMaze(Maze):
    def __init__(self,length, width, initialP, fire):
        '''
            initialize the number of the fire spots
        '''
        Maze.__init__(self, length, width, initialP)
        self.fire = fire
        self.fireSpots = []
        self.probabilityDistribution = {}
        for state in self.maze.keys():
            self.probabilityDistribution[state] = 0
        self.utility = {}
```

```
    def setFire(self):
        '''
            randomly distribute the fire spots
        '''
        coordinates = []
        for x in range((self.length // 4) + 1, 3 * self.length // 4):
            for y in range((self.width // 4) + 1, 3 * self.width // 4):
                coordinates.append((x, y))
        fire = self.fire
        random.shuffle(coordinates)
        for coordinate in coordinates:
            self.maze[coordinate] = 2
            self.probabilityDistribution[coordinate] = 1
            self.fireSpots.append(coordinate)
            fire -= 1
            if fire == 0:
                break

    def fireProbability(self, state):
        '''
            Input:
                - a state coordinate
            Output:
                - a number between 0 ~ 1 indicates the probability of the
state
                  changing to fire state
        '''

        (x, y) = state
        if self.maze[state] == 2:
            self.probabilityDistribution[state] = 1
            if state not in self.fireSpots:
                self.fireSpots.append(state)
            return 1
        '''
        probFire = 0
        if x - 1 >= 0:
            probFire += self.probabilityDistribution[(x - 1, y)]
        if y - 1 >= 0:
            probFire += self.probabilityDistribution[(x, y - 1)]
        if x + 1 < self.length:
            probFire += self.probabilityDistribution[(x + 1, y)]
        if y + 1 < self.width:
            probFire += self.probabilityDistribution[(x, y + 1)]
        return (0.25)**(probFire)
        '''
        distances = []
        for spot in self.fireSpots:
            distances.append(manhattanDistance(spot, state))
        distance = min(distances)
        return 0.25 ** (distance)

    def getNeighbor(self, state):
        '''
            Input: a given state with (x, y) as its coordinates
```

```
            Output:
                    the neighborhoods in four directions(n, e, s, w)
        '''
        (x, y) = state
        neighbor = []
        if x - 1 >= 0:
            neighbor.append((x - 1, y))
        if x + 1 < self.length:
            neighbor.append((x + 1, y))
        if y - 1 >= 0:
            neighbor.append((x, y - 1))
        if y + 1 < self.width:
            neighbor.append((x, y + 1))
        return neighbor

    def distributeFire(self):
        '''
            Distribute the fire probability to whole maze
            Using a dictionary to keep memorize the probability in each state
            Update probability when we spread fire
        '''
        probabilityLevel = self.fireSpots.copy()
        #print(probabilityLevel)
        #return None
        newProbability = {}
        for state in self.fireSpots:
            newProbability[state] = 1
        #n = 10
        for state in self.probabilityDistribution.keys():
            if self.probabilityDistribution[state] != 1:
                self.probabilityDistribution[state] = 0
        while len(newProbability) < len(self.probabilityDistribution):
            amount = len(probabilityLevel)
            for i in range(amount):
                state = probabilityLevel[i]
                #print('state:', state, 'neighbor:', self.getNeighbor(state))
                #print('current state ', state, 'has probability',
self.probabilityDistribution[state])
                for neighbor in self.getNeighbor(state):
                    #print('neighbor', neighbor, 'has probability',
self.probabilityDistribution[neighbor])
                    if self.probabilityDistribution[neighbor] <
self.probabilityDistribution[state] and neighbor not in
newProbability.keys():
                        newProbability[neighbor] =
self.fireProbability(neighbor)
                        probabilityLevel.append(neighbor)
                        #print('changed neighbor:', neighbor,'now has
probabilit', self.probabilityDistribution[neighbor] ,'but will have
probability:', newProbability[neighbor])
            for location in newProbability.keys():
                self.probabilityDistribution[location] =
newProbability[location]
            probabilityLevel = probabilityLevel[amount:]
            #print(self.probabilityDistribution)
```

```
    def spreadFire(self):
        '''
            Spread fire from the fire spots
        '''
        #print(1)
        #print(self.probabilityDistribution)
        newSpots = []
        for state in self.fireSpots:
            for neighbor in self.getNeighbor(state):
                if neighbor not in self.fireSpots and random.random() <=
self.probabilityDistribution[neighbor]:
                    self.probabilityDistribution[neighbor] = 1
                    if neighbor not in newSpots:
                        newSpots.append(neighbor)
        self.fireSpots += newSpots
        #print(1)
        #print(self.probabilityDistribution)



def DFS(maze):
        '''
            Using stack as the data structure
            return when we pop the goal state
            Output:
                a series of actions
        '''
        stack = structure.Stack()
        stack.push((maze.start, []))
        visited = {}
        global ctr
        ctr = 0
        while not stack.isEmpty():
            (state, path) = stack.pop()
            ctr += 1
            if maze.isGoalState(state):
                #print("Reach the goal!")
                return path
            for successor in maze.getSuccessor(state):
                (neighbor, action, _) = successor
                if neighbor not in visited.keys():
                    stack.push((neighbor, path + [action]))
            if state not in visited.keys():
                visited[state] = True
            if stack.isEmpty():
                #print("There is no such a path!")
                return path

def BFS(maze):
        '''
            Using queue as the data structure
            return when we meet the goal state
            Output:
```

```
            a series of actions
        '''
        queue = structure.Queue()
        queue.enqueue((maze.start, []))
        visited = {}
        global ctr
        ctr = 0
        while not queue.isEmpty():
            (state, path) = queue.dequeue()
            ctr += 1
            for successor in maze.getSuccessor(state):
                (neighbor, action, _) = successor
                if maze.isGoalState(neighbor):
                    #print("Reach the goal!")
                    return path + [action]
                elif neighbor not in visited.keys():
                    queue.enqueue((neighbor, path + [action]))
            if state not in visited.keys():
                visited[state] = True
            if queue.isEmpty():
                #print("There is no such a path!")
                return path


def Astar(maze, heuristic):
        '''
           input:
             - heuristic : a function which gives us the estimated value
           Using min-heap (priority queue) as data structure
           return when we pop the goal state
           Output:
                a series of actions
        '''
        Heap = structure.PriorityQueue()
        Heap.heap = [(0, 1, (maze.start, [], heuristic(maze.start,
maze.goal)))]
        visited = {}
        global ctr
        ctr = 0
        while not Heap.isEmpty():
            (state, path, priority) = Heap.pop()
            ctr += 1
            if maze.isGoalState(state):
                #print("Reach the goal!")
                return path
            for successor in maze.getSuccessor(state):
                (neighbor, action, cost) = successor
                if neighbor not in visited.keys():
                    visited[neighbor[0]] = True
                    Heap.update((neighbor, path + [action], priority + cost),
priority + cost + heuristic(neighbor, maze.goal))
            if state not in visited.keys():
                visited[state] = True
            if Heap.isEmpty():
                #print("There is no such a path!")
                return path
```

```
def approximateDistance(maze):
    return len(BFS(maze))

def Astar_Thinning(maze):
    Heap = structure.PriorityQueue()
    Heap.heap = [(0, 1, (maze.start, [], approximateDistance(maze)))]
    simple = maze.simplify(0.6)
    simple.printMaze(Astar, manhattanDistance)
    visited = {}
    while not Heap.isEmpty():
        (state, path, priority) = Heap.pop()
        if maze.isGoalState(state):
            #print("Reach the goal!")
            return path
        simple.start = state
        for successor in maze.getSuccessor(state):
            (neighbor, action, cost) = successor
            if neighbor not in visited.keys():
                visited[neighbor[0]] = True
                Heap.update((neighbor, path + [action], priority + cost),
priority + cost + approximateDistance(simple))
        if state not in visited.keys():
            visited[state] = True
        if Heap.isEmpty():
            #print("There is no such a path!")
            return path

def expectValue(fireMaze, state, visited):
    '''
        Input:
            fireMaze: a fireMaze object
            state: current state, determinate the probability of being fire
in
                   the future (utility)
            visited: a list of the nodes(states) which have already been
expand

        check the (current) state's utility by calculating its probability of
being
        fire in the future. Calculation is based on the utility of its
neighborhoods
        which means the probability of being fire is cumulative from the
nearest fire spot

        Output:
            the utility of current state
    '''
    if fireMaze.isGoalState(state):
        fireMaze.utility[state] = (fireMaze.length + fireMaze.width) ** 2
        return (fireMaze.length + fireMaze.width) ** 2
    elif fireMaze.probabilityDistribution[state] == 1:
        fireMaze.utility[state] = -10
        return -10
    value = 0
    for successor in fireMaze.getSuccessor(state):
```

```
        (neighbor, _, _) = successor
        if neighbor not in visited:
            probability = fireMaze.probabilityDistribution[neighbor]
            if neighbor not in fireMaze.utility.keys():
                fireMaze.utility[neighbor] = expectValue(fireMaze, neighbor,
visited)
            value += probability * fireMaze.utility[neighbor]
    fireMaze.utility[state] = value
    return value

def FireBFS(fireMaze, state):
        queue = structure.Queue()
        queue.enqueue((state, []))
        visited = {}
        states = [state]
        while not queue.isEmpty():
            (state, path) = queue.dequeue()
            for successor in fireMaze.getSuccessor(state):
                (neighbor, action, _) = successor
                if fireMaze.isGoalState(neighbor) or neighbor in
fireMaze.fireSpots:
                    #print("Reach the goal!")
                    path.append(action)
                    utility = 1
                    for action in path:
                        (x, y) = states[-1]
                        if action == "west":
                            states.append((x - 1, y))
                            utility *= fireMaze.probabilityDistribution[(x -
1, y)]
                        elif action == "north":
                            states.append((x, y - 1))
                            utility *= fireMaze.probabilityDistribution[(x, y
- 1)]
                        elif action == "east":
                            states.append((x + 1, y))
                            utility *= fireMaze.probabilityDistribution[(x +
1, y)]
                        elif action == "south":
                            utility *= fireMaze.probabilityDistribution[(x, y
+ 1)]
                    if states[-1] in fireMaze.fireSpots:
                        utility *= (-100)
                        print(utility)
                    else:
                        utility = (fireMaze.length + fireMaze.width) ** 2
                    return utility
                elif neighbor not in visited.keys():
                    queue.enqueue((neighbor, path + [action]))
            if state not in visited.keys():
                visited[state] = True
            if queue.isEmpty():
                #print("There is no such a path!")
                utility = 1
                for action in path:
                    (x, y) = states[-1]
                    if action == "west":
```

```
                        states.append((x - 1, y))
                        utility *= fireMaze.probabilityDistribution[(x - 1,
y)]
                    elif action == "north":
                        states.append((x, y - 1))
                        utility *= fireMaze.probabilityDistribution[(x, y -
1)]
                    elif action == "east":
                        states.append((x + 1, y))
                        utility *= fireMaze.probabilityDistribution[(x + 1,
y)]
                    elif action == "south":
                        utility *= fireMaze.probabilityDistribution[(x, y +
1)]
                if states[-1] in fireMaze.fireSpots:
                    utility *= (-100)
                else:
                    utility = utility /
fireMaze.probabilityDistribution[states[-1]]
                    utility *= (fireMaze.length + fireMaze.width) ** 2
                return utility

def makeDecision(fireMaze):
    visited = [fireMaze.start]
    path = []
    while True:
        state = visited[-1]
        v = -2 ** 64
        print('number of fire spots:', len(fireMaze.fireSpots))
        print('---------')
        print(fireMaze.fireSpots)
        print('---------')
        print('current state is', state, 'the probability of being fire is',
fireMaze.probabilityDistribution[state])
        if fireMaze.isGoalState(state):
            print("You escape the fire maze!")
            return path
        elif state in fireMaze.fireSpots:
            print("You are burned")
            return path
        optimalAction = ''
        nextState = state
        for successor in fireMaze.getSuccessor(state):
            (neighbor, action, _) = successor
            if neighbor not in visited and v < FireBFS(fireMaze, neighbor):
                v = FireBFS(fireMaze, neighbor)
                nextState = neighbor
                optimalAction = action
        print(v)
        visited.append(nextState)
        path.append(optimalAction)
        fireMaze.spreadFire()
        fireMaze.distributeFire()



def maxValue(fireMaze):
```

```
    '''
        go through the whole maze, when moving to a state, the maze will
update
        its fire probability distribution, and based on the new distribution
        function will choose the optimal direction
    '''
    visited = [fireMaze.start]
    path = []
    while visited != []:
        fireMaze.utility = {}
        state = visited[-1]
        nextState = state
        optimalAction = ''
        if fireMaze.isGoalState(state):
            return path
        value = -2 ** 64
        print(1)
        for successor in fireMaze.getSuccessor(state):
            (neighbor, action, _) = successor
            print('state ', neighbor, 'with utility', expectValue(fireMaze,
neighbor, visited))
            if neighbor not in visited and value < expectValue(fireMaze,
neighbor, visited):
                value = expectValue(fireMaze, neighbor, visited)
                nextState = neighbor
                optimalAction = action
            print(fireMaze.utility)
            return path
        if nextState != state:
            print('current state is ', state)
            print('next state will be', nextState, ', and the action will
be', optimalAction)
            visited.append(nextState)
        visited = visited[1:]
        if optimalAction != '':
            path.append(optimalAction)
        fireMaze.spreadFire()
        fireMaze.distributeFire()
        return path
    return path



maze = Maze(10, 10, 0.2)
maze.generateMaze()
path = []
ctr = 0
timeA = datetime.now()
maze.printMaze(BFS)
print("operations of BFS is:", ctr)
timeB = datetime.now()
maze.printMaze(DFS)
print("operations of DFS is:", ctr)
timeC = datetime.now()
maze.printMaze(Astar, manhattanDistance)
print("operations of Astar-manhattan is:", ctr)
```

```
timeD = datetime.now()
maze.printMaze(Astar, euclideanDistance)
print("operations of Astar-Euclidean is:", ctr)
timeE = datetime.now()
maze.printMaze(Astar_Thinning)
timeF = datetime.now()

print("Running time of BFS:",timeB-timeA)
print("Running time of DFS:",timeC-timeB)
print("Running time of Astar-Manhattan:",timeD-timeC)
print("Running time of Astar-Euclidean:",timeE-timeD)
print("Running time of Astar-Thinning:",timeF-timeE)

firemaze = FireMaze(10, 10, 0, 1)
firemaze.setFire()
firemaze.distributeFire()
print(makeDecision(firemaze))
```

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import heapq

class Stack:
    def __init__(self):
        self.list = []

    def push(self, item):
        self.list.append(item)

    def pop(self):
        return self.list.pop()

    def isEmpty(self):
        return self.list == []

class Queue:
    def __init__(self):
        self.list = []

    def enqueue(self, item):
        self.list.insert(0, item)

    def dequeue(self):
        return self.list.pop()

    def isEmpty(self):
        return self.list == []

class PriorityQueue:
    def __init__(self):
        self.heap = []
        self.count = 0
```

```
    def push(self, item, priority):
        entry = (priority, self.count, item)
        heapq.heappush(self.heap, entry)
        self.count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0

    def update(self, item, priority):
        # If item already in priority queue with higher priority, update its
priority and rebuild the heap.
        # If item already in priority queue with equal or lower priority, do
nothing.
        # If item not in priority queue, do the same thing as self.push.
        for index, (p, c, i) in enumerate(self.heap):
            if i == item:
                if p <= priority:
                    break
                del self.heap[index]
                self.heap.append((priority, c, item))
                heapq.heapify(self.heap)
                break
        else:
            self.push(item, priority)
```