8 QUEENS ALGORITHM

AIM:

To place 8 queens on a standard 8×8 chessboard such that no two queens threaten each other (i.e., no two queens share the same row, column, or diagonal).

```python
def print_board(board):
    for row in board:
        print(' '.join('Q' if x else '.' for x in row))
    print()
def is_safe(board, row, col):
    for i in range(row):
        if board[i][col]:
            return False
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i][j]:
            return False
    for i, j in zip(range(row-1, -1, -1), range(col+1, len(board))):
        if board[i][j]:
            return False
    return True
def solve_queens(board, row):
    if row == len(board):
        print_board(board)
        return True
    for col in range(len(board)):
        if is_safe(board, row, col):
            board[row][col] = True
            if solve_queens(board, row + 1):
                return True
            board[row][col] = False
    return False
def eight_queens():
    board = [[False] * 8 for _ in range(8)]
    if not solve_queens(board, 0):
        print("No solution found")
eight_queens()
```

RESULT:

One or more valid arrangements of 8 queens on the board where none attack each other.

DEPTH FIRST SEARCH

AIM:

To explore all vertices of a graph or tree by traversing as far as possible along each branch before backtracking.

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
    for neighbor in graph.get(start, []):
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# User input
graph = {}
nodes = int(input("Enter number of nodes: "))
for _ in range(nodes):
    node = input("Enter node name: ")
    neighbors = input(f"Enter neighbors of {node} (space-separated): ").split()
    graph[node] = neighbors

start_node = input("Enter start node for DFS: ")
print("DFS Traversal:")
dfs(graph, start_node)
```

RESULT:

A traversal order of nodes (or discovery of a path, component, or solution) depending on the problem, such as a list of visited nodes or detection of cycles, connected components, or paths.

A* ALGORITHM

AIM:

To find the shortest path from a start node to a goal node in a graph using heuristics to guide the search efficiently.

```
import heapq
def a_star(graph, heuristic, start, goal):
    queue = [(heuristic[start], 0, start, [start])]
    visited = set()
    while queue:
        est_total, cost_so_far, current, path = heapq.heappop(queue)
        if current == goal:
            return path
        if current in visited:
            continue
        visited.add(current)
        for neighbor, weight in graph[current]:
            if neighbor not in visited:
                new_cost = cost_so_far + weight
                est = new_cost + heuristic[neighbor]
                heapq.heappush(queue, (est, new_cost, neighbor, path + [neighbor]))
    return None
graph = {}
heuristic = {}
nodes = int(input("Enter number of nodes: "))
for _ in range(nodes):
    node = input("Node name: ")
    edges = input(f"Enter edges from {node} (format: neighbor:cost space-separated): ").split()
    graph[node] = [(e.split(':')[0], int(e.split(':')[1])) for e in edges]
    heuristic[node] = int(input(f"Heuristic value for {node}: "))
start = input("Start node: ")
goal = input("Goal node: ")
path = a_star(graph, heuristic, start, goal)
print("Shortest path:", path)
```

RESULT:

The optimal path from the start to the goal node, if one exists, with minimized total cost based on actual and estimated distances.

MIN / MAX ALGORITHM

AIM:

To determine the optimal move in a two-player game by simulating all possible moves, assuming both players play optimally.

```
import math
def check_win(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):
            return True
        if all(board[j][i] == player for j in range(3)):
            return True
    if board[0][0] == player and board[1][1] == player and board[2][2] == player:
        return True
    if board[0][2] == player and board[1][1] == player and board[2][0] == player:
        return True
    return False
def is_draw(board):
    return all(board[i][j] != 0 for i in range(3) for j in range(3))
def minmax(board, depth, is_maximizing):
    if check_win(board, 1):
        return 1
    if check_win(board, -1):
        return -1
    if is_draw(board):  # Draw
        return 0
    if is_maximizing:
        best = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == 0:
                    board[i][j] = 1
                    best = max(best, minmax(board, depth + 1, False))
                    board[i][j] = 0
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == 0:
```

```python
                    if board[i][j] == 0:
                        board[i][j] = -1
                        best = min(best, minmax(board, depth + 1, True))
                        board[i][j] = 0
            return best
def best_move(board):
    best_val = -math.inf
    move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                board[i][j] = 1
                move_val = minmax(board, 0, False)
                board[i][j] = 0
                if move_val > best_val:
                    move = (i, j)
                    best_val = move_val
    return move
def print_board(board):
    for row in board:
        print(" ".join(str(cell) for cell in row))
    print()
if __name__ == "__main__":
    board = [
        [1, -1, 1],
        [0, -1, 0],
        [0, 1, 0]
    ]
    print("Current Board:")
    print_board(board)
    move = best_move(board)
    print(f"Best move for X: {move}")
```

RESULT:

The best move for the current player that maximizes their minimum guaranteed outcome (for the maximizer) or minimizes the opponent's maximum gain (for the minimizer).

2.C)Implementation of Backward Chaining

AIM:

To implement Backward Chaining to determine whether a goal (query) can be inferred from a set of known facts and rules.

CODE:

```
def backward_chaining(rules, facts, goal):
    inferred_facts = set(facts)
    agenda = [goal]
    path = {}

    while agenda:
        current_goal = agenda.pop(0)

        if current_goal in inferred_facts:
            continue

        for rule in rules:
            if rule['consequent'] == current_goal:
                all_premises_true = True
                for premise in rule['antecedents']:
                    if premise not in inferred_facts:
                        agenda.append(premise)
                        path[premise] = current_goal
                        all_premises_true = False
                        break

                if all_premises_true:
                    inferred_facts.add(current_goal)
                    print(f"Inferred: {current_goal}")
                    break
```

```
    return goal in inferred_facts


def construct_path(path, goal):
    current = goal
    full_path = [goal]
    while current in path:
        current = path[current]
        full_path.append(current)
    return full_path[::-1]

rules = [
    {'antecedents': ['A', 'B'], 'consequent': 'C'},
    {'antecedents': ['C', 'D'], 'consequent': 'E'},
    {'antecedents': ['F'], 'consequent': 'D'},
    {'antecedents': ['G'], 'consequent': 'A'},
]

facts = ['A','F','B','G']
goal = 'E'

if backward_chaining(rules, facts, goal):
    print(f"Goal '{goal}' can be proven.")
else:
    print(f"Goal '{goal}' cannot be proven.")
```

OUTPUT:

Inferred: C

Goal 'E' cannot be proven.

RESULT:

The code is executed as expected and the output have been verified successfully.

2.D)Implementation of Forward Chaining

AIM:

To implement Forward Chaining to derive all possible conclusions (facts) from a given knowledge base of rules and initial known facts.

CODE:

```
def forward_chaining(rules, facts, goal):
   inferred_facts = set(facts)
   agenda = list(facts)
   path = {}

   while agenda:
       current_fact = agenda.pop(0)

       for rule in rules:
           if all(antecedent in inferred_facts for antecedent in
rule['antecedents']):
               if rule['consequent'] not in inferred_facts:
                   inferred_facts.add(rule['consequent'])
                   agenda.append(rule['consequent'])
                   path[rule['consequent']] = rule['antecedents']
                   print(f"Inferred: {rule['consequent']} from
{rule['antecedents']}")

   return goal in inferred_facts, path


def construct_path(path, goal):
 if goal not in path:
   return [goal]

 full_path = [goal]
```

```
 for antecedents in path[goal]:
     full_path.extend(construct_path(path,antecedents))
 return full_path


rules = [
   {'antecedents': ['A'], 'consequent': 'B'},
   {'antecedents': ['B'], 'consequent': 'C'},
   {'antecedents': ['C', 'D'], 'consequent': 'E'},
   {'antecedents': ['F'], 'consequent': 'D'},
]

facts = ['A', 'F']
goal = 'E'

result, path = forward_chaining(rules, facts, goal)

if result:
   print(f"Goal '{goal}' can be proven.")
   print("Inference Path:", construct_path(path, goal))
else:
   print(f"Goal '{goal}' cannot be proven.")
```

OUTPUT:

Inferred: B from ['A']
Inferred: C from ['B']
Inferred: D from ['F']
Inferred: E from ['C', 'D']
Goal 'E' can be proven.
Inference Path: ['E', 'C', 'B', 'A', 'D', 'F']

RESULT:

The code is executed as expected and the output have been verified successfully.

3.D)Implementation of Decision Tree

AIM:

To implement a Decision Tree classifier that can learn from labeled data and predict the label of new examples using a tree structure based on feature values.

CODE:

```
from sklearn import tree
X = [
   [150, 40],
   [160, 55],
   [170, 65],
   [155, 48],
   [180, 70],
   [165, 50]
]
y = [0, 0, 1, 0, 1, 1]
model = tree.DecisionTreeClassifier()

model.fit(X, y)

prediction = model.predict([[158, 52]])

if prediction[0] == 0:
   print("Predicted: Girl")
else:
   print("Predicted: Boy")
```

OUTPUT:

Predicted: Girl

RESULT:

The code is executed as expected and the output have been verified successfully.

3.E)Implementation of K-mean algorithm

AIM:

To implement the K-Means algorithm to cluster unlabeled data into K distinct groups based on feature similarity.

CODE:

```
import random

def euclidean_distance(point1, point2):
   squared_diff = [(a - b)**2 for a, b in zip(point1, point2)]
   return sum(squared_diff)**0.5

def k_means(data, k, max_iterations=100):

   centroids = random.sample(data, k)

   for _ in range(max_iterations):

       clusters = [[] for _ in range(k)]
       for point in data:
           distances = [euclidean_distance(point, centroid) for centroid
in centroids]
           cluster_index = distances.index(min(distances))
           clusters[cluster_index].append(point)

       new_centroids = []
       for cluster in clusters:
           if cluster:
               new_centroids.append([sum(dim) / len(cluster) for dim in
zip(*cluster)])
           else:
               new_centroids.append(centroids[clusters.index(cluster)])

       if new_centroids == centroids:
```

```
            break

        centroids = new_centroids

    return centroids, clusters

data = [[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [9, 11]]
k = 2

centroids, clusters = k_means(data, k)

print("Centroids:", centroids)
print("Clusters:", clusters)
```

OUTPUT:

Centroids: [[7.333333333333333, 9.0], [1.1666666666666667, 1.4666666666666666]]
Clusters: [[[5, 8], [8, 8], [9, 11]], [[1, 2], [1.5, 1.8], [1, 0.6]]]

RESULT:

The code is executed as expected and the output have been verified successfully.