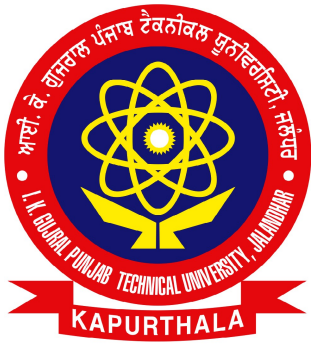
I.K. Gujral Punjab Technical University

Mohali Campus 1



Artificial Intelligence Lab

[BTCS 605-18]

Submitted to:

Mr. Jagpreet Singh

Submitted by:

Name - Sinesh Kumar

Roll No - 1917286

Branch - B.tech CSE

Semester - 6th

Table of Contents

1. WAP to implement Depth First Search (uninformed search).

Depth First Search: Depth-first search (DFS), is an algorithm for tree traversal on graph or tree data structures.

Algorithm:

1. Pick any node. If it is unvisited, mark it as visited and recur on all its adjacent nodes.
2. Repeat until all the nodes are visited, or the node to be searched is found.

Source Code:

```
# Using a Python dictionary to act as an adjacency list
graph = {
    'A' : ['B','C'],
    'B' : ['D', 'E'],
    'C' : ['F'],
    'D' : [],
    'E' : ['F'],
    'F' : []
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')]
```

OUTPUT:

```
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab>
A
B
D
E
F
C
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab>
```

2. WAP to implement Best First Search (informed search).

Best First Search: The best first search algorithm is a version of the depth first search using heuristics. Each node is evaluated with respect to its distance from the goal. Whichever node is closest to the final state is explored first. If the path fails to reach the goal, the algorithm backtracks and chooses some other node that didn't seem to be the best before.

Algorithm:

1. Create a priority queue.
2. Insert the starting node.
3. Remove a node from the priority queue.

   3.1. Check if it is the goal node. If yes, then exit.

   3.2. Otherwise, mark the node as visited and insert its neighbours into the priority queue. The priority of each will be its distance from the goal.

Source Code:

```
# implement best first search #
from queue import PriorityQueue
# store a graph in a dictionary
graph = dict()

def best_first_search(start, goal ):
    pq = PriorityQueue()
    pq.put((0, start))
    totalcost = 0
    visited = []
    while not pq.empty():
        cost, node = pq.get()
        totalcost+=cost
        visited.append(node)
        if node == goal:
            print("Path to reach goal : ",visited)
            return totalcost
        else:
            for n, c in graph[node].items():
                if n not in visited:
                    pq.put((c, n))
    return -1

def add_edge(x, y, cost):
    if x not in graph:
        graph[x] = dict()
```

```
    graph[x][y] = cost

add_edge('s', 'a', 3)
add_edge('s', 'b', 6)
add_edge('s', 'c', 5)

add_edge('a', 'd', 9)
add_edge('a', 'e', 8)

add_edge('b', 'f', 12)
add_edge('b', 'g', 14)

add_edge('c', 'h', 7)

add_edge('h', 'i', 5)
add_edge('h', 'j', 6)

add_edge('i', 'k', 1)
add_edge('i', 'l', 10)
add_edge('i', 'm', 2)

start = 's'
goal = 'i'
tc = best_first_search(start, goal)

if tc != -1:
    print(f'Found : {goal} and cost of the path is {tc}\n')
else:
    print('no path found')
```

OUTPUT:

```
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab> python -u "d:\Sinesh\

Path to reach goal :  ['s', 'a', 'c', 'b', 'h', 'i']
Found : i and cost of the path is 26

PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab>
```

3. WAP to implement A* algorithm. (A start Search).

A start Search: It is a graph traversal and path search algorithm, which is often used due to its completeness, optimality, and optimal efficiency.

A* uses the path of reaching to the current node from the starting node, and the path of reaching the goal from the current node. So, the heuristic function becomes:
`f(n) = g(n) + h(n)`

Where:
`f(n)`: cost of the optimal path from start to goal
`g(n)`: shortest path of the current node from the start
`h(n)`: shortest path of the goal from the current node.

Algorithm:

1. Create a Priority Queue.
2. Insert the starting node.
3. Remove a node from the priority queue.

   3.1. Check if it is the goal node. If yes, then exit.

   3.2. Otherwise, mark the node as visited and insert its neighbours into the priority queue. The priority of each node will be the sum of its cost from the start and the goal.

Source Code:

```
from collections import deque
# A star search
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }
        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
```

```
        closed_list = set([])
        g = {}
        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            open_list.remove(n)
```

```
            closed_list.add(n)

        print('Path does not exist!')
        return None

adjacency_list = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('A', 'D')
```

OUTPUT:

```
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab>
Path found: ['A', 'B', 'D']
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab>
```

4. WAP to construct Bayesian network from given data.

Bayesian Belief Network (BBN) is a Probabilistic Graphical Model (PGM) that represents a set of variables and their conditional dependencies via a Directed Acyclic Graph (DAG).

Source Code:

```
# Implement Bayesian Network
def stringMod1(n, fill):
    string = bin(n).replace("0b",
"").zfill(fill).replace("0","t").replace("1","0").replace("t","1")
    # print(string)
    ls = list(string)
    return ls

def stringMod2(ls):
    string =
"".join(ls).replace("0","t").replace("1","0").replace("t","1")
    # print(string)
    num = int(string, 2)
    return num
bbn = {
    "Bulgary" : ["Alarm"],
    "EarthQuake" : ["Alarm"],
    "Alarm" : ["JohnCalls", "MarryCalls"],
    "JohnCalls" : [],
    "MarryCalls" : []
}

dependencyGraph = {}
for i in bbn.keys():
    dependencyGraph[i] = list()

for (i,j) in bbn.items():
    for k in j:
        if k in dependencyGraph:
            # print(dependencyGraph)
            ls = dependencyGraph[k]
            ls.append(i)
            # print(ls)
            dependencyGraph[k] = ls
        else:
            dependencyGraph[k] = [i]

probabilityGraph = {}
for i in bbn.keys():
    probabilityGraph[i] = list()
```

```
# print(dependencyGraph)

for i,j in dependencyGraph.items():
    predValues = []
    #probabilities when happen
    for k in range(2**(len(j))):
        ls = stringMod1(k, len(j)+1)
        val = " ".join(j)
        prob = round(float(input(f"Probability for {ls} {i} {val} :
")),8)
        predValues.append(prob)
    #probabilities when not happen
    for l in range(len(predValues)):
        predValues.append(round(1-predValues[l],8))

    probabilityGraph[i] = predValues

# print(probabilityGraph)
def getJointProbability(ls):
    lsLen = len(ls)
    prob = 1
    for i in range(lsLen):
        string = ""
        lsKey = ls[i][0]
        lsValue = ls[i][1]
        string += str(lsValue)
        for k in dependencyGraph[lsKey]:
            for l,m in ls:
                if l == k:
                    string += str(m)

        # print(string)
        index = stringMod2(list(string))
        tempprob = probabilityGraph[lsKey][index]
        prob *= tempprob
        # print(tempprob)
    return round(prob,8)

print("\nBayesian Belief Network : ")
for i,v in bbn.items():
    print(f"{i} : {v}")
+
print("\nDependency Graph : ")
for i,v in dependencyGraph.items():
    print(f"{i} : {v}")

print("\nProbability Graph : ")
```

```python
for i,v in probabilityGraph.items():
    print(f"{i} : {v}")

joint_calc = getJointProbability([("Bulgary", 0), ("EarthQuake", 0),
("Alarm", 1), ("JohnCalls", 1), ("MarryCalls", 1)])
print(f"Joint Probability Distribution : {joint_calc}")
```

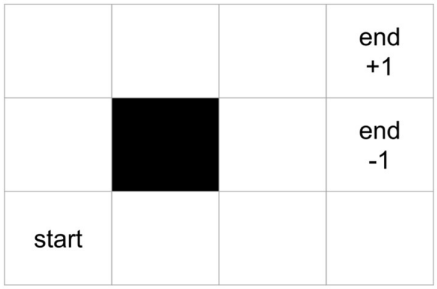
Output:

```
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab> python -u "d:\Sinesh\CSE\SEM 6\AI\AI_Lab\BasyenNetwork.py"
Probability for ['1'] Bulgary  : 0.3
Probability for ['1'] EarthQuake  : 0.1
Probability for ['1', '1', '1'] Alarm Bulgary EarthQuake : 0.7
Probability for ['1', '1', '0'] Alarm Bulgary EarthQuake : 0.5
Probability for ['1', '0', '1'] Alarm Bulgary EarthQuake : 0.7
Probability for ['1', '0', '0'] Alarm Bulgary EarthQuake : 0.5
Probability for ['1', '1'] JohnCalls Alarm : 0.4
Probability for ['1', '0'] JohnCalls Alarm : 0.8
Probability for ['1', '1'] MarryCalls Alarm : 0.9
Probability for ['1', '0'] MarryCalls Alarm : 0.8

Bayesian Belief Network :
Bulgary : ['Alarm']
EarthQuake : ['Alarm']
Alarm : ['JohnCalls', 'MarryCalls']
JohnCalls : []
MarryCalls : []

Dependency Graph :
Bulgary : []
EarthQuake : []
Alarm : ['Bulgary', 'EarthQuake']
JohnCalls : ['Alarm']
MarryCalls : ['Alarm']

Probability Graph :
Bulgary : [0.3, 0.7]
EarthQuake : [0.1, 0.9]
Alarm : [0.7, 0.5, 0.7, 0.5, 0.3, 0.5, 0.3, 0.5]
JohnCalls : [0.4, 0.8, 0.6, 0.2]
MarryCalls : [0.9, 0.8, 0.1, 0.2]
Joint Probability Distribution : 0.1134
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab>
```

5. WAP to construct value and policy iteration in a grid world.

Grid World Game is one of the most famous problem that can be solved using reinforcement learning.

The Grid World Game:



Game Rules:

The rule is simple. Your agent/robot starts at the left-bottom corner(the 'start' sign) and ends at either +1 or -1 which is the corresponding reward. At each step, the agent has 4 possible actions including up, down, left and right, whereas the black block is a wall where your agent won't be able to penetrate through.

Value Iteration:

At first, our gent knows nothing about the grid world (environment), so it would simply initialises all reward as 0. Then, it starts to explore the world by randomly walking around, surely it will endure lots of failure at the beginning. Once it reaches end of the game, either reward +1 or reward -1, the whole game reset and the reward propagates in a backward fashion and eventually the estimated value of all states along the way will be updated based on the formula:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[V(S_{t+1}) - V(S_t)\Big]$$

The Value Iteration Algorithm:

```
values = {each state : 0}
loop ITERATIONS times:
    previous = copy of values
    for all states:
        EVs = {each legal action : 0}
        for all legal actions:
            for each possible next_state:
                EVs[action] += prob * previous[next_state]
        values[state] = reward(state) + discount * max(EVs)
```

We will take the GRIDWORLD game to test Value Iteration algorithm.

Policy Iteration:

Policy Iteration is an algorithm in 'Reinforcement Learning', which helps in learning the optimal policy which maximizes the long term discounted reward.

What our agent will finally learn is a policy, and **a policy is a mapping from state to action**, simply instructs what the agent should do at each state.
At each state, our agent will choose the best action that gives the highest estimated reward.

Source Code:

```
import numpy as np

# global variables Rules for board.
BOARD_ROWS = 3
BOARD_COLS = 4
WIN_STATE = (0, 3)
LOSE_STATE = (1, 3)
START = (2, 0)
DETERMINISTIC = True

# Class to justify each state(position) of our agent,
#  giving reward according to its state.
class State:
    def __init__(self, state=START):
        self.board = np.zeros([BOARD_ROWS, BOARD_COLS])
        self.board[1, 1] = -1
        self.state = state
        self.isEnd = False
        self.determine = DETERMINISTIC

    def giveReward(self):
        if self.state == WIN_STATE:
            return 1
        elif self.state == LOSE_STATE:
            return -1
        else:
            return 0

    def isEndFunc(self):
        if (self.state == WIN_STATE) or (self.state == LOSE_STATE):
            self.isEnd = True

    def nxtPosition(self, action):
        """
        action: up, down, left, right
        -------------
        0 | 1 | 2| 3|
        1 |
```

```
        2 |
        return next position
        """
        if self.determine:
            if action == "up":
                nxtState = (self.state[0] - 1, self.state[1])
            elif action == "down":
                nxtState = (self.state[0] + 1, self.state[1])
            elif action == "left":
                nxtState = (self.state[0], self.state[1] - 1)
            else:
                nxtState = (self.state[0], self.state[1] + 1)
            # if next state legal
            if (nxtState[0] >= 0) and (nxtState[0] <= (BOARD_ROWS -1)):
                if (nxtState[1] >= 0) and (nxtState[1] <= (BOARD_COLS -1)):
                    if nxtState != (1, 1):
                        return nxtState
            return self.state

    def showBoard(self):
        self.board[self.state] = 1
        for i in range(0, BOARD_ROWS):
            print('-----------------')
            out = '| '
            for j in range(0, BOARD_COLS):
                if self.board[i, j] == 1:
                    token = '*'
                if self.board[i, j] == -1:
                    token = 'z'
                if self.board[i, j] == 0:
                    token = '0'
                out += token + ' | '
            print(out)
        print('-----------------')


# Agent of player

class Agent:

    def __init__(self):
        self.states = []
        self.actions = ["up", "down", "left", "right"]
        self.State = State()
        self.lr = 0.2
        self.exp_rate = 0.3

        # initial state reward
        self.state_values = {}
```

```
        for i in range(BOARD_ROWS):
            for j in range(BOARD_COLS):
                self.state_values[(i, j)] = 0  # set initial value to 0

    def chooseAction(self):
        # choose action with most expected value
        mx_nxt_reward = 0
        action = ""

        if np.random.uniform(0, 1) <= self.exp_rate:
            action = np.random.choice(self.actions)
        else:
            # greedy action
            for a in self.actions:
                # if the action is deterministic
                nxt_reward = self.state_values[self.State.nxtPosition(a)]
                if nxt_reward >= mx_nxt_reward:
                    action = a
                    mx_nxt_reward = nxt_reward
        return action

    def takeAction(self, action):
        position = self.State.nxtPosition(action)
        return State(state=position)

    def reset(self):
        self.states = []
        self.State = State()

    def play(self, rounds=10):
        i = 0
        while i < rounds:
            # to the end of game back propagate reward
            if self.State.isEnd:
                # back propagate
                reward = self.State.giveReward()
                # explicitly assign end state to reward values
                self.state_values[self.State.state] = reward  # this is
optional
                print("Game End Reward", reward)
                for s in reversed(self.states):
                    reward = self.state_values[s] + self.lr * (reward -
self.state_values[s])
                    self.state_values[s] = round(reward, 3)
                self.reset()
                i += 1
            else:
                action = self.chooseAction()
```

```
                # append trace
                self.states.append(self.State.nxtPosition(action))
                print("current position {} action
{}".format(self.State.state, action))
                # by taking the action, it reaches the next state
                self.State = self.takeAction(action)
                # mark is end
                self.State.isEndFunc()
                print("nxt state", self.State.state)
                print("---------------------")

    def showValues(self):
        for i in range(0, BOARD_ROWS):
            print('---------------------------------')
            out = '| '
            for j in range(0, BOARD_COLS):
                out += str(self.state_values[(i, j)]).ljust(6) + ' | '
            print(out)
        print('---------------------------------')


if __name__ == "__main__":
    ag = Agent()
    ag.play(50)
    print(ag.showValues())
```

Output:

```
Game End Reward 1
---------------------------------
| 0.952  | 0.95   | 0.917  | 1.0    |
---------------------------------
| 0.94   | 0      | 0.507  | -1.0   |
---------------------------------
| 0.719  | 0.18   | -0.005 | -0.2   |
---------------------------------
None
PS D:\Sinesh\CSE\SEM 6\AI\AI_Lab>
```

This is the estimates of each state after playing 50 rounds of game. As our action is deterministic, we can get best action at each state by following the highest estimate!