A FILE OF CLOUD COMPUTING LAB

At

BABA BANDA SINGH BAHADUR ENGINEERING COLLEGE

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENT FOR THE AWARD OF THE DEGREE OF

BACHELOR OF TECHNOLOGY

(Computer Science & Engineering)

SUBMITTED BY:

PRINCE KUMAR (2001308)

SUBMITTED TO:

PROF. JASSWINDER KAUR DHALIWAL

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

BABA BANDA SINGH BAHADUR ENGINEERING COLLEGE, FATEHGARH SAHIB

Practical no: 1

Aim: write a program to conduct uniformed search and informed search.

(a) Depth first search

Input

```
graph={
    'A':['B','C'],
    'B':['D','E'],
    'C':['F'],
    'D':[],
    'E':[],
    'F':[]
}
visited=set()
def dfs(visited,graph,node):
    if node not in visited:
        print(node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited,graph,neighbour)
print("FOLLOWING IS THE DPTH FIRST SEARCH")
dfs(visited,graph,'A')
```

Output

```
FOLLOWING IS THE DPTH FIRST SEARCH
A
B
D
E
C
F


...Program finished with exit code 0
Press ENTER to exit console.
```

(b) Breadth First Search

Input

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

visited = [] # List for visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
      visited.append(neighbour)
      queue.append(neighbour)
```

```
print("Following is the Breadth-First Search")

bfs(visited, graph, '5')
```

Output

```
Following is the Breadth-First Search
5 3 7 2 4 8

...Program finished with exit code 0
Press ENTER to exit console.
```

(c) Best first search

Input

```
from queue import PriorityQueue

v = 14

graph = [[] for i in range(v)]

def best_first_search(actual_Src, target, n):

    visited = [False] * n

    pq = PriorityQueue()

    pq.put((0, actual_Src))

    visited[actual_Src] = True

    while pq.empty() == False:

        u = pq.get()[1]

        print(u, end=" ")

        if u == target:

            break

        for v, c in graph[u]:

            if visited[v] == False:

                visited[v] = True

                pq.put((c, v))

    print()

def addedge(x, y, cost):

    graph[x].append((y, cost))

    graph[y].append((x, cost))

addedge(0, 1, 3)

addedge(0, 2, 6)

addedge(0, 3, 5)

addedge(1, 4, 9)
```

```
addedge(1, 5, 8)

addedge(2, 6, 12)

addedge(2, 7, 14)

addedge(3, 8, 7)

addedge(8, 9, 5)

addedge(8, 10, 6)

addedge(9, 11, 1)

addedge(9, 12, 10)

addedge(9, 13, 2)

source = 0

target = 9

best_first_search(source, target, v)
```

Output

```
0 1 3 2 8 9


...Program finished with exit code 0
Press ENTER to exit console.
```

Practical No: 2

Aim: Operators in Python

Input

```
a = 9

b = 4

add = a + b

sub = a - b

mul = a * b

mod = a % b

p = a ** b

print(add)

print(sub)

print(mul)

print(mod)

print(p)
```

Output

```
13
5
36
1
6561


...Program finished with exit code 0
Press ENTER to exit console.
```

Practical No: 3

Aim: Write a program to conduct game search for tic tac toe

Input

```
board = ["-", "-", "-",
         "-", "-", "-",
         "-", "-", "-"]
def print_board():
    print(board[0] + " | " + board[1] + " | " + board[2])
    print(board[3] + " | " + board[4] + " | " + board[5])
    print(board[6] + " | " + board[7] + " | " + board[8])
def take_turn(player):
    print(player + "'s turn.")
    position = input("Choose a position from 1-9: ")
    while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
        position = input("Invalid input. Choose a position from 1-9: ")
    position = int(position) - 1
    while board[position] != "-":
        position = int(input("Position already taken. Choose a different position: ")) -1
    board[position] = player
    print_board()
def check_game_over():
    if (board[0] == board[1] == board[2] != "-") or \
    (board[3] == board[4] == board[5] != "-") or \
    (board[6] == board[7] == board[8] != "-") or \
    (board[0] == board[3] == board[6] != "-") or \
    (board[1] == board[4] == board[7] != "-") or \
    (board[2] == board[5] == board[8] != "-") or \
```

```
        (board[0] == board[4] == board[8] != "-") or \
        (board[2] == board[4] == board[6] != "-"):
            return "win"
        elif "-" not in board:
            return "tie"
        else:
            return "play"
def play_game():
        print_board()
        current_player = "X"
        game_over = False
        while not game_over:
            take_turn(current_player)
            game_result = check_game_over()
            if game_result == "win":
                print(current_player + " wins!")
                game_over = True
            elif game_result == "tie":
                print("It's a tie!")
                game_over = True
            else:
                current_player = "O" if current_player == "X" else "X"
play_game()
```

Output

```
- | - | -
- | - | -
- | - | -
X's turn.
Choose a position from 1-9: 3
- | - | X
- | - | -
- | - | -
O's turn.
Choose a position from 1-9: 4
- | - | X
O | - | -
- | - | -
X's turn.
Choose a position from 1-9: 2
- | X | X
O | - | -
- | - | -
O's turn.
Choose a position from 1-9: 6
- | X | X
O | - | O
- | - | -
X's turn.
Choose a position from 1-9: 7
- | X | X
O | - | O
X | - | -
O's turn.
Choose a position from 1-9: 8
- | X | X
O | - | O
X | O | -
X's turn.
Choose a position from 1-9: 5
- | X | X
O | X | O
X | O | -
X wins!


...Program finished with exit code 0
Press ENTER to exit console.
```

Practical no: 4

Aim: Write a program to construct a Bayesian network for the given data

Input

```
import numpy as np import csv import pandas as pd from pgmpy.models import BayesianModel
from pgmpy.estimators import MaximumLikelihoodEstimator from pgmpy.inference import
VariableElimination
#read Cleveland Heart Disease data heartDisease
= pd.read_csv('heart.csv') heartDisease = heartDisease.replace('?',np.nan)
#display the data print('Few examples from the dataset are given below') print(heartDisease.head())
#Model Bayesian Network Model=BayesianModel([('age','trestbps'),('age','fbs'),
('sex','trestbps'),('exang','trestbps'),('trestbps','heartdise
ase'),('fbs','heartdisease'),('heartdisease','restecg'),
('heartdisease','thalach'),('heartdisease','chol')])
#Learning CPDs using Maximum Likelihood Estimators print('\n Learning CPD using Maximum
likelihood estimators') model.fit(heartDisease,estimator=MaximumLikelihoodEstimator
)
# Inferencing with Bayesian Network print('\n Inferencing with Bayesian Network:')
HeartDisease_infer = VariableElimination(model)
```

Given Dataset

age	sex	cp	tresbps	chol	fbs	restceg	thalach	exang	Old peak	slope	ca	thal	Heart disease
63	1	1	145	233	1	2	150	0	23	3	0	6	0
67	1	4	160	286	0	2	108	1	1.5	2	3	3	2
67	1	4	120	229	0	2	129	1	2.6	2	2	7	1
41	0	2	130	204	0	2	172	0	1.4	1	0	3	0
62	0	4	140	268	0	2	160	0	3.6	3	2	3	3
60	1	4	130	206	0	2	132	1	2.4	2	2	7	4

Output

Learning CPD using Maximum likelihood estimators Inferencing with Bayesian Network:

1. Probability of HeartDisease given Age=28

heartdisease	phi(heartdisease)
heartdisease_0	0.6791
heartdisease_1	0.1212
heartdisease_2	0.0810
heartdisease_3	0.0939
heartdisease_4	0.0247

2. Probability of HeartDisease given cholesterol=100

heartdisease	phi(heartdisease)
heartdisease_0	0.5400
heartdisease_1	0.1533
heartdisease_2	0.1303
heartdisease_3	0.1259
heartdisease_4	0.0506

Practical -5

Aim: Write a programme to run value and policy iteration in a grid world.

Input:

```
import numpy as np

# Define the grid world
grid = np.array([
    [-1, -1, -1, -1],
    [-1, -1, -1, -1],
    [-1, -1, -1, -1],
    [-1, -1, -1, -1]
])

# Define the rewards
rewards = np.array([
    [0, 0, 0, 1],
    [0, -1, 0, -1],
    [0, 0, 0, -1],
    [1, -1, 0, -1]
])

# Define the discount factor
gamma = 0.99

# Define the value function
v = np.zeros_like(grid, dtype=float)

# Define the policy
policy = np.zeros_like(grid, dtype=int)

# Define the number of iterations
num_iterations = 1000

# Value iteration algorithm
for i in range(num_iterations):
    for x in range(grid.shape[0]):
        for y in range(grid.shape[1]):
            if grid[x][y] == -1:
                v[x][y] = -1
            else:
                v[x][y] = rewards[x][y] + gamma * np.max([v[x-1][y] if x > 0 else -1,
                v[x+1][y] if x < grid.shape[0]-1 else -1,
                v[x][y-1] if y > 0 else -1,
                v[x][y+1] if y < grid.shape[1]-1 else -1])

    # Policy improvement step
    for x in range(grid.shape[0]):
        for y in range(grid.shape[1]):
            if grid[x][y] == -1:
                policy[x][y] = -1
```

```
            else:
                policy[x][y] = np.argmax([v[x-1][y] if x > 0 else -1,
                v[x+1][y] if x < grid.shape[0]-1 else -1,
                v[x][y-1] if y > 0 else -1,
                v[x][y+1] if y < grid.shape[1]-1 else -1])

# Print the results
print("Value Function:\n", v)
print("Policy:\n", policy)
```

Output :

```
Value Function:
 [[-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]
 [-1. -1. -1. -1.]]


Policy:
 [[-1 -1 -1 -1]
 [-1 -1 -1 -1]
 [-1 -1 -1 -1]
 [-1 -1 -1 -1]]
```

Practical - 6

Aim: Write a programme to do reinforcement learning in a grid world

Input:

```
import numpy as np

# define the grid world
grid = np.array([
    [0, 0, 0, 0],
    [0, -1, 0, -1],
    [0, 0, 0, 0],
    [-1, 0, 0, 1]
])

# define the reward function
def reward(state):
    return grid[tuple(state)]

# define the Q-learning algorithm
def q_learning(grid, learning_rate=0.8, discount_factor=0.95, epsilon=0.1, num_episodes=100):
    q_table = np.zeros((grid.shape[0], grid.shape[1], 4)) # 4 actions: up, down, left, right
    for episode in range(num_episodes):
        state = np.random.randint(grid.shape[0], size=2)
        while True:
            # select action using epsilon-greedy strategy
            if np.random.rand() < epsilon:
                action = np.random.randint(4)
            else:
                action = np.argmax(q_table[tuple(state)])
            # move to next state
            if action == 0 and state[0] > 0:
                next_state = state - [1, 0]
            elif action == 1 and state[0] < grid.shape[0]-1:
                next_state = state + [1, 0]
            elif action == 2 and state[1] > 0:
                next_state = state - [0, 1]
            elif action == 3 and state[1] < grid.shape[1]-1:
                next_state = state + [0, 1]
            else:
                next_state = state
            # update Q-table
            q_table[tuple(state)][action] += learning_rate * (reward(next_state) + discount_factor * np.
max(q_table[tuple(next_state)]) - q_table[tuple(state)][action])
            state = next_state
            if reward(state) != 0:
                break
    return q_table

# test the Q-learning algorithm
q_table = q_learning(grid)
print(q_table)
```

Output:

```
[[[ 0.          0.          0.          0.        ]
  [ 0.         -1.          0.          0.        ]
  [ 0.          0.          0.          0.        ]
  [ 0.         -1.          0.          0.        ]]

 [[ 0.          0.          0.         -0.8       ]
  [ 0.          0.          0.          0.        ]
  [ 0.          0.         -0.8        -0.8       ]
  [ 0.          0.          0.          0.        ]]

 [[ 0.         -0.96        0.          0.        ]
  [-0.992       0.          0.          0.        ]
  [ 0.          0.          0.          3.57822316]
  [-0.8         3.98547619  0.          0.        ]]

 [[ 0.          0.          0.          0.        ]
  [ 0.          0.         -0.9984      0.        ]
  [ 2.65812492  0.          0.          3.75626789]
  [ 3.14354137  0.          0.          0.        ]]]
```