VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Umesha H N(1BM24CS428)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug 2025 to Dec 2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Umesha H N(1BM24CS428),** who is bonafide student of **B.M.S College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Umeshahnn/1BM24CS428_AI

INDEX

Name: Umesha H N Std: Vth Sec:

Roll No.: Subject: AI School/College:

School/College Tel. No.: Parents Tel. No.:

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1	18/08/25	Implement Tic-Tac-Toe		10
2		vaccum cleaner.		21/8
3		a. BFS (Breath First search without Huristic approach		10
		b. implement BFS with Huristic approach		
		c. iterative Depending DF		10
4		Hill-climbing		10
		Simulated - Au nealig		
5		A* manhattan, misplaced		10
6	22/9/25	Propositional Logic.		22/9/25
7	13/10/25	unification		10
8		First order logic: Forward chaining		
9		Resolution in FOL		10
10		The Alpha-beta Algo		27/10

Completed

27/

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:

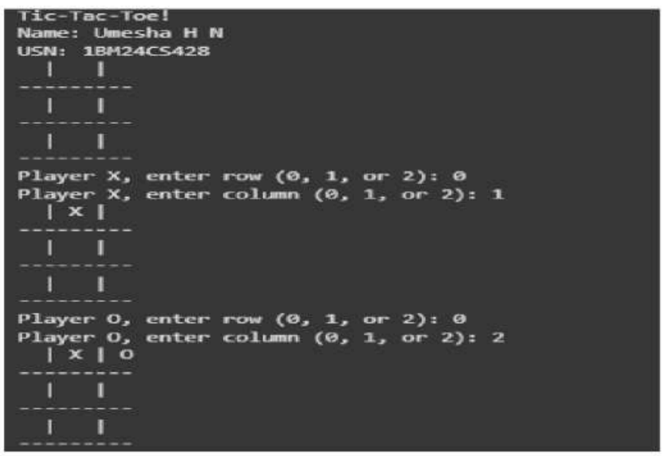
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

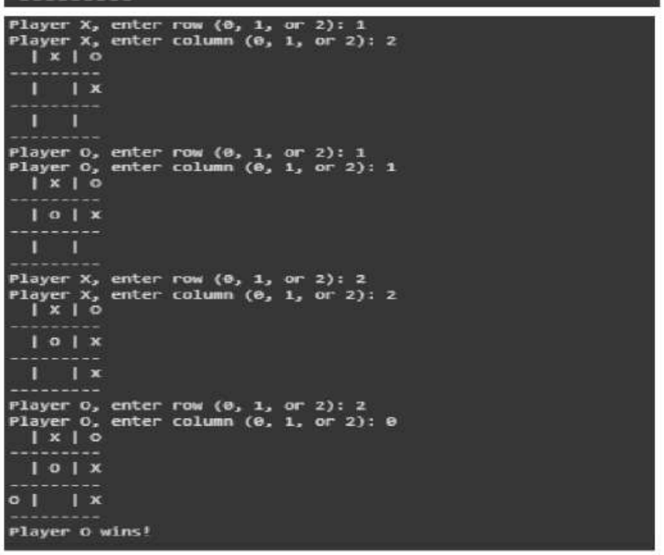
Code:

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_win(board, player):
    for row in board:
        if all([cell == player for cell in row]):
            return True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True
    if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player for i in range(3)]):
        return True
    return False

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

    print("Tic-Tac-Toe!")
    print("Name: Umesha H N")
    print("USN: 1BM24CS428")
    print_board(board)

    while True:
        player = players[turn % 2]
        row = int(input(f"Player {player}, enter row (0, 1, or 2): "))
        col = int(input(f"Player {player}, enter column (0, 1, or 2): "))

        if board[row][col] == " ":
            board[row][col] = player
            print_board(board)

            if check_win(board, player):
                print(f"Player {player} wins!")
                break
```

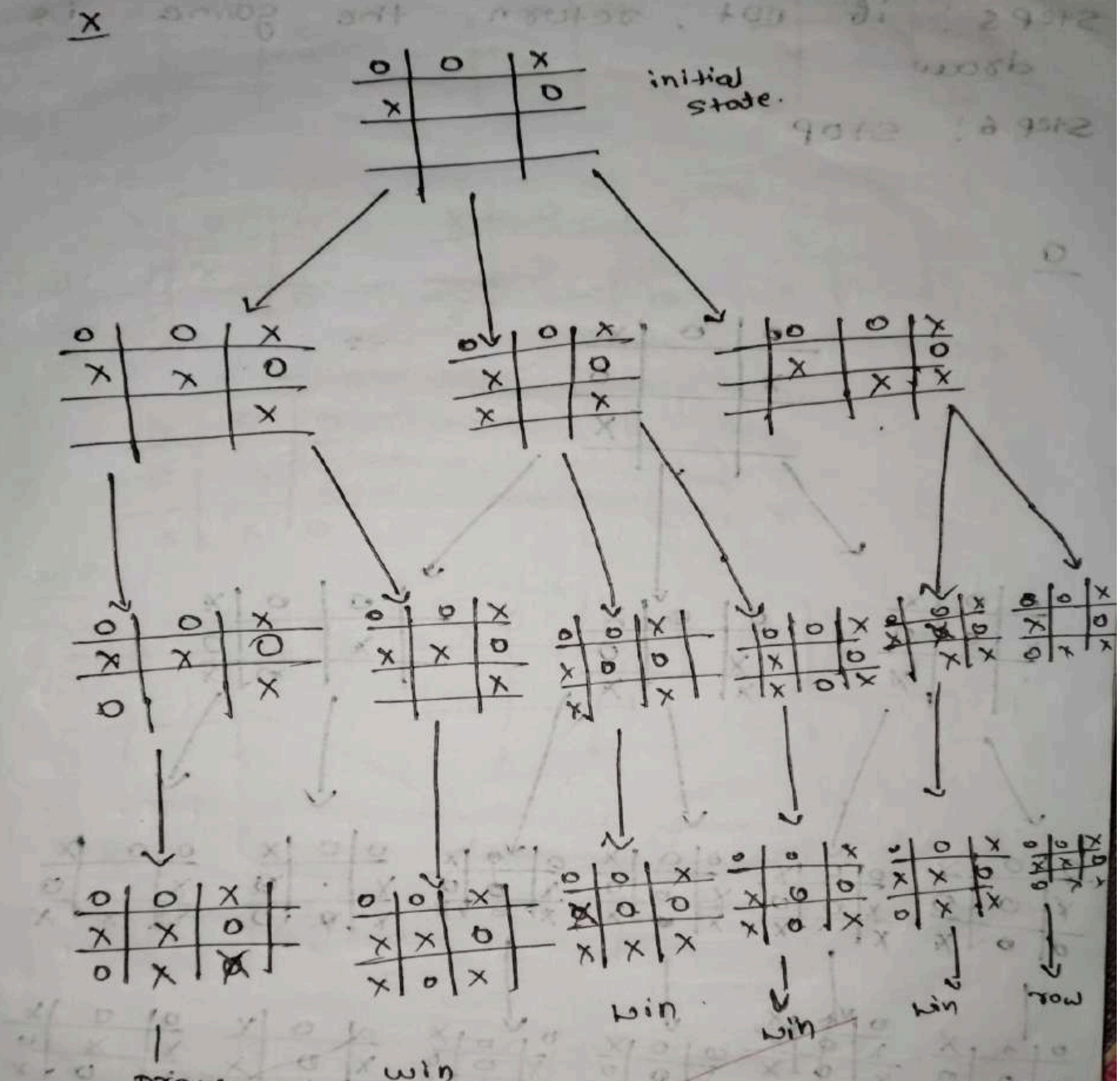
```
            elif all([cell != " " for row in board for cell in row]):
                print("It's a tie!")
                break
            else:
                turn += 1
        else:
            print("That spot is already taken! Try again.")

tic_tac_toe()
```

Output





① Implement Tic-Tac-Toe Game.



cost = 4

Algorithm

Step 1 :- Start.

Step 2 : create a matrix (n = 3×3).

Step 3: take the input from users 1 & 2 (1 → X 2 → O).

Step 4: if the three X or three O

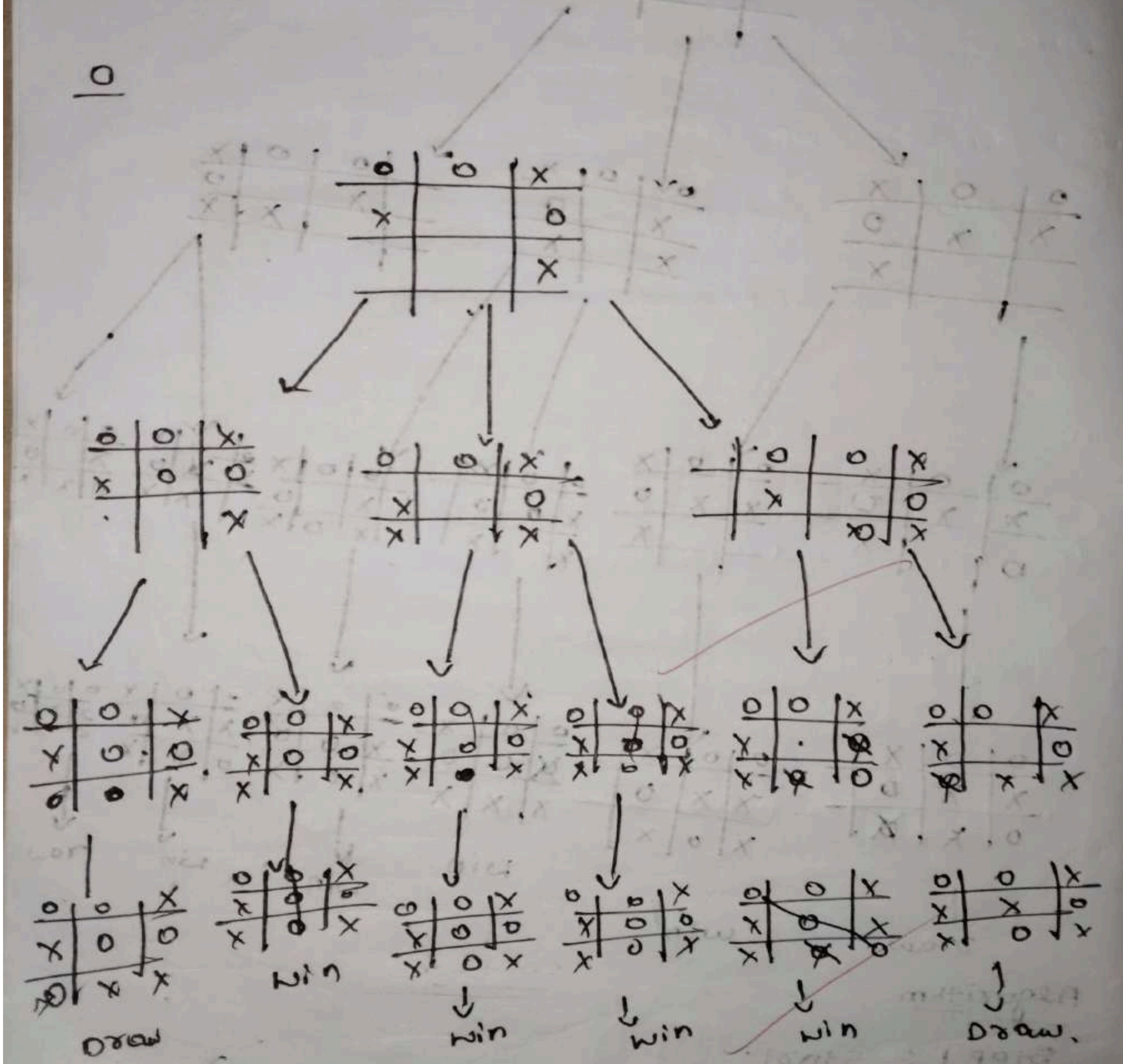
and Horizontally, print there is win

~~Step 1~~

Step 5 :- if not, return the game is draw

Step 6 : Stop

O



Implement vacuum cleaner agent

Code:
```
import random
import time

# Grid dimensions
ROWS, COLS = 5, 5

# Create a room (1 = dirty, 0 = clean)
room = [[random.choice([0, 1]) for _ in range(COLS)] for _ in range(ROWS)]

vacuum_x, vacuum_y = 0, 0

def display_room():
    for i in range(ROWS):
        for j in range(COLS):
            if i == vacuum_x and j == vacuum_y:
                print("", end=" ")
            elif room[i][j] == 1:
                print("", end=" ")  # dirt
            else:
                print("", end=" ")
        print()
    print()

def clean_cell(x, y):
    if room[x][y] == 1:
        print(f"Cleaning cell ({x}, {y})...")
        room[x][y] = 0
        time.sleep(0.5)

def move_vacuum():
    global vacuum_x, vacuum_y
    directions = [(0,1), (1,0), (0,-1), (-1,0)]
    dx, dy = random.choice(directions)
    nx, ny = vacuum_x + dx, vacuum_y + dy
    if 0 <= nx < ROWS and 0 <= ny < COLS:
        vacuum_x, vacuum_y = nx, ny

# Run simulation
print("Starting smart vacuum cleaner...\n")
for _ in range(50):
    display_room()
    clean_cell(vacuum_x, vacuum_y)
    move_vacuum()
    time.sleep(0.3)

print("All clean or time’s up! ")
display_room()
```

Output

```
Name:Umesha H N
USN:1BM24CS428

Current Room: Room A
Room Status: {'Room A': 0, 'Room B': 0}
Total Cost So Far: 0
Enter action (clean/move/clean and move): move

Moving to Room B.

Current Room: Room B
Room Status: {'Room A': 0, 'Room B': 0}
Total Cost So Far: 1
Enter action (clean/move/clean and move): clean

Cleaning Room B...
Room B is now clean.

Current Room: Room B
Room Status: {'Room A': 0, 'Room B': 1}
Total Cost So Far: 2
Enter action (clean/move/clean and move): clean and move

Room B is already clean.

Moving to Room A.

Current Room: Room A
Room Status: {'Room A': 0, 'Room B': 1}
Total Cost So Far: 3
Enter action (clean/move/clean and move): clean

Cleaning Room A...
Room A is now clean.

All rooms are clean! Simulation finished.
Total Cost of Cleaning: 4
```

2. vaccum cleaner. agent:

~~Algorithm~~

~~Step 1: Start~~

~~Step 2: ~~state~~ initialize the two rooms (in matrices)~~

~~Step 3: Assign the vaccum in A Room~~

~~Step 4: if there is any dirty in the room, suck it out~~

~~1 yes~~

~~2 NO~~

~~3 clean & 3 move to next room~~

~~Step 5:~~

Algorithm

Step 1 : Start

Step 2 : ~~assig.~~ initialize the two roo in array s (metrices)

Step 3 : if vaccum is in the a roo, clean it

Step 4 : if there is no dust in the A room

1 true

2 False

3 move to next room.

Step 5 : if both rooms are clean

1 stop

2 move to A room.

Step 6 : stop

output

* Curred room : Room A

Enter Action ((clean | move | clean and move) : clean

cleaning Room A....

Room A is now clear

* Curret Room : Room A

Enter Action : move

moving to Room B

* Curret Room : Room B

Enter action : clean and move

cleaning Room B

~~moving to Room~~ A

Room B is now clean

All Rooms are clean!

Total cost of cleaning : 3

25/8/25

<u>Program 2</u>

Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm

Algorithm:

```
MOVES = {
    'UP': (-1, 0),
    'DOWN': (1, 0),
    'LEFT': (0, -1),
    'RIGHT': (0, 1)
}

GOAL_STATE = ((1, 2, 3),
              (4, 5, 6),
              (7, 8, 0))

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_valid_pos(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    x1, y1 = pos1
    x2, y2 = pos2
    state_list[x1][y1], state_list[x2][y2] = state_list[x2][y2], state_list[x1][y1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    x, y = find_zero(state)
    for move in MOVES.values():
        new_x, new_y = x + move[0], y + move[1]
        if is_valid_pos(new_x, new_y):
            neighbors.append(swap_positions(state, (x, y), (new_x, new_y)))
    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
```

```
    return path

def dfs(start_state, depth_limit=50):
    stack = [(start_state, 0)]
    visited = set([start_state])
    came_from = {}

    while stack:
        current, depth = stack.pop()
        if current == GOAL_STATE:
            return reconstruct_path(came_from, current)
        if depth < depth_limit:
            for neighbor in get_neighbors(current):
                if neighbor not in visited:
                    visited.add(neighbor)
                    came_from[neighbor] = current
                    stack.append((neighbor, depth + 1))
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()

if __name__ == "__main__":
    start_state = ((1, 2, 3),
                   (4, 0, 6),
                   (7, 5, 8))

    print("Initial State:")
    print_state(start_state)

    print("Solving with DFS...")
    dfs_path = dfs(start_state, depth_limit=30)
    if dfs_path:
        print(f"Solution found in {len(dfs_path) - 1} moves!")
        for state in dfs_path:
            print_state(state)
    else:
        print("No solution found with DFS within depth limit.")
```

Output

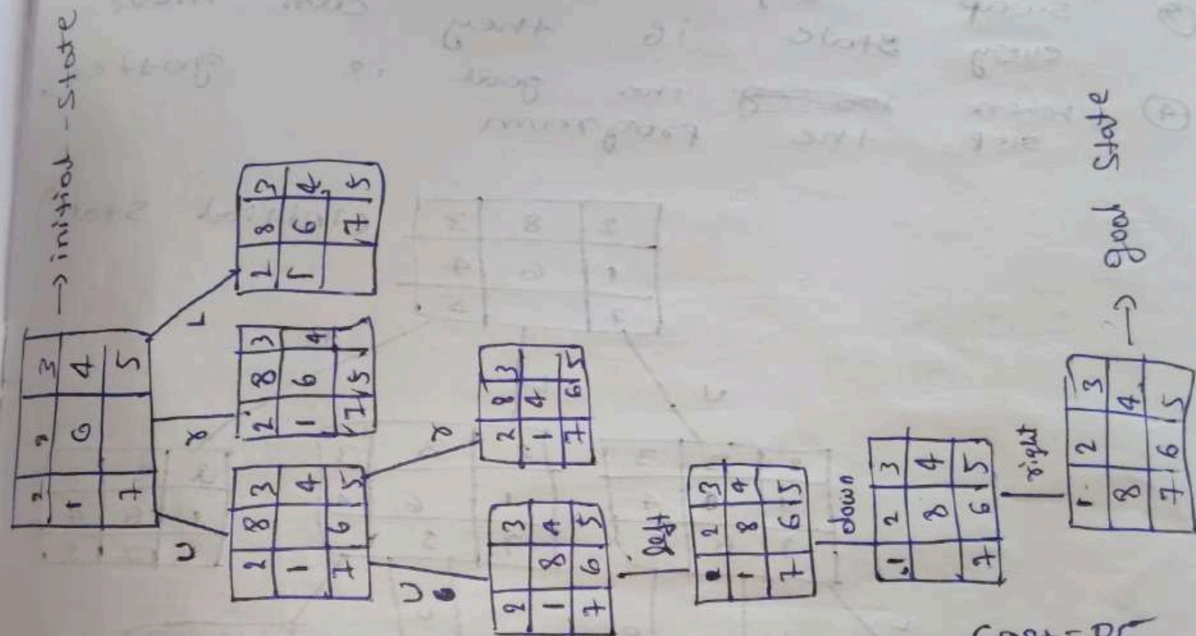
```
Initial State:
2 8 3
1 6 4
7 _ 5

Name:Umesha H N
USN:1BM24CS428

Solving with DFS...
Solution found in 15 moves!
2 8 3
1 6 4
7 _ 5

2 8 3
1 6 4
7 5 _

2 8 3
1 6 _
7 5 4

2 8 3
1 _ 6
7 5 4

2 8 3
_ 1 6
7 5 4

_ 8 3
2 1 6
7 5 4

8 _ 3
2 1 6
7 5 4

8 1 3
2 _ 6
7 5 4

8 1 3
2 6 _
7 5 4

8 1 3
2 6 4
7 5 _
```

```
2 8 3
_ 1 6
7 5 4

_ 8 3
2 1 6
7 5 4

8 _ 3
2 1 6
7 5 4

8 1 3
2 _ 6
7 5 4

8 1 3
2 6 _
7 5 4

8 1 3
2 6 4
7 5 _

8 1 3
2 6 4
7 _ 5

8 1 3
2 _ 4
7 6 5

8 1 3
_ 2 4
7 6 5

_ 1 3
8 2 4
7 6 5

1 _ 3
8 2 4
7 6 5

1 2 3
8 _ 4
7 6 5
```

Week-3

Using BFS solve 8 Puzzle without h-uristic.

DFS



Cost=05

Algorithm

① Start with the initial matrix.

② next start the operations for searching the goal matrix.

③ Swap right, left, down, up.

④ Swap until the goal is gotten.

⑤ when the goal is gotten

⑥ Stop the program.

Traverse level by level to left most side.

DFS

solving with DFS

solution found in 15 moves

2 8 3

1 0 4

7 0 5

2 8 3

1 6 4

7 5 0

=

1 2 3

8 0 4

7 6 5

Iterative Deepening Search

Code:

```
from collections import deque

GOAL_STATE = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

MOVES = {
    'UP': (-1, 0),
    'DOWN': (1, 0),
    'LEFT': (0, -1),
    'RIGHT': (0, 1)
}

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_valid_pos(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    x1, y1 = pos1
    x2, y2 = pos2
    state_list[x1][y1], state_list[x2][y2] = state_list[x2][y2], state_list[x1][y1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    x, y = find_zero(state)
    for move in MOVES.values():
        new_x, new_y = x + move[0], y + move[1]
        if is_valid_pos(new_x, new_y):
            neighbors.append(swap_positions(state, (x, y), (new_x, new_y)))
    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

```
def dls(state, depth_limit, came_from, visited):
    """Depth Limited Search"""
    if state == GOAL_STATE:
        return True
    if depth_limit <= 0:
        return False

    for neighbor in get_neighbors(state):
        if neighbor not in visited:
            visited.add(neighbor)
            came_from[neighbor] = state
            if dls(neighbor, depth_limit - 1, came_from, visited):
                return True
    return False

def iddfs(start_state, max_depth=50):
    """Iterative Deepening DFS"""
    for depth in range(max_depth):
        came_from = {}
        visited = {start_state}
        if dls(start_state, depth, came_from, visited):
            return reconstruct_path(came_from, GOAL_STATE)
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()

if __name__ == "__main__":
    start_state = ((2, 8, 3),
                   (1, 6, 4),
                   (7, 0, 5))

    print("Initial State:")
    print_state(start_state)
    print("Name: Umesha H N\nUSN: 1BM24CS428\n")

    print("Solving with Iterative Deepening DFS...")
    iddfs_path = iddfs(start_state)
    if iddfs_path:
        print(f"Solution found in {len(iddfs_path) - 1} moves!")
        for state in iddfs_path:
            print_state(state)
    else:
        print("No solution found with IDDFS.")
```

Output

```
Step-by-step solution:

Name:Umesha H N/nUSN:1BM24CS428
Step 0: Moves:
2 8 3
1 6 4
7   5

Step 1: Moves: U
2 8 3
1   4
7 6 5

Step 2: Moves: UU
2   3
1 8 4
7 6 5

Step 3: Moves: UUL
  2 3
1 8 4
7 6 5

Step 4: Moves: UULD
1 2 3
  8 4
7 6 5

Step 5: Moves: UULDR
1 2 3
8   4
7 6 5
```

Iterative deepning : DFS :

Algorithm :

S1 :- Start

S2 :- set depth unit d=0

S3 :- Perform a depth limited DFS with currect depth limited.

S4 : Explore the search tree to depth only if a goal is formed return to path.

S5 : if no solution is formed at depth d,
increment by 1

S6 : Repeat step 3 until solution is formed.

output

1 2 3 → 1 2 3
4 0 5   4 5 0
6 7 8   6 7 8

1 2 3   1 2 0   1 2 3
4 5 8   4 5 3   4 7 5
6 7 0   6 7 8   6 8 0

At depth = 2 solution is found

Program 3

Implement A* search algorithm

Manhattan

```
import heapq
import time

class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def heuristic(self):

        distance = 0
        for i, tile in enumerate(self.board):
            if tile != 0:
                goal_pos = self.goal.index(tile)
                distance += abs(i // self.size - goal_pos // self.size) + abs(i % self.size - goal_pos %
self.size)
        return distance

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y - 1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)

                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board), self.goal, self.path + move, self.cost + 1))
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
```

```
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map, current_state.board)

        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

if __name__ == "__main__":
    initial_state = (1, 5, 8,
                     3, 2, 0,
                     4, 6, 7)

    final_state = (1, 2, 3,
```

```
                4, 5, 6,
                7, 8, 0)

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)
    else:
        print("Name:Umesha H N ")
        print("USN:1BM24CS428")
        print("No solution found.")
```

Output

Misplaced Tiles

Code

```
import heapq
import time

class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
```

```
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def heuristic(self):

        misplaced = 0
        for i, tile in enumerate(self.board):
            if tile != 0 and tile != self.goal[i]:
                misplaced += 1
        return misplaced

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y - 1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)
                # Swap blank with the adjacent tile
                new_board[self.zero_pos], new_board[new_zero_pos] =
new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board), self.goal, self.path + move,
self.cost + 1))
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map, current_state.board)
```

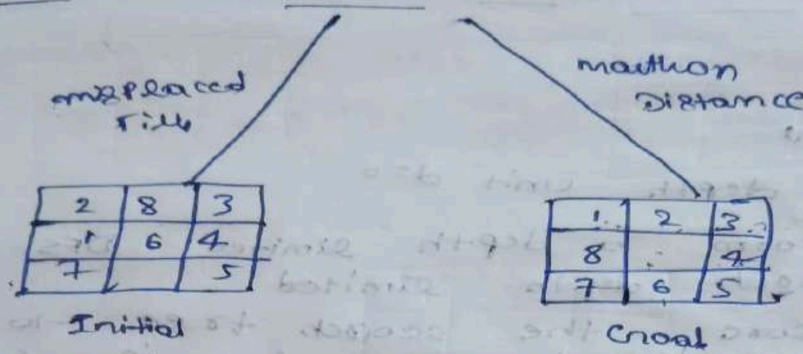
```
        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

if __name__ == "__main__":
    initial_state = (2, 8, 3,
                     1, 6, 4,
                     7, 0, 5)

    final_state = (1, 2, 3,
                   8, 0, 4,
                   7,6,5)

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        print("Name:Umesha H N/nUSN:1BM24CS428")
        for step_num, board in enumerate(solution_boards):
```

```
            moves_so_far = "".join(solution_moves[:step_num])
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)
    else:
        print("No solution found.")
```

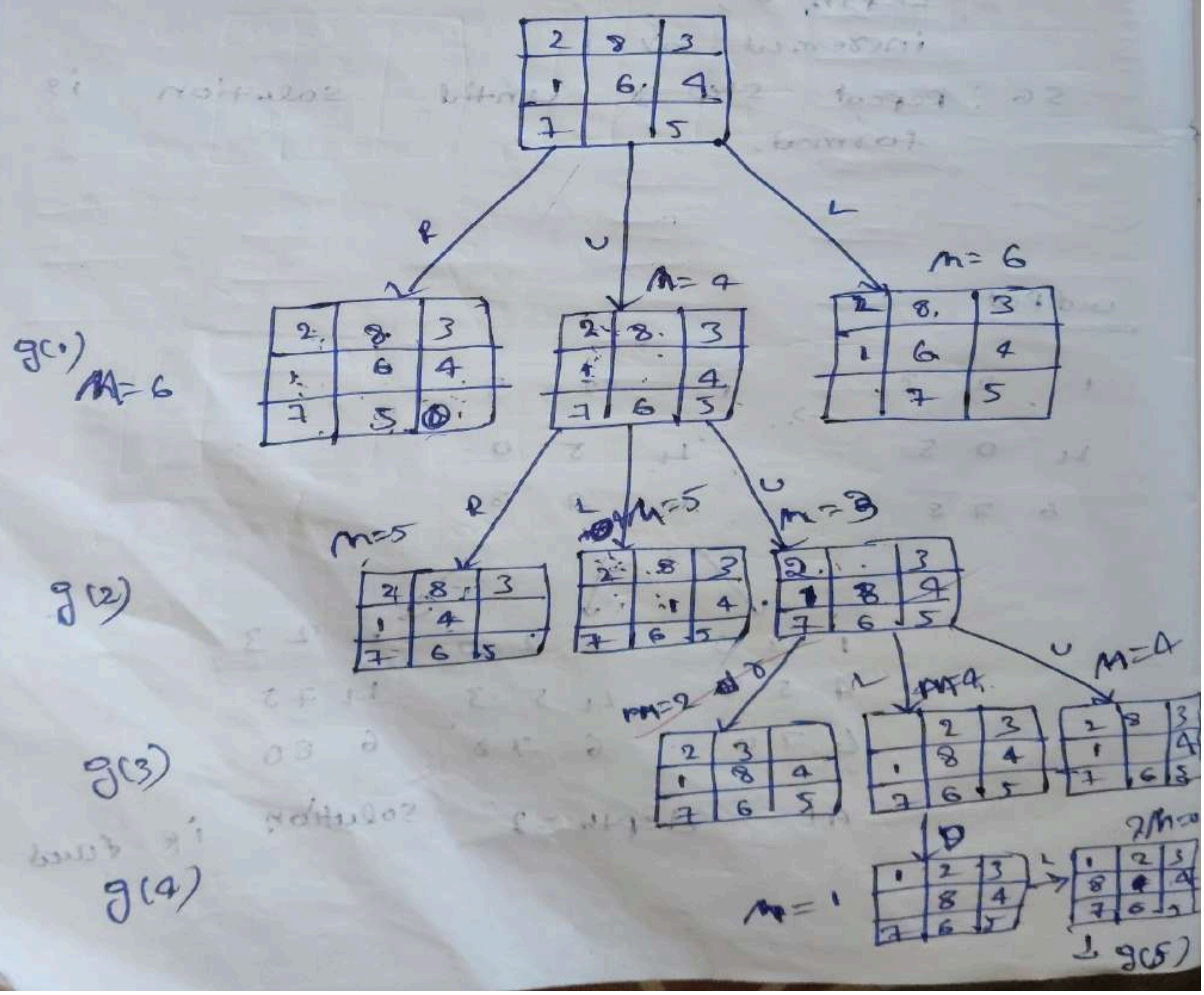
Output

```
Step-by-step solution:

Name:Umesha H N/nUSN:1BM24CS428
Step 0: Moves:
2 8 3
1 6 4
7   5

Step 1: Moves: U
2 8 3
1   4
7 6 5

Step 2: Moves: UU
2   3
1 8 4
7 6 5

Step 3: Moves: UUL
  2 3
1 8 4
7 6 5

Step 4: Moves: UULD
1 2 3
  8 4
7 6 5

Step 5: Moves: UULDR
1 2 3
8   4
7 6 5
```

Apply A* Algorithm



$$f(n) = g(n) + h(n)$$



Algorithm for manhattan Distance

Start : Initialize start node, with, g = 0, calculate h = sum of manhattan distances of tiles goal., f = g+h. put it in open list

select : pick node with lowest f from open list if goal done

Expand :- generate child states for each child. g = parent eg+1)

h = sum of manhattan distance

Algorithm for misplaced File

output

step - by - step solution

Step 0:

2 8 3
1 6 4
7 5

Step 1:

2 8 3
1 4
7 6 5

Step 2:

2 3
1 8 4
7 6 5

Step 3:

2 3
1 8 4
7 6 5

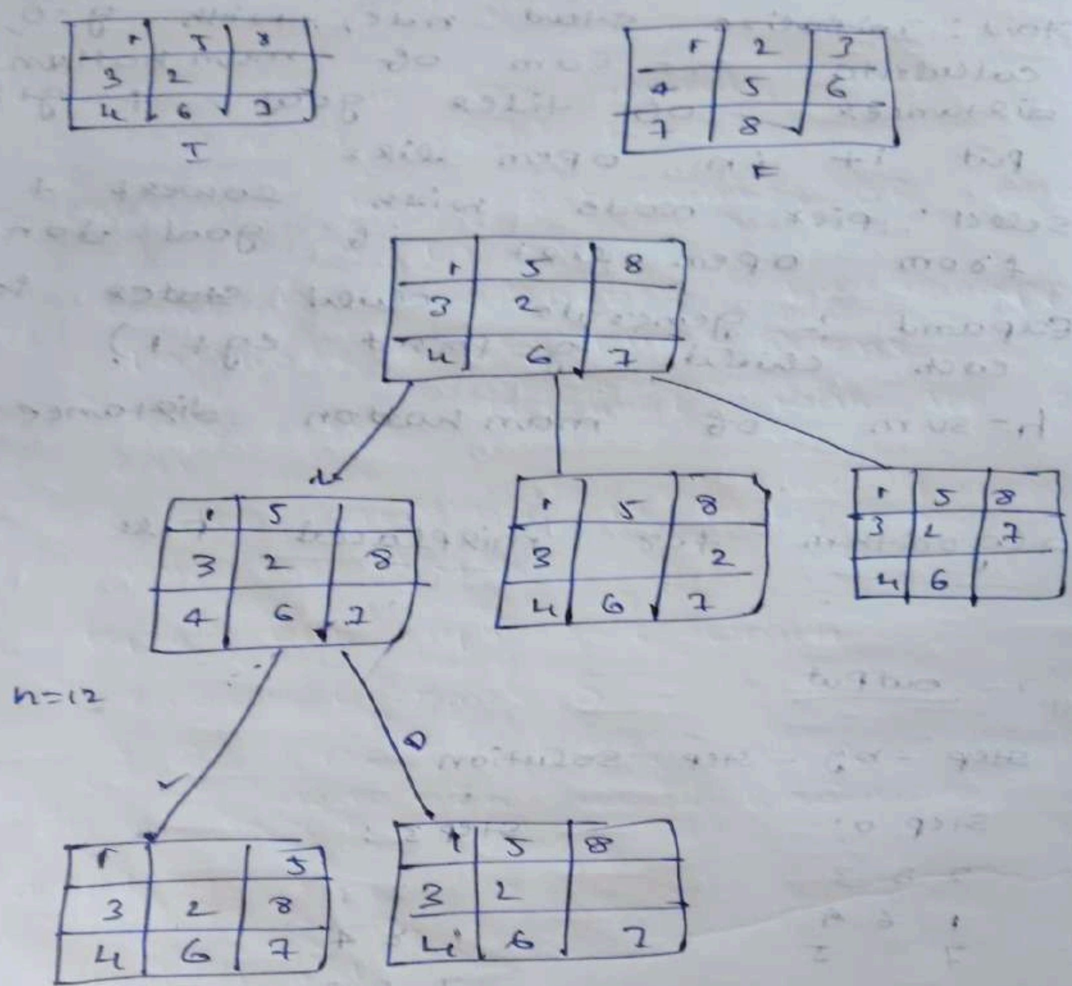
Step 4:

1 2 3
8 4
7 6 5

Step 5:

1 2 3
8 4
7 6 5

misplaced tiles



Algorithm for misplaced tiles

Initialize

start: Initialize Start node with g=0
calculate h= no of tiles out of place
f = g+h put it in open list

(2) Select: Take node with lowest from open list. if goal, stop

(3) expand: generator children from possible moves. For each child

calculate g = parent, g+1
h = misplaced tiles cond, f = g+h, Add to open list better.

④ Repeat : loop until goal found or no nodes left.

Output

~~No~~ Solution ~~found.~~

Step-by-Step solution

Step 0 : Moves:

2 8 3
1 6 4
7   5

Step 1 : Moves: U

2 8 3
1   4
7 6 5

Step 2 : Moves: UU

2 0 3
1 8 4
7 6 5

Step 3 : Moves : UUL

0 2 3
1 8 4
7 6 5

Step 4 : Moves : UULD

1 2 3
  8 4

Step 5 Moves : UULD

1 2 3
8   4
7 6 5

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Code:

```
def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(line))
    print()

def hill_climbing_step_by_step(board):
    n = len(board)
    current_state = board[:]
    current_conflicts = calculate_conflicts(current_state)

    step = 0
    print("Name:Umesha H N\nUSN:1BM24CS428\n")
    print(f"Initial board with conflicts = {current_conflicts}:")
    print_board(current_state)

    while current_conflicts > 0:
        step += 1
        print(f"Step {step}:")
        best_state = current_state[:]
        best_conflicts = current_conflicts

        for row in range(n):
            original_col = current_state[row]
            for col in range(n):
                if col != original_col:
                    current_state[row] = col
                    conflicts = calculate_conflicts(current_state)
```

```
            if conflicts < best_conflicts:
                best_conflicts = conflicts
                best_state = current_state[:]

            current_state[row] = original_col

        if best_conflicts == current_conflicts:
            print("No better neighbor found, stuck at local optimum.")
            break

        current_state = best_state
        current_conflicts = best_conflicts

        print(f"Board with conflicts = {current_conflicts}:")
        print_board(current_state)

    if current_conflicts == 0:
        print("Solution found!")
    else:
        print("No solution found.")
    return current_state

initial_board = [3, 0, 1, 2]
solution = hill_climbing_step_by_step(initial_board)
```

Output

```
Name:Umesha H N
USN:1BM24CS428

Initial board with conflicts = 4:
. . . Q
Q . . .
. Q . .
. . Q .

Step 1:
Board with conflicts = 2:
. . . Q
Q . . .
Q . . .
. . Q .

Step 2:
Board with conflicts = 1:
. . . Q
. Q . .
Q . . .
. . Q .

Step 3:
No better neighbor found, stuck at local optimum.
No solution found.
```

Hill - climbing Algorithm

Steps

① Start with an initial State
(e.g [3, 1, 2, 0]).

② calculate the cost (number of attackin -g Queen pairs)

③ Generate neighbors by swapping Positions of two Queens

④ choose the neighbor with the lowest cost

⑤ if the neighbor is better, move to it otherwise, Stop

⑥ Repeat steps 3 - 6 until
* cost becomes 0 → goal state found
* or no better neigbor exists
→ stuck in local maximum.

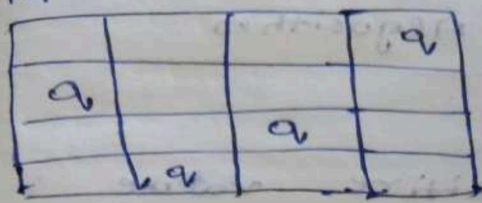
output

Initial Board!

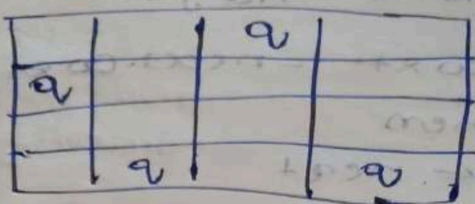
			Q
	q		
		q	
q			

cost = 2

step1 = cost = 1

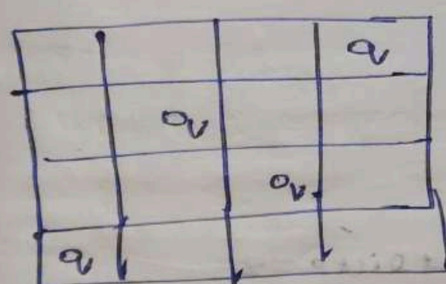


step2 : cost = 0

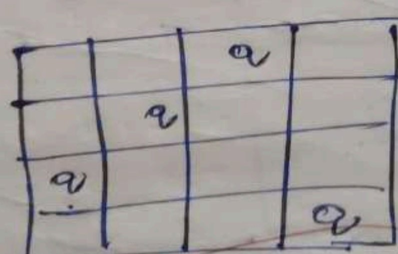


Goal reached at step 2!

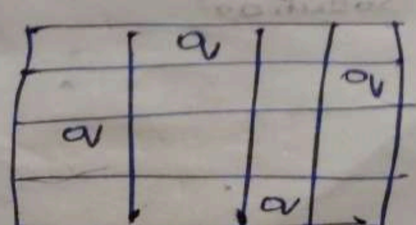
~~Step~~ (



cost = 2



cost = 1



Program 5

Simulated Annealing to Solve 8-Queens problem

Code:

```
import random
import math

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(line))
    print()

def simulated_annealing(n=8, max_iter=10000, initial_temp=100, cooling_rate=0.95):
    current_state = [random.randint(0, n - 1) for _ in range(n)]
    current_conflicts = calculate_conflicts(current_state)
    temperature = initial_temp
    iteration = 0

    while current_conflicts > 0 and iteration < max_iter and temperature > 0.1:
        iteration += 1
        neighbor = current_state[:]
        row = random.randint(0, n - 1)
        new_col = random.randint(0, n - 1)
        while new_col == neighbor[row]:
            new_col = random.randint(0, n - 1)
        neighbor[row] = new_col

        neighbor_conflicts = calculate_conflicts(neighbor)
        delta = neighbor_conflicts - current_conflicts

        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / temperature):
            current_state = neighbor
            current_conflicts = neighbor_conflicts
```

```
        temperature *= cooling_rate

    return current_state, current_conflicts

solution, conflicts = simulated_annealing(n=8)
print("Final board with conflicts =", conflicts)
print("Name:Umesha H N\nUSN:1BM24CS428\n")
print_board(solution)

if conflicts == 0:
    print("Solution found!")
else:
    print("Failed to find a solution.")
```

Output

```
Final board with conflicts = 2
Name:Umesha H N
USN:1BM24CS428

Q . . . . . . .
. . . . . Q . .
. Q . . . . . .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . Q . . . .
. . . . . . Q .

Failed to find a solution.
```

Simulated Anncaling Algorithm

1. current ← initial State
2. T ← a large positive value
3. While T > 0 do

next ← a random neighbour of current

ΔE ← current. cost - next.cost

if ΔE > 0 then

current ← next

else

current ← next with probability $P = e^{\frac{\Delta E}{T}}$

end if

decrease T

end while

return current

output

Final board with conflicts = 2.

```
Q . . . .
. . . Q .
. Q . . .
. . Q . .
Q . . . Q
. . Q . .
. . . . Q .
```

Failed to find a Solution

Program 6

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not**.**

Code:

```
import itertools
import pandas as pd
import re

def replace_implications(expr):
    """
    Replace every X => Y with (not X or Y).
    This uses regex with a callback to avoid partial string overwrites.
    """
    # Pattern: capture left side and right side around =>
    # Made more flexible to handle various expressions
    pattern = r'([^=><]+?)\s*=>\s*([^=><]+?)(?=\s|$|[&|)])'
    while re.search(pattern, expr):
        expr = re.sub(pattern,
                      lambda m: f"(not {m.group(1).strip()} or {m.group(2).strip()})",
                      expr,
                      count=1)
    return expr


def get_symbols(KB, alpha):
    symbols = set()
    for sentence in KB + [alpha]:
        # Find all alphabetic tokens (propositional variables)
        for token in re.findall(r'\b[A-Za-z]+\b', sentence):
            if token not in ['and', 'or', 'not']:  # Exclude boolean operators
                symbols.add(token)
    return sorted(list(symbols))

def tt_entails(KB, alpha):
    symbols = get_symbols(KB, alpha)
    rows = []
    entails = True

    for values in itertools.product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        try:
            kb_val = all(pl_true(sentence, model) for sentence in KB)
            alpha_val = pl_true(alpha, model)
```

```
            rows.append({**model, "KB": kb_val, "alpha": alpha_val})

            if kb_val and not alpha_val:
                entails = False
        except Exception as e:
            print(f"Error evaluating with model {model}: {e}")
            return False

    df = pd.DataFrame(rows)

    # Create a beautiful formatted table
    print("\n" + "="*50)
    print("                TRUTH TABLE")
    print("="*50)

    # Get column widths for proper alignment
    col_widths = {}
    for col in df.columns:
        col_widths[col] = max(len(str(col)), df[col].astype(str).str.len().max())

    # Calculate total table width
    table_width = sum(col_widths.values()) + len(df.columns) * 3 - 1

    # Print top border
    print("┌" + "─" * table_width + "┐")

    # Print header
    header = "│"
    for col in df.columns:
        header += f" {col:^{col_widths[col]}} │"
   print(header)

# Print separator
    separator = "├"
    for col in df.columns:
        separator += "─" * (col_widths[col] + 2) + "┼"
    separator = separator[:-1] + "┤"
    print(separator)

    # Print rows
    for _, row in df.iterrows():
        row_str = "│"
        for col in df.columns:
            value = str(row[col])
            row_str += f" {value:^{col_widths[col]}} │"
        print(row_str)
```

```
    # Print bottom border
    print(" └" + "─" * table_width + "┘ ")

    # Print result with styling
    print("\n" + "="*50)
    result_text = f"KB ENTAILS ALPHA: {'✓ YES' if entails else '✗ NO'}"
    print(f"{result_text:^50}")
    print("="*50)
    return entails

# --- Interactive input ---
print("Name :Umesha H N \nUSN:1BM24CS428")
print("Enter Knowledge Base (KB) sentences, separated by commas.")
print("Use symbols like A, B, C and operators: and, or, not, =>, <=>")
kb_input = input("KB: ").strip()
KB = [x.strip() for x in kb_input.split(",")]
alpha = input("Enter query (alpha): ").strip()
result = tt_entails(KB, alpha)
print(f"Result: {result}")
```

Output

```
Name :Umesha H N
USN:1BM24CS428
Enter Knowledge Base (KB) sentences, separated by commas.
Use symbols like A, B, C and operators: and, or, not, =>, <=>
KB: NOT
Enter query (alpha): T

==================================================
                TRUTH TABLE
==================================================
┌───────┬───────┬───────┬───────┐
│  NOT  │   T   │  KB   │ alpha │
├───────┼───────┼───────┼───────┤
│ True  │ True  │ True  │ True  │
│ True  │ False │ True  │ False │
│ False │ True  │ False │ True  │
│ False │ False │ False │ False │
└───────┴───────┴───────┴───────┘

==================================================
              KB ENTAILS ALPHA: ✗ NO
==================================================
Result: False
```

Propositional logic 22-09-25

Algorithm

① List all variables

* Find all the symbols that appear in KB and α.

Ex: A, B, C

② Try every possibility

* each symbol can be True or False.
* so we test all combinations

③ Check KB.

For each combination, see if KB is true

④ Check α

* if KB is true, then α must also be true.
* if KB is false, we dont care about α in that row

⑤ Final decision

* If in all cases, where KB is true α is also true → KB entails α
* If in any case KB is true but α false → KB does not entail α.

-25

appro

Fals.

true

but

output

22/9/25

~~NOT~~

KB : NOT

Enter Query (alpha): T

Truth Table

NOT	T	KB	alpha
True	True	True	True
True	False	True	False
False	True	False	true
False	Fals	False	False

Result: False

22/9/25

Program 7

Implement unification in first order logic

Code:

```
import re
from collections import namedtuple

Var = namedtuple('Var', ['name'])
Const = namedtuple('Const', ['name'])
Func = namedtuple('Func', ['name', 'args'])

def parse(s):
    s = s.strip()
    if '(' in s:
        n, rest = s[:s.index('(')], s[s.index('(')+1:-1]
        args = []
        depth = 0; current = []
        for c in rest + ',':
            if c == ',' and depth == 0:
                args.append(''.join(current).strip())
                current = []
            else:
                if c == '(': depth += 1
                elif c == ')': depth -= 1
                current.append(c)
        return Func(n, [parse(a) for a in args])
    if re.fullmatch(r'[a-z][a-z0-9]*', s): return Var(s)
    return Const(s)

def occurs(v, x, s):
    x = subst(x, s)
    if v == x: return True
    if isinstance(x, Func):
        return any(occurs(v, a, s) for a in x.args)
    return False

def subst(t, s):
    while isinstance(t, Var) and t.name in s:
        t = s[t.name]
    if isinstance(t, Func):
        return Func(t.name, [subst(a, s) for a in t.args])
    return t
```

```
def unify(t1, t2, s=None):
    if s is None: s = {}
    t1, t2 = subst(t1, s), subst(t2, s)
    if t1 == t2: return s
    if isinstance(t1, Var):
        if occurs(t1, t2, s): return None
        s[t1.name] = t2
        return s
    if isinstance(t2, Var):
        if occurs(t2, t1, s): return None
        s[t2.name] = t1
        return s
    if isinstance(t1, Func) and isinstance(t2, Func):
        if t1.name != t2.name or len(t1.args) != len(t2.args): return None
        for a1, a2 in zip(t1.args, t2.args):
            s = unify(a1, a2, s)
            if s is None: return None
        return s
    if isinstance(t1, Const) and isinstance(t2, Const) and t1.name == t2.name:
        return s
    return None

def to_str(t):
    if isinstance(t, Var) or isinstance(t, Const):
        return t.name
    return f"{t.name}({', '.join(to_str(a) for a in t.args)})"

def show_subs(s):
    if s is None:
        print("Unification failed.")
    elif not s:
        print("No substitution needed.")
    else:
        for k,v in s.items():
            print(f"{k} = {to_str(v)}")
print("Name:Umesha H N\nUSN:1BM24CS428\n\n")
tests = [
    ("p(b,X,f(g(Z)))", "p(z,f(Y),f(Y))"),
    ("Q(a,g(x,a),f(y))", "Q(a,g(f(b),a),x)"),
    ("p(f(a),g(Y))", "p(X,X)"),
    ("prime(11)", "prime(y)"),
    ("knows(John,x)", "knows(y,mother(y))"),
    ("knows(John,x)", "knows(y,Bill)")
```

```
]

for e1, e2 in tests:
    print(f"Unifying: {e1} and {e2}")
    s = unify(parse(e1), parse(e2))
    show_subs(s)
    print('-'*40)
```

Output

```
Name:Umesha H N
USN:1BM24CS428



Unifying: p(b,X,f(g(Z))) and p(z,f(Y),f(Y))
Unification failed.
----------------------------------------
Unifying: Q(a,g(x,a),f(y)) and Q(a,g(f(b),a),x)
x = f(b)
y = b
----------------------------------------
Unifying: p(f(a),g(Y)) and p(X,X)
Unification failed.
----------------------------------------
Unifying: prime(11) and prime(y)
y = 11
----------------------------------------
Unifying: knows(John,x) and knows(y,mother(y))
y = John
x = mother(John)
----------------------------------------
Unifying: knows(John,x) and knows(y,Bill)
y = John
x = Bill
----------------------------------------
```

Lab program : 7 13/10/25

① unification is a process to find substitution that make ... different for (first order logic)

① unify (knows (John, x) , knows (john, jane))

$\theta = x / Jane$

unify (knows (john, jane), know (john, jane))

② unify (knows (John, x) , known (y, Bill))

$\theta = y / John$

knows (John, x) , know (John, Bill)

$\theta = x / Bill$

knows (john, Bill), known (john, bill)

Algorithm

Step :- If ψ_1 or ψ_2 is a variable or constant, value

a) If ψ_1 or ψ_2 are identical, then return NIL.

b) Else if ψ_1 is a variable

a. then if ψ_1 occurs in ψ_2, then return Failure

b. Else return $\{(\psi_2 / \psi_1)\}$

c. Else if ψ_2 is a variable

a. if ψ_2 occurs in ψ_1 then return value

b. Else return {(ψ_1 / ψ_2)}. 13/10/25

d. Else return FAILURE

Step 2: If the initial predicate symbol in ψ_1 and ψ_2 are not same, then return. FAILURE

Step 3:- If ψ_1 and ψ_2 have a different n-umber of arguments, then return F-AILURE

Step 4: Set Substitution set(SUBST) to NIL

Step 5: For i = 1 to the number of el-ements in ψ_1.

a) Call unify function with the ith. element of ψ_2, and put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST = Append (S_SUBST)

Step 6: Return SUBST.

① Unify {Prime (11) and Prime (y)}

$\theta = 11/y$

Unify {Prime (11) and Prime (y)}

② Unify {Knows (John, x) Knows (y, mother(y))}

θ = John/y

{knows (y, x) , knows (11, mothegr(y))} 13/10/21

⟹ failure

3) unify {knows (John, x), knows (y, Bill)}

$\theta = y \mid John$

knows(John, x) , knows (John, Bill)

$\theta = x \mid Bill$

knows (John, Bill), knows (John, Bill)

4. find MGU of {P (f(a), g(y)), P(x, x)}

$\theta = f(a) \mid x$

{P (x, g(y)), P(x, x)}

unification fails

5 MGU of {Q (a, g(x, a), f(y)) and Q(a, g(f(b), a), x)

$\theta = x \mid f(b)$

unify {Q (a, g(f(b), a), f(y))}

$\theta = f(y) \mid x$

unify {(Q (a, g(f(b), a)), Q(a, g(f(b), a)))}

Program 8

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Code:

```
class Person:
    def __init__(self, name, nationality):
        self.name = name
        self.nationality = nationality

class Country:
    def __init__(self, name, hostile_to=None):
        self.name = name
        self.hostile_to = hostile_to if hostile_to else []

class Weapon:
    def __init__(self, name, owner=None):
        self.name = name
        self.owner = owner

robert = Person("Robert", "American")
countryA = Country("CountryA", hostile_to=["America"])

missiles = [
    Weapon("Missile1", owner=countryA),
    Weapon("Missile2", owner=countryA),
]

def sold_by(person, weapon):
    return weapon.owner == countryA and person == robert

def is_hostile(buyer, seller_country_name):
    return seller_country_name in buyer.hostile_to

def is_weapon(item):
    return isinstance(item, Weapon)

def prove_robert_criminal(person):
    print(f"Step 1: Check if {person.name} is American.")
    if person.nationality == "American":
        print(f"  {person.name} is American.")
    else:
```

```
        print(f"  {person.name} is NOT American. Proof ends here.")
        return False

    print(f"Step 2: Check if CountryA is hostile to America.")
    if is_hostile(countryA, "America"):
        print(f"  CountryA is hostile to America.")
    else:
        print(f"  CountryA is NOT hostile to America. Proof ends here.")
        return False

    print(f"Step 3: Check missiles owned by CountryA.")
    for missile in missiles:
        print(f"  Missile '{missile.name}' owned by {missile.owner.name}")

    print(f"Step 4: Check if {person.name} sold these missiles.")
    for missile in missiles:
        if sold_by(person, missile):
            print(f"  {person.name} sold {missile.name}.")
        else:
            print(f"  {person.name} did NOT sell {missile.name}. Proof ends here.")
            return False

    print(f"Step 5: Confirm missiles are weapons.")
    for missile in missiles:
        if is_weapon(missile):
            print(f"  {missile.name} is a weapon.")
        else:
            print(f"  {missile.name} is NOT a weapon. Proof ends here.")
            return False

    print(f"Step 6: Apply the law: American selling weapons to hostile nations is criminal.")
    print(f"Step 7: All conditions met, so {person.name} is criminal.")
    return True

if prove_robert_criminal(robert):
    print("\nConclusion: Robert is criminal.")
else:
    print("\nConclusion: Robert is NOT criminal.")
```

Output

```
Step 1: Check if Robert is American.
  Robert is American.
Step 2: Check if CountryA is hostile to America.
  CountryA is hostile to America.
Step 3: Check missiles owned by CountryA.
  Missile 'Missile1' owned by CountryA
  Missile 'Missile2' owned by CountryA
Step 4: Check if Robert sold these missiles.
  Robert sold Missile1.
  Robert sold Missile2.
Step 5: Confirm missiles are weapons.
  Missile1 is a weapon.
  Missile2 is a weapon.
Step 6: Apply the law: American selling weapons to hostile nations is criminal.
Step 7: All conditions met, so Robert is criminal.

Conclusion: Robert is criminal.
```

Week - 8 13/10/25

Lab program - 8 First order logic.

P => Q → conclusion

Rules:

$P \Rightarrow Q$

$L \land M \Rightarrow P$

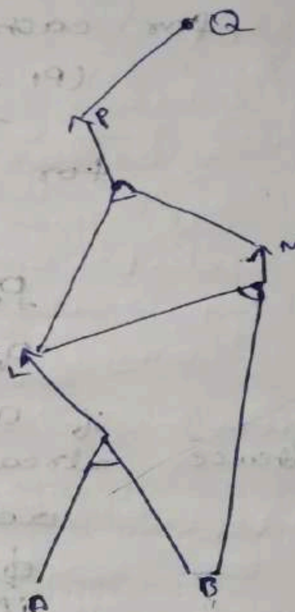
$B \land L \Rightarrow M$

$A \land P \Rightarrow L$

$A \land B \Rightarrow L$

facts: A, B

$\boxed{\text{Prove } Q}$



Q) The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has so missiles, and all of its missiles were sold to it by Colonel West, who is American. An enemy of America counts as "hostile".

- Prove that "West is Criminal"

Algorithm.

function FOL-FC-ASK(KB, α)

returns a substitution or false

inputs: KB, the knowledge base, a set of first-order definite clauses α, the query, an atomic sentence

local variables: new, the new sentence, inferred on each iteration

repeat until new is empty

new ← { ϕ }

for each rule in KB do

(P1 ∧ ... ∧ Pn ⇒ Q) ← STANDARDIZE-VARIABLES(rule)

for each θ such that SUBST(θ, P1 ∧ ... Pn) = SUBST(θ, P1 ∧ ... Pn')

for some P1' ... Pn' in KB

q' ← SUBST(θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

ϕ ← unify(q', α)

if ϕ is not fail then return ϕ

add new to KB

return false.

1) American(P) ∧ Weapon(Y) ∧ Sells(x, y, z) ∧ Hostile(z) ⇒ Criminal(x)

2) ∃ x missile(x) owns(nona, x) ⇒ Sells(cost x, Nano)

3) ∀ x Enemy (x, American) ⟹ Hostile (x) 13/10/25

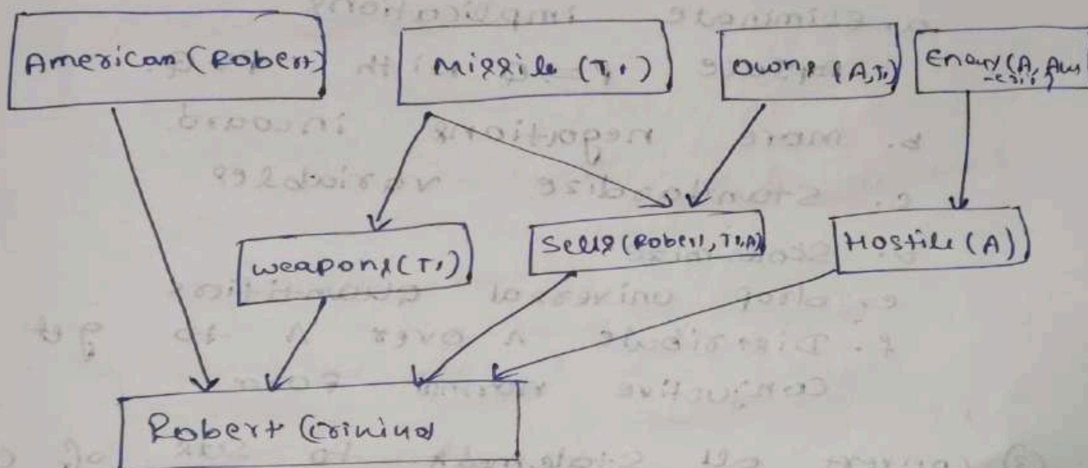
4) ∀ x Missile (x) = weapon(x)

5) American (Robert)

6) Enemy (Nona, America)

7) owns (Nono, M1) and

8) missile (M1)



American (p) ∧ weapon(q) sells(p,q,r) ∧ Hostile(r)

⟹ Criminal (p)

output

All conditions met, Robert is Criminal

Conclusion: Robert is criminal

13/10/25

Program 9

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution.

Code:

```
from typing import List, Set

class Predicate:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __eq__(self, other):
        return self.name == other.name and self.args == other.args
    def __hash__(self):
        return hash((self.name, tuple(self.args)))
    def __repr__(self):
        return f"{self.name}({', '.join(self.args)})"

def negate(pred):
    if pred.name.startswith("~"):
        return Predicate(pred.name[1:], pred.args)
    else:
        return Predicate("~" + pred.name, pred.args)

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x[0].islower():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y[0].islower():
        return unify_var(y, x, subst)
    elif isinstance(x, Predicate) and isinstance(y, Predicate):
        if x.name != y.name or len(x.args) != len(y.args):
            return None
        for a, b in zip(x.args, y.args):
            subst = unify(a, b, subst)
        return subst
    else:
        return None

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
```

```
    elif x in subst:
        return unify(var, subst[x], subst)
    else:
        subst[var] = x
        return subst

def resolution(kb: List[Set[Predicate]], query: Predicate):
    clauses = kb.copy()
    clauses.append({negate(query)})
    print("\nInitial Clauses:")
    for c in clauses:
        print(c)
    while True:
        new = []
        n = len(clauses)
        for i in range(n):
            for j in range(i + 1, n):
                resolvents = resolve(clauses[i], clauses[j])
                if set() in resolvents:
                    print("\nDerived empty clause {}. Hence, Query is PROVED.")
                    return True
                for res in resolvents:
                    if res not in clauses and res not in new:
                        new.append(res)
        if not new:
            print("\nNo new clauses derived. Query CANNOT be proved.")
            return False
        for c in new:
            clauses.append(c)

def resolve(ci: Set[Predicate], cj: Set[Predicate]):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di.name == "~" + dj.name or "~" + di.name == dj.name:
                subst = unify(di, negate(dj), {})
                if subst is not None:
                    new_clause = (ci.union(cj) - {di, dj})
                    new_clause = {apply_substitution(p, subst) for p in new_clause}
                    resolvents.append(new_clause)
    return resolvents

def apply_substitution(pred, subst):
    new_args = [subst.get(arg, arg) for arg in pred.args]
    return Predicate(pred.name, new_args)

KB = [
```

```
    {Predicate("~Food", ["x"]), Predicate("Likes", ["John", "x"])},
    {Predicate("Food", ["Apple"])},
    {Predicate("Food", ["Vegetable"])},
    {Predicate("~Eats", ["x", "y"]), Predicate("~Killed", ["x"]), Predicate("Food", ["y"])},
    {Predicate("Eats", ["Anil", "Peanut"])},
    {Predicate("Alive", ["Anil"])},
    {Predicate("~Eats", ["Anil", "x"]), Predicate("Eats", ["Harry", "x"])},
    {Predicate("~Alive", ["x"]), Predicate("~Killed", ["x"])},
    {Predicate("Killed", ["x"]), Predicate("Alive", ["x"])},
]

query = Predicate("Likes", ["John", "Peanut"])
print("Name:Umesha H N\nUSN:1BM24CS428\n")
print("RESOLUTION PROCESS ")
proved = resolution(KB, query)
print("\nRESULT:", "Query is TRUE (proved by resolution)" if proved else "Query is FALSE (not provable)")
```

Output

```
Name:Umesha H N
USN:1BM24CS428

RESOLUTION PROCESS

Initial Clauses:
{~Food(x), Likes(John, x)}
{Food(Apple)}
{Food(Vegetable)}
{~Killed(x), Food(y), ~Eats(x, y)}
{Eats(Anil, Peanut)}
{Alive(Anil)}
{Eats(Harry, x), ~Eats(Anil, x)}
{~Alive(x), ~Killed(x)}
{Alive(x), Killed(x)}
{~Likes(John, Peanut)}

Derived empty clause {}. Hence, Query is PROVED.

RESULT: Query is TRUE (proved by resolution)
```

week 9

Resolution in First order logic

Algorithm

1. write all the given facts and rules in First order logic (FOL)

2. Convert all FOL statements into conjuctive Normal Form
   a. Eliminate implications
      . Replace $P \Rightarrow Q$ with $\neg P \vee Q$.
   b. more negations inward.
   c. Standardize variables
   d. Skolemize
   e. drop universal quantifiers
   f. Distribute $\wedge$ over $\wedge$ to get conjuctive normal form.

3. convert all statements to sets of clauses.
   * Repeat the KB and $\neg Q$ (negation of query

4. Negate the Query ($\neg Q$).
   Add $\neg Q$ to the KB - this forms the basis for refutation

5. Apply the Resolution Rule.
   * Select two clauses containing complementary literals

(5) Add the resolvent to the clause set

(6) Repeat until

a) The empty clause (⊥) is derived - Query proven true

- (b) No new clauses ~~...~~ can be generated - Query not entailed.

Output:

Resolution Process

Initial Clauses:

{~Food(x), Likes(John, x)}

{Food(Apple)}

{Food(vegetable)}

{~killed(x), Food(y), ~Eats(x, y)}

{Eats(Anil, Peanut)}

{Alive(Anil)}

{Eats(Harry, x), ~Eats(Anil, x)}

{~Alive(x), ~killed(x)}

{Alive(x), killed(x)}

{~Likes(John, Peanut)}

Derived empty clause { }. Hence Query is PROVED.

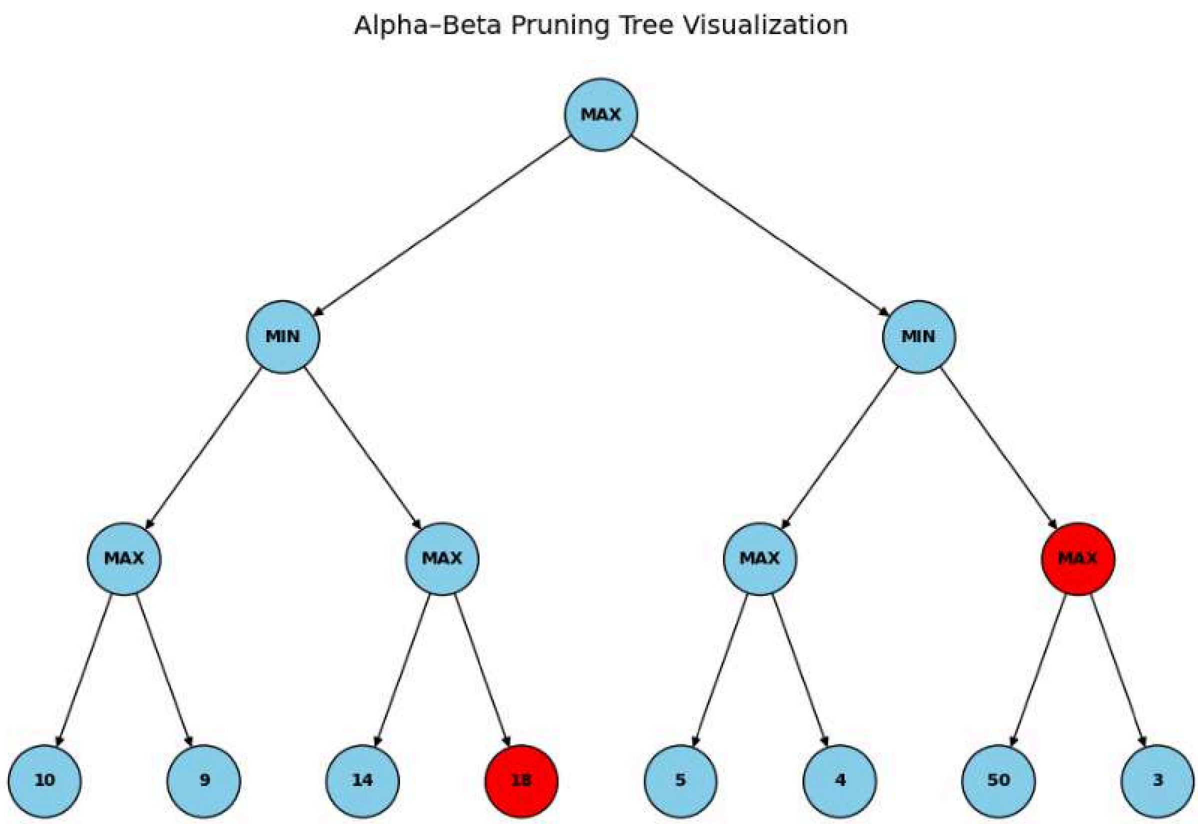
Result: Query is True (Proved by resolution)

Program 10
Implement Alpha-Beta Pruning

Code:
```
import math

def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta, max_depth):
    if depth == max_depth:
        return values[node_index]

    if maximizing_player:
        best = -math.inf
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth}, node {node_index}, α={alpha}, β={beta}")
                break
        return best
    else:
        best = math.inf
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth}, node {node_index}, α={alpha}, β={beta}")
                break
        return best


values = [10, 9, 14, 18, 5, 4, 50, 3]
max_depth = 3
print("Name:Umesha H N\nUSN:1BM24CS428\n")
print("ALPHA–BETA PRUNING PROCESS\n")
optimal_value = alpha_beta(0, 0, True, values, -math.inf, math.inf, max_depth)
print("\nOptimal value (Root Node):", optimal_value)
```

Output

```
Name:Umesha H N
USN:1BM24CS428

ALPHA-BETA PRUNING PROCESS

Pruned at depth 2, node 1, α=14, β=10
Pruned at depth 1, node 1, α=10, β=5

Optimal value (Root Node): 10
```



Week-10

The Alpha-beta Search Algorithm

Algorithm

function Alpha-Beta-Search (State)
returns an action
v ← MAX-VALUE (State, −∞, +∞)
return the action in Actions(State)
with value n

function MAX-value (State, α, β)
returns a utility value
if Terminal-Test (State)
then return utility (State)
v ← −∞
for each a in Actions (State)
do
v ← MAX (v, Min-value (Result (s, a), α, β))
if v ≥ β
then return v
α ← MAX (α, v)
return v.

function Min-value (State, α, β)
returns a utility value.
if Terminal-Test (State)
then return UTILITY (State)
v ← +∞
for each a in Actions (State)
do
v ← min (v, MAX-value (Result (s, a), α, β))

β ← Min (β, v)
return v.

output

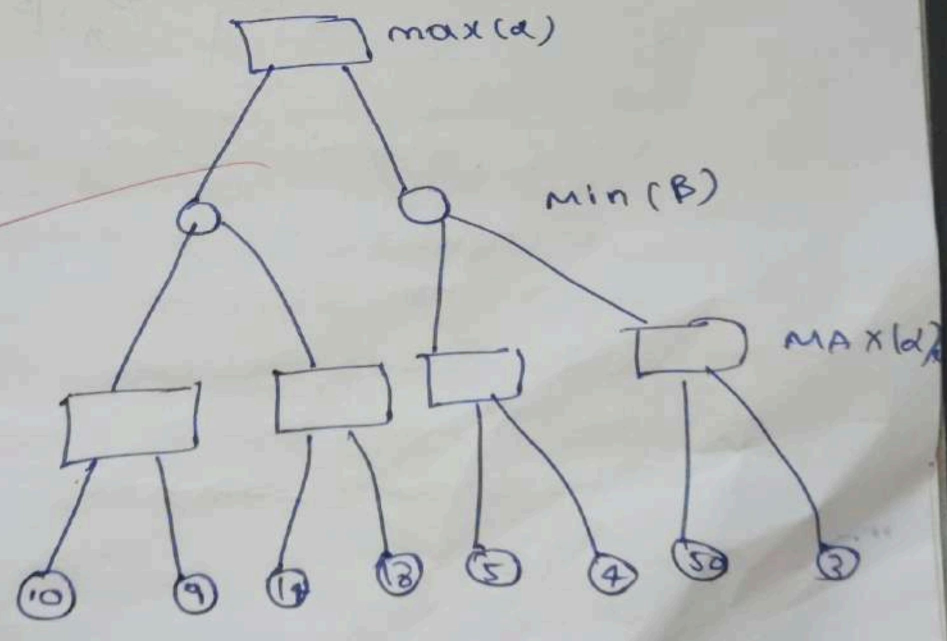
Alpha-Beta pruning process:

Pruned at depth 2, node 1, α = 14, β = 10
Pruned at depth 1, node 1, α = 10, β = 5

Optimal value (Root Node): 10

problem

Apply the Alpha-beta search Algorithm to find value of root node and path to root node (max node). Identify the paths which are pruned for exploration



Solution