CS7IS2 – Artificial Intelligence – Assignment 1
Search and MDPs

Abhik Subir Bhattacharjee
22305544
MSc Computer Science (Intelligent Systems)
Trinity College Dublin

1. INTRODUCTION

In this assignment, the main goal is to implement a variety of Search and MDP algorithms on a custom maze and compare the performance of each algorithm against the others. The maze generator used for this assignment can generate maze of varying sizes and the different Search and MDP algorithms are implemented on those randomly generated mazes. The subsequent sections discuss the implementation of the Maze Generator, Search along with the MDP algorithms, presenting the results and finally comparing the performance of each algorithm implemented.

2. IMPLEMENTATION

2.1 MAZE GENERATOR

The maze generator used for this assignment is Pyamaze [1]. Pyamaze is an opensource python package which gives the freedom to the user to generate custom mazes of random sizes. This package was specifically used to satisfy the major requirement for this assignment – Generate mazes of varying sizes. This maze generator gives flexibility to the user to generate mazes of varying attributes, like generating horizontal or vertical pattern maze, or create a *Perfect Maze* wherein there is only one path to the goal or create a maze which leads to the goal using multiple paths. The package also gives us the flexibility to randomly set the start state of the maze and lets us choose the end or target state.

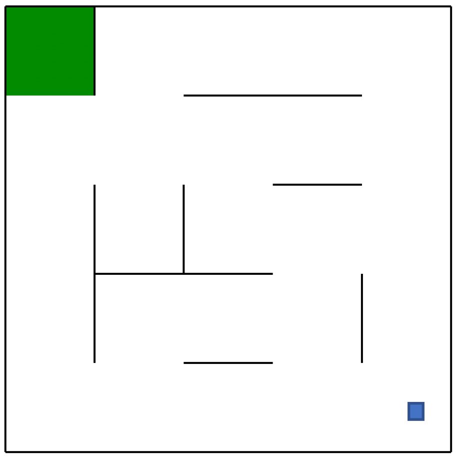
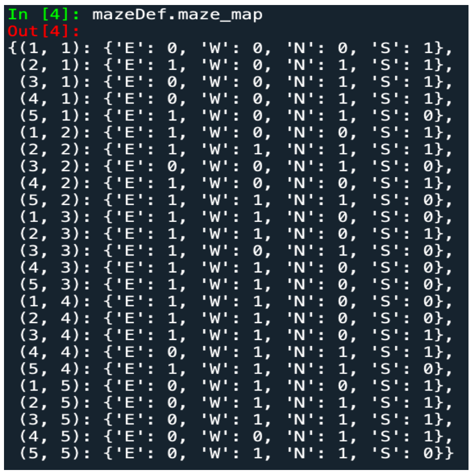


Figure 1.1: Sample 5x5 Maze



```
In [4]: mazeDef.maze_map
Out[4]:
{(1, 1): {'E': 0, 'W': 0, 'N': 0, 'S': 1},
 (2, 1): {'E': 1, 'W': 0, 'N': 1, 'S': 1},
 (3, 1): {'E': 0, 'W': 0, 'N': 1, 'S': 1},
 (4, 1): {'E': 0, 'W': 0, 'N': 1, 'S': 1},
 (5, 1): {'E': 1, 'W': 0, 'N': 1, 'S': 0},
 (1, 2): {'E': 1, 'W': 0, 'N': 0, 'S': 1},
 (2, 2): {'E': 1, 'W': 1, 'N': 1, 'S': 1},
 (3, 2): {'E': 0, 'W': 0, 'N': 1, 'S': 0},
 (4, 2): {'E': 1, 'W': 0, 'N': 0, 'S': 1},
 (5, 2): {'E': 1, 'W': 1, 'N': 1, 'S': 0},
 (1, 3): {'E': 1, 'W': 1, 'N': 0, 'S': 0},
 (2, 3): {'E': 1, 'W': 1, 'N': 0, 'S': 1},
 (3, 3): {'E': 1, 'W': 0, 'N': 1, 'S': 0},
 (4, 3): {'E': 1, 'W': 1, 'N': 0, 'S': 0},
 (5, 3): {'E': 1, 'W': 1, 'N': 0, 'S': 0},
 (1, 4): {'E': 1, 'W': 1, 'N': 0, 'S': 0},
 (2, 4): {'E': 1, 'W': 1, 'N': 0, 'S': 0},
 (3, 4): {'E': 1, 'W': 1, 'N': 0, 'S': 1},
 (4, 4): {'E': 0, 'W': 1, 'N': 1, 'S': 1},
 (5, 4): {'E': 1, 'W': 1, 'N': 1, 'S': 0},
 (1, 5): {'E': 0, 'W': 1, 'N': 0, 'S': 1},
 (2, 5): {'E': 0, 'W': 1, 'N': 1, 'S': 1},
 (3, 5): {'E': 0, 'W': 1, 'N': 1, 'S': 1},
 (4, 5): {'E': 0, 'W': 0, 'N': 1, 'S': 1},
 (5, 5): {'E': 0, 'W': 1, 'N': 1, 'S': 0}}
```

Figure 1.2: Maze Map of the 5x5 Maze

The Figure 1.1 above shows a sample *5x5* maze generated by the Pyamaze package with multiple paths to the goal. In the maze shown above, the cell *(5, 5)* is the starting node of the maze whereas the node *(1, 1)* is the target node. The maze generator also gives us the flexibility to place an *Agent* inside the maze which can be used to show the track or path taken by various algorithms to reach the goal. The blue box at the bottom right of the Figure 1.1 is a sample agent on the starting node. The package also provides a maze map of each state (Figure 1.2). The maze map is used to identify the possible next nodes in which an agent can traverse. The next nodes can be identified by the key value pair provided for each node, wherein if the value is 0 for a direction, it implies that if the agent takes that direction, it would be met with a wall and that direction can't be explored further.

For this assignment, I have kept the *Loop Percentage* as 100 to create a maze which has *multiple paths* leading to the goal every time. This design decision is specifically taken to test various search algorithms and the distinct steps taken by each algorithm to reach the goal. Based on the path taken by a given algorithm, its performance can be measured, and effective comparison can be made between the algorithms.

2.2 SEARCH ALGORITHMS

2.2.1 DEPTH FIRST SEARCH (DFS)

Depth first search (DFS) is an un-informed search algorithm that expands and explores the deepest unexpanded node first. This search algorithm is implemented by using the **L**ast **I**n **F**irst **O**ut (LIFO) strategy of queue, in other words, a stack. This algorithm starts at the designated first or starting node and explores the first branch completely to check for the goal before it starts backtracking. This search for goal is carried out in every branch recursively until the goal state is achieved. Based on its definition, DFS can be a good choice of tree search algorithm where the solution is likely to be found in a *Deeper* state. However, this choice of algorithm may not be a good choice if the target state is likely to be present near the root node. Additionally, this algorithm would fail to converge in cases of infinite depth spaces or spaces with loops *(Ref: Lecture Notes).*

The figure below (Figure 2) shows the traversal of an agent from the start node *(10, 10)* to the target node *(3, 2)* in a *10x10* maze. The pseudo code is also presented below for this DFS maze search algorithm.

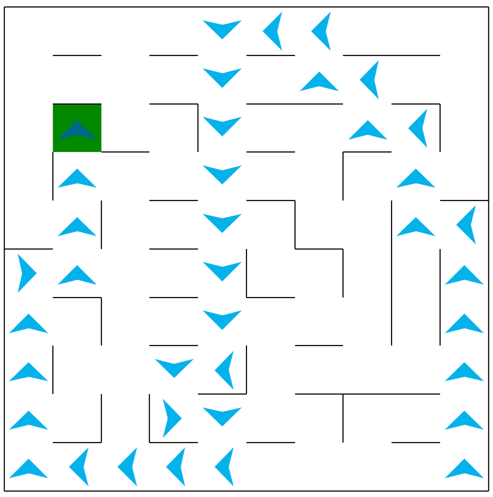


Figure 2: DFS on *10x10* Maze

```
startNode <- x, y
target <- tx, ty
stack[startNode]
path{}
while len(stack) > 0:
  currNode = stack.pop()
  if currNode = target: break;
  for direction in 'NSEW':
     if currNode[direction] exists:
        nextNode = currNode[direction]
        stack.append(nextNode)
        path[nextNode] = currNode
End
```

Code 1: Pseudo Code for DFS (Maze Search)

2.2.2 BREADTH FIRST SEARCH (BFS)

The Breadth First Search (BFS) technique uses the strategy to explore the shallowest node first as opposed to DFS. This algorithm first explores all the nodes at a given depth before moving on to the next level. The search algorithm is implemented using the **F**irst **I**n **F**irst **O**ut (FIFO) strategy. The BFS technique explores the shallowest nodes first and hence the new successors go to the end of the queue and are explored at a later stage. BFS is complete if the number of nodes is finite and optimal if the cost of each step is uniformly set to 1 *(Ref: Lecture Notes)*. BFS would not be the best choice of search algorithm if the target node is expected to be found at a *Deeper* state. Additionally, BFS is also memory expensive, and space can be a huge problem for large tree searches.
The figure below (Figure 3) shows the traversal of an agent from the start node *(10, 10)* to the target node *(3, 2)* in a *10x10* maze. The pseudo code is also presented below for this BFS maze search algorithm.

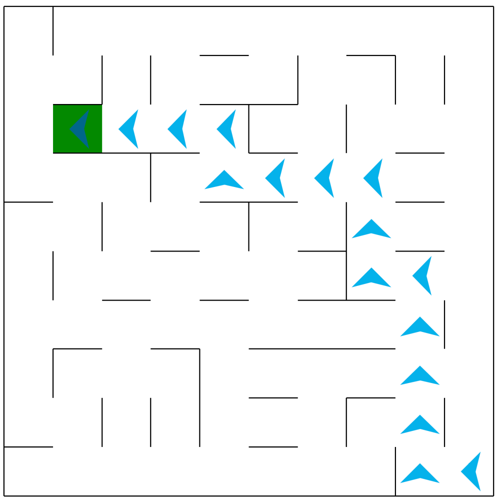


Figure 3: BFS on *10x10* Maze

```
startNode <- x, y
target <- tx, ty
queue[startNode]
path{}
while len(queue) > 0:
  currNode = queue.pop()
  if currNode = target: break;
  for direction in ‘NSEW’:
     if currNode[direction] exists:
        nextNode = currNode[direction]
        queue.append(nextNode)
        path[nextNode] = currNode
End
```

Code 2: Pseudo Code for BFS (Maze Search)

2.2.2 A^* (STAR) SEARCH

A^* Search is a weighted graph and informed search algorithm which is technically is a combination of Uniform cost search and Greedy Best First Search algorithms. A^* uses a heuristic function to calculate and estimate the distance from current node to the target. A^* is said to be a combination of Uniform Cost Search and Greedy Best First search algorithm for the following reasons:

- *Uniform Cost* orders by path cost *g(n)*
- *Greedy* orders by Goal Proximity *h(n)*
- A^* orders by the sum $f(n) = g(n) + h(n)$

This is interpreted as A^* is eventually the actual cost of the path so far plus the estimated cost to the goal *(Ref: Lecture Notes)*. The major advantage of using A^* is that it very well avoids the local maxima traps, and its major goal is to avoid the paths that are already expensive. A^* uses a priority queue and calculates the node cost using two possible ways – Euclidian Distance and Manhattan Distance. For this assignment I have used Manhattan distance as the heuristic for the following two reasons:

- Manhattan distance is a consistent heuristic which never overestimates the actual distance to the target node

- It is also considered to be faster than Euclidian distance as the later explores diagonal along with the adjacent nodes whereas Manhattan distance only considers the top-down and sideway movements.

The figure below (Figure 4) shows the traversal of an agent from the start node *(10, 10)* to the target node *(1, 1)* in a *10x10* maze. The pseudo code is also presented below for this BFS maze search algorithm.

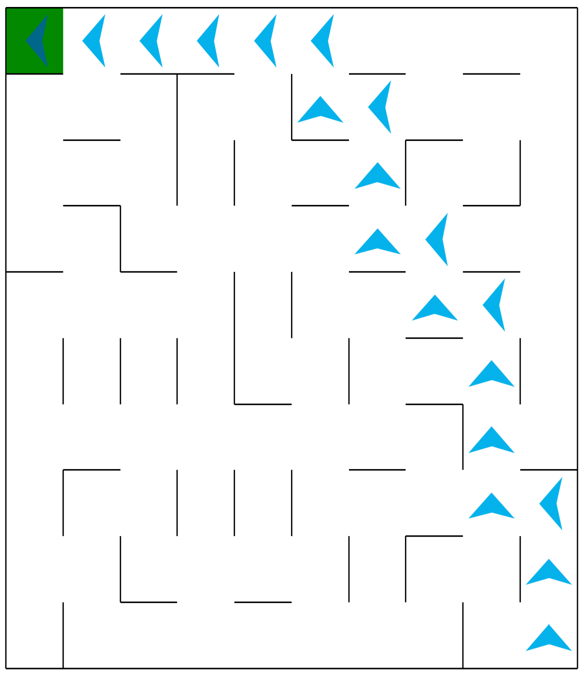


Figure 4: A* on *10x10* Maze

```
startNode <- x, y
target <- tx, ty
g[allNodes] <- inf; g[startNode] <- 0
f[allNodes] <- inf;
f[start] <- Manhattan(start, target)
priorityQueue.put((Manhattan(start,target)),
Manhattan(start,target),start))
while !priorityQueue.empty():
   currCell = priorityQueue[2]
   if currCell = target: break;
   for direction in 'NSEW':
        if currNode[direction] exists:
        nextNode = currNode[direction]
        tempg=g[currCell]+1
        tempf =tempg + Manhattan
                      (nextNode,target)
        if tempf < f[nextNode]:
           g[nextNode] = tempg;
           f[nextNode] = tempf;
           path[nextNode] = currCell
```

Code 3: Pseudo Code for A* (Maze Search)

2.3 MARKOV DECISION PROCESS

A Markov Decision Process (MDP) is a sequential decision problem in a stochastic environment with a state transition model and additive rewards. Every MDP consists a set of states' *S*, a set of actions for each state denoted as *ACTIONS (s)*, a transition model *P(s'|s,a)* and a reward function *R(s)* [2]. The goal of a given MDP is to maximize the expected Reward. MDPs are non-deterministic search problems, and it always tries to identify the optimal policy for each state. Policy is nothing but the action specified for each state if the agent is currently present in that state. In other words, choice of action set for each state. An optimal policy is the policy that yields the highest expected utility wherein utility is the sum of discounted rewards. A given MDP can be solved using two methods described below – Value Iterations and Policy Iterations

2.3.1 VALUE ITERATIONS

The basic idea of value iterations is to calculate the utility of each state and then use the utility of each state to identify the best policy for the given state. The utility of any given state can be calculated using the Bellman Equation given below –

$$U(S) = R(S) + \gamma \max_{a \in A(s)} P(s' \mid s, a) U(s') \; ..... (1)$$

In the equation above, *U(S)* is the utility of the current state, *R(S)* is the reward for the current state, γ is the discount factor, *P(S'|a,S)* is the state transition probabilities and *U(S')* is the utility of the next state. The main advantage of Value Iteration is its ability to handle MDPs with stochastic transitions and rewards. On the other hand, one of the major disadvantages of Value Iteration is that the agent is assumed to have complete knowledge of the MDP [2]. Value Iteration is expected to have full convergence and it can be an optimal choice of algorithm for a problem containing discrete number of states and discrete actions for each state. This condition is satisfied for the maze search problem that we are tackling.

For the maze search problem, I have given an option to set the transition probabilities in two ways:

- Stochastic – where the agent is expected to act *'N'* 80% time, *'W'* 10% time, *'E' and 'S'* 5% time each.
- Deterministic – where the agent is expected to act and explore in all directions.

The transition probabilities are set such because the start node is to the bottom right of the maze and the target goal node is to the north-western frontier from the start. To solve the MDP, I have used the pseudo code mentioned in Code 4 below which in turn is inspired from the chapter 'Making Complex Decisions' by Russel and Norvig [2]. The reward for target state is set as the highest whereas the living reward for the states in MDP is set to -4. The discount γ is set to 0.8 which is in turn is used to recalculate the utility of the state. The discount factor is deliberately kept higher to encourage faster convergence. The Figure 4 below shows the traversal of agent starting from node *(10, 10)* till goal *(3, 2)* using value iterations.

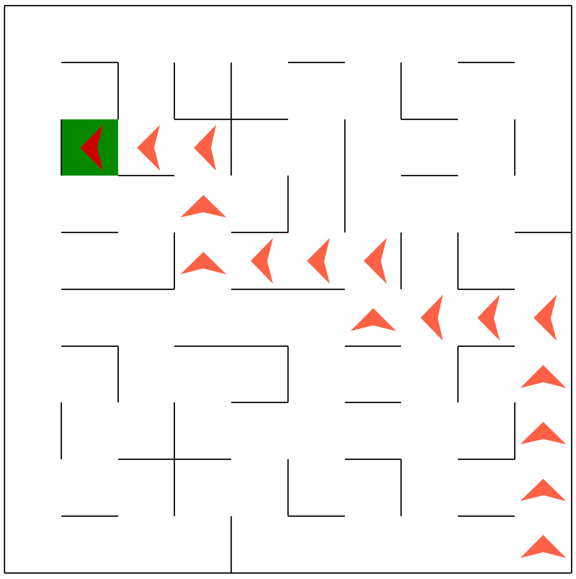
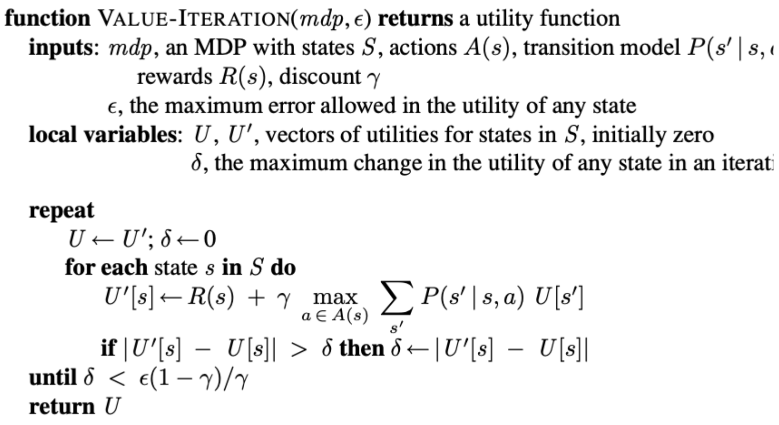


Figure 5: Value Iteration on *10x10* Maze



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

  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 $U$
```

Code 4: Pseudo Code for Value Iteration (Reference [2])

2.3.2 POLICY ITERATIONS

The policy iteration algorithm is based on the notion that to find the optimal policy, it is not important to find the optimal utility of each state. As there are only a finite option of policies in a finite-state MDP, is assumed that the search over all policies would terminate after a finite number of steps. Policy Iteration algorithm basically fluctuates between the following two steps:

- **Policy Evaluation:** Calculate utilities for fixed policies until convergence.

- **Policy Improvement:** Update the calculated policies for each state using one-step look ahead utilities as future values.

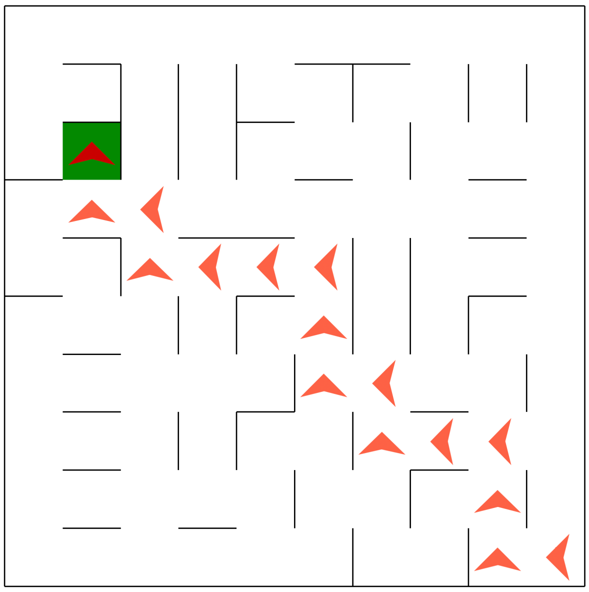
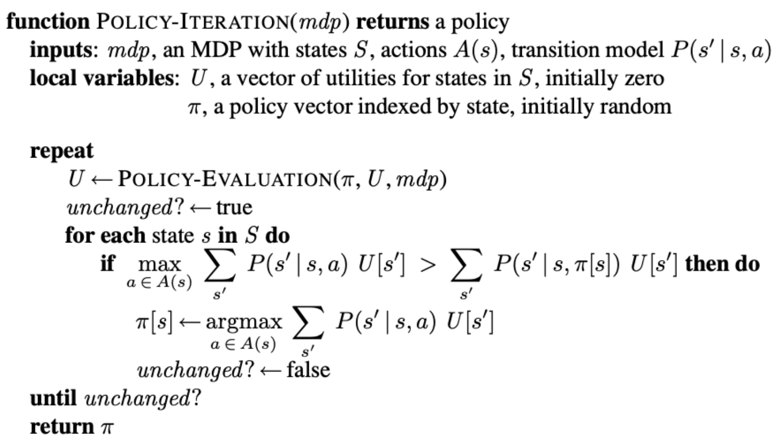


Figure 6: Policy Iteration on *10x10* Maze



```
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
                   $\pi$, a policy vector indexed by state, initially random

  repeat
     $U \leftarrow$ POLICY-EVALUATION($\pi, U, mdp$)
     $unchanged? \leftarrow$ 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 $\pi$
```

Code 5: Pseudo Code for Policy Iteration (Reference [2])

The first step in Policy is the policy evaluation. This step is just like the value iteration but with fixed policies by solving recursive Bellman equations. The next step is to improve the policy first by evaluating it and then improving it by acting greedily, using one-step look ahead based on the utility of the next state. *(Ref: Lecture Notes, [2]).* The function used to carry out this policy improvement as mentioned in the pseudo code is as follows: $\pi[s] \leftarrow \underset{a \in A(s)}{argmax} \sum_{s'} P(s'|s,a)U[s']$.

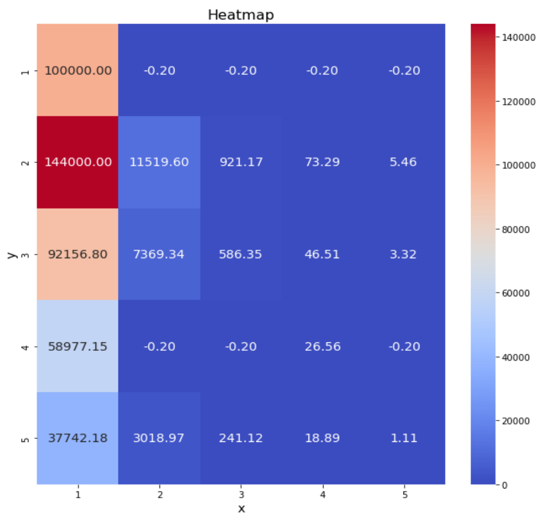


Figure 7.1: Final Utilities for each state – Value Iteration

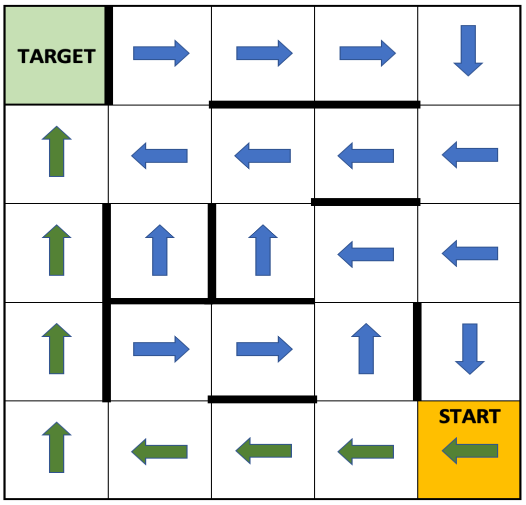


Figure 7.2: Final Policies for each state – Policy Iteration

The process of policy improvement is carried out iteratively and the process is terminated when the improvement step yields no significance improvement in utilities. This process of policy improvement is extremely simple because of the fact that each state is fixed by the policy.

In terms of design, like value iteration, I have given an option to set the transition probabilities in two ways – Stochastic and Deterministic. The hyperparameters of reward, discount factor, max error, etc are kept constant with value iteration. Please refer to the image 7 demonstrating the utility of each state generated by the value iteration algorithm along the optimal policy generated by the policy iteration algorithm for a *5x5* maze.

3 RESULTS AND DISCUSSION

3.1 COMPARING PERFORMANCE OF SEARCH ALGORITHMS (BFS vs DFS vs A*)

In this section, I have tested the search algorithms on a variety of maze, each of a different size to compare the performance of each algorithm. As a metric, I have used the number of nodes in the path taken by an algorithm and the time taken by each algorithm to reach the goal state to judge the performance of the algorithms. The algorithms were tested over the following mazes: *5x5, 10x10, 15x15, 30x30, 25x50 and 50x50*. This analysis gave a holistic view of the performance for all the three search algorithms. The figure below shows the path taken by all three search algorithms together over a *30x30* maze.

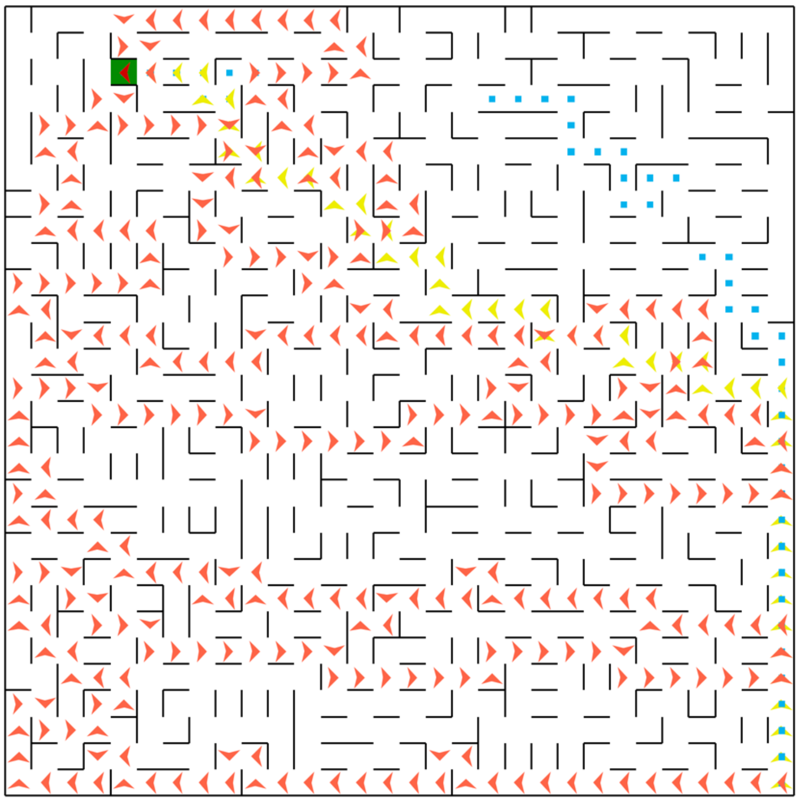


Figure 8: BFS vs DFS vs A* on *30x30* maze (Red Arrow: DFS, Yellow Arrow: BFS, Blue Square: A*)

From the above figure, we can see that the path taken by DFS (Red arrow) is the largest and it also explores the maximum nodes. From the perspective of number of nodes taken, BFS (Yellow arrow) is better as it tries to converge to the goal sooner. The best amongst all is A* as it traverses through the least number of nodes in order to reach to the goal.

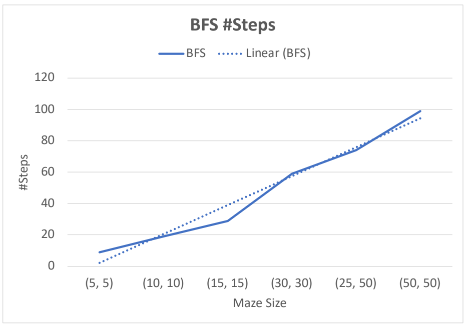


Figure 9.1: #Steps for Different Mazes in BFS

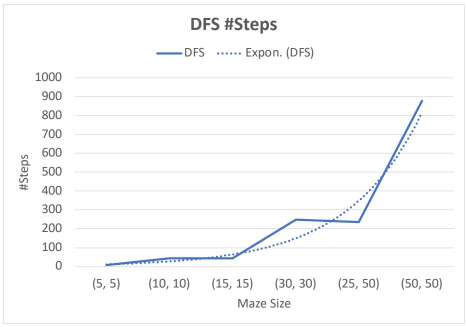


Figure 9.2: #Steps for Different Mazes in DFS

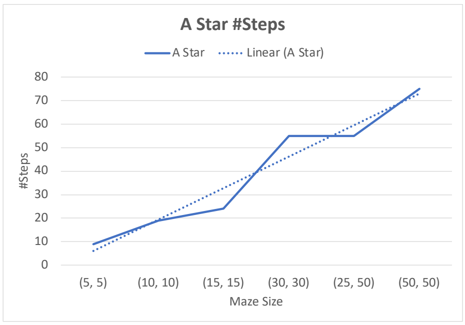


Figure 9.3: #Steps for Different Mazes in A*

From the plots shown above in Figure 9, we can see that both BFS and A* follow a linear trend for the number of steps, that is, as we go on increasing the size of the maze, the number of steps taken by both of those algorithms to converge to the goal also increases linearly. On the other hand, if we consider DFS, the increase in the number of steps taken by the algorithm to converge to the goal is exponential. This also hints that DFS is memory expensive and is not a good choice in cases where the target node is near the root node. Please note that the target node in all the cases from figure 9 were *(1, 1)*.

A similar kind of trend was also observed in the time taken by each algorithm to converge to the target node for different maze sizes. Please refer to figure 10 given below. From the plots given, we can see that A* converges to the target sooner when compared to BFS and DFS. BFS is the next best and DFS takes the largest amount of time to converge to the target node. One peculiar observation is that for A*, the time taken follows a linear trend whereas for BFS and DFS both, it follows an exponential trend. Hence the time taken by BFS and DFS to converge to the target node goes on increasing exponentially when we increase the size of the maze.

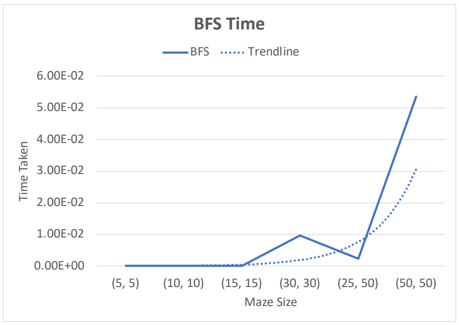


Figure 10.1: Time taken by BFS for Different Mazes

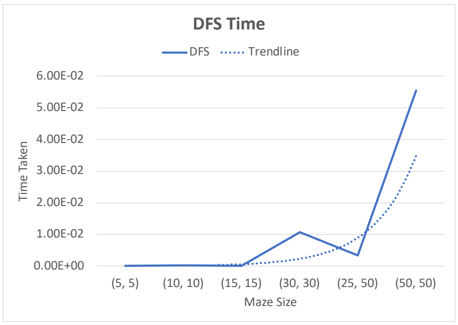


Figure 10.2: Time taken by DFS for Different Mazes

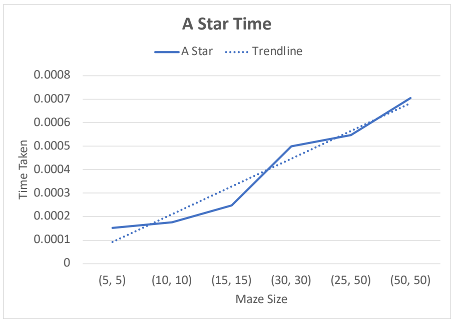


Figure 10.3: Time taken by A* for Different Mazes

Hence, when the performance is compared between the three search algorithms, we can conclude that A* is superior search algorithm and can be the optimal choice for any maze size as it is not memory expensive since it takes the least number of steps to reach the target node and the time taken to converge to the goal node is the smallest.

3.2 COMPARING PERFORMANCE OF MDP ALGORITHMS (VALUE vs POLICY ITERATION)

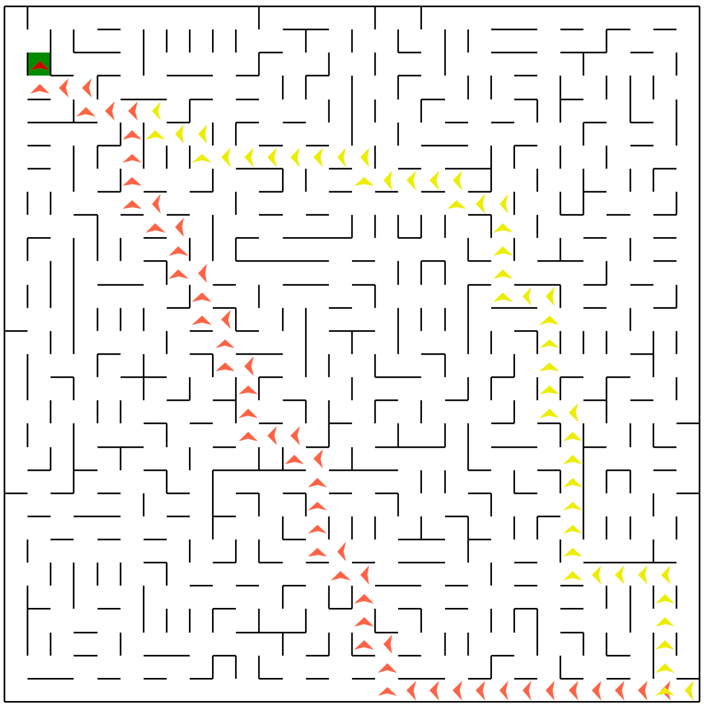


Figure 11: MDP Value vs Policy Iteration on *30x30* maze (Red Arrow: Policy Iteration, Yellow Arrow: Value Iterations

The performance of the MDP algorithms is benchmarked using the similar metrics as used for the search algorithms. The MDP algorithms were also subjected to a wide range of maze sizes in order to test their performance is both deterministic and stochastic environments. The maze plot shown in figure 11 above shows the path taken by Value Iteration algorithm (Yellow arrow) and the Policy Iteration algorithm in a stochastic setting. Based on our theoretical knowledge, it is expected that the number of steps required by value iteration should be more when compared to the policy iteration algorithm. To test that understanding, I have plotted the graph containing the number of steps required by value iteration and policy iteration for different maze sizes in figure 11 below.

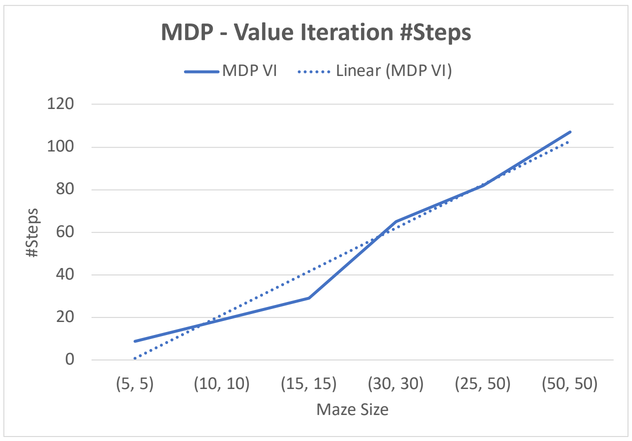


Figure 12.1 #Steps for Different Mazes in Value Iteration

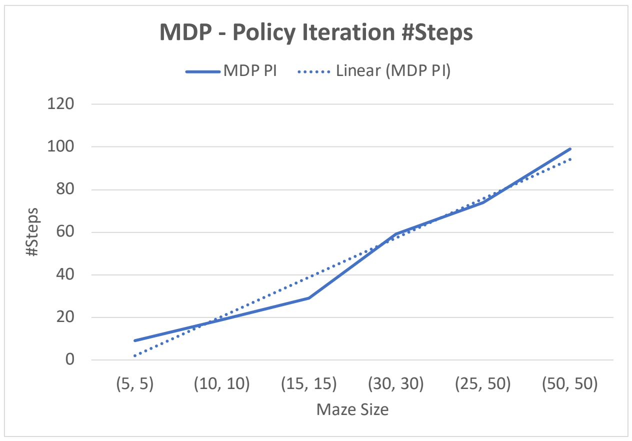


Figure 12.2 #Steps for Different Mazes in Policy Iteration

From the plot above we cannot make much difference between the number of steps taken by value iteration and policy iteration as both roughly take the similar number of steps to converge to the target or goal node. Both follow a linear trend in the number of steps and if we look minutely, we can see that Policy iteration performs a bit better when compared to value iteration as the number of steps required to converge is slightly lesser.

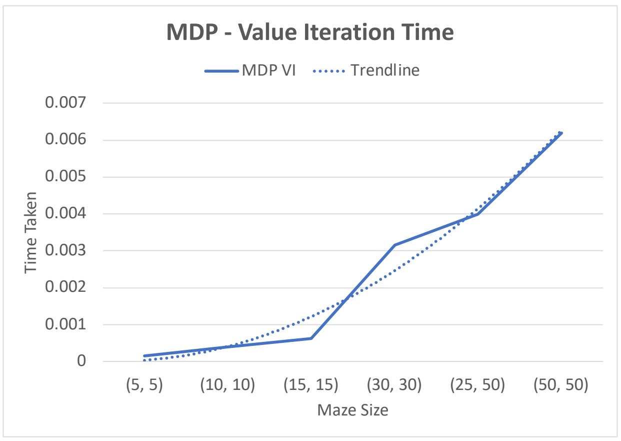


Figure 13.1 Time taken by Value Iteration for Different Mazes

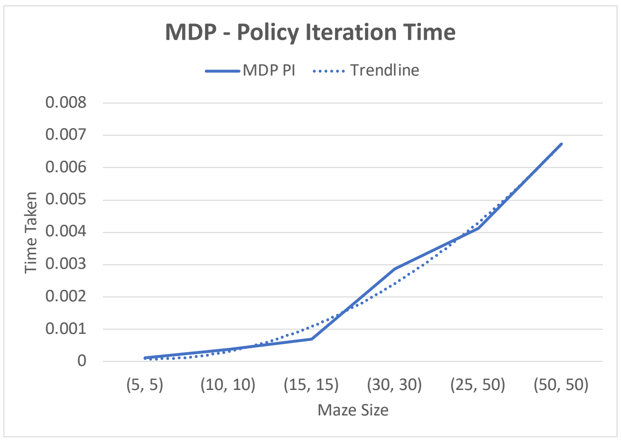


Figure 13.2 Time taken by Policy Iteration for Different Mazes

Similarly, when the time taken to converge each algorithm is considered, we can see that policy iteration performs a bit better when compared to value iteration. This is mainly because policy iteration has only a finite option of policies in a finite-state MDP, and they eventually converge in a finite number of steps. Both the algorithms follow a linear trend when time is considered as a metric. Based on the preliminary understanding of the working of these algorithms, we can conclude that policy iteration would be a better choice for the maze search problem.

3.3 COMPARING PERFORMANCE OF MDP ALGORITHMS AND SEARCH ALGORITHMS

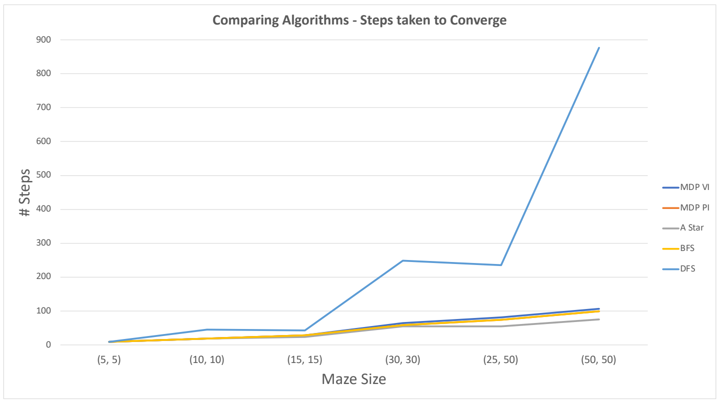


Figure 14.1 #Steps taken by Search and MDP Algorithms for Different Mazes

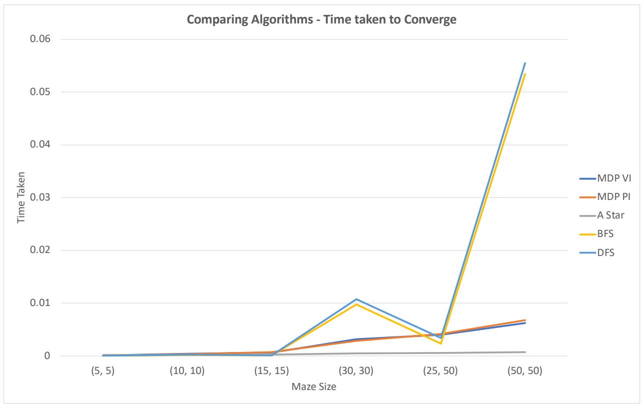


Figure 14.2 Time taken by Search and MDP Algorithms for Different Mazes

When the similar kind of analysis where the number of steps and the time taken by each algorithm to converge to the target nodes are compared, we can see that DFS performs the worst for both the metrics we have considered for this assignment. BFS takes lesser number of steps to converge to the goal, but the time taken to complete the process is relatively higher when compared to A* and the MDP algorithms. The MDP algorithms – Value Iteration and Policy Iteration have comparable performance for the maze solver problem, and both have superior performance when compared to BFS and DFS for both the metrics – number of steps and total time taken to converge. In this problem of solving the maze and finding the optimal path, A* search algorithm performs the best – both in terms of the steps and time taken. The same observation can be reinforced from the plots given in figure 14. The plot given below shows the path taken by the agent for all the 5 algorithms in a *30x30* maze wherein the start node is *(30, 30)* and the goal node is *(1, 1)*.

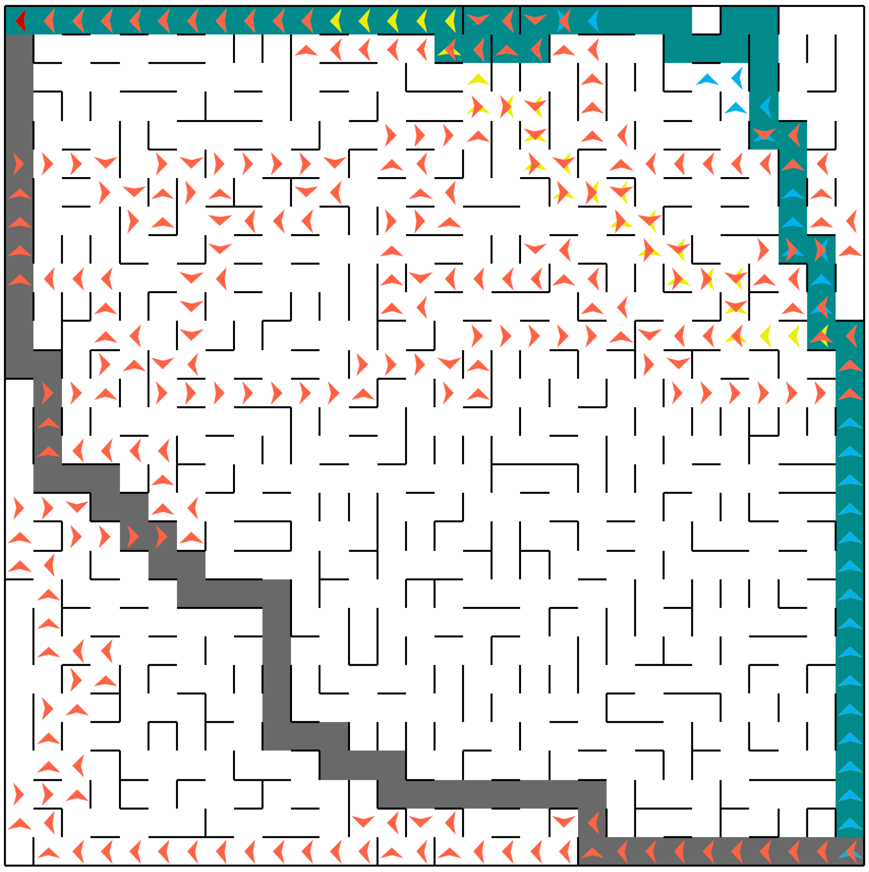


Figure 15: Search and MDP algorithms paths on a *30x30* maze

4 CONCLUSIONS

In this assignment, we have implemented three search algorithms namely BFS, DFS and A* along with the MDP algorithms including Value Iterations and Policy Iterations. The algorithms were executed on a variety of maze sizes to compare their performance against one another. Time and number of steps taken by the algorithm to converge to the goal were considered as the key metrics to compare the performance of the algorithms. From all the five algorithms implemented, DFS performed in a least optimal way taking both the metrics in consideration, whereas A* performed the best in all the situations. In this case, A* would be the best algorithm to consider traversing the agent from start node to the goal nide in a maze. Both the MDP algorithms had a similar performance, but A* fared a bit well compared to both.

NOTE:

Please note that if in case the MDP algorithms fail to converge and throws a *'Key Error'*, we need to change the rewards set for the independent algorithms. The current state of the algorithms is designed to work optimally for mazes till the range of 50x50. The rewards can be changed from the independent scripts of value iteration and policy iteration from the following lines. Details below –
CS7IS2-AI-Maze-Solver/MazeSolver/classMDP.py : Lines 91 and 171
CS7IS2-AI-Maze-Solver/OutTakes/mdpInit.py : Lines 90 and 170

REFERENCE:

[1] Pyamaze GitHub Repository: https://github.com/MAN1986/pyamaze

[2] Russell, Stuart J., et al. Artificial Intelligence: A Modern Approach. 3rd ed. Upper Saddle River, NJ, Prentice Hall, 2010.

[3] aima-code: Implementation of algorithms from Russell And Norvig's "Artificial Intelligence - A Modern Approach". GitHub Repository: https://github.com/aimacode/aima-python

[4] MDP_VI_PI_Q-learning_AIMA.ipynb by Tirthajyoti Sarkar. Link: https://github.com/tirthajyoti/RL_basics

[5] Markov Decision Process by Joseph Su. Department of Computer Science, Georgia Institute of Technology. Link: https://jsu800.github.io/docs/ml_mdp.pdf

[6] MazeMDP by Sally Gao. GitHub Repository: https://github.com/sally-gao/mazemdp

APPENDIX:

[1] Class to Create the Maze, Agent, and Labels

```
Class
mazeCreate:

          def agntMake(self, mazeDef, traceShape, traceColor, fill = False):
              agnt = agent(mazeDef, footprints=True, shape = traceShape, color =
          traceColor, filled=fill)
              return agnt

          def stepCount(self, mazeDef, name, algo):
              stepLabel = textLabel(mazeDef, f'{name} Steps: ', len(algo)+1)
              return stepLabel

          def algoTime(self, mazeDef, name, delta):
              stepLabelT = textLabel(mazeDef, f'{name} Time: ', round(delta, 5))
              return stepLabelT
```

[2] Function to traverse to the next feasible node from current node based on Direction

```
def mazeTrace(self, mazeDef, currentNode, direction):
        if direction == 'E':
            return (currentNode[0], currentNode[1]+1)
        elif direction == 'W':
            return (currentNode[0], currentNode[1]-1)
        elif direction == 'N':
            return (currentNode[0]-1, currentNode[1])
        elif direction == 'S':
            return (currentNode[0]+1, currentNode[1])
```

[2] Function to implement BFS and DFS Algorithms

```
def aiAlgo(self, algo, mazeDef, xtarget, ytarget):
        start = time.time()
        startNode = (mazeDef.rows, mazeDef.cols)
        target = (xtarget, ytarget)
        cell = (xtarget, ytarget)
        container = [startNode]

        adjNode = [startNode]

        algoPath = {}
        tracePath = {}
        while len(container) > 0:
            if algo == 'DFS':
                currentNode = container.pop() #Stack Pop
            elif algo == 'BFS':
                currentNode = container.pop(0) #Queue Pop
            if currentNode == target:
                break
            for direction in 'NSEW':
                if mazeDef.maze_map[currentNode][direction]==True:
                    nextNode = self.mazeTrace(mazeDef, currentNode, direction)
                    if nextNode in adjNode:
                        continue
                    adjNode.append(nextNode)
                    container.append(nextNode)

                    algoPath[nextNode] = currentNode

        while cell != startNode:
            tracePath[algoPath[cell]] = cell
            cell = algoPath[cell]
        end = time.time()
        return tracePath, (end-start)
```

[3] Function to Calculate Manhattan Distance

```
def calNodeCost(self, node1, node2):
        nodex1,nodey1 = node1
        nodex2,nodey2 = node2
        return (abs(nodex1 - nodex2) + abs(nodey1 - nodey2))
```

[4] Function to implement A*

```
def aStarInit(self, mazeDef, xtarget, ytarget):
        startNode = (mazeDef.rows, mazeDef.cols)
        nodeDist = dict.fromkeys(mazeDef.grid, float('inf'))
        totalNodeCost = dict.fromkeys(mazeDef.grid, float('inf'))
        return startNode, nodeDist, totalNodeCost

def aStar(self, mazeDef, xtarget, ytarget):
        start = time.time()
        startNode,  nodeDist,  totalNodeCost  =  self.aStarInit(mazeDef,  xtarget,
ytarget)
        nodeDist[startNode] = 0
        totalNodeCost[startNode] = self.calNodeCost(startNode, (xtarget, ytarget))
        algoPath = {}
        tracePath = {}
        nextNode = None
        tempNodeDist = None
        tempTotalNodeCost = None
        currentNode = None
        cell = (xtarget, ytarget)
        container = PriorityQueue()
        container.put((self.calNodeCost(startNode,      (xtarget,      ytarget)),
self.calNodeCost(startNode, (xtarget, ytarget)), startNode))

        while not container.empty():
            currentNode = container.get()[2]

            if currentNode == (xtarget, ytarget):
                break
            for direction in 'NSEW':
                if mazeDef.maze_map[currentNode][direction]==True:
                    nextNode    =    searchAlgo().mazeTrace(mazeDef,    currentNode,
direction)

                tempNodeDist = nodeDist[currentNode] + 1
                tempTotalNodeCost   =   tempNodeDist   +   self.calNodeCost(nextNode,
(xtarget, ytarget))

                if tempTotalNodeCost < totalNodeCost[nextNode]:
                    nodeDist[nextNode] = tempNodeDist
                    totalNodeCost[nextNode] = tempTotalNodeCost
                    container.put((tempTotalNodeCost,    self.calNodeCost(nextNode,
(xtarget, ytarget)), nextNode))
                    algoPath[nextNode] = currentNode

        while cell != startNode:
            tracePath[algoPath[cell]] = cell
            cell = algoPath[cell]
        end = time.time()
        return tracePath, (end-start)
```

[5] Function to set MDP Actions

```
def mdpVIActions(self, mazeDef, isStochastic = True):
        self.stochastic = isStochastic
        for key, val in mazeDef.maze_map.items():
            self.ACTIONS[key] = [(k, v) for k, v in val.items() if v == 1]

        for k, v in self.ACTIONS.items():
```

```
                    self.ACTIONS[k] = dict(v)

                if self.stochastic:
                    for key, val in self.ACTIONS.items():
                        for k, v in val.items():
                            if k == 'N':
                                val[k] = 0.80
                            elif k == 'W':
                                val[k] = 0.1
                            elif k == 'E':
                                val[k] = 0.05
                            elif k == 'S':
                                val[k] = 0.05
                else:
                    for key, val in self.ACTIONS.items():
                        for k, v in val.items():
                            if k == 'N':
                                val[k] = 1
                            elif k == 'W':
                                val[k] = 1
                            elif k == 'E':
                                val[k] = 1
                            elif k == 'S':
                                val[k] = 1

                self.U = {state: 0 for state in self.ACTIONS.keys()}
                self.U[self.target[0]] = 1

                return self.ACTIONS, self.U
```

[6] Function to implement Value Iteration

```
        def mdpValIter(self, xTarget, yTarget, reward, gamma, error, mazeDef, stochastic =
        True):
                REWARD, DISCOUNT, MAX_ERROR, ACTIONS, U, target, policy, tracePath, algoPath
        = self.mdpVIInit(xTarget, yTarget, reward, gamma, error)
                ACTIONS, U = self.mdpVIActions(mazeDef, stochastic)
                while True:
                    delta = 0
                    for state in ACTIONS.keys():
                        if state == target[0]:
                            continue
                        max_utility = float("-infinity")
                        max_action = None
                        for action, prob in ACTIONS[state].items():
```

```
                    for direction in action:
                        if mazeDef.maze_map[state][direction]==True:
                            next_state = self.mazeTrace(mazeDef, state, direction)
                    utility = 0
                    reward = REWARD
                    if next_state == target[0]:
                        reward = 100000
                    utility = reward + DISCOUNT*(prob*U[next_state])
                    if utility > max_utility:
                        max_utility = utility
                        max_action = action
                delta = max(delta, abs(max_utility - U[state]))
                U[state] = max_utility
                policy[state] = max_action
            if delta < MAX_ERROR:
                break
            return U, policy, ACTIONS
```

[7] Function to implement Policy Iteration

```
def mdpPolIter(self, x, y, tarx, tary, mazeDef):
        start = time.time()
        target = [(tarx, tary)]
        actions = {}
        for key, val in mazeDef.maze_map.items():
            actions[key] = [(k, v) for k, v in val.items() if v == 1]

        for k, v in actions.items():
            actions[k] = dict(v)

        U = {state: 0 for state in actions.keys()}
        U[target[0]] = 10**(8)
        policy = {s: random.choice('NSEW') for s in actions.keys()}

        REWARD = {state: -40 for state in actions.keys()}
        REWARD[target[0]] = 10**(8)
        DISCOUNT = 0.9
        is_policy_changed = True
        iterations = 0

        while is_policy_changed:
            is_policy_changed = False
            is_value_changed = True
            while is_value_changed:
                is_value_changed = False
                for state in actions.keys():
                    if state == target[0]:
                        continue
                    max_utility = float("-infinity")
                    max_action = None
                    for action, prob in actions[state].items():
                        for direction in action:
                            if mazeDef.maze_map[state][direction]==True:
                                next_state = mdpVI().mazeTrace(mazeDef, state,
direction)
                        reward = REWARD[state]
                        if next_state == target[0]:
                            reward = 10**(7)
                        utility = reward + DISCOUNT*(prob*U[next_state])
                        if utility > max_utility:
                            max_utility = utility
                            max_action = action
                        policy[state] = max_action
                        U[state] = max_utility
                        if policy[state] != max_action:
                            is_policy_changed = True
                            policy[state] = max_action
                    iterations += 1
        currNode = (x,y)
        tracePath = {}
        while currNode != target[0]:
            test = mdpVI().mazeTrace(mazeDef, currNode, policy[currNode])
            tracePath[currNode] = test
            currNode = test
        end = time.time()
        return tracePath, U, policy, (end-start)
```

[8] GitHub Repository: https://github.com/abhikbhattacharjee/CS7IS2-AI-Maze-Solver