1.Write a python program to implement Breadth First Search.

**Aim:** To write a python program to implement Breadth First Search.

**Description:**

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or any arbitrary node) and explores all of the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. BFS employs a queue to keep track of the next vertex to visit, ensuring that vertices are visited in the order of their distance from the root, making it suitable for finding the shortest path in unweighted graphs.

**Code:**

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def BFS(self, s):
        visited = [False] * (max(self.graph) + 1)
        queue = []
        queue.append(s)
        visited[s] = True
        while queue:
            s = queue.pop(0)
            print(s, end=" ")
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
if __name__ == '__main__':
    g = Graph()
    g.addEdge(0, 1)
```

```
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print("Following is Breadth First Traversal"" (starting from vertex 2)")
g.BFS(2)
```

**Output:**

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

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

2. Write a python program to implement Depth First Search.

**Aim:** To write a python program to implement Depth First Search.

**Description:**

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root (or any arbitrary node) and explores as far as possible along each branch before backtracking. DFS uses a stack to keep track of the next vertex to visit, which allows it to explore deeply into the structure before moving on to shallower nodes. This algorithm is commonly used to explore all the vertices in a graph or to find a specific target vertex or path.

**Code:**

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)
if __name__ == "__main__":
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
```

```
g.addEdge(2, 3)
g.addEdge(3, 3)
print("Following is Depth First Traversal (starting from vertex 2)")
g.DFS(2)
```

**Output:**

```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3

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

3. Write a python program to implement Best First Search.

**Aim:** To write a python program to implement Best First Search.

**Description:**

Best First Search selects nodes for expansion based on a heuristic function estimating proximity to the goal, utilizing a priority queue to prioritize exploration of the most promising nodes. This algorithm efficiently navigates graphs or search spaces, often employed in pathfinding or optimization problems where heuristic information is available.

**Code:**

```
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.
```

4. Write a python program to implement A * Search.

**Aim:** To write a python program to implement A * Search.

**Description:**

A* Search is an informed search algorithm that efficiently finds the shortest path from a start node to a goal node in a graph, using both the actual cost from the start node and a heuristic estimate of the cost to reach the goal. It explores nodes in order of their total cost, which is the sum of the actual cost and the heuristic estimate, ensuring optimality while efficiently pruning the search space.

**Code:**

```
import math
import heapq
class Cell:
    def __init__(self):
        self.parent_i = 0
        self.parent_j = 0
        self.f = float('inf')
        self.g = float('inf')
        self.h = 0
ROW = 9
COL = 10
def is_valid(row, col):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)
def is_unblocked(grid, row, col):
    return grid[row][col] == 1
def is_destination(row, col, dest):
    return row == dest[0] and col == dest[1]
def calculate_h_value(row, col, dest):
    return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5
def trace_path(cell_details, dest):
    print("The Path is ")
```

```
    path = []
    row = dest[0]
    col = dest[1]
    while not (cell_details[row][col].parent_i == row and cell_details[row][col].parent_j == col):
        path.append((row, col))
        temp_row = cell_details[row][col].parent_i
        temp_col = cell_details[row][col].parent_j
        row = temp_row
        col = temp_col
    path.append((row, col))
    path.reverse()
    for i in path:
        print("->", i, end=" ")
    print()
def a_star_search(grid, src, dest):
    if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
        print("Source or destination is invalid")
        return
    if not is_unblocked(grid, src[0], src[1]) or not is_unblocked(grid, dest[0], dest[1]):
        print("Source or the destination is blocked")
        return
    if is_destination(src[0], src[1], dest):
        print("We are already at the destination")
        return
    closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
    cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]
    i = src[0]
    j = src[1]
```

```
    cell_details[i][j].f = 0
    cell_details[i][j].g = 0
    cell_details[i][j].h = 0
    cell_details[i][j].parent_i = i
    cell_details[i][j].parent_j = j
    open_list = []
    heapq.heappush(open_list, (0.0, i, j))
    found_dest = False
    while len(open_list) > 0:
        p = heapq.heappop(open_list)
        i = p[1]
        j = p[2]
        closed_list[i][j] = True
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]
        for dir in directions:
            new_i = i + dir[0]
            new_j = j + dir[1]
            if is_valid(new_i, new_j) and is_unblocked(grid, new_i, new_j) and
not closed_list[new_i][new_j]:
                if is_destination(new_i, new_j, dest):
                    cell_details[new_i][new_j].parent_i = i
                    cell_details[new_i][new_j].parent_j = j
                    print("The destination cell is found")
                    trace_path(cell_details, dest)
                    found_dest = True
                    return
                else:
                    g_new = cell_details[i][j].g + 1.0
                    h_new = calculate_h_value(new_i, new_j, dest)
```

```
                    f_new = g_new + h_new
                    if cell_details[new_i][new_j].f == float('inf') or cell_details[new_i][new_j].f > f_new:
                        heapq.heappush(open_list, (f_new, new_i, new_j))
                        cell_details[new_i][new_j].f = f_new
                        cell_details[new_i][new_j].g = g_new
                        cell_details[new_i][new_j].h = h_new
                        cell_details[new_i][new_j].parent_i = i
                        cell_details[new_i][new_j].parent_j = j
    if not found_dest:
        print("Failed to find the destination cell")
def main():
    grid = [
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
        [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
    ]
    src = [8, 0]
    dest = [0, 0]
    a_star_search(grid, src, dest)
if __name__ == "__main__":
    main()
```

**Output:**

```
The destination cell is found
The Path is
-> (8, 0) -> (7, 0) -> (6, 0) -> (5, 0) -> (4, 1) -> (3, 2) -> (2, 1) -> (1, 0) -> (0, 0)


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

5. Write a python program to implement Min- Max Algorithm.

**Aim:** To write a python program to implement Min- Max Algorithm.

**Description:**

The Minimax algorithm recursively explores the game tree, alternating between maximizing and minimizing player objectives to determine the optimal move. It evaluates possible moves until reaching a terminal state or predetermined depth, providing a strategy for decision-making in adversarial games.

**Code:**

```
import math
def minimax (curDepth, nodeIndex,maxTurn, scores,targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2,False, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,False, scores, targetDepth))
    else:
        return min(minimax(curDepth + 1, nodeIndex * 2,True, scores, targetDepth),
                minimax(curDepth + 1, nodeIndex * 2 + 1,True, scores, targetDepth))
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores), 2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

**Output:**

```
The optimal value is : 12


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

6. Write a python program to implement Alpha- Beta Pruning.

**Aim:** To write a python program to implement Alpha- Beta Pruning.

**Description:**

Alpha-Beta Pruning optimizes the Minimax algorithm by efficiently eliminating irrelevant branches in the search tree. It maintains bounds (alpha for max player, beta for min player) to determine whether further exploration is necessary, significantly reducing computational complexity while preserving optimality.

**Code:**

```
MAX, MIN = float('inf'), float('-inf')

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]
    if maximizingPlayer:
        best = MIN
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best
```

```python
if __name__ == "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

**Output:**

```
The optimal value is : 5


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

7. Write a python program to implement the Bayesian belief networks.

**Aim:** To write a python program to implement Bayesian belief networks.

**Description:**

A Bayesian Network is a probabilistic graphical model that represents a set of random variables and their conditional dependencies using a directed acyclic graph (DAG). Each node in the graph represents a random variable, and the edges indicate probabilistic dependencies between them. The network allows for efficient inference about the probability distribution of variables given evidence or observations, making it a powerful tool for reasoning under uncertainty in various domains such as artificial intelligence, decision making, and machine learning.

**Code:**

```
from pgmpy.models import BayesianNetwork

from pgmpy.estimators import MaximumLikelihoodEstimator

from pgmpy.inference import VariableElimination

import pandas as pd

data = pd.DataFrame({

    'Cloudy': [True, True, False, False],

    'Sprinkler': [True, False, True, False]

})

model = BayesianNetwork([('Cloudy', 'Sprinkler')])

model.fit(data, estimator=MaximumLikelihoodEstimator)

inference = VariableElimination(model)

print("Inferencing P(Sprinkler=True | Cloudy=False):")

print(inference.query(variables=['Sprinkler'], evidence={'Cloudy': False7

}))
```

**Output:**

```
Inferencing P(Sprinkler=True | Cloudy=False):
+------------------+------------------+
| Sprinkler        |   phi(Sprinkler) |
+==================+==================+
| Sprinkler(False) |           0.5000 |
+------------------+------------------+
| Sprinkler(True)  |           0.5000 |
+------------------+------------------+
```

8. Program to implement the Q-learning.

**Aim:** To write a program to implement the Q-learning.

**Description:**

Q-learning is a reinforcement learning algorithm that enables an agent to learn an optimal policy for sequential decision-making tasks in an unknown environment. It uses a Q-table to store action-value estimates, representing the expected cumulative reward for taking a specific action in a particular state. Through iterative exploration and exploitation, the agent updates the Q-values based on observed rewards, aiming to maximize long-term rewards.

**Code:**

```
import numpy as np

n_states = 16

n_actions = 4

goal_state = 15

Q_table = np.zeros((n_states, n_actions))

learning_rate = 0.8

discount_factor = 0.95

exploration_prob = 0.2

epochs = 1000


for epoch in range(epochs):

    current_state = np.random.randint(0, n_states)

    while current_state != goal_state:

        if np.random.rand() < exploration_prob:

            action = np.random.randint(0, n_actions)

        else:

            action = np.argmax(Q_table[current_state])

        next_state = (current_state + 1) % n_states

        reward = 1 if next_state == goal_state else 0

        Q_table[current_state, action] += learning_rate * \

            (reward + discount_factor *
```

```
            np.max(Q_table[next_state]) - Q_table[current_state, action])
        current_state = next_state
print("Learned Q-table:")
print(Q_table)
```

**Output:**

Output

```
Learned Q-table:
[[0.48767498 0.48767373 0.         0.39013998]
 [0.51334208 0.51333551 0.51334201 0.51333551]
 [0.54036009 0.54035981 0.54035317 0.5403255 ]
 [0.56880009 0.56880009 0.56880009 0.56880009]
 [0.59873694 0.59873694 0.59873694 0.59873694]
 [0.63024941 0.63024941 0.63024941 0.63024941]
 [0.66342043 0.66342043 0.66342043 0.66342043]
 [0.6983373  0.6983373  0.6983373  0.6983373 ]
 [0.73509189 0.73509189 0.73509189 0.73509189]
 [0.77378094 0.77378094 0.77378094 0.77378094]
 [0.81450625 0.81450625 0.81450625 0.81450625]
 [0.857375   0.857375   0.857375   0.857375  ]
 [0.9025     0.9025     0.9025     0.9025    ]
 [0.95       0.95       0.95       0.95      ]
 [1.         1.         1.         1.        ]
 [0.         0.         0.         0.        ]]

=== Code Execution Successful ===
```

9. Program to implement Bayes Theorem, Joint probability, Conditional probability Using python.

**Aim:** To write a program to implement Bayes Theorem, Joint probability, Conditional probability Using python.

**Description:**

Bayes' Theorem computes the probability of a hypothesis given evidence. Joint probability assesses the likelihood of multiple events occurring together. Conditional probability estimates the probability of an event given another has already happened. These concepts are fundamental in probability theory and find applications across various domains, including statistics, machine learning, and decision-making.

**Code:**

```
import numpy as np


def bayes_theorem(p_a, p_b_given_a, p_b):
    return (p_b_given_a * p_a) / p_b


def joint_probability(p_a, p_b_given_a):
    return p_a * p_b_given_a


def conditional_probability(p_a_and_b, p_a):
    return p_a_and_b / p_a


p_disease = 0.01
p_positive_given_disease = 0.9
p_positive_given_no_disease = 0.05
p_no_disease = 1 - p_disease


p_positive = (p_positive_given_disease * p_disease) + (p_positive_given_no_disease * p_no_disease)
p_disease_given_positive = bayes_theorem(p_disease, p_positive_given_disease, p_positive)
print("Probability of having the disease given a positive test result:", p_disease_given_positive)
```

```
p_red_cards = 26 / 52
p_face_cards_given_red = 6 / 26
p_red_and_face = joint_probability(p_red_cards, p_face_cards_given_red)
print("Joint probability of drawing a red card and a face card:", p_red_and_face)


p_red_cards = 26 / 52
p_face_cards_given_red = 6 / 26
p_face_given_red = conditional_probability(p_face_cards_given_red, p_red_cards)
print("Conditional probability of drawing a face card given the card drawn is red:", p_face_given_red)
```

**Output:**

Output Clear

```
Probability of having the disease given a positive test result: 0.15384615384615385
Joint probability of drawing a red card and a face card: 0.11538461538461539
Conditional probability of drawing a face given the card is red: 0.46153846153846156

=== Code Execution Successful ===
```

10.Program to implement reinforcement learning.

**Aim:** To write a python program to implement reinforcement learning.

**Description:**

This Python code implements the Value Iteration algorithm for reinforcement learning. It initializes the value function and defines the reward and transition matrices. The algorithm iteratively updates the value function until convergence, then extracts the optimal policy based on the learned values. Finally, it prints the optimal value function and policy for the given problem.

**Code:**

```
import numpy as np
gamma = 0.9
theta = 1e-6
num_states = 10
num_actions = 2

V = np.zeros(num_states)

reward_matrix = np.random.rand(num_states, num_actions)
transition_matrix = np.zeros((num_states, num_actions), dtype=int)

for state in range(num_states):
    transition_matrix[state, 0] = (state + 1) % num_states
    transition_matrix[state, 1] = (state - 1) % num_states

def reward(state, action):
    return reward_matrix[state, action]

def next_state(state, action):
    return transition_matrix[state, action]
```

```
def value_iteration():
    while True:
        delta = 0
        for state in range(num_states):
            v = V[state]
            max_value = float('-inf')
            for action in range(num_actions):
                next_s = next_state(state, action)
                r = reward(state, action)
                max_value = max(max_value, r + gamma * V[next_s])
            V[state] = max_value
            delta = max(delta, abs(v - V[state]))
        if delta < theta:
            break


def extract_policy():
    policy = np.zeros(num_states, dtype=int)
    for state in range(num_states):
        max_value = float('-inf')
        best_action = 0
        for action in range(num_actions):
            next_s = next_state(state, action)
            r = reward(state, action)
            value = r + gamma * V[next_s]
            if value > max_value:
                max_value = value
                best_action = action
        policy[state] = best_action
    return policy
```

```
value_iteration()
policy = extract_policy()


print("Optimal Value Function:")
print(V)
print("Optimal Policy:")
print(policy)
```

**Output:**

```
Optimal Value Function:
[8.63312572 8.77908639 9.07426195 8.97547958 8.86325356 8.46885469
 7.6456346  7.86095783 8.19563499 8.11289272]
Optimal Policy:
[0 0 0 1 1 1 1 0 0 0]
```

11. Implement the bellman equation, Temporal dependencies for Q-Table.

**Aim:** To write a python program to implement the bellman equation , Temporal dependencies for Q-table.

**Description:**

This Python code implements the Bellman equation to update the Q-table, representing the expected cumulative rewards for state-action pairs in reinforcement learning. By iteratively updating Q-values based on observed rewards and future expected rewards, it enables the agent to learn optimal actions over time, considering temporal dependencies in decision-making tasks.

**Code:**

```
import numpy as np


gamma = 0.9

num_states = 10

num_actions = 2


Q = np.zeros((num_states, num_actions))


def reward(state, action):

    return np.random.rand()


def next_state(state, action):

    return (state + action) % num_states


def bellman_update(state, action, reward, next_state):

    best_next_action = np.argmax(Q[next_state, :])

    Q[state, action] = reward + gamma * Q[next_state, best_next_action]


for episode in range(1000):

    state = np.random.randint(num_states)

    while True:
```

```
        action = np.random.randint(num_actions)
        next_s = next_state(state, action)
        r = reward(state, action)
        bellman_update(state, action, r, next_s)
        state = next_s
        if state == 0:
            break

print("Q-Table after applying Bellman updates:")
print(Q)
```

**Output:**

```
Q-Table after applying Bellman updates:
[[6.64647047 6.97750022]
 [7.3131742  6.57996773]
 [6.81965123 6.2955066 ]
 [5.71165467 6.80835921]
 [6.85769632 6.41178038]
 [6.28291072 6.00007225]
 [6.49933447 7.16979182]
 [6.82279842 6.85639025]
 [6.66330103 6.84613052]
 [6.28455908 6.74491288]]
```

12. Program to implement adaptive dynamic programming.

**Aim:** To write a program to implement adaptive dynamic programming.

**Description:**

ADP iteratively updates value functions or policies based on observed data, adjusting strategies over time to improve decision-making in dynamic environments. It enables agents to adapt and learn optimal behaviors through interactions with the environment, utilizing techniques like value iteration or policy iteration for learning.

**Code:**

```
import numpy as np
num_states = 10
num_actions = 2
gamma = 0.6

V = np.zeros(num_states)

def policy_evaluation(policy):
    for state in range(num_states):
        action = policy[state]
        next_state = (state + action) % num_states
        reward = np.random.rand()
        V[state] = reward + gamma * V[next_state]

def policy_improvement():
    policy = np.zeros(num_states, dtype=int)
    for state in range(num_states):
        q_values = np.zeros(num_actions)
        for action in range(num_actions):
            next_state = (state + action) % num_states
            reward = np.random.rand()
            q_values[action] = reward + gamma * V[next_state]
```

```
        policy[state] = np.argmax(q_values)
    return policy

policy = np.zeros(num_states, dtype=int)
for _ in range(100):
    policy_evaluation(policy)
    policy = policy_improvement()

print("Final policy:", policy)
print("Value function:", V)
```

**Output:**

```
Final policy: [1 1 1 1 0 1 0 0 0 0]
Value function: [1.37789747 1.14051112 1.34068519 1.50484019 1.89542883 0.61504918
 1.25984049 1.84707482 1.77719479 1.32055923]
```

13. Program to implement the hidden Markov process.

**Aim:** To write a python program to implement the Hidden Markov process.

**Description:**

This Python code defines parameters for a Hidden Markov Model (HMM) with states, observations, and transition probabilities. It implements the forward algorithm to calculate the probabilities of observing a sequence of observations given the model, enabling inference of the underlying state sequence. Finally, it prints the forward probabilities for the given observed sequence.

**Code:**

```
states = ['Rainy', 'Sunny']

observations = ['walk', 'shop', 'clean']

start_probability = {'Rainy': 0.6, 'Sunny': 0.4}

transition_probability = {
    'Rainy': {'Rainy': 0.7, 'Sunny': 0.3},
    'Sunny': {'Rainy': 0.4, 'Sunny': 0.6}
}

emission_probability = {
    'Rainy': {'walk': 0.1, 'shop': 0.4, 'clean': 0.5},
    'Sunny': {'walk': 0.6, 'shop': 0.3, 'clean': 0.1}
}


def forward_algorithm(observed_sequence):
    fwd = [{}]
    for state in states:
        fwd[0][state] = start_probability[state] * emission_probability[state][observed_sequence[0]]
    for t in range(1, len(observed_sequence)):
        fwd.append({})
        for state in states:
            fwd[t][state] = sum((fwd[t-1][prev_state] * transition_probability[prev_state][state] * emission_probability[state][observed_sequence[t]]) for prev_state in states)
```

```
    return fwd


observed_sequence = ['walk', 'shop', 'clean']

fwd_probs = forward_algorithm(observed_sequence)

print(fwd_probs)
```

**Output:**

```
[{'Rainy': 0.06, 'Sunny': 0.24}, {'Rainy': 0.0552, 'Sunny': 0.0486}, {'Rainy': 0.02903999999999996, 'Sunny': 0.004572}]
PS C:\Users\Laxmikanth\OneDrive\Desktop\AI>
```