



SAVEETHA

SCHOOL OF ENGINEERING

Name of the Student :

Register Number :

Department :

Semestor :

Subject :

LABORATORY RECORD NOTE BOOK



SAVEETHA

INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES
(Declared as Deemed to be University under Section 3 of UGC Act 1956)

Saveetha Nagar, Thandalam, Chennai - 602 105,
Tamil Nadu, India. Phone : +91 44 6672 6672
Website : www.saveetha.com,
Email : principal.sse@saveetha.com



SAVEETHA

SCHOOL OF ENGINEERING

Department Of

LABORATORY RECORD NOTE BOOK

20 - 20

This is certify that this is a bonafide record of that work done by

Mr. / Ms Register Number

of the year B.E / B.Tech., Department of

in the Laboratory in the Semester

University Examination held on

Staff in - Charge

Head of the Department

Internal Examiner

External Examiner



SAVEETHA

INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES
(Declared as Deemed to be University under Section 3 of UGC Act 1956)

Saveetha Nagar, Thandalam, Chennai - 602 105,
Tamil Nadu, India. Phone : +91 44 6672 6672
Website : www.saveetha.com,
Email : principal.sse@saveetha.com



SAVEETHA INSTITUTE OF MEDICAL AND TECHNICAL SCIENCES
SIMATS ENGINEERING



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SUBCODE & NAME: CSA17 – ARTIFICIAL INTELLIGENCE

LIST OF EXPERIMENTS

S. No	LIST OF PROGRAMS
1	Write the python program to solve 8-Puzzle problem
2	Write the python program to solve 8-Queen problem
3	Write the python program for Water Jug Problem
4	Write the python program for Crypt-Arithmetic problem
5	Write the python program for Missionaries Cannibal problem
6	Write the python program for Vacuum Cleaner problem
7	Write the python program to implement BFS.
8	Write the python program to implement DFS.
9	Write the python to implement Travelling Salesman Problem
10	Write the python program to implement A* algorithm
11	Write the python program for Map Coloring to implement CSP.
12	Write the python program for Tic Tac Toe game
13	Write the python program to implement Minimax algorithm for gaming
14	Write the python program to implement Alpha & Beta pruning algorithm for gaming
15	Write the python program to implement Decision Tree
16	Write the python program to implement Feed forward neural Network
17	Write a Prolog Program to Sum the Integers from 1 to n.
18	Write a Prolog Program for A DB WITH NAME, DOB.
19	Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.
20	Write a Prolog Program for PLANETS DB.
21	Write a Prolog Program to implement Towers of Hanoi.
22	Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.
23	Write the prolog program to implement family tree.
24	Write a Prolog Program to suggest Dieting System based on Disease.
25	Write a Prolog program to implement Monkey Banana Problem
26	Write a Prolog Program for fruit and its color using Back Tracking.
27	Write a Prolog Program to implement Best First Search algorithm
28	Write the prolog program for Medical Diagnosis
29	Write a Prolog Program for forward Chaining. Incorporate required queries.
30	Write a Prolog Program for backward Chaining. Incorporate required queries.
31	Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc.

1. Write the python program to solve 8-Puzzle problem

AIM:

To write the python program to solve the 8 Puzzle problem

ALGORITHM:

1. Define Start and Goal States: Set up the initial and goal configurations.
2. Choose Heuristic: Use Manhattan Distance or Misplaced Tiles to estimate distance to the goal.
3. Initialize Priority Queue: Add the start state with priority based on $f(\text{state}) = g + h$.
4. Expand and Track Moves: Repeatedly extract the state with the lowest cost, generate valid moves, and add unvisited states to the queue.
5. Goal Check: If the goal state is reached, output the solution; if not, continue searching.

PROGRAM:

```
import heapq

def heuristic(board, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0:
                x, y = divmod(goal.index(board[i][j]), 3)
                distance += abs(x - i) + abs(y - j)
    return distance

def get_neighbors(board):
    x, y = [(ix, iy) for ix, row in enumerate(board) for iy, i in enumerate(row) if i == 0][0]
    directions = [("up", x - 1, y), ("down", x + 1, y), ("left", x, y - 1), ("right", x, y + 1)]
    neighbors = []
    for move, nx, ny in directions:
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_board = [row[:] for row in board]
            new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
            neighbors.append((new_board, move))
    return neighbors

def solve_puzzle(start, goal):
    goal_flat = [item for sublist in goal for item in sublist]
```

```

start_state = (heuristic(start, goal_flat), 0, start, "")
open_list = []
heapq.heappush(open_list, start_state)
closed_set = set()
while open_list:
    _, depth, current_board, path = heapq.heappop(open_list)
    if current_board == goal:
        return path.split()
    closed_set.add(tuple(map(tuple, current_board)))
    for neighbor, move in get_neighbors(current_board):
        if tuple(map(tuple, neighbor)) not in closed_set:
            heapq.heappush(open_list, (heuristic(neighbor, goal_flat) + depth + 1, depth + 1, neighbor,
path + " " + move))
    return None
start = [
    [1, 2, 3],
    [4, 0, 5],
    [7, 8, 6]
]
goal = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
solution = solve_puzzle(start, goal)
if solution:
    print("Solution found!")
    for step in solution:
        print(step)
else:
    print("No solution exists.")

```


INPUT AND OUTPUT:

Input:

```
python
```

```
initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

Output:

```
plaintext
```

```
Solution steps:  
Move blank left  
Move blank down  
...  
Goal state reached!
```

Result:

The python program to solve the 8 Puzzle problem was executed

2 Write the python program to solve 8-Queen problem

AIM:

To write the python program to solve the 8 Queen problem

ALGORITHM:

1. Initialize the Board: Create an $N \times N \times N$ board filled with 0s, where a 1 will represent a queen.
2. Define Safety Check: Create a function to check if placing a queen at any position (row,col) (row,col) is safe (no other queens in the same column, upper-left diagonal, or upper-right diagonal).
3. Place Queens with Backtracking:
 - Start from the first row and attempt to place a queen in each column of the current row.
 - If the position is safe, place the queen and recursively try to place a queen in the next row.
 - If placing a queen in any column of a row fails, backtrack by removing the queen from the previous row and try the next column.
4. Check for Completion:
 - If queens are successfully placed in all rows, a solution is found. Print or store the board configuration.
5. Repeat for All Solutions (Optional): After finding a solution, backtrack further to explore other possible solutions if needed.

PROGRAM:

```
def is_safe(board, row, col):  
    # Check this row on the left side  
    for i in range(col):  
        if board[row][i] == 1:  
            return False  
    # Check the upper diagonal on the left side  
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  
        if board[i][j] == 1:  
            return False  
    # Check the lower diagonal on the left side  
    for i, j in zip(range(row, len(board), 1), range(col, -1, -1)):  
        if board[i][j] == 1:  
            return False  
    return True
```

```

def solve_n_queens(board, col):
    if col >= len(board):
        return True
    for i in range(len(board)):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_n_queens(board, col + 1):
                return True
            board[i][col] = 0
    return False

def print_board(board):
    for row in board:
        print(" ".join(str(x) for x in row))

def solve():
    n = 8
    board = [[0 for _ in range(n)] for _ in range(n)]
    if solve_n_queens(board, 0):
        print_board(board)
    else:
        print("No solution exists")

solve()

```

Input:

```

python

board_size = 8

```

Output:

```

plaintext

Solution:
. Q . . . . .
. . . . Q . .
Q . . . . . .
...

```

Result:

The python program to solve the 8 Queen problem was executed.

3. Write the python program for Water Jug Problem

AIM:

To write the python program to solve the Water Jug problem

ALGORITHM:

1. Define Capacities and Goal: Set jug sizes and target volume.
2. Initialize State: Start with both jugs empty.
3. Generate Moves: Fill, empty, or pour between jugs.
4. Search with BFS: Use a queue to explore moves, tracking visited states.
5. Check Goal: If target volume is reached, return the solution; otherwise, continue searching.

Program:

```
def water_jug_problem(jug1_capacity, jug2_capacity, target):  
    jug1 = 0  
    jug2 = 0  
    while jug1 != target and jug2 != target:  
        if jug2 == jug2_capacity:  
            jug2 = 0  
        elif jug1 == 0:  
            jug1 = jug1_capacity  
        else:  
            amount_to_pour = min(jug1, jug2_capacity - jug2)  
            jug1 -= amount_to_pour  
            jug2 += amount_to_pour  
    return jug1, jug2  
  
if __name__ == "__main__":  
    jug1_capacity = 4  
    jug2_capacity = 3  
    target = 2  
  
    jug1_final, jug2_final = water_jug_problem(jug1_capacity, jug2_capacity, target)  
    print(f"Final state: Jug1: {jug1_final} gallons, Jug2: {jug2_final} gallons")
```

Input:

```
python

jug1_capacity = 4
jug2_capacity = 3
target_amount = 2
```

Output:

```
plaintext

Steps to reach target:
Fill Jug1
Pour Jug1 into Jug2
...
Reached 2 liters in Jug1.
```

Result:

The python program to solve the WATER JUG problem was executed.

4 Write the python program for Cript-Arithmetic problem.

AIM:

To write the python program to solve the Cript-Arithmetic problem

ALGORITHM:

1. Define Variables and Constraints: Identify unique letters as variables and ensure no two letters map to the same digit. Set constraints (e.g., sum equals the target number).
2. Assign Digits with Backtracking: Try assigning digits (0-9) to each letter, ensuring unique assignments.
3. Check Partial Solutions: For each assignment, check if it satisfies partial sums of the target (to prune invalid paths early).
4. Validate Solution: When all letters are assigned, verify if the arithmetic equation holds.
5. Return or Backtrack: If valid, return the solution. If not, backtrack and try other assignments.

Program:

```
import itertools

def is_solution(s, m, o, r, y, e, n, d):
    # Form the two numbers from the letters
    send = s * 1000 + e * 100 + n * 10 + d
    more = m * 1000 + o * 100 + r * 10 + e
    money = m * 10000 + o * 1000 + n * 100 + e * 10 + y
    # Check if the sum of SEND and MORE equals MONEY
    return send + more == money

# List of unique digits (0-9) and the unique letters in the problem
digits = range(10)
letters = 'smoreynd'

# Try all possible permutations of the digits
for perm in itertools.permutations(digits, len(letters)):
    # Map letters to the corresponding digits
    mapping = dict(zip(letters, perm))
    # Extract the digits
    s, m, o, r, e, y, n, d = [mapping[letter] for letter in letters]
    # Skip permutations where S or M are zero, as leading zeros are not allowed
    if s == 0 or m == 0:
        continue
```

```
# Check if the current permutation solves the problem
if is_solution(s, m, o, r, y, e, n, d):
    print(f"SEND + MORE = MONEY: {mapping}")
    break
```

Input:

```
python

problem = "SEND + MORE = MONEY"
```

Output:

```
plaintext

Solution:
S=9, E=5, N=6, D=7, M=1, O=0, R=8, Y=2
```

Result:

The python program to solve the Crypt arithmetic problem was executed.

5. Write the python program for Missionaries Cannibal problem from collections import deque

AIM:

To write the python program to solve the Missionaries Cannibal problem from collections import deque

ALGORITHM:

- Initial state: All missionaries (M) and cannibals (C) are on the left bank.
- Goal: Move all M and C to the right bank.
- Constraints: At no point on either bank should cannibals outnumber missionaries if there are any missionaries present.

PROGRAM:

```
def is_valid_state(m, c):  
    if m < 0 or c < 0 or m > 3 or c > 3 or (m != 0 and m < c) or (m != 3 and (3 - m) < (3 - c)):  
        return False  
    return True  
  
def get_possible_moves(m, c, b):  
    moves = []  
    if b == 1:  
        for i in range(1, min(m, 2) + 1):  
            for j in range(0, min(c, i) + 1):  
                if is_valid_state(m - i, c - j):  
                    moves.append((m - i, c - j, 0))  
    else:  
        for i in range(1, min(3 - m, 2) + 1):  
            for j in range(0, min(3 - c, i) + 1):  
                if is_valid_state(m + i, c + j):  
                    moves.append((m + i, c + j, 1))  
    return moves  
  
def bfs():  
    start_state = (3, 3, 1)  
    goal_state = (0, 0, 0)  
    queue = deque([(start_state, [])])  
    visited = set([start_state])  
    while queue:  
        (m, c, b), path = queue.popleft()
```

```

if (m, c, b) == goal_state:
    return path
for move in get_possible_moves(m, c, b):
    if move not in visited:
        visited.add(move)
        queue.append((move, path + [(m, c, b)]))
return None

def print_solution(solution):
    if solution:
        print("Solution found!")
        for i, (m, c, b) in enumerate(solution):
            direction = "left to right" if b == 1 else "right to left"
            print(f"Step {i + 1}: Move {m} missionaries and {c} cannibals from {direction}.")
    else:
        print("No solution found.")

if __name__ == "__main__":
    solution = bfs()
    print_solution(solution)

```

Input:

```

python

missionaries = 3
cannibals = 3
boat_capacity = 2

```

Output:

```

plaintext

Steps:
Move 2 Cannibals to other side
Move 1 Cannibal back
Move 2 Missionaries to other side
...
All missionaries and cannibals safely across!

```

Result:

The python program to solve the Missionaries Cannabials problem was executed.

6. Write the python program for Vacuum Cleaner problem

AIM:

To write the python program to solve the Vacuum Cleaner problem

ALGORITHM:

- Environment: Two rooms (Room A and Room B).
- States: Each room can be clean or dirty, and the vacuum cleaner can be in either room.
- Actions: The vacuum can:
 - Suck: Clean the current room.
 - Move Left: Move to Room A if in Room B.
 - Move Right: Move to Room B if in Room A.

Goal

Clean both rooms.

PROGRAM:

```
class VacuumCleaner:
```

```
    def __init__(self, grid, start_position):
```

```
        self.grid = grid
```

```
        self.position = start_position
```

```
        self.cleaned_cells = 0
```

```
    def clean(self):
```

```
        x, y = self.position
```

```
        if self.grid[x][y] == 1:
```

```
            self.grid[x][y] = 0
```

```
            self.cleaned_cells += 1
```

```
    def move(self, direction):
```

```
        x, y = self.position
```

```
        if direction == 'UP' and x > 0:
```

```
            self.position = (x - 1, y)
```

```
        elif direction == 'DOWN' and x < len(self.grid) - 1:
```

```
            self.position = (x + 1, y)
```

```
        elif direction == 'LEFT' and y > 0:
```

```
            self.position = (x, y - 1)
```

```
        elif direction == 'RIGHT' and y < len(self.grid[0]) - 1:
```

```
            self.position = (x, y + 1)
```



```

def start_cleaning(self, movements):
    for move in movements:
        self.clean()
        self.move(move)
    self.clean()
# Define the grid and start position
grid = [
    [1, 1, 0, 1],
    [0, 1, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 1, 0]
]
start_position = (0, 0)
# Create VacuumCleaner instance and define movements
vacuum = VacuumCleaner(grid, start_position)
movements = ['RIGHT', 'DOWN', 'DOWN', 'LEFT', 'UP', 'UP', 'RIGHT', 'RIGHT', 'DOWN']
# Start cleaning
vacuum.start_cleaning(movements)
print("Cleaned cells:", vacuum.cleaned_cells)
print("Final grid state:")
for row in grid:
    print(row)

```

Input:

```

python

room_state = {"A": "Dirty", "B": "Clean"}
initial_position = "A"

```

Output:

```

plaintext

Cleaning Room A
Moving to Room B
Room B is already clean
All rooms are clean!

```

Result:

The python program to solve the Vacuum cleaner problem was executed.

7. Write the python program to implement BFS

AIM:

To write the python program to implement BFS

ALGORITHM:

- Environment: A grid (for simplicity, assume two rooms).
- States: Each room can either be clean or dirty.
- Actions: The vacuum can:
 - Suck: Clean the current room.
 - Move Left: Move to Room A if in Room B.
 - Move Right: Move to Room B if in Room A.

The goal is to clean both rooms.

PROGRAM:

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)
    while queue:
        node = queue.popleft()
        print(node, end=' ')
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)


# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
```

```
start_node = 'A'
```

```
bfs(graph, start_node)
```

Input:


python

 Copy code

```
graph = {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []}  
start_node = 'A'  
goal_node = 'F'
```

Output:

plaintext

 Copy code

```
Path found: A -> C -> F
```

Result:

The python program to solve the BFS was executed.

8 .Write the python program to implement DFS.

AIM:

To write the python program to implement DFS

ALGORITHM:

- Environment: Two rooms (Room A and Room B).
- States: Each room can either be "Clean" or "Dirty".
- Actions:
 - Suck: Clean the current room.
 - Move Left: Move to Room A if in Room B.
 - Move Right: Move to Room B if in Room A.

The goal is to clean both rooms.

PROGRAM:

```
def dfs(graph, start, visited=None):
```

```
    if visited is None:
```

```
        visited = set()
```

```
    visited.add(start)
```

```
    print(start, end=' ')
```

```
    for neighbor in graph[start]:
```

```
        if neighbor not in visited:
```

```
            dfs(graph, neighbor, visited)
```

```
# Example usage
```

```
graph = {
```

```
    'A': ['B', 'C'],
```

```
    'B': ['A', 'D', 'E'],
```

```
    'C': ['A', 'F'],
```

```
    'D': ['B'],
```

```
    'E': ['B', 'F'],
```

```
    'F': ['C', 'E']
```


```
}
```

```
start_node = 'A'
```

```
dfs(graph, start_node)
```

Input:


python

 Copy code

```
graph = {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []}  
start_node = 'A'  
goal_node = 'F'
```

Output:

plaintext

 Copy code

```
Path found: A -> B -> E -> F
```

Result:

The python program to solve the DFS was executed.

9. Write the python to implement Travelling Salesman Problem

AIM:

To write the python program to implement Travelling Salesman Problem

ALGORITHM:

1. Input: Distance matrix dist_matrix of size n x n.
2. Generate Permutations: Generate all permutations of cities (except the starting city).
3. Calculate Total Distance: For each permutation, calculate the total distance, including returning to the starting city.
4. Find Minimum Distance: Track the permutation with the minimum total distance.
5. Output: Return the optimal route and its total distance.

PROGRAM:

```
from itertools import permutations

def calculate_total_distance(graph, path):
    return sum(graph[path[i]][path[i+1]] for i in range(len(path)-1)) + graph[path[-1]][path[0]]


def travelling_salesman(graph):
    cities = list(graph.keys())
    min_distance = float('inf')
    best_path = []
    for perm in permutations(cities):
        current_distance = calculate_total_distance(graph, perm)
        if current_distance < min_distance:
            min_distance = current_distance
            best_path = perm
    return best_path, min_distance

# Example usage
graph = {
    'A': {'A': 0, 'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'B': 0, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'C': 0, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30, 'D': 0}
}

best_path, min_distance = travelling_salesman(graph)
print("Best path:", best_path)
print("Minimum distance:", min_distance)
```

Input:


python

 Copy code

```
cities = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
```

Output:

plaintext

 Copy code

Minimum cost path: 80

Path: 0 -> 1 -> 3 -> 2 -> 0

Result:

The python program to solve the Travelling Salesman problem was executed.

10. Write the python program to implement A* algorithm

AIM:

To write the python program to implement A* algorithm

ALGORITHM:

1. Initialize: Add the start node to the open list with $f = g + h$, where g is the cost from start and h is the heuristic.
2. Loop:
 - Select the node with the lowest f from the open list.
 - If it's the goal, return the path.
 - Otherwise, move it to the closed list and evaluate its neighbors.
3. Update: For each neighbor, calculate $f = g + h$. If it has a lower f , update and add it to the open list.
4. End: If the open list is empty and the goal isn't found, return failure.

PROGRAM:

```
import heapq

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1]) # Manhattan distance

def astar(start, goal, graph):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0
    f_score = {node: float('inf') for node in graph}
    f_score[start] = heuristic(start, goal)
    while open_set:
        _, current = heapq.heappop(open_set)
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]
```

```

for neighbor in graph[current]:
    tentative_g_score = g_score[current] + graph[current][neighbor]
    if tentative_g_score < g_score[neighbor]:
        came_from[neighbor] = current
        g_score[neighbor] = tentative_g_score
        f_score[neighbor] = g_score[neighbor] + heuristic(neighbor, goal)
        heapq.heappush(open_set, (f_score[neighbor], neighbor))

return None

# Example usage
graph = {
    (0, 0): {(0, 1): 1, (1, 0): 1},
    (0, 1): {(0, 0): 1, (1, 1): 1},
    (1, 0): {(0, 0): 1, (1, 1): 1},
    (1, 1): {(0, 1): 1, (1, 0): 1}
}

start, goal = (0, 0), (1, 1)
path = astar(start, goal, graph)
print("Path:", path)

```

Input:

```

python

graph = {...}
start_node = 'A'
goal_node = 'G'

```

Output:

```

plaintext

Path found: A -> C -> G
Total cost: 5

```

Result:

The python program to solve the A* algorithm problem was executed.

11. Write the python program for Map Coloring to implement CSP.

AIM:

To write the python program for Map Coloring to implement CSP.

ALGORITHM:

1. Initialize: List regions and their possible colors.
2. Assign Color: Choose an uncolored region and assign a valid color.
3. Backtrack: If no valid color, backtrack and try a different color.
4. Repeat: Continue until all regions are colored or no solution is found.
5. End: Return the solution or failure.

PROGRAM:


```
def is_valid(coloring, node, color, graph):  
    for neighbor in graph[node]:  
        if coloring.get(neighbor) == color:  
            return False  
    return True  
  
def map_coloring(graph, colors, coloring={}, node_list=None):  
    if node_list is None:  
        node_list = list(graph.keys())  
    if not node_list:  
        return coloring  
    node = node_list[0]  
    remaining_nodes = node_list[1:]  
    for color in colors:  
        if is_valid(coloring, node, color, graph):  
            coloring[node] = color  
            result = map_coloring(graph, colors, coloring, remaining_nodes)  
            if result:  
                return result  
            del coloring[node]  
    return None
```

Example usage

```
graph = {  
    'A': ['B', 'C'],  
    'B': ['A', 'D', 'E'],  
    'C': ['A', 'F'],  
    'D': ['B'],  
    'E': ['B', 'F'],  
    'F': ['C', 'E']  
}  
colors = ['Red', 'Green', 'Blue']  
coloring = map_coloring(graph, colors)  
print("Coloring:", coloring)
```

Input:


python

 Copy code

```
regions = {'WA': ['SA', 'NT'], 'NT': ['WA', 'SA', 'Q'], 'SA': ['WA', 'NT', 'Q', 'NSW', 'V'],  
           'Q': ['NT', 'SA', 'WA', 'NSW', 'V'], 'NSW': ['SA', 'Q', 'WA', 'V'], 'V': ['SA', 'Q', 'NSW', 'WA']}  
colors = ['Red', 'Green', 'Blue']
```

Output:

plaintext

 Copy code

Solution:

WA = Red, NT = Green, SA = Red, Q = Green, NSW = Blue, V = Green, T = Red

Result:

The python program to solve the map coloring problem was executed.

12. Write the python program for Tic Tac Toe game

AIM:

To write the python program for Tic Tac Toe game.

ALGORITHM:

1. Initialize: Create a 3x3 grid and set it as empty.
2. Player Move:
 - Alternate between Player 1 (X) and Player 2 (O).
 - Each player places their symbol in an empty cell.
3. Check Win: After each move, check for a winner (three same symbols in a row, column, or diagonal).
4. Check Draw: If all cells are filled and no winner, declare a draw.
5. End: Game ends when a player wins or a draw occurs.

PROGRAM:

```
def print_board(board):
    for row in board:
        print(' '.join(row))
    print()

def check_winner(board, player):
    win_conditions = [
        [board[0][0], board[0][1], board[0][2]], # Row 1
        [board[1][0], board[1][1], board[1][2]], # Row 2
        [board[2][0], board[2][1], board[2][2]], # Row 3
        [board[0][0], board[1][0], board[2][0]], # Column 1
        [board[0][1], board[1][1], board[2][1]], # Column 2
        [board[0][2], board[1][2], board[2][2]], # Column 3
        [board[0][0], board[1][1], board[2][2]], # Diagonal 1
        [board[2][0], board[1][1], board[0][2]] # Diagonal 2
    ]
    return [player, player, player] in win_conditions

def tic_tac_toe():
    board = [[' ']*3 for _ in range(3)]
    players = ['X', 'O']
    turn = 0
```

```

for _ in range(9):
    print_board(board)
    player = players[turn % 2]
    row, col = map(int, input(f"Player {player}, enter row and column (0-2): ").split())

    if board[row][col] != ' ':
        print("Cell already taken, try again.")
        continue
    board[row][col] = player
    if check_winner(board, player):
        print_board(board)
        print(f"Player {player} wins!")
        return
    turn += 1
print_board(board)
print("It's a tie!")
# Run the game
tic_tac_toe()

```

Input:

```
python
```

```
moves = [(1, 1), (0, 0), (2, 2), (0, 1), (1, 0), (1, 2)]
```

Output:

```
plaintext
```

```
Board:
```

```
X | O | .
```

```
-----
```

```
O | X | .
```

```
-----
```

```
. | . | X
```

```
Player X wins!
```

Result:

The python program to solve the Tic Toc Toe problem was executed.

13. Write the python program to implement Minimax algorithm for gaming

AIM:

To write the python program to implement Minimax algorithm for gaming

ALGORITHM:

1. Initialize: Define game state and possible moves.
2. Maximize: Maximize score for the current player.
3. Minimize: Minimize score for the opponent.
4. Evaluate: Use a heuristic to score terminal states.
5. Backtrack: Recursively compute and choose the optimal move.

PROGRAM:

```
import math

def minimax(board, depth, is_maximizing):
    if check_winner(board, 'X'):
        return 10 - depth
    if check_winner(board, 'O'):
        return depth - 10
    if not any(cell == '' for row in board for cell in row):
        return 0
    if is_maximizing:
        best_score = -math.inf
        for (i, j) in get_empty_cells(board):
            board[i][j] = 'X'
            score = minimax(board, depth + 1, False)
            board[i][j] = ''
            best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for (i, j) in get_empty_cells(board):
            board[i][j] = 'O'
            score = minimax(board, depth + 1, True)
            board[i][j] = ''
            best_score = min(score, best_score)
        return best_score
```



```

def get_empty_cells(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def check_winner(board, player):
    win_conditions = [board[i] for i in range(3)] + \
        [[board[i][j] for i in range(3)] for j in range(3)] + \
        [[board[i][i] for i in range(3)]] + \
        [[board[i][2-i] for i in range(3)]]
    return [player]*3 in win_conditions

def best_move(board):
    best_score = -math.inf
    move = None
    for (i, j) in get_empty_cells(board):
        board[i][j] = 'X'
        score = minimax(board, 0, False)
        board[i][j] = ' '
        if score > best_score:
            best_score = score
            move = (i, j)
    return move

# Example usage
board = [[' ']*3 for _ in range(3)]
move = best_move(board)
print("Best move:", move)

```

Input:

python

```
initial_board = [['', '', ''], ['', '', ''], ['', '', '']]
```

Output:

plaintext

```
Best move for AI: (1, 1)
```

Result:

The python program to solve the Minmax algorithm was executed.

14. Write the python program to implement Apha & Beta pruning algorithm for gaming

AIM:

To write the python program to implement Apha & Beta pruning algorithm for gaming

ALGORITHM:

1. Initialize: Set $\alpha = -\infty$ (max score for maximizer) and $\beta = +\infty$ (min score for minimizer).
2. Traverse: Recursively evaluate the game tree, alternating between maximizing and minimizing players.
3. Prune:
 - If $\alpha \geq \beta$, stop evaluating further nodes (prune the branch).
 - Update α or β during the traversal to prune irrelevant branches.
4. Evaluate: Use the minimax evaluation function to score terminal nodes.
5. End: Return the optimal move based on the pruned search tree.

PROGRAM:

```
import math

def alpha_beta(board, depth, alpha, beta, is_maximizing):
    if check_winner(board, 'X'):
        return 10 - depth
    if check_winner(board, 'O'):
        return depth - 10
    if not any(cell == '' for row in board for cell in row):
        return 0
    if is_maximizing:
        best_score = -math.inf
        for (i, j) in get_empty_cells(board):
            board[i][j] = 'X'
            score = alpha_beta(board, depth + 1, alpha, beta, False)
            board[i][j] = ''
            best_score = max(score, best_score)
            alpha = max(alpha, best_score)
            if beta <= alpha:
                break
        return best_score
    else:
```

```

best_score = math.inf

for (i, j) in get_empty_cells(board):
    board[i][j] = 'O'
    score = alpha_beta(board, depth + 1, alpha, beta, True)
    board[i][j] = ''
    best_score = min(score, best_score)
    beta = min(beta, best_score)
    if beta <= alpha:
        break
return best_score

def get_empty_cells(board):
    return [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']

def check_winner(board, player):
    win_conditions = [board[i] for i in range(3)] + \
        [[board[i][j] for i in range(3)] for j in range(3)] + \
        [[board[i][i] for i in range(3)]] + \
        [[board[i][2-i] for i in range(3)]]
    return [player]*3 in win_conditions

def best_move(board):
    best_score = -math.inf
    move = None
    alpha = -math.inf
    beta = math.inf
    for (i, j) in get_empty_cells(board):
        board[i][j] = 'X'
        score = alpha_beta(board, 0, alpha, beta, False)
        board[i][j] = ''
        if score > best_score:
            best_score = score
            move = (i, j)
    return move

```

```
# Example usage
board = [[' ']*3 for _ in range(3)]
move = best_move(board)
print("Best move:", move)
```

Input:

```
python

tree = {...}
```

Output:

```
plaintext

Optimal value with pruning: 3
```

Result:

The python program to solve the Alpha and Beta Pruning algorithm was executed.

15. Write the python program to implement Decision Tree

AIM:

To write the python program to implement Decision Tree

ALGORITHM:

1. **Select Attribute:** Choose the best attribute to split the data (often using metrics like Gini or Information Gain).
2. **Split Data:** Divide the data into subsets based on the chosen attribute values.
3. **Create Nodes:** For each subset, create a tree node and connect it to the parent node.
4. **Repeat:** Recursively apply steps 1–3 on each subset until reaching a stopping condition (e.g., all data in a subset belong to one class).
5. **End:** The tree is complete when all branches end in a decision (leaf) node.

PROGRAM:

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy_score

# Load dataset from CSV file

data = pd.read_csv('IRIS.csv')

X = data.drop('species', axis=1) # assuming 'species' is the target column

y = data['species']

# Split the dataset into training and testing sets

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize and train the Decision Tree classifier

clf = DecisionTreeClassifier()

clf.fit(X_train, y_train)

# Make predictions on the test set


y_pred = clf.predict(X_test)

# Print the accuracy of the model

print(f"Accuracy: {accuracy_score(y_test, y_pred):.2f}")
```

Input:


python

 Copy code

```
data = {"age": [22, 35, 26, ...], "income": ["high", "medium", ...], "buys": ["yes", "no",
```

Output:

plaintext

 Copy code

```
Decision tree:  
IF age <= 30 THEN buys = yes  
...
```

Result:

The python program to solve the Decision tree problem was executed.

16. Write the python program to implement Feed forward neural Network

AIM:

To write the python program to implement Feed forward neural Network

ALGORITHM:

1. Initialize: Set up layers, weights, and biases.
2. Forward Pass: Compute outputs layer by layer.
3. Loss: Calculate error between output and target.
4. Backpropagate: Calculate gradients.
5. Update: Adjust weights and biases to reduce loss.

PROGRAM:

```
import pandas as pd

from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import LabelEncoder

# Load dataset from CSV file
data = pd.read_csv('IRIS.csv')

# Separate features and target variable
X = data.drop('species', axis=1) # Assuming 'species' is the target column
y = data['species']

# Encode the target variable if it's categorical
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.3, random_state=42)

# Initialize and train the Feedforward Neural Network
clf = MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000, random_state=42)
clf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = clf.predict(X_test)

# Decode the predictions back to original labels
y_pred_labels = label_encoder.inverse_transform(y_pred)
```



```
# Decode the actual test labels back to original labels
y_test_labels = label_encoder.inverse_transform(y_test)

# Print the accuracy of the model
print(f"Accuracy: {accuracy_score(y_test_labels, y_pred_labels):.2f}")
```

Input:

```
python

inputs = [0.5, 0.1, 0.4]
weights = {...}
```

Output:

```
plaintext

Predicted output: [0.76, 0.32, ...]
```

Result:

The python program to solve the Free forward neural network problem was executed.

17 Write a Prolog Program to Sum the Integers from 1 to n

AIM:

To write the prolog program to sum of integers.

ALGORITHM:

1. Define the base case where the sum of integers from 1 to 0 is 0.
2. For any N, the sum is calculated as $N + \text{Sum}(N-1)$.
3. Implement recursion to calculate $\text{Sum}(N)$.
4. Query to obtain the result for any integer N.

Program:

sum(N,Sum):-

Sum is $(N+1)*N/2$.

Example Input and Output:

```
prolog
```

```
?- sum_to_n(5, Sum).
```

```
Sum = 15.
```

Result:

The prolog program to solve the sum of n digits problem was executed.

18 Write a Prolog Program for A DB WITH NAME, DOB.

AIM:

To write the prolog program for A DB with Name and DOB.

ALGORITHM:

1. Define a fact structure person(Name, DOB).
2. Insert multiple entries as facts.
3. Write queries to retrieve DOB by name.
4. Query the database using person/2.

Program:

person(sai, 2003, 11, 24).

dob(Name, Year, Month, Day) :-

 person(Name, Year, Month, Day).

name(Year, Month, Day, Name) :-

 person(Name, Year, Month, Day).

Example Input and Output:

```
prolog

?- person(Name, '1998-05-14').
Name = mary.
```

RESULT:

The prolog program to solve the name with DOB problem was executed.

19 Write a Prolog Program for STUDENT-TEACHER-SUB-CODE.

AIM:

To write the prolog program for STUDENT-TEACHER-SUB-CODE.

ALGORITHM:

1. Define facts as teaches(Teacher, Subject, Code), studies(Student, Subject, Code).
2. Add records for each relationship.
3. Write queries to find which teacher teaches which student.
4. Query to list all subjects taught by a specific teacher.

Program:

```
student(sai, csa1738).
```

```
teacher(kumar, csa1738).
```

```
subject_code(csa1738, 'AI').
```

```
student_subject(Student, SubjectCode) :-
```

```
    student(Student, SubjectCode).
```

```
student_teacher(Student, Teacher) :-
```

```
    student(Student, SubjectCode), teacher(Teacher, SubjectCode).
```

```
teacher_subject(Teacher, SubjectCode) :-
```

```
    teacher(Teacher, SubjectCode).
```

```
subject_name(SubjectCode, SubjectName) :-
```

```
    subject_code(SubjectCode, SubjectName).
```

```
student_subject_name(Student, SubjectName) :-
```

```
    student(Student, SubjectCode), subject_code(SubjectCode, SubjectName).
```

```
student_teacher_subject(Student, Teacher, SubjectName) :-
```

```
    student(Student, SubjectCode), teacher(Teacher, SubjectCode),  
    subject_code(SubjectCode, SubjectName).
```

Example Input and Output:

```
prolog  
  
?- teaches(Teacher, Subject, 101).  
Teacher = mr_smith,  
Subject = math.
```

Result:

The prolog program to solve STUDENT TEACHER SUBCODE problem was executed.

20 Write a Prolog Program for PLANETS DB

AIM:

To write the prolog program for Planets DB .

ALGORITHM:

1. Define each planet as a fact, e.g., planet(Name, Distance, Size, OrbitTime).
2. Input details of each planet.
3. Query to retrieve specific planet information.
4. Query by planet name to get details.

Program:

```
planet(mercury, 1, terrestrial).
```

```
planet(venus, 2, terrestrial).
```

```
planet(earth, 3, terrestrial).
```

```
planet(mars, 4, terrestrial).
```

```
planet(jupiter, 5, gas_giant).
```

```
planet(saturn, 6, gas_giant).
```

```
planet(uranus, 7, gas_giant).
```

```
planet(neptune, 8, gas_giant).
```

```
planetName(Order, Name) :-
```

```
    planet(Name, Order, _).
```

```
planetType(Name, Type) :-
```

```
    planet(Name, _, Type).
```

```
terrestrialPlanets(Planets) :-
```

```
    findall(Name, planet(Name, _, terrestrial), Planets).
```

```
gasGiants(Planets) :-
```

```
    findall(Name, planet(Name, _, gas_giant), Planets).
```

Example Input and Output:

```
prolog

?- planet(Name, medium, Moons).
Name = earth,
Moons = 1.
```

Result:

The prolog program to solve PLANETS DB problem was executed.

21 Write a Prolog Program to implement Towers of Hanoi

AIM:

To write the prolog program to implement Towers of Hanoi.

ALGORITHM:

1. Define base case for moving one disk.
2. For N disks, recursively move N-1 disks to the auxiliary pole.
3. Move the Nth disk directly to the destination.
4. Continue until all disks are moved.

Program:

```
hanoi(0, _, _, _, []) :- !.
```

```
hanoi(N, Source, Destination, Auxiliary, Moves) :-
```

```
    N > 0,
```

```
    N1 is N - 1,
```

```
    hanoi(N1, Source, Auxiliary, Destination, Moves1),
```

```
    MoveN = move(Source, Destination),
```

```
    hanoi(N1, Auxiliary, Destination, Source, Moves2),
```

```
    append(Moves1, [MoveN|Moves2], Moves).
```

Example Input and Output:

```
prolog

?- hanoi(3, left, right, center).
Move disk from left to right.
Move disk from left to center.
Move disk from right to center.
Move disk from left to right.
Move disk from center to left.
Move disk from center to right.
Move disk from left to right.
```

Result:

The prolog program to solve Towers of Hanoi problem was executed.

22 Write a Prolog Program to print particular bird can fly or not. Incorporate required queries.

AIM:

To write the prolog program to print particular bird can fly or not.

ALGORITHM:

1. Define birds that can and cannot fly, e.g., `can_fly(eagle).` and `cannot_fly(penguin).`
2. Query if a particular bird can fly.
3. Include a rule to check if a bird can fly.
4. Provide a response based on the bird type.

Program:

`canfly(sparrow).`

`canfly(eagle).`

`canfly(peacock).`

`cannotfly(ostrich).`

`cannotfly(penguin).`

Example Input and Output:

```
prolog

?- can_fly(eagle).
true.

?- can_fly(penguin).
false.
```

Result:

The prolog program to solve Birds can fly or not problem was executed.

23 Write the prolog program to implement family tree.

AIM:

To write the prolog program to implement Family tree.

ALGORITHM:

1. Define family relations as facts, e.g., parent(Parent, Child).
2. Add rules for relationships like sibling, cousin.
3. Query relationships (e.g., grandparents, siblings).
4. Define rules to answer relationship-based questions.

Program:

```
% Facts
parent(john, mary).
parent(john, david).
parent(mary, susan).
parent(david, tom).
parent(david, anna).
male(john).
male(david).
male(tom).
female(mary).
female(susan).
female(anna).

% Rules
father(F, C) :- parent(F, C), male(F).
mother(M, C) :- parent(M, C), female(M).
sibling(X, Y) :- parent(P, X), parent(P, Y), X \= Y.
```

Example Input and Output:

```
prolog
?- grandparent(john, alice).
true.
```

Result:

The prolog program to solve Family tree problem was executed.

24 Write a Prolog Program to suggest Dieting System based on Disease.

AIM:

To write the prologProgram to suggest Dieting System based on Disease.

ALGORITHM:

1. Define dietary recommendations based on diseases, e.g., diet(diabetes, DietPlan).
2. Input disease-diet pairs.
3. Query for diet suggestions based on disease.
4. Return diet plan according to the queried disease.

Program:

% Facts

diet(diabetes, 'Low sugar, high fiber, whole grains, vegetables, lean protein').

diet(hypertension, 'Low sodium, high potassium, fruits, vegetables, whole grains').

diet(heart_disease, 'Low saturated fat, high omega-3, fruits, vegetables, whole grains').

diet(obesity, 'Balanced diet, portion control, high fiber, lean protein').

diet(anemia, 'High iron, vitamin C, leafy greens, red meat, beans').

diet(gastrointestinal_disorder, 'Low fiber, bland diet, avoid spicy foods, small frequent meals').

% Rules

suggest_diet(Disease, Diet) :- diet(Disease, Diet).

% Example Queries

% suggest_diet(diabetes, Diet).

% suggest_diet(hypertension, Diet).

Example Input and Output:

```
prolog

?- diet(diabetes, Diet).
Diet = low_sugar.
```

Result:

The prolog program to solve Dieting system problem was executed.

25 Write a Prolog program to implement Monkey Banana Problem

AIM:

To write the prolog program to implement Monkey Banana Problem.

ALGORITHM:

1. Define initial state of monkey and banana.
2. Define actions (climb, move, grasp).
3. Recursively apply actions to reach banana.
4. Query the sequence to find if monkey gets banana.

Program:

```
% Define the initial state
initial_state(state(at_door, on_floor, at_window, has_not)).

% Define the goal state
goal_state(state(_, _, _, has)).

% Define the possible actions
action(state(middle, on_box, middle, has_not), grasp, state(middle, on_box, middle, has)).
action(state(P, on_floor, P, H), climb_box, state(P, on_box, P, H)).
action(state(P1, on_floor, P1, H), push_box(P1, P2), state(P2, on_floor, P2, H)).
action(state(P1, on_floor, B, H), walk(P1, P2), state(P2, on_floor, B, H)).

% Define a plan to achieve the goal state
plan(State, [], State) :- goal_state(State).
plan(State1, [Action | RestActions], State3) :-
    action(State1, Action, State2),
    plan(State2, RestActions, State3).

% Query to find the plan
find_plan(Plan) :-
    initial_state(State),
    plan(State, Plan, _).
```

Example Input and Output:

```
prolog
?- can_reach(monkey, banana).
true.
```

Result:

The prolog program to solve Monkey Banana problem was executed.

26 Write a Prolog Program for fruit and its color using Back Tracking

AIM:

To write the prolog program for fruit and its color using Back Tracking.

ALGORITHM:

1. Define fruit-color pairs, e.g., color(apple, red).
2. Query with different fruits for colors.
3. Use backtracking to retrieve multiple colors.
4. Return colors for queried fruits.

Program:

```
% Facts: fruit and its color

fruit_color(apple, red).
fruit_color(banana, yellow).
fruit_color(grape, purple).
fruit_color(lemon, yellow).
fruit_color(orange, orange).
fruit_color(cherry, red).
fruit_color(plum, purple).

% Rule to find fruits of a specific color
fruits_of_color(Color, Fruit) :-
    fruit_color(Fruit, Color).

% Rule to list all fruits of a specific color using backtracking
list_fruits_of_color(Color) :-
    fruits_of_color(Color, Fruit),
    writeln(Fruit),
    fail.

list_fruits_of_color(_).
```

Example Input and Output:

```
prolog

?- fruit(Fruit, yellow).
Fruit = banana.
```

Result:

The prolog program to solve Fruit Coloring problem was executed.

27 Write a Prolog Program to implement Best First Search algorithm

AIM:

To write the prolog program to implement Best First Search algorithm.

ALGORITHM:

1. Define graph edges and heuristic costs.
2. Implement a best-first strategy to traverse nodes.
3. Define goal-checking conditions.
4. Query for shortest path from start to goal.

Program:

```
% Define edges of the graph with their costs
edge(a, b, 1).
edge(a, c, 3).
edge(b, d, 3).
edge(b, e, 6).
edge(c, e, 2).
edge(d, f, 1).
edge(e, f, 2).

% Define the heuristic values (estimated cost to reach the goal)
heuristic(a, 6).
heuristic(b, 4).
heuristic(c, 5).
heuristic(d, 2).
heuristic(e, 1).
heuristic(f, 0). % Goal node

% Best First Search algorithm
best_first_search(Start, Goal, Path) :-
    heuristic(Start, H),
    bfs([[Start, H]], Goal, [], % Define edges of the graph with their costs
edge(a, b, 1).
edge(a, c, 3).
edge(b, d, 3).
edge(b, e, 6).
```

```

edge(c, e, 2).
edge(d, f, 1).
edge(e, f, 2).

% Define the heuristic values (estimated cost to reach the goal)
heuristic(a, 6).
heuristic(b, 4).
heuristic(c, 5).
heuristic(d, 2).
heuristic(e, 1).
heuristic(f, 0). % Goal node

% Best First Search algorithm
best_first_search(Start, Goal, Path) :-
    heuristic(Start, H),
    bfs([[Start, H]], Goal, [], Path).


% Helper predicate to implement BFS
bfs([[[Goal|Path]|_], Goal, _, [Goal|Path]]).
bfs([[[Current|Path]|Rest], Goal, Visited, FinalPath) :-
    findall([Next, H, Current|Path],
        (edge(Current, Next, _),
         \+ member(Next, Visited),
         heuristic(Next, H)),
        Neighbors),
    append(Rest, Neighbors, NewFrontier),
    sort(2, @=<, NewFrontier, SortedFrontier),
    bfs(SortedFrontier, Goal, [Current|Visited], FinalPath).

% Query to find the path
find_path(Start, Goal, Path) :-
    best_first_search(Start, Goal, RevPath),
    reverse(RevPath, Path).

```

Input:


python

 Copy code

```
graph = {'A': ['B', 'C'], 'B': ['D', 'E'], 'C': ['F'], 'D': [], 'E': ['F'], 'F': []}  
start_node = 'A'  
goal_node = 'F'
```

Output:

plaintext

 Copy code

```
Path found: A -> C -> F
```

Result:

The prolog program to solve BFS problem was executed.

28 Write the prolog program for Medical Diagnosis

AIM:

To write the prolog program for Medical Diagnosis.

ALGORITHM:

1. Define symptoms and corresponding diseases.
2. Add rules to infer diseases from symptoms.
3. Query symptoms to get possible diseases.
4. Suggest probable diagnoses based on inputs.

Program:

```
% Define symptoms
symptom(john, fever).
symptom(john, cough).
symptom(john, headache).
symptom(mary, sore_throat).
symptom(mary, cough).
symptom(mary, fatigue).
symptom(tom, rash).
symptom(tom, fever).
symptom(tom, headache).

% Define diseases and their associated symptoms
disease(flu, [fever, cough, headache, fatigue]).
disease(cold, [cough, sore_throat, fatigue]).
disease(measles, [rash, fever, headache]).

% Rule to diagnose a disease based on symptoms
diagnose(Patient, Disease) :-
    symptom(Patient, Symptom1),
    symptom(Patient, Symptom2),
    symptom(Patient, Symptom3),
    disease(Disease, Symptoms),
    member(Symptom1, Symptoms),
    member(Symptom2, Symptoms),
    member(Symptom3, Symptoms).
```

% Query example

% ?- diagnose(john, Disease).

Example Input and Output:

```
prolog
```

```
?- diagnose(fever, Disease).
```

```
Disease = flu.
```

Result:

The prolog program to medical diagnosis solve problem was executed.

29 Write a Prolog Program for forward Chaining. Incorporate required queries.

AIM:

To write the prolog program for forward Chaining. Incorporate required queries.

ALGORITHM:

1. Define facts and rules for forward chaining.
2. Process known facts to infer new facts.
3. Continue until no new facts can be derived.
4. Query based on forward-chained conclusions

Program:

rainy(chennai).

rainy(coimbatore).

rainy(ooty).

cold(ooty).

snowy(X):-rainy(X),cold(X).

Example Query and Output:

```
prolog

?- snowy(X).
X = ooty.
```

Result:

The prolog program to solve Forward chaining problem was executed.

30 Write a Prolog Program for backward Chaining. Incorporate required queries

AIM:

To write the prolog program for backward Chaining. Incorporate required queries.

ALGORITHM:

1. Define rules and target goal.
2. Check if goal can be satisfied by facts.
3. Trace back to determine if goal is true.
4. Return result based on backtracking.

Program:

```
% Facts
fact(sunny).
fact(weekend).
fact(raining).
fact(weekday).

% Rules
rule(go_beach) :-
    fact(sunny),
    fact(weekend).
rule(watch_movie) :-
    fact(raining).
rule(stay_home) :-
    fact(sunny),
    fact(weekday).

% To deduce a fact
deduce(Fact) :-
    fact(Fact).
deduce(Fact) :-
    rule(Fact),
    \+ fact(Fact),
    assertz(fact(Fact)),
    write('Derived: '), write(Fact), nl.
```

Output:

```
makefile
```

```
F = sunny ;  
F = weekend ;  
F = raining ;  
F = weekday ;  
F = go_beach ;  
F = watch_movie ;  
F = stay_home.
```

Result:

The prolog program to solve backward chaining problem was executed.

31 Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc

AIM:

To Create a Web Blog using Word press to demonstrate Anchor Tag, Title Tag, etc

ALGORITHM:

1. Define page structure with anchor tags.
2. Add titles, headers, and tags for SEO.
3. Style with CSS for a responsive layout.
4. Publish the blog page.

Program:

```
<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>My Blog Post</title>

</head>

<body>

<h1>Welcome to My Blog</h1>

<p>This is an example of a blog post demonstrating various HTML elements.</p>

<h2>Using Anchor Tags</h2>

<p><a href="https://example.com">Click here to visit Example</a></p>

<h2>Using Bold and Italics</h2>

<p>This is <b>bold text</b> and this is <i>italic text</i>.</p>

<h2>Using Lists</h2>

<ul>

<li>First item in unordered list</li>

<li>Second item in unordered list</li>

</ul>

<ol>

<li>First item in ordered list</li>

<li>Second item in ordered list</li>

</ol>
```

<h2>Using Images</h2>

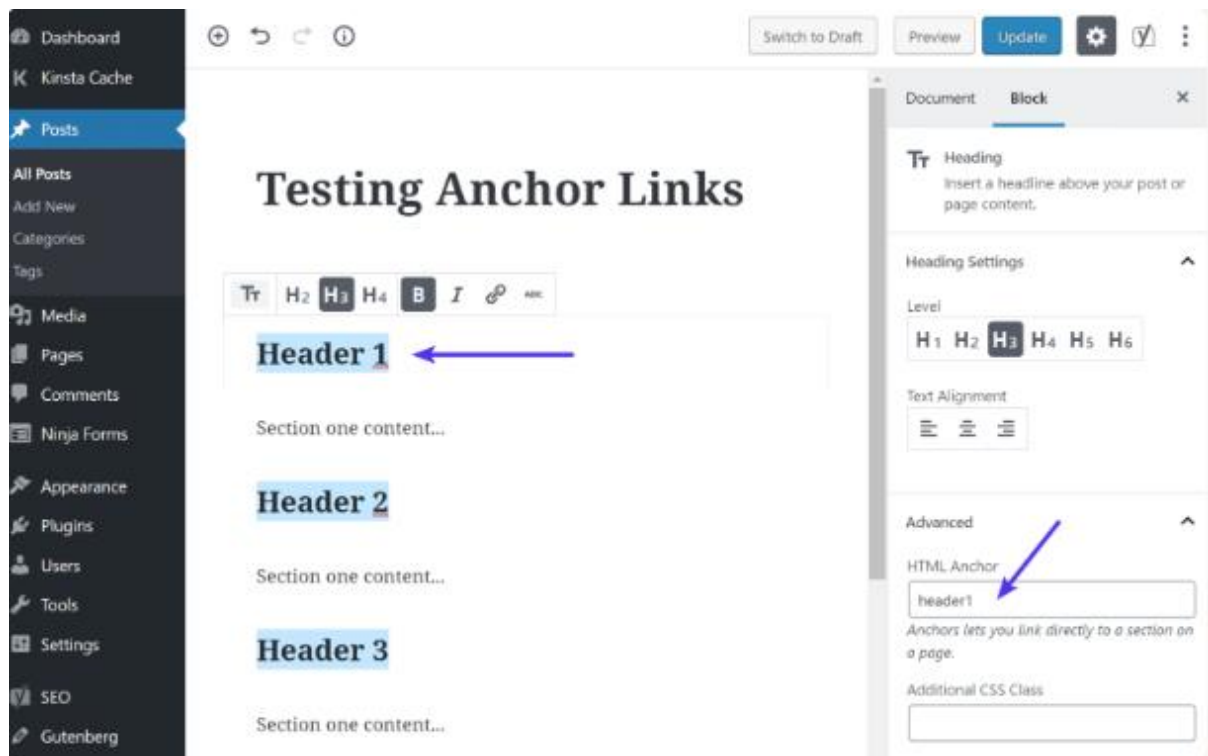
<p></p>

<p>Thank you for reading!</p>

</body>

</html>

Sample Output:



Result:

The web blog program was executed.