

Artificial Intelligence Lab Report



Submitted by

Varsha Prasanth(1BM22CS321)

Batch: 3F

Course: Artificial Intelligence

Course Code: 23CS5PCAIN

Sem & Section: 5F

**BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING**



B. M. S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

2022-2023

Table of contents

Program Number	Date	Program Title Page Number	Page No.
1	04/10/2024	Tic Tac Toe	1 - 6
2	18/10/2024	8-Puzzle problem using BFS	7 - 10
3	18/10/2024	8-Puzzle problem using DFS	11 -15
4	18/10/2024	8-Puzzle problem using A*	16 - 20
5	18/10/2024	Vacuum Cleaner Agent	21 - 25
6	08/11/2024	Hill Climbing	26 - 31
7	15/11/2024	Simulated Annealing	32 - 35
8	22/11/2024	Knowledge Base -Resolution	36 - 39
9	29/11/2024	Unification in First Order Logic	40 - 56
10	29/11/2024	First Order Logic to Conjunctive Normal Form	57 - 60
11	29/11/2024	Forward Reasoning	61 - 64
12	29/11/2024	Alpha - Beta Pruning	65 - 67

Program 1 - Tic Tac toe

Algorithm

LAB 1 : TIC TAC TOE GAME

Date 4, 10, 24
Page 1

```
function minimax (node, depth, is Maximizing Player) :  
    if node is a terminal state :  
        return evaluate(node)  
  
    if is Maximizing Player :  
        best Value = -infinity  
        for each child in node :  
            value = minimax (child, depth + 1, false)  
            best Value = max (best Value, value)  
        return best Value  
  
    else :  
        best Value = +infinity  
        for each child in node :  
            value = minimax (child, depth + 1, true)  
            best Value = min (best Value, value)  
        return best Value
```

2 u10/24

0 0 X
X 0 X 0
X
↓
O O X
X X 0
X
↓
O O X
X X 0
X
↓
O O X
X X 0
X
↓
Tie X X Tie X X

Code

```
board = {1: ' ', 2: ' ', 3: ' ',
         4: ' ', 5: ' ', 6: ' ',
         7: ' ', 8: ' ', 9: ' '}

def printBoard(board):
    print(board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('---+---')
    print(board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('---+---')
    print(board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

def checkWin():
    win_conditions = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9), # Rows
        (1, 4, 7), (2, 5, 8), (3, 6, 9), # Columns
        (1, 5, 9), (3, 5, 7) # Diagonals
    ]
    for a, b, c in win_conditions:
        if board[a] == board[b] == board[c] and board[a] != ' ':
            return True
```

```

        return False

def checkMoveForWin(move):
    win_conditions = [
        (1, 2, 3), (4, 5, 6), (7, 8, 9),
        (1, 4, 7), (2, 5, 8), (3, 6, 9),
        (1, 5, 9), (3, 5, 7)
    ]
    for a, b, c in win_conditions:
        if board[a] == board[b] == move and board[c] != ' ':
            return True
    return False

def checkDraw():
    return all(board[key] != ' ' for key in board.keys())

def insertLetter(letter, position):
    if spaceFree(position):
        board[position] = letter
        printBoard(board)
        if checkDraw():
            print('Draw!')
        elif checkWin():
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
        return
    else:
        print('Position already filled')
        return False

def checkWin():
    for condition in win_conditions:
        if board[condition[0]] == board[condition[1]] == board[condition[2]] != ' ':
            return True
    return False

```

```

        print('Position taken, please pick a different position.')
    position = int(input('Enter new position: '))
    insertLetter(letter, position)

player = 'O'
bot = 'X'

def playerMove():
    position = int(input('Enter position for O: '))
    insertLetter(player, position)

def compMove():
    bestScore = -1000
    bestMove = 0
    for key in board.keys():
        if board[key] == ' ':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key
    insertLetter(bot, bestMove)

def minimax(board, isMaximizing):
    if checkMoveForWin(bot):
        return 1
    elif checkMoveForWin(player):
        return -1

```

```

    elif checkDraw():
        return 0

    if isMaximizing:

        bestScore = -1000

        for key in board.keys():

            if board[key] == ' ':

                board[key] = bot

                score = minimax(board, False)

                board[key] = ' '

                bestScore = max(score, bestScore)

        return bestScore

    else:

        bestScore = 1000

        for key in board.keys():

            if board[key] == ' ':

                board[key] = player

                score = minimax(board, True)

                board[key] = ' '

                bestScore = min(score, bestScore)

        return bestScore

while not checkWin() and not checkDraw():

    compMove()

    if checkWin() or checkDraw():

        break

    playerMove()

    name = "Varsha Prasanth"

usn = "1BM22CS321"

```

```
print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

```
→ x| |
-+-
| |
-+-
| |
```

Enter position for 0: 5

```
x| |
-+-
|o|
-+-
| |
```

```
x|x|
-+-
|o|
-+-
| |
```

Enter position for 0: 3

```
x|x|o
-+-
|o|
-+-
| |
```

```
x|x|o
-+-
x|o|
-+-
| |
```

Enter position for 0: 7

```
x|x|o
-+-
x|o|
-+-
o| |
```

You win!

Name: Varsha Prasanth, USN: 1BM22CS321

Program 2 - 8 Puzzle Using BFS

Algorithm

program :- 02

8 puzzle.

problem statement:- execute BFS to solve the 8 puzzle problem starting from the initial state.

initial state

1	2	5
3	4	0
6	7	8

good state

0	1	2
3	4	5
6	7	8

BFS algorithm:-

s start node

initialize a queue with s;

v = pop(a);

if v is "goal" return success;

make node v as visited;

operate on v;

for each node w accessible from node v do

 if w is not marked as visited then push
 w at the back of a;

end for;

Code

```
from collections import deque

# Define the goal state
GOAL_STATE = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

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

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # right, down, left,
    up

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state] # create a copy
            new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)

    return neighbors

def bfs(start_state):
    start_tuple = tuple(tuple(row) for row in start_state)
    goal_tuple = tuple(tuple(row) for row in GOAL_STATE)

    queue = deque([start_state]) # Use the list representation for the
queue
    visited = {start_tuple: None}

    while queue:
        current_state = queue.popleft()
```

```

        if tuple(tuple(row) for row in current_state) == goal_tuple:
            break

    for neighbor in get_neighbors(current_state):
        neighbor_tuple = tuple(tuple(row) for row in neighbor)
        if neighbor_tuple not in visited:
            visited[neighbor_tuple] = current_state
            queue.append(neighbor)

# Backtrack to find the solution path
path = []
while current_state is not None:
    path.append(current_state)
    current_state = visited[tuple(tuple(row) for row in current_state)]


return path[::-1] # Return reversed path

def print_solution(path):
    for state in path:
        for row in state:
            print(row)
        print()

# Example usage
start_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]
solution_path = bfs(start_state)
print_solution(solution_path)
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")

```

Output Snapshot

[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Name: Varsha Prasanth, USN: 1BM22CS321

State Space Tree

BFS output	1 2 3	1 2 3	1 2 3
	4 5 6	4 5 0	4 5 6
	7 0 8	4 0 6	0 7 6
	move up	move left	move right
	:	:	:
	1 2 3	1 2 3	1 2 3
	0 5 6	4 5 6	4 5 6
	7 4 8	0 7 8	7 0 8
	move up	move down	move down
	:		
	1 2 3		
	4 5 6		
	7 8 0		
	Goal		

Program 3 - 8 puzzle using DFS

Algorithm

8-puzzle Problem Using BFS & DFS

Date _____
Page _____

BFS (queue)

Let fringe be a list containing the initial state
loop

- if fringe is empty return failure
- Node \leftarrow remove-first (fringe)
- if Node is a goal
 - then return the path from initial state to Node
 - else generate all successors of Node, and add generated nodes to the back of fringe

End Loop

DFS (stack)

Let fringe be a list containing the initial state
loop

- if fringe is empty return failure
- Node \leftarrow remove-first (fringe)
- if Node is a goal
 - then return the path from initial state to Node
 - else generate all successors of Node, and add generated nodes to the front of fringe

END LOOP

~~8P~~ State space tree

Start state:	1 2 3	Goal state:	1 2 3
	4 5 6		4 5 6
	0 7 8		7 8 0

Code

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

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)] # right, down, left, up

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state] # Create a copy
            new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)

    return neighbors

def iterative_dfs(start_state):
    stack = [start_state]
    visited = set()
    path_map = {tuple(map(tuple, start_state)): None}

    while stack:
        current_state = stack.pop()

        if current_state == GOAL_STATE:
            return reconstruct_path(path_map, current_state)

        visited.add(tuple(map(tuple, current_state)))
```

```

        for neighbor in get_neighbors(current_state):
            neighbor_tuple = tuple(map(tuple, neighbor))
            if neighbor_tuple not in visited and neighbor_tuple not in
path_map:
                path_map[neighbor_tuple] = current_state
                stack.append(neighbor)

    return None

def reconstruct_path(path_map, goal_state):
    path = []
    current = goal_state
    while current is not None:
        path.append(current)
        current = path_map[tuple(map(tuple, current))]
    return path[::-1] # Reverse to get the correct order

def print_solution(path):
    if path:
        for state in path:
            for row in state:
                print(row)
            print()
    else:
        print("No solution found.")

# Example usage
GOAL_STATE = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
start_state = [[1, 2, 3], [4, 0, 6], [7, 5, 8]]

solution_path = iterative_dfs(start_state)
print_solution(solution_path)
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")

```

Output Snapshot

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

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

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

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

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

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

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

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

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

Name: Varsha Prasanth, USN: 1BM22CS321

17

StateSpaceTree

DFS		
	1	2
	3	
	4	5
	6	
	0	4
	8	
	move up	
	1	2
	3	
	0	5
	6	
	4	7
	8	
	move up	
	1	2
	3	
	4	5
	6	
	7	0
	8	
	Goal state	

Program 04 - 8 Puzzle Using A*

Algorithm

THE A* ALGORITHM

Date 25/10/2024
Page _____

For A* search (problem) returns a solution or failure

$n \leftarrow$ a node n with n .state = Problem. initial state

$n.g = 0$

$\text{frontier} \leftarrow$ a priority queue ordered by ascending $g + h$
only element n

loop do

- if empty? (frontier) then return failure
- $n \leftarrow \text{pop}(\text{frontier})$
- if problem.goalTest(n .state) then return solution(n)
- for each action a in Problem.actions(n .state) do
- $n' \leftarrow \text{childNode}(\text{Problem}, n, a)$
- insert $(n', g+n') + h(n')$, frontier)

Misplaced

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

$n=3$ $f=4$
 $g=1$

$h=1$ $g=1$ $f=2$	$h=3$ $g=1$ $f=4$	$h=5$ $g=1$ $f=4$
$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 8 & 5 \\ 7 & 0 & 6 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 7 & 8 & 6 \end{bmatrix}$
$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 6 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & 3 \\ 4 & 2 & 5 \\ 7 & 8 & 6 \end{bmatrix}$

$h=0$ $g=2$ $f=4$ $h=2$ $g=2$ $f=4$

$h=0$ $g=2$ $f=4$	$h=2$ $g=2$ $f=4$
$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 3 \\ 7 & 8 & 6 \end{bmatrix}$

goal

Code

```
import heapq

class PuzzleState:

    def __init__(self, board, depth=0, path=''):
        self.board = board
        self.blank_index = board.index(0) # Index of the blank tile
        self.depth = depth
        self.path = path # Path taken to reach this state

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = divmod(self.blank_index, 3)

        if row > 0: # Up
            moves.append(-3)
        if row < 2: # Down
            moves.append(3)
        if col > 0: # Left
            moves.append(-1)
        if col < 2: # Right
            moves.append(1)

        return moves

    def generate_new_state(self, move):
        new_index = self.blank_index + move
        new_board = self.board[:]
        new_board[self.blank_index], new_board[new_index] =
        new_board[new_index], new_board[self.blank_index]

        return PuzzleState(new_board, self.depth + 1, self.path +
str(new_board))
```

```

def heuristic_misplaced(self):
    return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i]
!= i + 1)

def heuristic_manhattan(self):
    distance = 0
    for i in range(9):
        if self.board[i] != 0:
            target_row, target_col = divmod(self.board[i] - 1, 3)
            current_row, current_col = divmod(i, 3)
            distance += abs(target_row - current_row) + abs(target_col -
current_col)
    return distance

def __lt__(self, other):
    return False # Needed for priority queue to compare states

def a_star(initial_board, heuristic_type='misplaced'):
    start_state = PuzzleState(initial_board)
    if start_state.is_goal():
        return start_state.path

    # Priority queue for open states
    open_set = []
    heapq.heappush(open_set, (0, start_state))

    # Set to track visited states
    visited = set()

    while open_set:
        current_cost, current_state = heapq.heappop(open_set)

        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))

```

```

        for move in current_state.get_possible_moves():
            new_state = current_state.generate_new_state(move)
            if tuple(new_state.board) in visited:
                continue

            # Calculate h(n) based on selected heuristic
            if heuristic_type == 'misplaced':
                h = new_state.heuristic_misplaced()
            elif heuristic_type == 'manhattan':
                h = new_state.heuristic_manhattan()
            else:
                raise ValueError("Invalid heuristic type")

            # Calculate f(n)
            f = new_state.depth + h

            # Print f(n) for the current state
            print(f"Current State: {new_state.board}, g(n): {new_state.depth}, h(n): {h}, f(n): {f}")

            heapq.heappush(open_set, (f, new_state))

        return None # No solution found

# Example usage
initial_board = [1, 2, 3, 4, 5, 6, 0, 7, 8] # 0 represents the blank tile
print("Solution Path (Misplaced Tiles):", a_star(initial_board,
heuristic_type='misplaced'))
print("Solution Path (Manhattan Distance):", a_star(initial_board,
heuristic_type='manhattan'))

name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")

```

Output Snapshot

```

→ Current State: [1, 2, 3, 0, 5, 6, 4, 7, 8], g(n): 1, h(n): 3, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 0, 8], g(n): 1, h(n): 1, f(n): 2
Current State: [1, 2, 3, 4, 0, 6, 7, 5, 8], g(n): 2, h(n): 2, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 8, 0], g(n): 2, h(n): 0, f(n): 2
Solution Path (Misplaced Tiles): [1, 2, 3, 4, 5, 6, 7, 0, 8][1, 2, 3, 4, 5, 6, 7, 8, 0]
Current State: [1, 2, 3, 0, 5, 6, 4, 7, 8], g(n): 1, h(n): 3, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 0, 8], g(n): 1, h(n): 1, f(n): 2
Current State: [1, 2, 3, 4, 0, 6, 7, 5, 8], g(n): 2, h(n): 2, f(n): 4
Current State: [1, 2, 3, 4, 5, 6, 7, 8, 0], g(n): 2, h(n): 0, f(n): 2
Solution Path (Manhattan Distance): [1, 2, 3, 4, 5, 6, 7, 0, 8][1, 2, 3, 4, 5, 6, 7, 8, 0]
Name: Varsha Prasanth, USN: 1BM22CS321

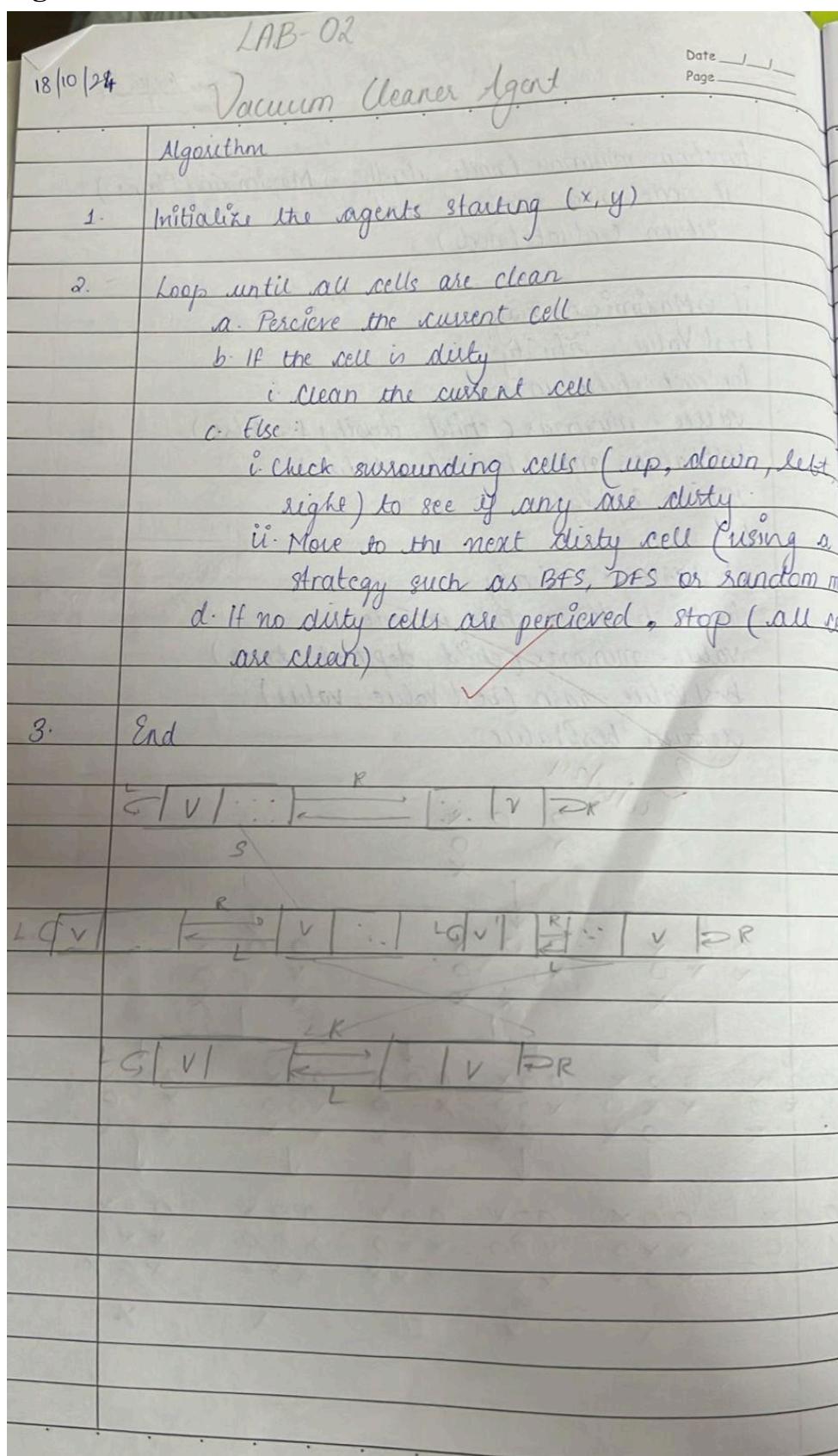
```

		Misplaced	
		1 2 3	1 2 3
		4 0 5	4 5 6
		7 8 6	7 8
$g=1$	$f=4$		
		$h=1$ $g=1$ $f=2$	$h=2$ $g=1$ $f=4$
		$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 8 & 5 \\ 7 & 0 & 6 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 0 \\ 7 & 8 & 6 \end{bmatrix}$
			$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 7 & 8 & 6 \end{bmatrix}$
			$\begin{bmatrix} 1 & 0 & 3 \\ 4 & 2 & 5 \\ 7 & 8 & 6 \end{bmatrix}$
		$h=0$ $g=2$ $f=2$	$h=2$ $g=2$ $f=4$
		$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 7 & 8 & 6 \end{bmatrix}$
			$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 3 \\ 7 & 8 & 6 \end{bmatrix}$
		goal	

		Manhattan	
		1 2 3	1 2 3
		4 0 5	4 5 6
		7 8 6	7 8
$g=1$	$h=3$ $f=4$		
		$g=1$ $h=1$ $f=2$	$g=1$ $h=3$ $f=4$
		$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 8 & 5 \\ 7 & 0 & 6 \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 0 \\ 7 & 8 & 6 \end{bmatrix}$
			$\begin{bmatrix} 1 & 2 & 3 \\ 6 & 4 & 5 \\ 7 & 8 & 6 \end{bmatrix}$
			$\begin{bmatrix} 1 & 0 & 3 \\ 4 & 2 & 5 \\ 7 & 8 & 6 \end{bmatrix}$
		$g=2$ $h=0$ $f=2$	$g=2$ $h=2$ $f=4$
		$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \end{bmatrix}$	$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 5 \\ 7 & 8 & 6 \end{bmatrix}$
			$\begin{bmatrix} 1 & 2 & 0 \\ 4 & 5 & 3 \\ 7 & 8 & 6 \end{bmatrix}$
		goal	

Program 5 - Vacuum Cleaner

Algorithm



Code

```
def vacuum_world():

    # Initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty

    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum: ") # User input for
location vacuum is placed

    status_input = input("Enter status of " + location_input + " (0 for Clean,
1 for Dirty): ") # User input if location is dirty or clean

    status_input_complement = input("Enter status of other room (0 for Clean,
1 for Dirty): ")

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")

        if status_input == '1':
            print("Location A is Dirty.")

            # Suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1 # Cost for suck
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                # If B is Dirty
                print("Location B is Dirty.")

                print("Moving right to Location B.")
                cost += 1 # Cost for moving right
                print("COST for moving RIGHT: " + str(cost))

                # Suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1 # Cost for suck
                print("COST for SUCK: " + str(cost))

                print("Location B has been Cleaned.")
```

```

else:

    print("No action needed; Location B is already clean.")

else:

    print("Location A is already clean.")

if status_input_complement == '1': # If B is Dirty

    print("Location B is Dirty.")

    print("Moving RIGHT to Location B.")

    cost += 1 # Cost for moving right

    print("COST for moving RIGHT: " + str(cost))

    # Suck the dirt and mark it as clean

    goal_state['B'] = '0'

    cost += 1 # Cost for suck

    print("COST for SUCK: " + str(cost))

    print("Location B has been Cleaned.")

else:

    print("No action needed; Location B is already clean.")


else: # Vacuum is placed in location B

    print("Vacuum is placed in Location B")

    if status_input == '1':

        print("Location B is Dirty.")

        # Suck the dirt and mark it as clean

        goal_state['B'] = '0'

        cost += 1 # Cost for suck

        print("COST for CLEANING B: " + str(cost))

        print("Location B has been Cleaned."


if status_input_complement == '1': # If A is Dirty

    print("Location A is Dirty.")

    print("Moving LEFT to Location A.")

    cost += 1 # Cost for moving left

    print("COST for moving LEFT: " + str(cost))

    # Suck the dirt and mark it as clean

    goal_state['A'] = '0'

    cost += 1 # Cost for suck

    print("COST for SUCK: " + str(cost))

    print("Location A has been Cleaned.")

else:

```

```

        print("No action needed; Location A is already clean.")

    else:

        print("Location B is already clean.")

        if status_input_complement == '1': # If A is Dirty
            print("Location A is Dirty.")

            print("Moving LEFT to Location A.")

            cost += 1 # Cost for moving left

            print("COST for moving LEFT: " + str(cost))

            # Suck the dirt and mark it as clean

            goal_state['A'] = '0'

            cost += 1 # Cost for suck

            print("COST for SUCK: " + str(cost))

            print("Location A has been Cleaned.")

        else:

            print("No action needed; Location A is already clean.")

# Done cleaning

print("GOAL STATE: ")

print(goal_state)

print("Performance Measurement: " + str(cost))

# Call the function to run the vacuum world simulation

vacuum_world()

name = "Varsha Prasanth"

usn = "1BM22CS321"

print(f"Name: {name}, USN: {usn}")

```

25

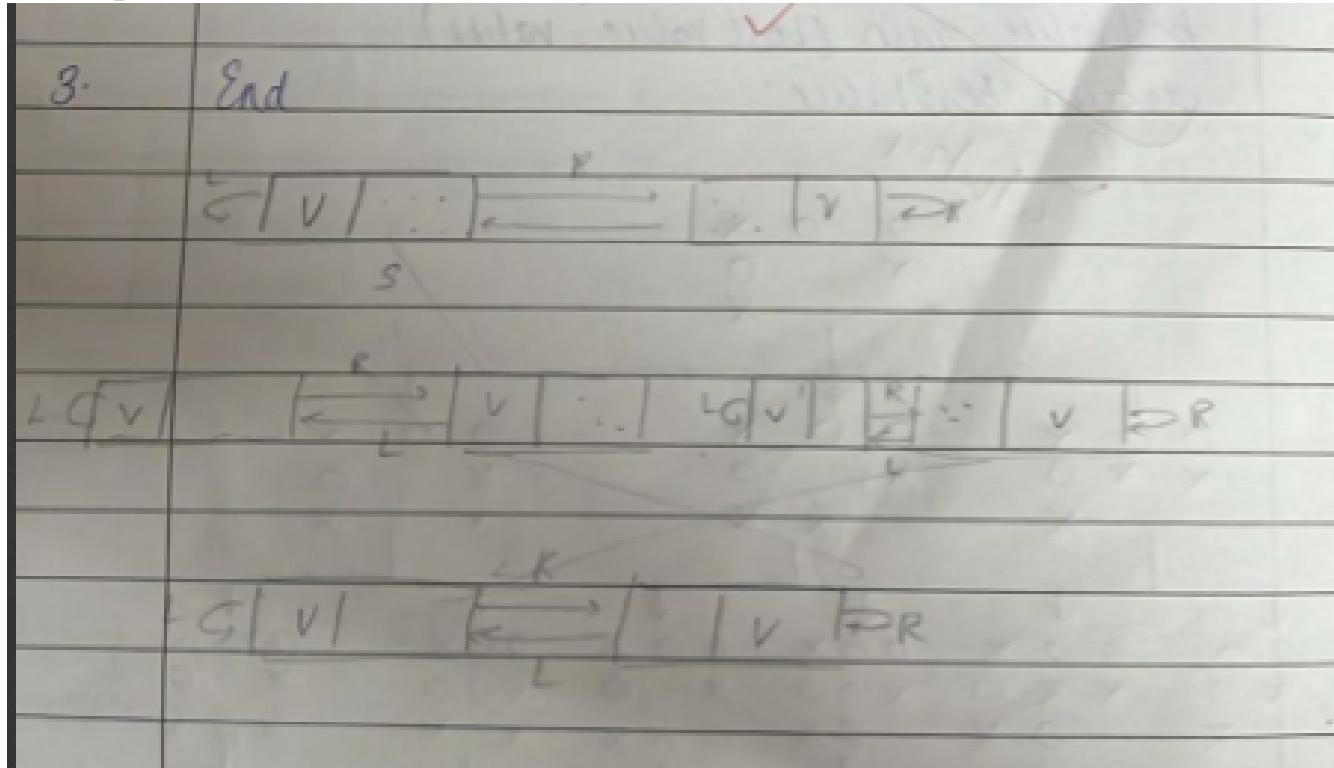
Output Snapshot

```

> Enter Location of Vacuum: A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of other room (0 for Clean, 1 for Dirty): 0
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A: 1
Location A has been Cleaned.
No action needed; Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
Name: Varsha Prasanth, USN: 1BM22CS321

```

State Space Tree



Program-06 Hill Climbing

Algorithm

Hill Climbing (N-Queens) Date _____
Page _____

function HILL-CLIMBING (problem) returns state that is local maximum
current ← MAKE-NODE (problem, INITIAL-STATE)
loop do
 neighbor ← highest-valued successor of node
 if neighbor.VALUE < current.VALUE then return current.STATE
 current ← neighbor

✓ 0 ~~Current~~ 2 0 (3, 1, 2, 0)
 |
 1 Q 1 3 2 0
 |
 2 Q 2 1 3 0
 |
 3 Q 0 1 2 3
 |
 4 Q 3 2 1 0
 |
 h-5 Q 3 0 2 1
 |
 h-6 Q 3 1 0 2
 |
 h-7 Q 0 2 1 3
 |
 h-8 Q 1 0 3 2
 |
 h-9 Q 2 3 0 1
 |
 h-10 Q 3 1 2 0
 |
 h-11 Q 0 3 1 2
 |
 h-12 Q 1 2 0 3
 |
 h-13 Q 2 0 3 1
 |
 h-14 Q 3 0 1 2
 |
 h-15 Q 0 1 2 3
 |
 h-16 Q 1 3 2 0
 |
 h-17 Q 2 1 3 0
 |
 h-18 Q 0 2 1 3
 |
 h-19 Q 3 2 0 1
 |
 h-20 Q 1 0 3 2
 |
 h-21 Q 2 3 0 1
 |
 h-22 Q 3 1 0 2
 |
 h-23 Q 0 2 1 3
 |
 h-24 Q 1 0 3 2
 |
 h-25 Q 2 3 0 1
 |
 h-26 Q 3 1 0 2
 |
 h-27 Q 0 2 1 3
 |
 h-28 Q 1 3 2 0
 |
 h-29 Q 2 1 3 0
 |
 h-30 Q 0 1 2 3
 |
 h-31 Q 3 2 0 1
 |
 h-32 Q 1 0 3 2
 |
 h-33 Q 2 3 0 1
 |
 h-34 Q 3 1 0 2
 |
 h-35 Q 0 2 1 3
 |
 h-36 Q 1 3 2 0
 |
 h-37 Q 2 1 3 0
 |
 h-38 Q 0 1 2 3
 |
 h-39 Q 3 2 0 1
 |
 h-40 Q 1 0 3 2
 |
 h-41 Q 2 3 0 1
 |
 h-42 Q 3 1 0 2
 |
 h-43 Q 0 2 1 3
 |
 h-44 Q 1 3 2 0
 |
 h-45 Q 2 1 3 0
 |
 h-46 Q 0 1 2 3
 |
 h-47 Q 3 2 0 1
 |
 h-48 Q 1 0 3 2
 |
 h-49 Q 2 3 0 1
 |
 h-50 Q 3 1 0 2
 |
 h-51 Q 0 2 1 3
 |
 h-52 Q 1 3 2 0
 |
 h-53 Q 2 1 3 0
 |
 h-54 Q 0 1 2 3
 |
 h-55 Q 3 2 0 1
 |
 h-56 Q 1 0 3 2
 |
 h-57 Q 2 3 0 1
 |
 h-58 Q 3 1 0 2
 |
 h-59 Q 0 2 1 3
 |
 h-60 Q 1 3 2 0
 |
 h-61 Q 2 1 3 0
 |
 h-62 Q 0 1 2 3
 |
 h-63 Q 3 2 0 1
 |
 h-64 Q 1 0 3 2
 |
 h-65 Q 2 3 0 1
 |
 h-66 Q 3 1 0 2
 |
 h-67 Q 0 2 1 3
 |
 h-68 Q 1 3 2 0
 |
 h-69 Q 2 1 3 0
 |
 h-70 Q 0 1 2 3
 |
 h-71 Q 3 2 0 1
 |
 h-72 Q 1 0 3 2
 |
 h-73 Q 2 3 0 1
 |
 h-74 Q 3 1 0 2
 |
 h-75 Q 0 2 1 3
 |
 h-76 Q 1 3 2 0
 |
 h-77 Q 2 1 3 0
 |
 h-78 Q 0 1 2 3
 |
 h-79 Q 3 2 0 1
 |
 h-80 Q 1 0 3 2
 |
 h-81 Q 2 3 0 1
 |
 h-82 Q 3 1 0 2
 |
 h-83 Q 0 2 1 3
 |
 h-84 Q 1 3 2 0
 |
 h-85 Q 2 1 3 0
 |
 h-86 Q 0 1 2 3
 |
 h-87 Q 3 2 0 1
 |
 h-88 Q 1 0 3 2
 |
 h-89 Q 2 3 0 1
 |
 h-90 Q 3 1 0 2
 |
 h-91 Q 0 2 1 3
 |
 h-92 Q 1 3 2 0
 |
 h-93 Q 2 1 3 0
 |
 h-94 Q 0 1 2 3
 |
 h-95 Q 3 2 0 1
 |
 h-96 Q 1 0 3 2
 |
 h-97 Q 2 3 0 1
 |
 h-98 Q 3 1 0 2
 |
 h-99 Q 0 2 1 3
 |
 h-100 Q 1 3 2 0
 |
 h-101 Q 2 1 3 0
 |
 h-102 Q 0 1 2 3
 |
 h-103 Q 3 2 0 1
 |
 h-104 Q 1 0 3 2
 |
 h-105 Q 2 3 0 1
 |
 h-106 Q 3 1 0 2
 |
 h-107 Q 0 2 1 3
 |
 h-108 Q 1 3 2 0
 |
 h-109 Q 2 1 3 0
 |
 h-110 Q 0 1 2 3
 |
 h-111 Q 3 2 0 1
 |
 h-112 Q 1 0 3 2
 |
 h-113 Q 2 3 0 1
 |
 h-114 Q 3 1 0 2
 |
 h-115 Q 0 2 1 3
 |
 h-116 Q 1 3 2 0
 |
 h-117 Q 2 1 3 0
 |
 h-118 Q 0 1 2 3
 |
 h-119 Q 3 2 0 1
 |
 h-120 Q 1 0 3 2
 |
 h-121 Q 2 3 0 1
 |
 h-122 Q 3 1 0 2
 |
 h-123 Q 0 2 1 3
 |
 h-124 Q 1 3 2 0
 |
 h-125 Q 2 1 3 0
 |
 h-126 Q 0 1 2 3
 |
 h-127 Q 3 2 0 1
 |
 h-128 Q 1 0 3 2
 |
 h-129 Q 2 3 0 1
 |
 h-130 Q 3 1 0 2
 |
 h-131 Q 0 2 1 3
 |
 h-132 Q 1 3 2 0
 |
 h-133 Q 2 1 3 0
 |
 h-134 Q 0 1 2 3
 |
 h-135 Q 3 2 0 1
 |
 h-136 Q 1 0 3 2
 |
 h-137 Q 2 3 0 1
 |
 h-138 Q 3 1 0 2
 |
 h-139 Q 0 2 1 3
 |
 h-140 Q 1 3 2 0
 |
 h-141 Q 2 1 3 0
 |
 h-142 Q 0 1 2 3
 |
 h-143 Q 3 2 0 1
 |
 h-144 Q 1 0 3 2
 |
 h-145 Q 2 3 0 1
 |
 h-146 Q 3 1 0 2
 |
 h-147 Q 0 2 1 3
 |
 h-148 Q 1 3 2 0
 |
 h-149 Q 2 1 3 0
 |
 h-150 Q 0 1 2 3
 |
 h-151 Q 3 2 0 1
 |
 h-152 Q 1 0 3 2
 |
 h-153 Q 2 3 0 1
 |
 h-154 Q 3 1 0 2
 |
 h-155 Q 0 2 1 3
 |
 h-156 Q 1 3 2 0
 |
 h-157 Q 2 1 3 0
 |
 h-158 Q 0 1 2 3
 |
 h-159 Q 3 2 0 1
 |
 h-160 Q 1 0 3 2
 |
 h-161 Q 2 3 0 1
 |
 h-162 Q 3 1 0 2
 |
 h-163 Q 0 2 1 3
 |
 h-164 Q 1 3 2 0
 |
 h-165 Q 2 1 3 0
 |
 h-166 Q 0 1 2 3
 |
 h-167 Q 3 2 0 1
 |
 h-168 Q 1 0 3 2
 |
 h-169 Q 2 3 0 1
 |
 h-170 Q 3 1 0 2
 |
 h-171 Q 0 2 1 3
 |
 h-172 Q 1 3 2 0
 |
 h-173 Q 2 1 3 0
 |
 h-174 Q 0 1 2 3
 |
 h-175 Q 3 2 0 1
 |
 h-176 Q 1 0 3 2
 |
 h-177 Q 2 3 0 1
 |
 h-178 Q 3 1 0 2
 |
 h-179 Q 0 2 1 3
 |
 h-180 Q 1 3 2 0
 |
 h-181 Q 2 1 3 0
 |
 h-182 Q 0 1 2 3
 |
 h-183 Q 3 2 0 1
 |
 h-184 Q 1 0 3 2
 |
 h-185 Q 2 3 0 1
 |
 h-186 Q 3 1 0 2
 |
 h-187 Q 0 2 1 3
 |
 h-188 Q 1 3 2 0
 |
 h-189 Q 2 1 3 0
 |
 h-190 Q 0 1 2 3
 |
 h-191 Q 3 2 0 1
 |
 h-192 Q 1 0 3 2
 |
 h-193 Q 2 3 0 1
 |
 h-194 Q 3 1 0 2
 |
 h-195 Q 0 2 1 3
 |
 h-196 Q 1 3 2 0
 |
 h-197 Q 2 1 3 0
 |
 h-198 Q 0 1 2 3
 |
 h-199 Q 3 2 0 1
 |
 h-200 Q 1 0 3 2
 |
 h-201 Q 2 3 0 1
 |
 h-202 Q 3 1 0 2
 |
 h-203 Q 0 2 1 3
 |
 h-204 Q 1 3 2 0
 |
 h-205 Q 2 1 3 0
 |
 h-206 Q 0 1 2 3
 |
 h-207 Q 3 2 0 1
 |
 h-208 Q 1 0 3 2
 |
 h-209 Q 2 3 0 1
 |
 h-210 Q 3 1 0 2
 |
 h-211 Q 0 2 1 3
 |
 h-212 Q 1 3 2 0
 |
 h-213 Q 2 1 3 0
 |
 h-214 Q 0 1 2 3
 |
 h-215 Q 3 2 0 1
 |
 h-216 Q 1 0 3 2
 |
 h-217 Q 2 3 0 1
 |
 h-218 Q 3 1 0 2
 |
 h-219 Q 0 2 1 3
 |
 h-220 Q 1 3 2 0
 |
 h-221 Q 2 1 3 0
 |
 h-222 Q 0 1 2 3
 |
 h-223 Q 3 2 0 1
 |
 h-224 Q 1 0 3 2
 |
 h-225 Q 2 3 0 1
 |
 h-226 Q 3 1 0 2
 |
 h-227 Q 0 2 1 3
 |
 h-228 Q 1 3 2 0
 |
 h-229 Q 2 1 3 0
 |
 h-230 Q 0 1 2 3
 |
 h-231 Q 3 2 0 1
 |
 h-232 Q 1 0 3 2
 |
 h-233 Q 2 3 0 1
 |
 h-234 Q 3 1 0 2
 |
 h-235 Q 0 2 1 3
 |
 h-236 Q 1 3 2 0
 |
 h-237 Q 2 1 3 0
 |
 h-238 Q 0 1 2 3
 |
 h-239 Q 3 2 0 1
 |
 h-240 Q 1 0 3 2
 |
 h-241 Q 2 3 0 1
 |
 h-242 Q 3 1 0 2
 |
 h-243 Q 0 2 1 3
 |
 h-244 Q 1 3 2 0
 |
 h-245 Q 2 1 3 0
 |
 h-246 Q 0 1 2 3
 |
 h-247 Q 3 2 0 1
 |
 h-248 Q 1 0 3 2
 |
 h-249 Q 2 3 0 1
 |
 h-250 Q 3 1 0 2
 |
 h-251 Q 0 2 1 3
 |
 h-252 Q 1 3 2 0
 |
 h-253 Q 2 1 3 0
 |
 h-254 Q 0 1 2 3
 |
 h-255 Q 3 2 0 1
 |
 h-256 Q 1 0 3 2
 |
 h-257 Q 2 3 0 1
 |
 h-258 Q 3 1 0 2
 |
 h-259 Q 0 2 1 3
 |
 h-260 Q 1 3 2 0
 |
 h-261 Q 2 1 3 0
 |
 h-262 Q 0 1 2 3
 |
 h-263 Q 3 2 0 1
 |
 h-264 Q 1 0 3 2
 |
 h-265 Q 2 3 0 1
 |
 h-266 Q 3 1 0 2
 |
 h-267 Q 0 2 1 3
 |
 h-268 Q 1 3 2 0
 |
 h-269 Q 2 1 3 0
 |
 h-270 Q 0 1 2 3
 |
 h-271 Q 3 2 0 1
 |
 h-272 Q 1 0 3 2
 |
 h-273 Q 2 3 0 1
 |
 h-274 Q 3 1 0 2
 |
 h-275 Q 0 2 1 3
 |
 h-276 Q 1 3 2 0
 |
 h-277 Q 2 1 3 0
 |
 h-278 Q 0 1 2 3
 |
 h-279 Q 3 2 0 1
 |
 h-280 Q 1 0 3 2
 |
 h-281 Q 2 3 0 1
 |
 h-282 Q 3 1 0 2
 |
 h-283 Q 0 2 1 3
 |
 h-284 Q 1 3 2 0
 |
 h-285 Q 2 1 3 0
 |
 h-286 Q 0 1 2 3
 |
 h-287 Q 3 2 0 1
 |
 h-288 Q 1 0 3 2
 |
 h-289 Q 2 3 0 1
 |
 h-290 Q 3 1 0 2
 |
 h-291 Q 0 2 1 3
 |
 h-292 Q 1 3 2 0
 |
 h-293 Q 2 1 3 0
 |
 h-294 Q 0 1 2 3
 |
 h-295 Q 3 2 0 1
 |
 h-296 Q 1 0 3 2
 |
 h-297 Q 2 3 0 1
 |
 h-298 Q 3 1 0 2
 |
 h-299 Q 0 2 1 3
 |
 h-300 Q 1 3 2 0
 |
 h-301 Q 2 1 3 0
 |
 h-302 Q 0 1 2 3
 |
 h-303 Q 3 2 0 1
 |
 h-304 Q 1 0 3 2
 |
 h-305 Q 2 3 0 1
 |
 h-306 Q 3 1 0 2
 |
 h-307 Q 0 2 1 3
 |
 h-308 Q 1 3 2 0
 |
 h-309 Q 2 1 3 0
 |
 h-310 Q 0 1 2 3
 |
 h-311 Q 3 2 0 1
 |
 h-312 Q 1 0 3 2
 |
 h-313 Q 2 3 0 1
 |
 h-314 Q 3 1 0 2
 |
 h-315 Q 0 2 1 3
 |
 h-316 Q 1 3 2 0
 |
 h-317 Q 2 1 3 0
 |
 h-318 Q 0 1 2 3
 |
 h-319 Q 3 2 0 1
 |
 h-320 Q 1 0 3 2
 |
 h-321 Q 2 3 0 1
 |
 h-322 Q 3 1 0 2
 |
 h-323 Q 0 2 1 3
 |
 h-324 Q 1 3 2 0
 |
 h-325 Q 2 1 3 0
 |
 h-326 Q 0 1 2 3
 |
 h-327 Q 3 2 0 1
 |
 h-328 Q 1 0 3 2
 |
 h-329 Q 2 3 0 1
 |
 h-330 Q 3 1 0 2
 |
 h-331 Q 0 2 1 3
 |
 h-332 Q 1 3 2 0
 |
 h-333 Q 2 1 3 0
 |
 h-334 Q 0 1 2 3
 |
 h-335 Q 3 2 0 1
 |
 h-336 Q 1 0 3 2
 |
 h-337 Q 2 3 0 1
 |
 h-338 Q 3 1 0 2
 |
 h-339 Q 0 2 1 3
 |
 h-340 Q 1 3 2 0
 |
 h-341 Q 2 1 3 0
 |
 h-342 Q 0 1 2 3
 |
 h-343 Q 3 2 0 1
 |
 h-344 Q 1 0 3 2
 |
 h-345 Q 2 3 0 1
 |
 h-346 Q 3 1 0 2
 |
 h-347 Q 0 2 1 3
 |
 h-348 Q 1 3 2 0
 |
 h-349 Q 2 1 3 0
 |
 h-350 Q 0 1 2 3
 |
 h-351 Q 3 2 0 1
 |
 h-352 Q 1 0 3 2
 |
 h-353 Q 2 3 0 1
 |
 h-354 Q 3 1 0 2
 |
 h-355 Q 0 2 1 3
 |
 h-356 Q 1 3 2 0
 |
 h-357 Q 2 1 3 0
 |
 h-358 Q 0 1 2 3
 |
 h-359 Q 3 2 0 1
 |
 h-360 Q 1 0 3 2
 |
 h-361 Q 2 3 0 1
 |
 h-362 Q 3 1 0 2
 |
 h-363 Q 0 2 1 3
 |

Code

```
import random

def print_board(board, n):
    """Prints the current state of the board."""
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += " Q "
            else:
                line += " . "
        print(line)
    print()

def calculate_conflicts(board, n):
    """Calculates the number of conflicts (attacks) between queens."""
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same row or diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def get_best_neighbor(board, n):
    """
    Finds the best neighboring board with the fewest conflicts.
    Returns the best board and its conflict count.
    """
    current_conflicts = calculate_conflicts(board, n)
    best_board = board[:]
    best_conflicts = current_conflicts
    neighbors = []

    for col in range(n):
        original_row = board[col]
```

```

        for row in range(n):
            if row == original_row:
                continue
            # Move queen to a new row and calculate conflicts
            board[col] = row
            new_conflicts = calculate_conflicts(board, n)
            neighbors.append((board[:], new_conflicts))

        # Restore the original row before moving to the next column
        board[col] = original_row

    # Sort neighbors by the number of conflicts (ascending)
    neighbors.sort(key=lambda x: x[1])
    if neighbors:
        best_neighbor = neighbors[0]
        if best_neighbor[1] < best_conflicts:
            return best_neighbor
    return board, current_conflicts

def hill_climbing_with_restarts(n, initial_board, max_restarts=100):
    """
    Performs Hill Climbing with random restarts to solve the N-Queens
    problem.

    Returns the final board configuration and its conflict count.
    """
    current_board = initial_board[:]
    current_conflicts = calculate_conflicts(current_board, n)

    print("Initial board:")
    print_board(current_board, n)
    print(f"Initial conflicts: {current_conflicts}\n")

    steps = 0
    restarts = 0

    while current_conflicts > 0 and restarts < max_restarts:
        new_board, new_conflicts = get_best_neighbor(current_board, n)

        steps += 1

```

```

print(f"Step {steps}:")
print_board(new_board, n)
print(f"Conflicts: {new_conflicts}\n")

if new_conflicts < current_conflicts:
    current_board = new_board
    current_conflicts = new_conflicts
else:
    # If no better neighbor is found, perform a random restart
    restarts += 1
    print(f"Restarting... (Restart number {restarts})\n")
    current_board = [random.randint(0, n-1) for _ in range(n)]
    current_conflicts = calculate_conflicts(current_board, n)
    print("New initial board:")
    print_board(current_board, n)
    print(f"Conflicts: {current_conflicts}\n")

return current_board, current_conflicts

# Main function
def main():
    n = 4
    print("Enter the initial positions of queens (row numbers from 0 to 3
for each column):")
    initial_board = []
    for i in range(n):
        while True:
            try:
                row = int(input(f"Column {i}: "))
                if 0 <= row < n:
                    initial_board.append(row)
                    break
                else:
                    print(f"Please enter a number between 0 and {n-1}.")
            except ValueError:
                print("Invalid input. Please enter an integer.")

solution, conflicts = hill_climbing_with_restarts(n, initial_board)

```

```

print("Final solution:")
print_board(solution, n)
if conflicts == 0:
    print("A solution was found with no conflicts!")
else:
    print(f"No solution was found after {100} restarts. Final number of
conflicts: {conflicts}")

if __name__ == "__main__":
    main()

name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")

```

OUTPUT

```

Enter the initial positions of queens (row numbers from 0 to 3 for each column):
Column 0: 3
Column 1: 1
Column 2: 2
Column 3: 0
Initial board:
. . . Q
. Q .
. . Q .
Q . .

Initial conflicts: 2

Step 6:
Q Q -
- - - Q
- - -
- . Q -
Conflicts: 1

Step 7:
- Q -
- - - Q
Q - -
- . Q -
Conflicts: 0

Final solution:
- Q -
- . - Q
Q - -
- . Q -
A solution was found with no conflicts!
Name: Varsha Prasanth, USN: 1BM22CS321

```

	\checkmark Encuentro	α	$(3, 1, 2, 0)$
1	α		$1 \ 3 \ 2 \ 0$
2		α	$2 \ 1 \ 3 \ 0$
3	α		$0 \ 1 \ 2 \ 3$
			$5 \ 2 \ 3 \ 0$
$b=3$		α	$3 \ 0 \ 2 \ 3$
	α		$3 \ 1 \ 0 \ 2$
	α		
$b=3$		α	α
	α		α
$b=6$	α		α
	α		α
	α		α

\rightarrow	$4 \ 3 \ 2 \ 0$	α	α	α	α
	$3 \ 1 \ 2 \ 0$	α	α	α	α
	$2 \ 3 \ 1 \ 0$	α	α	α	α
	$0 \ 3 \ 2 \ 1$	α	α	α	α
	$1 \ 0 \ 2 \ 3$	α	α	α	α
	$1 \ 3 \ 0 \ 2$	α	α	α	α
$b=10$	$b=2$	α	α	α	α
	α	α	α	α	α
$b=3$	α	α	α	α	α
	α	α	α	α	α
$b=21$	α	α	α	α	α
$b=11$	α	α	α	α	α
$b=17$	α	α	α	α	α
$b=1$	α	α	α	α	α

Program-07 Simulated Annealing

Algorithm

Simulated Annealing

Date _____
Page _____

```
current ← initial state
T ← a large positive value
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
```

Q.P.

Enter the size of the board (N) : 4
Enter initial temperature : 1000
Enter cooling rate : 0.99
Enter maximum number of iterations : 200
Initial board

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

Initial conflict ^{15/12}

Final solution

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

Code

```
import random
import math

def print_board(board, n):
    """Prints the current state of the board."""
    for row in range(n):
        line = ""
        for col in range(n):
            if board[col] == row:
                line += " Q " # Queen is represented by "Q"
            else:
                line += " . " # Empty space represented by "."
        print(line)
    print()

def calculate_conflicts(board, n):
    """Calculates the number of conflicts (attacks) between queens."""
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are in the same row or diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def simulated_annealing(n, initial_temp=1000, cooling_rate=0.995,
max_iterations=10000):
    """Simulated Annealing algorithm to solve N-Queens."""
    # Initial random board configuration (one queen in each column)
    board = [random.randint(0, n - 1) for _ in range(n)]
    current_conflicts = calculate_conflicts(board, n)
    temperature = initial_temp
    iteration = 0

    print("Initial board:")
    print_board(board, n)
    print(f"Initial conflicts: {current_conflicts}\n")
```

```

while current_conflicts > 0 and iteration < max_iterations:
    # Generate a neighboring state by moving a queen to another row in
    its column

    col = random.randint(0, n - 1)
    original_row = board[col]
    new_row = random.randint(0, n - 1)
    while new_row == original_row:
        new_row = random.randint(0, n - 1) # Ensure we are moving the
    queen to a new row
    board[col] = new_row

    # Calculate the number of conflicts in the new configuration
    new_conflicts = calculate_conflicts(board, n)

    # If the new state has fewer conflicts, accept it.
    # If the new state has more conflicts, accept it with a certain
    probability.

    if new_conflicts < current_conflicts or random.random() <
math.exp((current_conflicts - new_conflicts) / temperature):
        current_conflicts = new_conflicts
    else:
        # If no improvement, revert the move
        board[col] = original_row

    # Reduce the temperature according to the cooling schedule
    temperature *= cooling_rate

    iteration += 1

    # Display progress
    if iteration % 1000 == 0:
        print(f"Iteration {iteration}: Conflicts = {current_conflicts},"
Temperature = {temperature}")
        print_board(board, n)

return board, current_conflicts

def main():
    # Input dynamic parameters
    print("Welcome to the N-Queens Problem Solver using Simulated
Annealing!")
    n = int(input("Enter the size of the board (N): "))

```

```

initial_temp = float(input("Enter the initial temperature (e.g., 1000):
"))
cooling_rate = float(input("Enter the cooling rate (e.g., 0.995): "))
max_iterations = int(input("Enter the maximum number of iterations
(e.g., 10000): "))

solution, conflicts = simulated_annealing(n, initial_temp,
cooling_rate, max_iterations)

print("Final solution:")
print_board(solution, n)
if conflicts == 0:
    print("A solution was found with no conflicts!")
else:
    print(f"No solution was found after {max_iterations} iterations.
Final number of conflicts: {conflicts}")

if __name__ == "__main__":
    main()

name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")

```

OUTPUT

```

→ Welcome to the N-Queens Problem Solver using Simulated Annealing!
Enter the size of the board (N): 4
Enter the initial temperature (e.g., 1000): 1000
Enter the cooling rate (e.g., 0.995): 0.995
Enter the maximum number of iterations (e.g., 10000): 200
Initial board:
. . .
. . Q
Q . Q .
. Q .

Initial conflicts: 5

Final solution:
. Q .
. . Q
Q . .
. . Q .

A solution was found with no conflicts!
Name: Varsha Prasanth, USN: 1BM22CS321

```

Program-08 KnowledgeBase - Resolution

Algorithm

~~1/2~~ Creating KB using Propositional Logic
and proving query using resolution

Date _____
Page _____

Initialize a knowledge-base with propositional logic statements

Input Query:

Convert KB and query into CNF

Add -query to CNF clauses

while True:

Select two clauses from CNF clauses

Resolve the clauses to produce new clauses

If new clause is empty:
 print("Query is proven using resolution")
 break

If new clause is already in CNF clauses:
 Add new clause to CNF clauses

If new clause can be generated:
 print("Query can't be proven using resolution")
 break

For KB: ~~[~A, ~B, ~A-B-C, C-D]~~

Query D

Query is proven using resolution.

$\neg R$ $\neg C \vee R$
 $\neg C$ $\neg H \vee R$
 H $\neg H \vee S$
 S $\neg S$
 $\{S\}$

Code

```
# Function to negate a literal
def negate(literal):
    """Negate a literal."""
    if isinstance(literal, tuple) and literal[0] == "not":
        # If the literal is already negated, return the positive form
        return literal[1]
    else:
        # Otherwise, return the negated form
        return ("not", literal)

# Function to resolve two clauses
def resolve(clause1, clause2):
    """Return the resolvent of two clauses."""
    resolvents = set()
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == negate(literal2):
                resolvent = (clause1 - {literal1}) | (clause2 - {literal2})
                print(f" Resolving literal: {literal1} with {literal2}")
                print(f" Resulting Resolvent: {resolvent}")
                resolvents.add(frozenset(resolvent))
    return resolvents

# Function to perform resolution on the KB and query with detailed output
def resolution_algorithm(KB, query):
    """Perform the resolution algorithm to check if the query can be proven."""
    print("\n--- Step-by-Step Resolution Process ---")
    # Add the negation of the query to the knowledge base
    negated_query = negate(query)
    KB.append(frozenset([negated_query]))
    print(f"Negated Query Added to KB: {negated_query}")

    # Initialize the set of clauses to process
    clauses = set(KB)

    step = 1
    while True:
        new_clauses = set()
        print(f"\nStep {step}: Resolving Clauses")
        for c1 in clauses:
```

```

        for c2 in clauses:
            if c1 != c2:
                print(f" Resolving clauses: {c1} and {c2}")
                resolvent = resolve(c1, c2)

                for res in resolvent:
                    if frozenset([]) in resolvent:
                        print("\nEmpty clause derived! The query is
provable.")
                        return True # Empty clause found, contradiction,
query is provable

                    new_clauses.add(res)

            if new_clauses.issubset(clauses):
                print("\nNo new clauses can be derived. The query is not
provable.")
                return False # No new clauses, query is not provable

        clauses.update(new_clauses)
        step += 1

# Knowledge Base (KB) from the image facts
KB = [
    frozenset([('not', "food(x)'), ('likes', "John", "x"))], # 1
    frozenset([('food', "Apple")]), # 2
    frozenset([('food', "vegetables")]), # 3
    frozenset([('not', "eats(y, z)'), ("killed", "y"), ("food", "z")]), # 4
    frozenset([('eats', "Anil", "Peanuts")]), # 5
    frozenset([('alive', "Anil")]), # 6
    frozenset([('not', "eats(Anil, w)'), ("eats", "Harry", "w")]), # 7
    frozenset([('killed', "g"), ("alive", "g")]), # 8
    frozenset([('not', "alive(k)'), ("not", "killed(k)")]), # 9
    frozenset([('likes', "John", "Peanuts")]) # 10
]

# Query to prove
query = ("likes", "John", "Peanuts")

# Perform resolution to check if the query is provable
result = resolution_algorithm(KB, query)
if result:
    print("\nQuery is provable.")

```

```

else:
    print("\nQuery is not provable.")

name = "Varsha Prasanth"
usn = "1BM22CS321"

print(f"Name: {name}, USN: {usn}")

```

Output Snapshot

```

Resolving clauses: frozenset([('eats', 'Anil', 'Peanuts'))) and frozenset([('not', 'food(x)'), ('likes', 'John', 'x')))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('alive', 'Anil'))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('eats', 'Anil', 'Peanuts'))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('food', 'Apple'))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('not', ('likes', 'John', 'Peanuts')))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('eats', 'Harry', 'w'), ('not', 'eats(Anil, w')))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('food', 'vegetables'))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('not', 'killed(k)'), ('not', 'alive(k')))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y'))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('likes', 'John', 'Peanuts'))))
Resolving clauses: frozenset([('killed', 'g'), ('alive', 'g')))) and frozenset([('not', 'food(x)'), ('likes', 'John', 'x'))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('alive', 'Anil'))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('eats', 'Anil', 'Peanuts'))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('killed', 'g'), ('alive', 'g'))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('not', ('likes', 'John', 'Peanuts')))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('eats', 'Harry', 'w'), ('not', 'eats(Anil, w')))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('food', 'vegetables'))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('not', 'killed(k)'), ('not', 'alive(k')))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y'))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('likes', 'John', 'Peanuts'))))
Resolving clauses: frozenset([('food', 'Apple')))) and frozenset([('not', 'food(x)'), ('likes', 'John', 'x'))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('alive', 'Anil'))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('eats', 'Anil', 'Peanuts'))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('killed', 'g'), ('alive', 'g'))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('not', 'food', 'Apple'))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('eats', 'Harry', 'w'), ('not', 'eats(Anil, w')))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('food', 'vegetables'))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('not', 'killed(k)'), ('not', 'alive(k')))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y'))))
Resolving clauses: frozenset([('not', ('likes', 'John', 'Peanuts')))) and frozenset([('likes', 'John', 'Peanuts'))))
Resolving literal: ('not', ('likes', 'John', 'Peanuts')) with ('likes', 'John', 'Peanuts')

Resulting Resolvent: frozenset()

```

Empty clause derived! The query is provable.

Query is provable.

Name: Varsha Prasanth, USN: 1BM22CS321

Program-09 Unification in first order logic

Algorithm

Date _____
Page _____

32/11/24 FOL Unification

Unify (ψ_1, ψ_2)

Step 1 : If ψ_1 or ψ_2 is a variable or constant, Then
(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 failure,
 b. Else return $\{(\psi_1 / \psi_2)\}$
 d) Else return FAILURE

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

Step 3 : If ψ_1 and ψ_2 have a different no. of arguments, then return

Step 4 : Set substitution set (SUBST) to NIL

Step 5 : For $i=1$ to number of elements in ψ_1
(a) call unify function with i^{th} element of ψ_2 and ψ_1
 put the result into S
(b) If $S = \text{failure}$ then return FAILURE
(c) If $S \neq \text{NIL}$ then do,
 a. apply S to the remainder of both ψ_1 and ψ_2
 b. ~~Subset = append (S, subset)~~

Step 6 : return ~~Subset~~.

~~OR~~ Please enter the ~~execute~~ expression in list formal

Example : $["\text{Eats}", "X", "\text{Apple}"]$

Enter Expression 1 : $["\text{Eats}", "x", "\text{Apple}"]$
2 : $["\text{Eats}", "Orange", "y"]$

Unification successful
 $\{x = \text{orange}, y = \text{Apple}\}$

~~32/11/24~~

Code

```
import ast

from typing import Union, List, Dict, Tuple

from collections import deque

# Define Term Classes

class Term:

    def substitute(self, subs: Dict[str, 'Term']) -> 'Term':

        raise NotImplementedError

    def occurs(self, var: 'Variable') -> bool:

        raise NotImplementedError
```

```
def __eq__(self, other):
    raise NotImplementedError

def __str__(self):
    raise NotImplementedError

class Variable(Term):
    def __init__(self, name: str):
        self.name = name

    def substitute(self, subs: Dict[str, Term]) -> Term:
        if self.name in subs:
```

```
        return subs[self.name].substitute(subs)

    return self

def occurs(self, var: 'Variable') -> bool:
    return self.name == var.name

def __eq__(self, other):
    return isinstance(other, Variable) and self.name == other.name

def __str__(self):
    return self.name

class Constant(Term):
```

```
def __init__(self, name: str):
    self.name = name

def substitute(self, subs: Dict[str, Term]) -> Term:
    if self.name in subs:
        return subs[self.name]
    else:
        return self

def occurs(self, var: 'Variable') -> bool:
    if self.name == var:
        return True
    else:
        return False

def __eq__(self, other):
    if not isinstance(other, Constant):
        return False
    else:
        return self.name == other.name

def __str__(self):
```

```

    return self.name

class Function(Term):

    def __init__(self, name: str, args: List[Term]):
        self.name = name
        self.args = args

    def substitute(self, subs: Dict[str, Term]) -> Term:
        substituted_args = [arg.substitute(subs) for arg in self.args]
        return Function(self.name, substituted_args)

    def occurs(self, var: 'Variable') -> bool:
        return any(arg.occurs(var) for arg in self.args)

```

```
def __eq__(self, other):  
  
    return (  
  
        isinstance(other, Function) and  
  
        self.name == other.name and  
  
        len(self.args) == len(other.args) and  
  
        all(a == b for a, b in zip(self.args, other.args))  
  
)  
  
def __str__(self):  
  
    return f'{self.name}({', '.join(str(arg) for arg in self.args)})'  
  
def parse_expression(expr: List) -> Term:
```

```
if not isinstance(expr, list) or not expr:  
  
    raise ValueError("Expression must be a non-empty list")  
  
func_name = expr[0]  
  
args = expr[1:]  
  
parsed_args = []  
  
for arg in args:  
  
    if isinstance(arg, list):  
  
        parsed_args.append(parse_expression(arg))  
  
    elif isinstance(arg, str):  
  
        if arg[0].islower():  
  
            parsed_args.append(Variable(arg))
```

```

        elif arg[0].isupper():

            parsed_args.append(Constant(arg))

    else:

        raise ValueError(f"Invalid argument format: {arg}")

else:

    raise ValueError(f"Unsupported argument type: {arg}")

return Function(func_name, parsed_args)

def unify_terms(term1: Term, term2: Term) -> Union[Dict[str, Term], str]:
    # Initialize substitution set S
    S: Dict[str, Term] = {}

```

```

# Initialize the equation list

equations: deque[Tuple[Term, Term]] = deque()

equations.append((term1, term2))

while equations:

    s, t = equations.popleft()

    s = s.substitute(S)

    t = t.substitute(S)

    if s == t:
        continue

    elif isinstance(s, Variable):

        if t.occurs(s):

```

```

        return "FAILURE"

S[s.name] = t

# Apply the substitution to existing substitutions

for var in S:

    S[var] = S[var].substitute({s.name: t})

elif isinstance(t, Variable):

    if s.occurs(t):

        return "FAILURE"

S[t.name] = s

for var in S:

    S[var] = S[var].substitute({t.name: s})

elif isinstance(s, Function) and isinstance(t, Function):

    if s.name != t.name or len(s.args) != len(t.args):

```

```
        return "FAILURE"

    for s_arg, t_arg in zip(s.args, t.args):
        equations.append((s_arg, t_arg))

    elif isinstance(s, Constant) and isinstance(t, Constant):
        if s.name != t.name:
            return "FAILURE"

    # else, they are equal; continue

    else:
        return "FAILURE"

    return s

def format_substitution(S: Dict[str, Term]) -> str:
```

```
if not S:

    return "{}"

return "{ " + ", ".join(f"{{var}} = {term}" for var, term in S.items()) + "}"
}

def unify(expression1: List, expression2: List) -> Union[Dict[str, Term], str]:
    try:
        term1 = parse_expression(expression1)

        term2 = parse_expression(expression2)

    except ValueError as e:
        return f"FAILURE: {e}"

    result = unify_terms(term1, term2)
```

```
    return result

def main():

    print("==== Unification Algorithm ====\n")

    print("Please enter the expressions in list format.")

    print("Example: [\"Eats\", \"x\", \"Apple\"]\n")

    try:

        expr1_input = input("Enter Expression 1: ")

        expression1 = ast.literal_eval(expr1_input)

        if not isinstance(expression1, list):

            raise ValueError("Expression must be a list.")
```

```
expr2_input = input("Enter Expression 2: ")

expression2 = ast.literal_eval(expr2_input)

if not isinstance(expression2, list):
    raise ValueError("Expression must be a list.")

except (SyntaxError, ValueError) as e:
    print(f"Invalid input format: {e}")

return

result = unify(expression1, expression2)

if isinstance(result, str):
    print("\nUnification Result:")
```

```
    print(result)

else:

    print("\nUnification Successful:")

print(format_substitution(result))

if __name__ == "__main__":
    main()

name = "Varsha Prasanth"

usn = "1BM22CS321"

print(f"Name: {name}, USN: {usn}")
```

Output Snapshot

```
→ === Unification Algorithm ===  
Please enter the expressions in list format.  
Example: ["Eats", "x", "Apple"]  
Enter Expression 1: ["Eats", "x", "Apple"]  
Enter Expression 2: ["Eats", "orange", "y"]  
Unification Successful:  
{ x = orange, y = Apple }  
Name: Varsha Prasanth, USN: 1BM22CS321
```

STATE SPACE TREE

Enter Expression 1: ["Eats", "x", "Apple"]
2: ["Eats", "orange", "y"]

Unification successful
{ x = orange , y = Apple }

Program-10 First Order Logic to Conjunctive Normal Form

Algorithm

29/11/24 Converting FOL \rightarrow CNF	
	Date 29/11/24 Page _____
	Input first order logic statement :
	Eliminate implication : Replace $(A \rightarrow B)$ with $(\neg A \vee B)$
	Move \neg (negation) inwards using De Morgan's Law
	Standardize variables : Ensure each quantifier has a unique variable
	Move quantifiers to the front (prenex form)
	Skolemize : Eliminate existential quantifiers by introducing Skolem variables or functions.
	Drop universal quantifiers
	Distribute \vee over \wedge
	Output CNF clause
Q.F.	Original statement $(A \wedge B) \rightarrow C$ CNF : $\neg A \vee \neg B \vee C$

Code

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def eliminate_implications(expr):
    """Eliminate implications and equivalences."""
    if isinstance(expr, Implies):
        return Or(Not(eliminate_implications(expr.args[0])), 
                  eliminate_implications(expr.args[1]))
    elif isinstance(expr, Equivalent):
        left = eliminate_implications(expr.args[0])
        right = eliminate_implications(expr.args[1])
        return And(Or(Not(left), right), Or(Not(right), left))
    elif expr.is_Atom:
        return expr
    else:
        return expr.func(*[eliminate_implications(arg) for arg in expr.args])

def push_negations(expr):
    """Push negations inward using De Morgan's laws."""
    if expr.is_Not:
        arg = expr.args[0]
        if isinstance(arg, And):
            return Or(*[push_negations(Not(sub_arg)) for sub_arg in arg.args])
        elif isinstance(arg, Or):
            return And(*[push_negations(Not(sub_arg)) for sub_arg in
                        arg.args])
        elif isinstance(arg, Not):
            return push_negations(arg.args[0])
        else:
            return Not(push_negations(arg))
    elif expr.is_Atom:
        return expr
    else:
        return expr.func(*[push_negations(arg) for arg in expr.args])
```

```

def distribute_ands(expr):
    """Distribute AND over OR to obtain CNF."""
    if isinstance(expr, Or):
        and_args = [arg for arg in expr.args if isinstance(arg, And)]
        if and_args:
            first_and = and_args[0]
            rest = [arg for arg in expr.args if arg != first_and]
            return And(*[distribute_ands(Or(arg, *rest)) for arg in
first_and.args])
    elif isinstance(expr, And) or expr.is_Atom or expr.is_Not:
        return expr
    return expr.func(*[distribute_ands(arg) for arg in expr.args])

def to_cnf(expr):
    """Convert the given logical expression to CNF."""
    expr = eliminate_implications(expr)
    expr = push_negations(expr)
    expr = distribute_ands(expr)
    return expr

# Example usage:
A, B, C = symbols('A B C')

fol_expr = Implies(A, Or(B, Not(C))) # Example FOL expression
cnf_expr = to_cnf(fol_expr)

print("FOL expression:", fol_expr)
print("CNF expression:", cnf_expr)
name = "Varsha Prasanth"
usn = "1BM22CS321"

print(f"Name: {name}, USN: {usn}")

```

Output Snapshot

→ FOL expression: $\text{Implies}(A, B \mid \neg C)$
CNF expression: $B \mid \neg A \mid \neg C$
Name: Varsha Prasanth, USN: 1BM22CS321

Program-11 Forward Reasoning

Algorithm

Forward Chaining

Date _____
Page _____

```
function FOL-FC-ASK (KB, α) return 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 sentences inferred on each
                  iteration
repeat until new is empty
    new ← {}
    for each rule in KB do
        (p, 1...1 pn ⇒ q) ← STANDARDIZE-VARIABLES (rule)
        for each θ such that SUBST(θ, p, 1...1 pn) = SUBST(θ, p', 1...
            ...1 p'n)
            for some p', ..., p'n 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 not fail then return φ
                add new, KB to new
    return false

Processing fact: Criminal (Robert)
Found query match: Criminal (Robert)

Final result: True

Exercise q's: Answers
a. Occupation (Emily, Surgeon) ∨ occupation (Emily, Lawyer)
b. occupation (Joe, Actor) 1 → (o * Actor 1 occupation (Joe, o))
c. ∀x: Occupation (x, Surgeon) → occupation (x, Doctor)
```

Code

```
def fol_fc_ask(KB, query):
    """
    Implements the Forward Chaining algorithm.

    :param KB: The knowledge base, a list of first-order definite clauses.
    :param query: The query, an atomic sentence.
    :return: True if the query can be proven, otherwise False.
    """

    inferred = set() # Keep track of inferred facts
    agenda = [fact for fact in KB if not fact.get('premises')] # Initial facts
    rules = [rule for rule in KB if rule.get('premises')] # Rules with premises

    # Debugging output: Initial agenda and inferred facts
    print(f"Initial agenda: {[fact['conclusion'] for fact in agenda]}")
    print(f"Initial inferred: {inferred}")

    while agenda:
        fact = agenda.pop(0)
        print(f"\nProcessing fact: {fact['conclusion']}")

        # Check if this fact matches the query
        if fact['conclusion'] == query:
            print(f"Found query match: {fact['conclusion']}") # Rule premise satisfied
            return True

        # Infer new facts if this fact hasn't been inferred before
        if fact['conclusion'] not in inferred:
            inferred.add(fact['conclusion'])
            print(f"Inferred facts: {inferred}")

        # Process rules that match this fact as a premise
        for rule in rules:
            if fact['conclusion'] in rule['premises']:
                print(f"Rule premise satisfied: {rule['premises']} -> {rule['conclusion']}") # Remove satisfied premise
                rule['premises'].remove(fact['conclusion']) # Remove satisfied premise
                if not rule['premises']: # All premises satisfied
                    new_fact = {'conclusion': rule['conclusion']}
                    agenda.append(new_fact) # Add new fact to agenda
```

```

        print(f"New fact inferred: {new_fact['conclusion']}")

    # Debugging output after each iteration
    print(f"Current agenda: {[fact['conclusion'] for fact in agenda]})")
    print(f"Current inferred: {inferred}")

    # If the loop finishes without finding the query
    print(f"Query {query} not found.")
    return False

# Example Knowledge Base
KB = [
    {'premises': [], 'conclusion': 'American(Robert)'},
    {'premises': [], 'conclusion': 'Missile(T1)'},
    {'premises': [], 'conclusion': 'Owns(A, T1)'},
    {'premises': [], 'conclusion': 'Enemy(A, America)'},
    {'premises': ['Missile(T1)'], 'conclusion': 'Weapon(T1)'},
    {'premises': ['American(Robert)', 'Weapon(T1)', 'Sells(Robert, T1, A)'],
     'conclusion': 'Hostile(A)'}, {'premises': ['Owns(A, T1)', 'Enemy(A, America)'],
     'conclusion': 'Hostile(A)'}, {'premises': [], 'conclusion': 'Sells(Robert, T1, A)'}

]

# Query
query = 'Criminal(Robert)'

# Run the algorithm
result = fol_fc_ask(KB, query)
print("\nFinal Result:", result)
name = "Varsha Prasanth"
usn = "1BM22CS321"
print(f"Name: {name}, USN: {usn}")

```

OutputSnapshot

```

Initial agenda: ['American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)']
Initial inferred: set()

Processing fact: American(Robert)
Inferred facts: {'American(Robert)'}

Rule premise satisfied: ['American(Robert)', 'Weapon(T1)', 'Sells(Robert, T1, A)', 'Hostile(A)'] -> Criminal(Robert)
Current agenda: ['Missile(T1)', 'Owns(A, T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)']
Current inferred: {'American(Robert)'}

Processing fact: Missile(T1)
Inferred facts: {'American(Robert)', 'Missile(T1)'}
Rule premise satisfied: ['Missile(T1)'] -> Weapon(T1)
New fact inferred: Weapon(T1)
Current agenda: ['Owns(A, T1)', 'Enemy(A, America)', 'Sells(Robert, T1, A)', 'Weapon(T1)']
Current inferred: {'American(Robert)', 'Missile(T1)'}

Processing fact: Owns(A, T1)
Inferred facts: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)', 'Enemy(A, America)'}
Rule premise satisfied: ['Owns(A, T1)', 'Enemy(A, America)'] -> Hostile(A)
Current agenda: ['Enemy(A, America)', 'Sells(Robert, T1, A)', 'Weapon(T1)']
Current inferred: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)'}

Processing fact: Enemy(A, America)
Inferred facts: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)', 'Enemy(A, America)'}
Rule premise satisfied: ['Enemy(A, America)'] -> Hostile(A)
New fact inferred: Hostile(A)
Current agenda: ['Sells(Robert, T1, A)', 'Weapon(T1)', 'Hostile(A)']
Current inferred: {'Owns(A, T1)', 'American(Robert)', 'Missile(T1)', 'Enemy(A, America)'}

Processing fact: Sells(Robert, T1, A)
Inferred facts: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Sells(Robert, T1, A)', 'Enemy(A, America)'}
Rule premise satisfied: ['Weapon(T1)', 'Sells(Robert, T1, A)', 'Hostile(A)'] -> Criminal(Robert)
Current agenda: ['Weapon(T1)', 'Hostile(A)']
Current inferred: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Sells(Robert, T1, A)', 'Enemy(A, America)'}

Processing fact: Hostile(A)
Inferred facts: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Hostile(A)', 'Sells(Robert, T1, A)', 'Weapon(T1)', 'Enemy(A, America)'}
Rule premise satisfied: ['Hostile(A)'] -> Criminal(Robert)
New fact inferred: Criminal(Robert)
Current agenda: ['Criminal(Robert)']
Current inferred: {'American(Robert)', 'Missile(T1)', 'Owns(A, T1)', 'Hostile(A)', 'Sells(Robert, T1, A)', 'Weapon(T1)', 'Enemy(A, America)'}

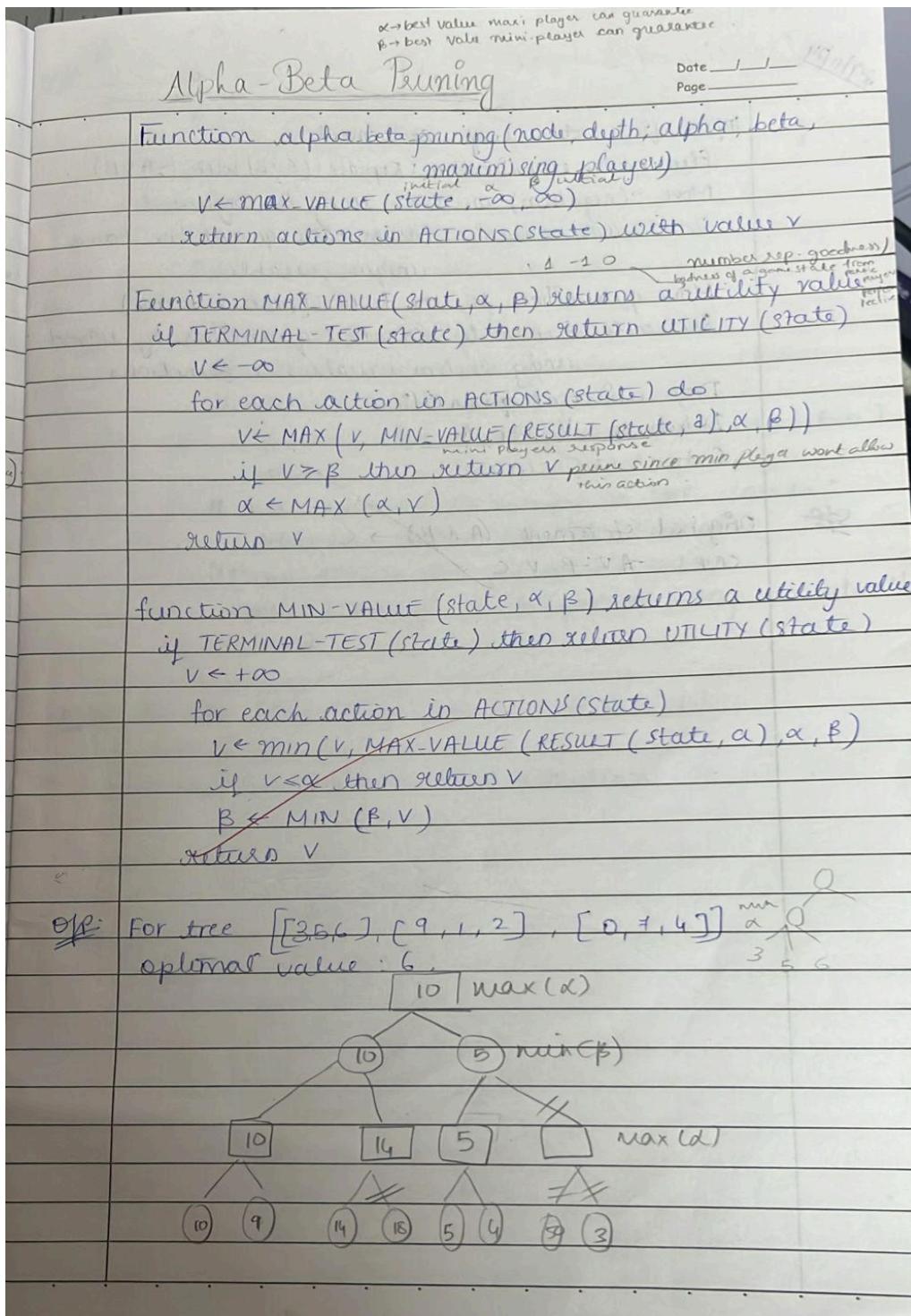
Processing fact: Criminal(Robert)
Found query match: Criminal(Robert)

Final Result: True
Name: Varsha Prasanth, USN: 1BM22CS321

```

Program-11 Alpha - Beta Pruning

Algorithm



Code

```
import math
```

```

# Alpha-Beta Pruning Algorithm

def alpha_beta_search(depth, index, is_max, values, alpha, beta,
target_depth):
    """Recursive function for Alpha-Beta Pruning."""
    # Base case: If the target depth is reached, return the leaf node value
    if depth == target_depth:
        return values[index]

    if is_max:
        # Maximizer's turn
        best = -math.inf
        for i in range(2):
            val = alpha_beta_search(depth + 1, index * 2 + i, False, values,
alpha, beta, target_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break # Prune remaining branches
        return best

    else:
        # Minimizer's turn
        best = math.inf
        for i in range(2):
            val = alpha_beta_search(depth + 1, index * 2 + i, True, values,
alpha, beta, target_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break # Prune remaining branches
        return best

def main():
    # User Input: Values of leaf nodes
    print("Enter the values of leaf nodes separated by spaces:")
    values = list(map(int, input().split()))

    # Calculate depth of the game tree
    target_depth = math.log2(len(values))
    if target_depth != int(target_depth):
        print("Error: The number of leaf nodes must be a power of 2.")
        return
    target_depth = int(target_depth)

```

```

# Run Alpha-Beta Pruning
result = alpha_beta_search(0, 0, True, values, -math.inf, math.inf,
target_depth)

# Display the result
print(f"The optimal value determined by Alpha-Beta Pruning is: {result}")

if __name__ == "__main__":
    main()
    name = "Varsha Prasanth"
    usn = "1BM22CS321"
    print(f"Name: {name}, USN: {usn}")

```

OUTPUT

→ Enter the values of leaf nodes separated by spaces:
10 9 14 18 5 4 50 3
The optimal value determined by Alpha-Beta Pruning is: 10
Name: Varsha Prasanth, USN: 1BM22CS321

STATE SPACE TREE

