Name Milan Mukherjee ID: 22303417

Introduction:

Here a maze generator is implemented using python pyamaze library ("https://pypi.org/project/pyamaze/") which creates a maze and also provides an agent to traverse the maze as per specified traversal path.

All the 5 specified algorithms are implemented: Breadth-First Search, Depth First Search and A* , MDP value iteration and MDP policy iteration.

Here a total of 6 mazes of different sizes are generated and saved. 10*10,20*20, 20*30, 30*30 , 50*50, 80*80 . Alternatively, an user can also create her own maze as per required.
Among them, the mazes are used for comparison are of size 20*20, 50*50 and 80*80 . For better benchmark all the mazes are pre-generated and saved for reuse as mentioned.

All the parameters are customizable. For example: for BFS,DFS, and A* the customizable parameters are:
1. maze size
2. loopPercent of the maze
3. target or goal cell

For MDP the customizable parameters are:
1. maze size
2. loopPercent of the maze
3. target or goal cell
4. discount factor
5. Noise (probability to move straight, right or left when policy is to move straight.)

In noise, for example, lets say an user specifies straight probability as 0.8, right as 0.1 and left as 0.1 and if for a state policy is to move north. In that case, during the calculation of utilities, it will be considered that the probability to move north is 80%, the probability to move east is 0.1 and the probability to move west is 0.1. Probability to move south would be zero in this case. This makes the MDP stochastic or non-deterministic.

Important to note that, during comparison of MDP algorithms with search algorithms, probability is set for straight move is 1 and for right and left 0 to keep the benchmark equivalent with search algorithms.

Logic of the algorithm:
To run and test any algorithm, the user only needs to run the "main.py" file. This file asks for other inputs from the user and calls a suitable module of the algorithm the user decides.

Pyamaze provides a maze where each cell is defined and can be accessed as a tuple (x,y). Default goal state is (1,1)

Depth-First Search:

Logic:

```
Depth_First(agent,goal):
  Begin = LIFO queue
  visited = list of nodes visited(initialized with start node)
  Journey = an empty dictionary
  while(length of visited>0):
          Square = queue.pop()

          For all the directions for the state:
                  If direction is open:
                          next_square = handle_agent_movement(direction,state)
                  If next_square in visited:
                          continue
                  Journey[next_square] = square
                  visited.append(next_square)
                  queue.append(next_square)
                  If square == goal:
                          Break from all the loop
  backtrace(goal,begin,journey) #find final path from the dictionary journey
  return path, count of traversal
```

The module is separated into 3 parts:
find_path method is called from the main.py. It is called the Depth_First algorithm. The algorithm maintains a queue named “queue”. It is a LAST IN FIRST OUT(LIFO) queue which essentially means a stack. It is used to maintain and check which square or cell is next to consider. The LIFO implementation ensures the depth-first procedure that is all the nodes in the same level of the graph are considered before moving to the next level.

The list “visited” keeps track of the nodes which are already visited so that those nodes are not considered again. It helps to avoid loops.

The handle_agent_movement function takes the agent's current state and next direction as input. It returns the agent's next state.
The dictionary “journey” keeps track of node to node journey. For example, if there is a key-value pair as “State (4,5): State (3,4)”, it means the previous state of state(4,5) is state (3,4). The backtrace function helps to create the final path starting from start state to goal state.
The Depth_First function returns two variables called “path” and “count”.
The “path” stores the final path agent should follow according to the algorithm and “count”. The count gives the number of nodes visited in total.

Breadth_First Algorithm:

Logic:

```
Breadth_First(agent,goal):
  Begin = FIFO queue
  visited = list of nodes visited(initialized with start node)
  Journey = an empty dictionary
  while(length of visited>0):
        Square = queue.pop()
        For all the directions for the state:
              If direction is open:
                    next_square = handle_agent_movement(direction,state)
              If next_square in visited:
                    continue
              Journey[next_square] = square
              visited.append(next_square)
              queue.append(next_square)
              If square == goal:
                    Break from all the loop
  backtrace(goal,begin,journey) #find final path from start to goal using the dictionary journey
  return path, count of traversal
```

Design and observation:

BFS and DFS implementations have one major difference that is: In BFS FIFO queue is used to store and pop states but in DFS LIFO queue(stack) is used to pop and store elements. The FIFO logic in BFS helps to ensure that all the nodes in the same level of the graph are considered before moving to next level.

So, DFS may not find the most optimal path to goal unlike BFS.

A_Star Algorithm:

Logic:

```
A_star(agent,goal_state):
  Begin = FIFO queue
  visited = list of nodes visited(Initialized with start node)
  Journey = an empty dictionary
  previous_cost =  initially set infinity for all nodes(dictionary of cost of reaching each particular node)
  total_cost =  initially set infinity for all nodes (previous_cost + heuristic_cost to reach goal)
  queue = stores tuple of total cost & square for each square
  # Above is a priority queue. If  (a,b) square and total cost for the square is c then tuple is (c,(a,b))
priority is set based on the least value of total cost that is c.
  while(length of visited>0):
        square = pop the min cost square from queue
        If (square == goal)
              break
        For each direction of the square:
              If the direction is open:
              next_square = handle_agent_movement(direction,state)
              next_square_cost = previous_cost[square] + 1+next_square_heuristic_cost
              If next_square is visited:
              update next_square total cost and previous cost if they are new nodes
               or the new total cost is less than previously calculated total cost
              queue.append(next_square_cost,next_sqare)
              Update journey
              Add next_square to visited if not already.
  return path, count of traversal

  backtrace(goal,begin,journey) #find final path from start to goal using the dictionary journey
  return path
```

Design decision:

Here the heuristic function is selected in such a way so that it is always admissible and consistent. Here the Manhattan distance is calculated in the heuristic function. It is tried to ensure that for any other admissible heuristic functions and a state n, current_heuristic_result(n) > other_heuristic_result(n). For example, euclidean distance could also be considered as heuristic and it is both admissible and consistent. But for a given state and goal, manhatton_distance > euclidian distance that is. Manhattan distance is closer to original cost. Hence Manhatton distance is selected

The two major difference between BFS or DFS and A* is :

1. The BFS or DFS blindly visits all the node based on LIFO or FIFO however A* keeps a optimistic cost calculation to find the goal state
2. In BFS or DFS, loop breaks or search stops when the goal state is fetched for the first time and it is not enqueued in the queue. However, in A* search, the goal state is also queued and the search stops when the goal state is popped from the queue.

Value Iteration:

Logic:

```
val_iter(agent,rows,cols,discount,reward,delta,prob,goal):
        Value = dictionary of utilities
        Set all utility as zero
        error = minus infinity
steps = dictionary of path. Initially empty
        while(delta<error):
                For each square or state:
                        If state == goal:
                                Set reward = reward[goal]
                                Continue

Find max of( total possible score for each open direction based on Bellman equation for the square)

update the steps and utility accordingly
                error = max(difference in previous utility and updated utility for each square)
        Return steps, total loop count
```

The following pseudo code is followed:

function VALUE-ITERATION(mdp, ϵ) **returns** a utility function
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$,
      rewards $R(s)$, discount γ
    ϵ, the maximum error allowed in the utility of any state
  local variables: U, U', vectors of utilities for states in S, initially zero
      δ, the maximum change in the utility of any state in an iteration

  repeat
    $U \leftarrow U'$; $\delta \leftarrow 0$
    for each state s **in** S **do**
      $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
      if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  until $\delta < \epsilon(1-\gamma)/\gamma$
  return U

Figure 17.4 The value iteration algorithm for calculating utilities of states. The termination condition is from Equation (17.8).

Credit: AI- A modern approach by Russell and Norvig

Observation:

Here the utilities are set to zero initially and the algorithm tries to find the best utility for all the states. From the utility, the direction is tracked.

Policy Iteration:

Logic:

First a random policy is created and passed to the function

```
pol_iter(agent,rows,cols,discount,reward,delta,policy,prob,goal):
Utility = dictionary of utilities of states.
error = minus infinity
While true:
        while(delta<error):
                Policy_evaluation based on current policy
        old_policy = policy
        For each square best policy is checked and policy is set accordingly
        if(policy == old_policy):
                Break
return policy,total loop count
```

The following pseudo code is followed:

function POLICY-ITERATION(mdp) **returns** a policy
  inputs: mdp, an MDP with states S, actions $A(s)$, transition model $P(s' \mid s, a)$
  local variables: U, a vector of utilities for states in S, initially zero
    π, a policy vector indexed by state, initially random

  repeat
    $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
    $unchanged? \leftarrow$ true
    for each state s **in** S **do**
      if $\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s'] > \sum_{s'} P(s' \mid s, \pi[s])\, U[s']$ **then do**
        $\pi[s] \leftarrow \operatorname{argmax}_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$
        $unchanged? \leftarrow$ false
  until $unchanged?$
  return π

Figure 17.7 The policy iteration algorithm for calculating an optimal policy.

Credit: AI- A modern approach by Russell and Norvig

Observation:

For MDP To reach the goal for larger mazes, reward should be high for goal state.

Low Discount factor ensures the goal will be reached with shorter steps but too low discount factor traps the agent in deadlock.
Here Path length, Total number of loops and time taken to execute an algorithm, these three major metrics are considered. The time shows how good an algorithm is and perfect for comparison between mdp and search algorithms.

Number of iteration is proportional to algorithm complexity.
The memory taken is calculated only for maze 50*50 because only for large maze, it will be significant.
Analysis:

Time taken is in seconds
Maze 10*10

Algo	Path Length	Total node or loop	Wall time taken	CPU execution time	Remark
BFS	31	99	0.015	Almost zero	optimal
DFS	47	51	Almost zero	Almost zero	Not optimal
A*	31	84	0.031	Almost zero	optimal
policyIteration	31	25	0.437	0.328	optimal
Value Iteration	31	18	0.422	0.265	optimal

Maze 20*20

Algo	Path Length	Total node or loop	Wall time taken	CPU execution time	Remark
BFS	97	399	0.015	Almost zero	optimal
DFS	121	340	Almost zero	Almost zero	Not optimal
A*	97	409	0.015	Almost zero	optimal
policyIteration	97	87	1.28	1.09	optimal
Value Iteration	97	61	0.58	0.35	optimal

Maze 30*30

Algo	Path Length	Total node or loop	Wall time taken	CPU execution time	Remark
BFS	77	848	0.015	0.015	optimal
DFS	105	113	Almost zero	Almost zero	Not optimal
A*	77	479	0.015	Almost zero	optimal
policyIteration	97	87	0.813	0.718	Not optimal
Value Iteration	97	61	0.595	0.390	Not optimal

Maze 50*50

Algo	Path Length	Total node or loop	Wall time taken	CPU execution time	Memory (MiB)	Remark
BFS	191	2442	0.0945	0.078	32.5429687	optimal
DFS	567	959	0.016	Almost zero	32.4375	Not optimal
A*	191	2218	0.06	0.031	33.1289062	optimal
policyIteration	191	127	4.43	4.17	68.453125	optimal
Value Iteration	191	58	1.06	0.46	68.4609375	optimal

Maze 80*80

Algo	Path Length	Total node or loop	Wall time taken	CPU execution time	Remark
BFS	323	6399	0.578	0.578	optimal
DFS	1445	5141	0.37	0.328	Not optimal
A*	323	6343	0.54	0.5	optimal

policyIteration	323	219	15.78	15.29	optimal
Value Iteration	-	-	-	-	deadlock

Optimality:
While comparing with search algorithms, the probability factor was removed from mdp and it was set as deterministic. As it can be observed that, BFS and A * always returns optimal results. But incase of policy iteration and value iteration, the optimality depends on the reward, discount factor.

Path length:
For maze 30*30 the mdp doesn't return an optimal path initially. But when, reward was increased for goal state and living reward for other states were set as negative and discount factor was decreased(0.9 to 0.4), the algorithm tended to converge faster.

Also, setting discount factor too low for large maze might cause deadlock as the goal state reward wouldn't have much impact in that case.

Time:
Overall, it can be observed that, DFS takes the least time followed by A*, BFS ,policy iteration and value iteration respectively.
Node traversed and Loop executed:
Between BFS ,DFS and A*, DFS traverse through least number of nodes followed by A* and BFS.

For the search algorithms, mostly the iteration over the outermost loop is calculated. For example, for Policy_iteration, the loop for policy_evaluation is not considered. Only the number of times policy improvement occurs is calculated.

Here although, loop execution is better for value iteration, but it fails to find optimal solution for 80*80 maze. Tuning the discount factor , living reward, it can be managed.

Memory:
The memory consumption is DFS<BFS<A*<MDP
This is expected since DFS doesn't store much information in the queue because of last in first out execution. On the other hand, A* and MDP maintains lots of information related to heuristic and utility.

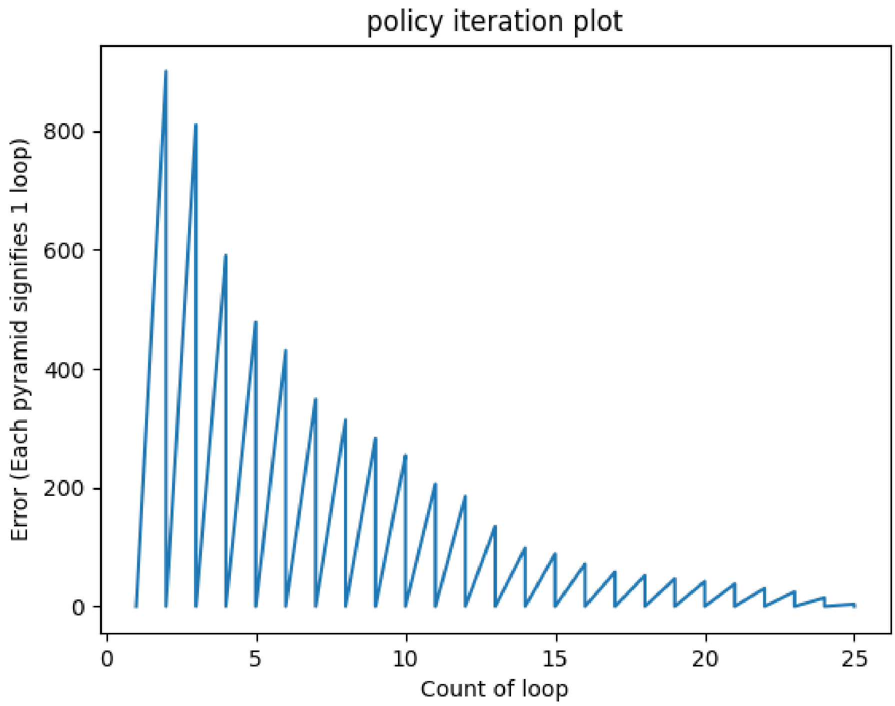

Fig 1 (discount = 0.9)

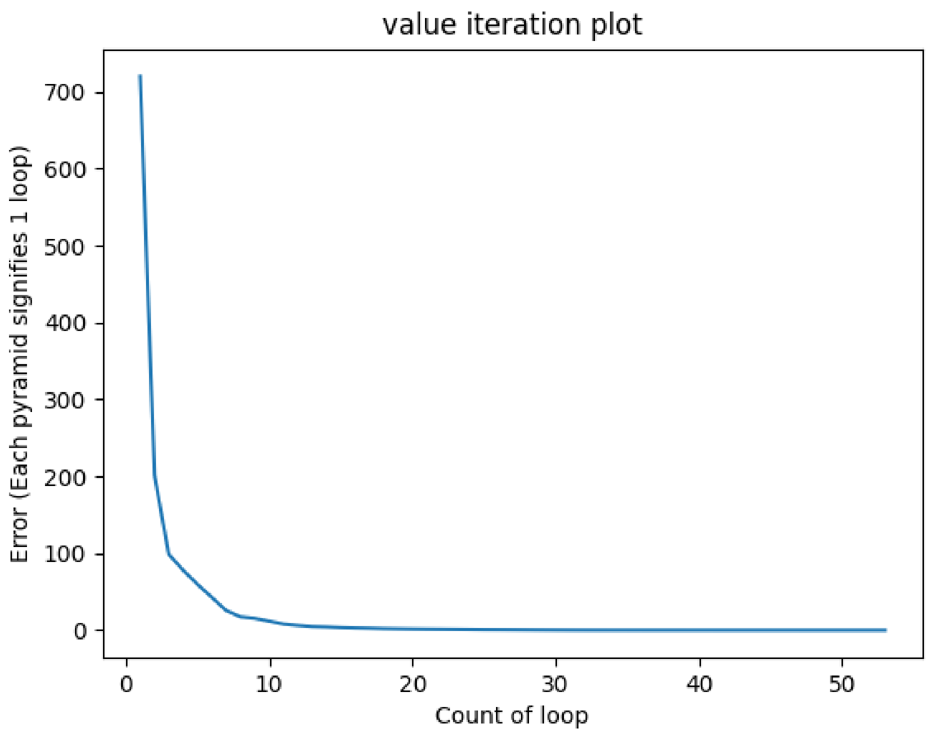

Fig 2 (discount = 0.9)

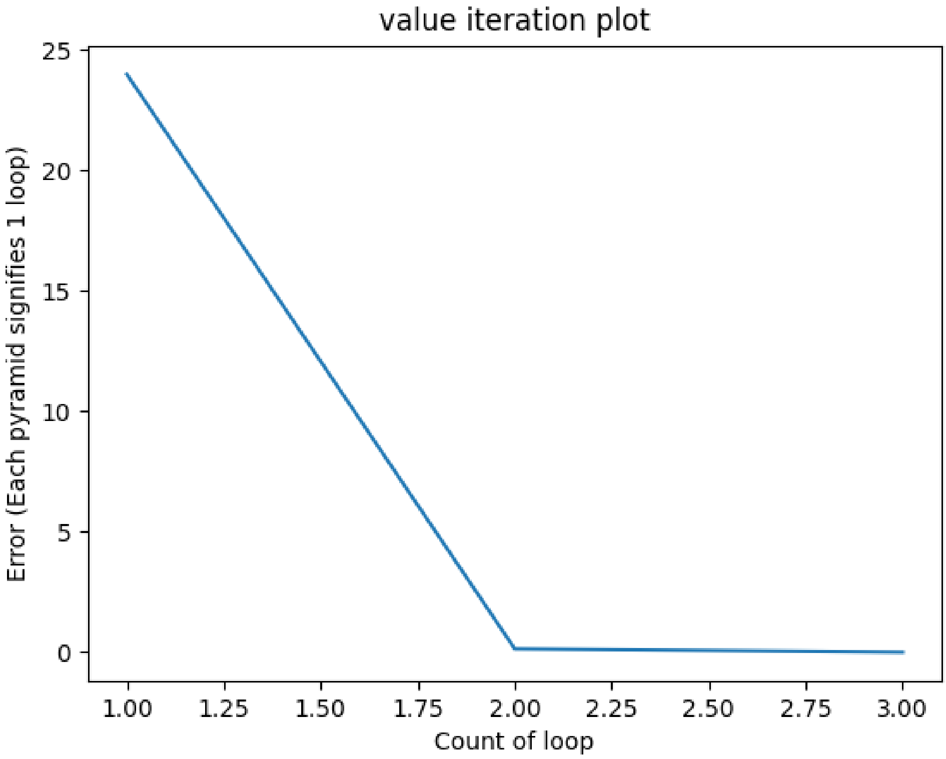

Fig 3 (discount factor = 0.03)

The above figures plots the change in error as the algorithm converge. For Fig 1, it can be seen that at the start of each policy evaluation the error is high and gradually it decreases. Again for the next policy evaluation it is high initially and reduce with time. Gradually the graph converse and error tends to zero.

For Fig 2 and Fig 3, same error is plotted with different discount factor. For discount factor 0.9, the algorithm converges in around 80 iteration. However, as the discount factor reduces, the algorithm converges in just 3 iteration.
Also, its better to use high rewards for goal state to avoid deadlock.

<u>Plan and Future:</u>
When enough mazes are run on these 5 algorithms, a machine learning model can be formed (train on mazes as input and execution time, optimality as output) used to feed the output data such and predict which algorithm would be best for which maze. Also, it the model is trained on different discount factors and rewards, the model can predict the suitable discount factor and rewards for a mdp running on a maze..

But for this, enough mazes need to be trained.

Appendix:

main_script(main.py):

```
import sys
from pyamaze import maze, COLOR, agent, textLabel
from time import process_time
import time

algo = input("Enter the algorithm you want to use: \n type 'policy' for
policy iteration \n type 'value' for value_iteration \n type 'astar' for
A_STAR \n type 'bfs' for Breadth First Search \n type 'dfs' for Depth First
Search \n>> ")

predefined_maze = input("type 'y' if you want to use predefined maze?\n>> ")
if(predefined_maze == "y"):
    maze_name = input("give your maze name\n options:    name\n 50*50 maze:
'maze5050.csv' \n 30*30 maze: 'maze3030.csv' \n 20*20 maze: 'maze2020.csv'\n
10*10 maze: 'maze1010.csv'\n 20*30 maze: 'maze2030.csv'\n 80*80 maze:
'maze8080.csv' \n>> ")
    my_maze=maze()
else:
    rows = int(input("Enter maze number of rows\n>> "))
    cols = int(input("Enter maze number of cols\n>> "))
    my_maze=maze(rows,cols)
    looppercent = input("type 'y' if you want to use defalt loopPercent of
the maze? default = 10 \n>> ")
    if(looppercent != "y"):
        looppercent = int(input("input loopPercent raange 0-100\n>> "))
    else:
        looppercent = 10
goal_state = input("type 'y' to use default goal state?default is (1,1) top
left cell of the maze\n>> ")
if(goal_state == "y"):
    goal = (1,1)
else:
    goal_row = int(input("give row number of goal state\n>> "))
    goal_col = int(input("give column number of goal state\n>> "))
    goal = (int(goal_row),int(goal_col))


if(algo == "policy" or algo == "value"):
    path_string = 'length of path'
    iteration_node = 'total iteration '
    discount_factor = input("type 'y' to use default discount factor? default
0.9\n>> ")
```

```
    if (discount_factor != "y"):
        discount_factor = float(input("give your preferred discount
factor\n>> "))
    else:
        discount_factor = 0.9

    use_default_prob = input("type 'y' to use default probability. Default is
0.8 for stragiht,0.1 for left and 0.1 for right\n>> ")
    if(use_default_prob == "y"):
        prob = [0.8,0,0]

    else:
        straight_prob = float(input("give straight moving prob. sample:
0.8\n>> "))
        right_prob = float(input("give right moving prob. sample: 0.1\n>> "))
        left_prob = float(input("give left moving prob. sample 0.1\n>> "))
        prob = [straight_prob,right_prob,left_prob]

    use_default_reward = input("type 'y' to use default reward. Default is 0
for normal state and 1000 for goal state\n>> ")
    if(use_default_reward == "y"):
        reward_normal_state = 0
        reward_goal_state = 1000
    else:
        reward_normal_state = int(input("give reward for normal states apart
from goal state\n>> "))
        reward_goal_state = int(input("give reward for goal states apart from
goal state\n>> "))
else:
    path_string = 'length of path'
    iteration_node = 'total nodes traversed '

if(predefined_maze == "y"):
    my_maze.CreateMaze(goal[0],goal[1],loopPercent=10,loadMaze=maze_name)
else:
    my_maze.CreateMaze(goal[0],goal[1],loopPercent=10,saveMaze = True)

print("maze solver is running. Grab a coffee or relax a bit :) ")

t1_start = process_time()
t1_wall_start = time.time()
if(algo == "policy"):
    import policy_iter
```

```
    final_path,count = 
policy_iter.find_path(my_maze,goal,discount_factor,prob,reward_normal_state,r
eward_goal_state)
elif(algo == "value"):
    import value_iter
    final_path,count = 
value_iter.find_path(my_maze,goal,discount_factor,prob,reward_normal_state,re
ward_goal_state)
elif(algo == "astar"):
    import a_star
    final_path,count = a_star.find_path(my_maze,goal)
elif(algo == "bfs"):
    import bfs
    final_path,count = bfs.find_path(my_maze,goal)
elif(algo == "dfs"):
    import dfs
    final_path,count = dfs.find_path(my_maze,goal)

t1_wall_stop = time.time()
t1_stop = process_time()


try:
    my_agent=agent(my_maze,footprints=True)

    my_maze.tracePath({my_agent:final_path})

    my_path=textLabel(my_maze,path_string,len(final_path)+1)
    my_iteration=textLabel(my_maze,iteration_node,count)
    my_algo = textLabel(my_maze,"algorithm name",algo)

    wall_time_taken = textLabel(my_maze,"Wall 
time",t1_wall_stop-t1_wall_start)
    cpu_execution_time = textLabel(my_maze,"CPU execution 
time",t1_stop-t1_start)
    my_maze.run()
except:
    print("error at line 73 to 77 in main.py")
```

DFS:

```
from pyamaze import maze, COLOR, agent, textLabel
```

```python
def handle_agent_movement(direction,square):
    next_move = {'E':(0,1),'W':(0,-1),'N':(-1,0),'S':(1,0)}
    return
(square[0]+next_move[direction][0],square[1]+next_move[direction][1])
def backtrace(temp_goal,begin,journey):
    final = {}
    while begin != temp_goal:
        final[journey[temp_goal]] = temp_goal
        temp_goal = journey[temp_goal]
    return final
def Depth_First(agent,goal):
    begin = (agent.rows,agent.cols)
    queue = [begin]
    visited = [begin]
    journey = {}
    count = 0
    while len(queue)>0:
        square = queue.pop() #last in first out or stack
        count = count+1
        if square == None:
            break
        try:
            for direction in agent.maze_map[square]:
                if agent.maze_map[square][direction] == 1:
                    next_square = handle_agent_movement(direction,square)
                    if next_square in visited:
                        continue
                    journey[next_square] = square
                    visited.append(next_square)
                    queue.append(next_square)
                    if(next_square == goal):
                        break
        except:
            print("error")
        if(next_square == goal):
            break
    temp_goal = goal
    final = backtrace(temp_goal,begin,journey)
    print("count: " + str(count))
    return final,count
```

```python
def find_path(my_maze,goal):
    path,count=Depth_First(my_maze,goal)
    return path,count
```

BFS:

```python
from pyamaze import maze, COLOR, agent, textLabel
import sys


def handle_agent_movement(direction,square):
    next_move = {'E':(0,1),'W':(0,-1),'N':(-1,0),'S':(1,0)}
    return
(square[0]+next_move[direction][0],square[1]+next_move[direction][1])
def backtrace(temp_goal,begin,journey):
    final = {}
    while begin != temp_goal:
        final[journey[temp_goal]] = temp_goal
        temp_goal = journey[temp_goal]
    return final
def Breadth_First(agent,goal):
    begin = (agent.rows,agent.cols)
    queue = [begin]
    visited = [begin]
    journey = {}
    count = 0
    while len(queue)>0:
        square = queue.pop(0) # first in first out
        count = count+1
        if square == None:
            break
        try:
            for direction in agent.maze_map[square]:
                if agent.maze_map[square][direction] == 1:
                    next_square = handle_agent_movement(direction,square)
                    if next_square in visited:
                        continue
                    journey[next_square] = square
                    visited.append(next_square)
                    queue.append(next_square)
                    if(next_square == goal):
                        break
        except:
            print("error")
```

```
        if(next_square == goal):
            break
    temp_goal = goal
    final = backtrace(temp_goal,begin,journey)
    print("count: " + str(count))
    return final,count

arg={}
for i in range(1,len(sys.argv)):
    k,v = sys.argv[i].strip().split("=")
    arg[k] = v


def find_path(my_maze,goal):
    path,count=Breadth_First(my_maze,goal)
    return path,count
```

A_Star:

```
from pyamaze import maze, COLOR, agent, textLabel

def my_heuristic(squareA,squareB):
    x1,y1=squareA
    x2,y2=squareB
    return abs(x1-x2) + abs(y1-y2)

def handle_agent_movement(direction,square):
    next_move = {'E':(0,1),'W':(0,-1),'N':(-1,0),'S':(1,0)}
    return
(square[0]+next_move[direction][0],square[1]+next_move[direction][1])
def backtrace(temp_goal,begin,journey):
    final = {}
    while begin != temp_goal:
        final[journey[temp_goal]] = temp_goal
        temp_goal = journey[temp_goal]
    return final
def a_star(agent,goal):
    begin = (agent.rows,agent.cols)
    previous_cost = {}

    total_cost = {}
    visited = [begin]
    goal = goal
```

```
    for i in range(1,agent.rows+1):
        for j in range(1,agent.cols+1):
            previous_cost[(i,j)] =  float('inf')
            total_cost[(i,j)] =  float('inf')
    previous_cost[begin] = 0
    total_cost[begin] = my_heuristic(begin,goal)


    queue = [(total_cost[begin],begin)]

    journey = {}
    count = 0

    while len(queue)>0:
        square_tuple = min(queue)
        queue.remove(square_tuple)
        square = square_tuple[1]
        count = count+1
        if square == None or square == goal:
            break
        try:

            for direction in agent.maze_map[square]:

                if agent.maze_map[square][direction] == 1:
                    next_square = handle_agent_movement(direction,square)
                    next_square_cost = previous_cost[square]+1 +
my_heuristic(next_square,goal)

                    if next_square in visited:
                        if next_square_cost < total_cost[next_square]:
                            previous_cost[next_square] =
previous_cost[square]+1
                            total_cost[next_square] = next_square_cost

queue.append((total_cost[next_square],next_square))
                            journey[next_square] = square
                    else:
                        previous_cost[next_square] = previous_cost[square]+1
                        total_cost[next_square] = previous_cost[square]+1 +
my_heuristic(next_square,goal)
                        queue.append((total_cost[next_square],next_square))
                        journey[next_square] = square
```

```
                        visited.append(next_square)

        except:
            print("error")
    temp_goal = goal
    final = backtrace(temp_goal,begin,journey)
    print("count: " + str(count))
    return final,count




def find_path(my_maze,goal):
    path,count=a_star(my_maze,goal)
    return path,count
```

Value Iteration:

```
from pyamaze import maze, COLOR, agent, textLabel
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.lines import Line2D

def handle_agent_movement(direction,square):
    next_move = {'E':(0,1),'W':(0,-1),'N':(-1,0),'S':(1,0)}
    return
(square[0]+next_move[direction][0],square[1]+next_move[direction][1])
def handle_agent_movement_prob(agent,direction,square,prob):
    next_move = {'E':(0,1),'W':(0,-1),'N':(-1,0),'S':(1,0)}
    moves = ['W','N','E','S']
    if(agent.maze_map[square][direction] == 1):
        straight_square =
(square[0]+next_move[direction][0],square[1]+next_move[direction][1])
    else:
        straight_square = square
    if(moves.index(direction) == 3):
        right_direction = moves[0]
    else:
        right_direction = moves[moves.index(direction)+1]
    if(agent.maze_map[square][right_direction] == 1):
```

```
        right_square =
(square[0]+next_move[right_direction][0],square[1]+next_move[right_direction]
[1])
    else:
        right_square = square

    left_direction = moves[moves.index(direction)-1]
    if(agent.maze_map[square][left_direction] == 1):
        left_square =
(square[0]+next_move[left_direction][0],square[1]+next_move[left_direction][1
])
    else:
        left_square = square

    P={"straight":{"prob": prob[0],"square":straight_square},"right":{"prob":
prob[1],"square":right_square},"left":{"prob": prob[2],"square":left_square}}

    return P

def val_iter(agent,rows,cols,discount,reward,delta,prob,goal):
    value = {}
    action={}
    steps={}
    for i in range(1,agent.rows+1):
        for j in range(1,agent.cols+1):
            value[(i,j)] =  0
    #value[goal] = 10
    error = float('inf')
    count = 0

    while(delta<error):
        count = count+1

        temp_error = float("-inf")
        for square in agent.maze_map:

            if(square == goal):
                value[square] = reward[goal]
                continue

            temp_score = float("-inf")
            for direction in agent.maze_map[square]:
                if agent.maze_map[square][direction] == 1:
```

```
                    next_square =
handle_agent_movement_prob(agent,direction,square,prob)
                    direction_score =
reward[square]+(discount*value[next_square["straight"]["square"]]*next_square
["straight"]["prob"])+(discount*value[next_square["right"]["square"]]*next_sq
uare["right"]["prob"])+(discount*value[next_square["left"]["square"]]*next_sq
uare["left"]["prob"])
                    if  direction_score> temp_score:
                        temp_score = direction_score
                        steps[square] = next_square["straight"]["square"]


            if (abs(temp_score-value[square]) > temp_error):
                temp_error = abs(temp_score-value[square])
            #asynchronous update of utilities
            value[square] = temp_score

        error = temp_error

    return steps,count

def
find_path(my_maze,goal,discount_factor,prob,reward_normal_state,reward_goal_s
tate):
    print("maze calculation running. Please give some time")
    rows = my_maze.rows
    cols = my_maze.cols
    reward = {}
    for i in range(1,rows+1):
        for j in range(1,cols+1):
            reward[(i,j)] =  reward_normal_state
    reward[goal] = reward_goal_state

    #m.CreateMaze(loopPercent=20,saveMaze=True)
    path,count =
val_iter(my_maze,rows,cols,discount_factor,reward,0.001,prob,goal)


    start = (rows,cols)
    final_policy = {}

    while start != goal:
        final_policy[start] = path[start]
```

```
        start = path[start]

    return final_policy,count
```

Policy Iteraton:

```
from pyamaze import maze, COLOR, agent, textLabel
import numpy as np
import copy

import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from matplotlib.lines import Line2D



def handle_agent_movement(direction,square):
    next_move = {'E':(0,1),'W':(0,-1),'N':(-1,0),'S':(1,0)}
    return
(square[0]+next_move[direction][0],square[1]+next_move[direction][1])

def handle_agent_movement_prob(agent,direction,square,prob):
    next_move = {'E':(0,1),'W':(0,-1),'N':(-1,0),'S':(1,0)}
    moves = ['W','N','E','S']
    if(agent.maze_map[square][direction] == 1):
        straight_square =
(square[0]+next_move[direction][0],square[1]+next_move[direction][1])
    else:
        straight_square = square
    if(moves.index(direction) == 3):
        right_direction = moves[0]
    else:
        right_direction = moves[moves.index(direction)+1]
    if(agent.maze_map[square][right_direction] == 1):
        right_square =
(square[0]+next_move[right_direction][0],square[1]+next_move[right_direction]
[1])
    else:
        right_square = square
```

```
    left_direction = moves[moves.index(direction)-1]
    if(agent.maze_map[square][left_direction] == 1):
        left_square =
(square[0]+next_move[left_direction][0],square[1]+next_move[left_direction][1
])
    else:
        left_square = square

    P={"straight":{"prob": prob[0],"square":straight_square},"right":{"prob":
prob[1],"square":right_square},"left":{"prob": prob[2],"square":left_square}}

    return P

def find_direction(square,next_square):
    move={(0,1):'E',(0,-1):'W',(-1,0):'N',(1,0):'S'}
    return move[(next_square[0]-square[0],next_square[1]-square[1])]

def pol_iter(agent,rows,cols,discount,reward,delta,policy,prob,goal):
    value = {}
    for i in range(1,agent.rows+1):
        for j in range(1,agent.cols+1):
            value[(i,j)] =  0
    value[goal] = reward[goal]

    error = float('inf')
    count = 0
    check_no_change = False
    temp_policy={}

    while True:
        count = count+1

        #policy evaluation

        while(delta<error):
            temp_error = float("-inf")

            for square in agent.maze_map:
                if (square == goal):
                    value[(square)] = reward[goal]
                    continue
                next_square = policy[square]
```

```
                temp_score = value[square]

                #asynchronous update of uitility

                if(
agent.maze_map[square][find_direction(square,next_square)] == 1):
                    next_square =
handle_agent_movement_prob(agent,find_direction(square,next_square),square,pr
ob)
                    direction_score =
reward[square]+(discount*value[next_square["straight"]["square"]]*next_square
["straight"]["prob"])+(discount*value[next_square["right"]["square"]]*next_sq
uare["right"]["prob"])+(discount*value[next_square["left"]["square"]]*next_sq
uare["left"]["prob"])
                    value[square] = direction_score

                if (abs(temp_score-value[square]) > temp_error):
                    temp_error = abs(temp_score-value[square])
            error = temp_error



        #policy improvement code

        for square in agent.maze_map:
            temp_score = float("-inf")

            if(square == goal):
                continue
            for direction in agent.maze_map[square]:

                if agent.maze_map[square][direction] == 1:
                    next_square =
handle_agent_movement_prob(agent,direction,square,prob)
                    direction_square =
reward[square]+(discount*value[next_square["straight"]["square"]]*next_square
["straight"]["prob"])+(discount*value[next_square["right"]["square"]]*next_sq
uare["right"]["prob"])+(discount*value[next_square["left"]["square"]]*next_sq
uare["left"]["prob"])

                    if direction_square > temp_score:
                        temp_score = direction_square
```

```
                        temp_policy[square] =
next_square["straight"]["square"]
        if(temp_policy == policy):
            break
        else:
            policy = copy.deepcopy(temp_policy)
            error = float('inf')

    return policy,count



def
find_path(my_maze,goal,discount_factor,prob,reward_normal_state,reward_goal_s
tate):

    rows = my_maze.rows
    cols = my_maze.cols
    reward = {}
    for i in range(1,rows+1):
        for j in range(1,cols+1):
            reward[(i,j)] =  reward_normal_state
    reward[goal] = reward_goal_state
    policy = {}


    for square in my_maze.maze_map:
        if(square == goal):
            continue
        for direction in my_maze.maze_map[square]:
            if my_maze.maze_map[square][direction] == 1:
                next_square = handle_agent_movement(direction,square)
                policy[square] = next_square
    prob = [1,0,0]
    path,count =
pol_iter(my_maze,rows,cols,discount_factor,reward,0.001,policy,prob,goal)

    start = (rows,cols)
    final_path = {}

    while start != goal:
        final_path[start] = path[start]
        start = path[start]
```

```
    return final_path,count
```