VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Uday Shankar Y(1BM24CS427)

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 **Uday Shankar Y(1BM24CS427),** 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.

Lab faculty Incharge Name Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/UdayShankarY/AI

INDEX

Name UDAY SHANKAR Y Std ____ Sec ____

Roll No. ____ Subject ____ School/College ____

School/College Tel. No. ____ Parents Tel. No. ____

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1	18/8/25	Tic Tac Toe.	10	10. 25-8--
2	25/8/25	VACCUM CLEANER		
3a	25/8/25	BFS without heuristic	10	25-8
3b	25/8/25	BFS with heuristic		
3C	25/8/25	iterative Deepning DFS.	10	
4.	08/9/25	A* routeFinder, Misplaced	10	8/9 15/9
5	15/9/25	Navens way Hill climbing		
6	15/9/25	Simulated Annealing.	10	SRG
7	22/9/25.	propositional Logic		SRG 15/9. 22/9/25
8	6/10/25	unification.		
9	13/10/25	First order logic.	10	SRG
10	.	Resolution in FOL.		31/10/25
11		The Alpha-beta Algo		

Cryptic

Program 1

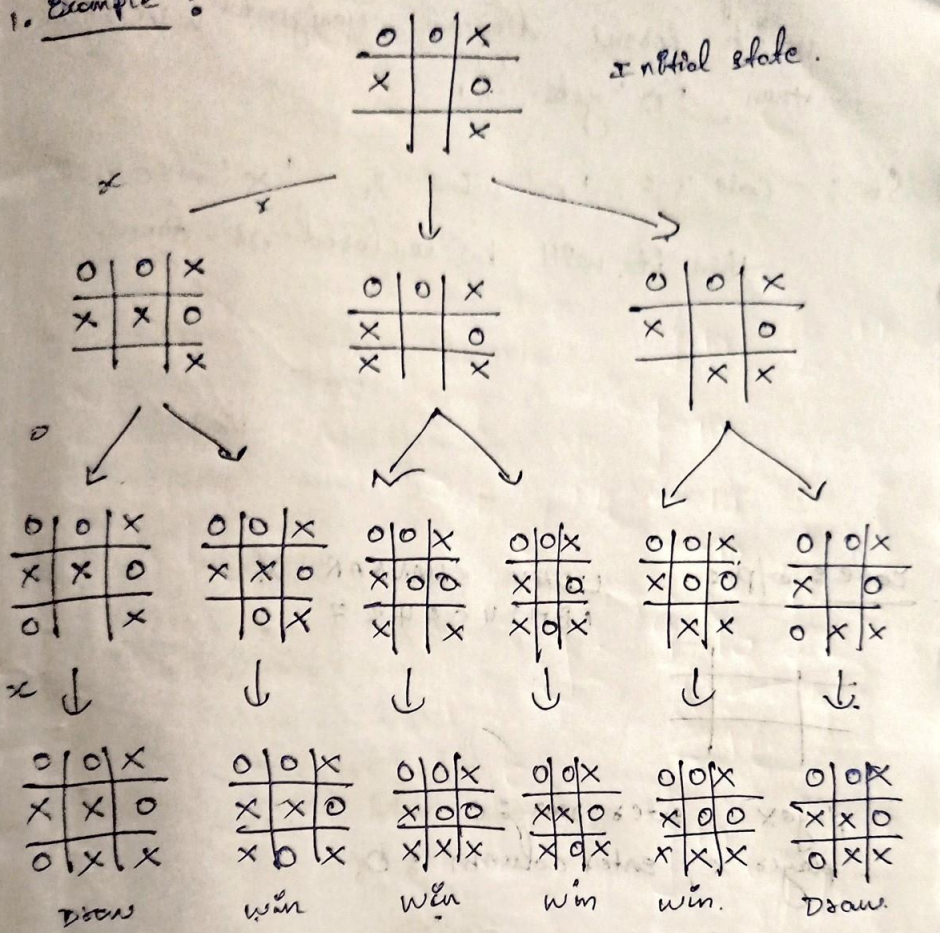
Implement Tic –Tac –Toe Game I

```
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()
```

```
Tic-Tac-Toe!
Name: Uday Shankar Y
USN: 1BM24CS427
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter row (0, 1, or 2):  0
Player X, enter column (0, 1, or 2):  0
X |   |
---------
  |   |
---------
  |   |
---------
Player O, enter row (0, 1, or 2):  0
Player O, enter column (0, 1, or 2):  1
X | O |
---------
  |   |
---------
  |   |
---------
Player X, enter row (0, 1, or 2):  1
Player X, enter column (0, 1, or 2):  1
X | O |
---------
  | X |
---------
  |   |
---------
Player O, enter row (0, 1, or 2):  2
Player O, enter column (0, 1, or 2):  2
X | O |
---------
  | X |
---------
  |   | O
---------
Player X, enter row (0, 1, or 2):  1
Player X, enter column (0, 1, or 2):  0
X | O |
---------
X | X |
---------
  |   | O
---------
Player O, enter row (0, 1, or 2):  1
Player O, enter column (0, 1, or 2):  2
X | O |
---------
X | X | O
---------
  |   | O
---------
Player X, enter row (0, 1, or 2):  2
Player X, enter column (0, 1, or 2):  0
X | O |
---------
X | X | O
---------
X |   | O
---------
Player X wins!
```

Implement Tic-Tac-Toe game

1. Example :



2. Algorithm.

S1 : Start Accepting the I/p from the user wheather 'X' or 'O'

S2 : or we can start from the initial state

S3 : if it forms same letter 'X' or 'O' diagonally, vertically, Horizontally then that player wins.

S4 : case 1 : player 1 wins (X)
if X is placed Horizontally, vertically or diagonally then 'X' gets +1

S5 : case 2 : 'O' wins (player 2)

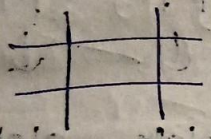
if 'O' is placed by the user (player 2) which forms diagonal, horizontal, vertical then 'O' gets +1

S6 : case 3 : 'O' wins & 'X' also wins.

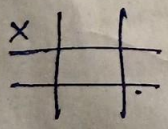
then it will be declared as draw.

code : o/p :

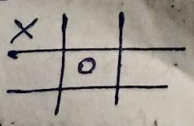
UDAY SHANKAR Y.
1BM24CS427.



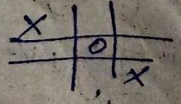
player X enter row : 0.
player X, enter column : 0.



player O, enter row : 1.
player O, enter colum : 1.



player X, enter row : 2.
player X, enter colum : 2.



Implement vacuum cleaner agent

```
rooms = {
    'A': int(input("Enter state of A (0 for clean, 1 for dirty): ")),
    'B': int(input("Enter state of B (0 for clean, 1 for dirty): ")),
    'C': int(input("Enter state of C (0 for clean, 1 for dirty): ")),
    'D': int(input("Enter state of D (0 for clean, 1 for dirty): "))
}

start = input("Enter starting location (A, B, C, or D): ").upper()
order = ['A', 'B', 'C', 'D']

if start not in order:
    print("Invalid starting location.")
    exit()

start_index = order.index(start)
visited_order = order[start_index:] + order[:start_index]

cost = 0

for i in range(len(visited_order)):
    current = visited_order[i]
    print(f"\nVacuum is in room {current}")

    if rooms[current] == 1:
        print(f"{current} is dirty.")
        print(f"Cleaning {current}...")
        rooms[current] = 0
        cost += 1
    else:
        print(f"{current} is clean.")

    if i < len(visited_order) - 1:
        next_room = visited_order[i + 1]
        print(f"Moving vacuum to {next_room}")
        cost += 1

print(f"\nCost: {cost}")
print(rooms)
```

```
Uday Shankar Y
1BM24CS427
Is Room A1 clean or dirty? (enter 'clean' or 'dirty', or 'exit' to quit):  dirty
Is Room A2 clean or dirty? (enter 'clean' or 'dirty', or 'exit' to quit):  clean
Is Room B1 clean or dirty? (enter 'clean' or 'dirty', or 'exit' to quit):  clean
Is Room B2 clean or dirty? (enter 'clean' or 'dirty', or 'exit' to quit):  clean

Current Room: Room A1
Room Status: {'Room A1': 0, 'Room A2': 1, 'Room B1': 1, 'Room B2': 1}
Total cost so far: 0
Enter action (clean/move/clean and move) or 'exit' to quit:  clean

Cleaning Room A1...
Room A1 is now clean.

All rooms are clean! Simulation finished.

Total cost of cleaning and moving: 1
```

2. VACCUM CLEANER

Algorithm

Initialize 2 × 2 matrix with dirt

S1: Start

S2: options:
- check Room (Dirt)
- clean Room (suck Dirt)
- move to left
- move to right
- move to up
- move to down.

S3: check_Dirt() → true.
clean_Room() → true
check All room cleaned() → false

S4: move to right
check_Dirt() → true
clean_Room() → true
check All room cleaned() → false

S5: move to down.
check_Dirt() → true.
clean_room() → true.
check All rooms cleaned() → false.

S6: move Left
check_Dirt() → false.
check All rooms cleaned → true
End program (Stop)

if check Dirt → true
then only check clean_Room()

O/P : Name : UDAY SHANKAR Y

USN : 1BM24CS427

Current room : A

Room Status : {'RoomA' : 0, 'RoomB' : 0}

Total cost so far : 0

Enter action (clean / move / clean and move)

: move

Moving to room B

Current ROOM : ROOM B

Room Status : {'ROOA' : 0, 'RoomB' : 0}.

Total cost so far : 1.

Enter action (clean / move / clean and move)

: clean

cleaning ROOM B - - -

Room B is clean now.

All rooms are clean ! & simulation finished

Total cost of cleaning : 4

9/8/24

Utility

- DFS
- Iterative
- BFS
- 8 puzzle

Program 2

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

Algorithm:

Code:

a) DFS

```
from collections import deque

    def input_state(prompt):
    print("Uday Shankar Y\n1BM24CS427")
    print(prompt)
    state = []
    for _ in range(3):
    row = input().strip().split()
    if len(row) != 3:
    raise ValueError("Each row must have exactly 3 numbers")
    state.extend(row)
    return tuple(state)

    def print_state(state):
    for i in range(0, 9, 3):
    print(' '.join(state[i:i+3]))
    print()

    def get_neighbors_with_moves(state):
    neighbors = []
    zero_index = state.index('0')
    r, c = zero_index // 3, zero_index % 3

    moves = [(-1,0,'UP'), (1,0,'DOWN'), (0,-1,'LEFT'), (0,1,'RIGHT')]
    for dr, dc, move_name in moves:
    nr, nc = r + dr, c + dc
    if 0 <= nr < 3 and 0 <= nc < 3:
    new_index = nr * 3 + nc
    new_state = list(state)
    new_state[zero_index], new_state[new_index] = new_state[new_index],
    new_state[zero_index]
    neighbors.append((tuple(new_state), move_name))

    return neighbors

    def is_solvable(state):
```

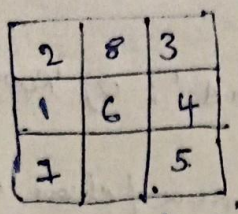
```
lst = [int(x) for x in state if x != '0']
inv_count = 0
for i in range(len(lst)):
for j in range(i+1, len(lst)):
if lst[i] > lst[j]:
inv_count += 1
return inv_count % 2 == 0

def bfs(start, goal):
queue = deque([(start, [], [], 0)])  # (state, path_states, path_moves, cost)
visited = set([start])
visited_count = 1

while queue:
current, path_states, path_moves, cost = queue.popleft()
if current == goal:
return path_states + [current], path_moves, cost, visited_count
for neighbor, move in get_neighbors_with_moves(current):
if neighbor not in visited:
visited.add(neighbor)
visited_count += 1
queue.append((neighbor, path_states + [current], path_moves + [move],
cost + 1))
return None, None, None, visited_count

def dfs(start, goal, max_depth=50):
stack = [(start, [], [], 0, 0)]  # (state, path_states, path_moves, cost, depth)
visited = set([start])
visited_count = 1

while stack:
current, path_states, path_moves, cost, depth = stack.pop()
if current == goal:
return path_states + [current], path_moves, cost, visited_count
if depth < max_depth:
for neighbor, move in get_neighbors_with_moves(current):
if neighbor not in visited:
visited.add(neighbor)
visited_count += 1
stack.append((neighbor, path_states + [current], path_moves + [move],
cost + 1, depth + 1))
return None, None, None, visited_count

def main():
try:
```

```
initial_state = input_state("Enter the initial state (3 rows, 3 numbers each,
use 0 for blank):")
goal_state = input_state("Enter the goal state (3 rows, 3 numbers each,
use 0 for blank):")
except ValueError as e:
print(f"Invalid input: {e}")
return

if not is_solvable(initial_state):
print("This initial state is unsolvable!")
return
if not is_solvable(goal_state):
print("This goal state is unsolvable!")
return

method = input("Enter search method (bfs or dfs): ").strip().lower()
if method not in ['bfs', 'dfs']:
print("Invalid method. Please enter 'bfs' or 'dfs'.")
return

print("\nSearching...\n")
if method == 'bfs':
states, moves, cost, visited_count = bfs(initial_state, goal_state)
else:
states, moves, cost, visited_count = dfs(initial_state, goal_state)

if states is None:
print("No solution found.")
else:
print(f"Solution found with cost {cost} moves.")
print(f"Total visited states: {visited_count}\n")

# Print initial state
print("Step 0:")
print_state(states[0])

# Print moves and subsequent states
for i in range(1, len(states)):
print(f"Move {i}: {moves[i-1]}")
print_state(states[i])

if __name__ == "__main__":
main()
:
```

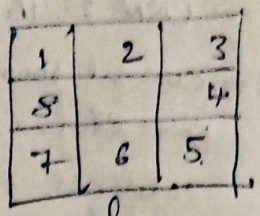
```
Uday Shankar Y
1BM24CS427
Enter the initial state (3 rows, 3 numbers each, use 0 for blank):
0 1 2
3 4 5
6 7 8
Uday Shankar Y
1BM24CS427
Enter the goal state (3 rows, 3 numbers each, use 0 for blank):
1 0 2
3 4 5
6 7 8
Enter search method (bfs or dfs): dfs

Searching...

Solution found with cost 1 moves.
Total visited states: 3

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

Move 1: RIGHT
1 0 2
3 4 5
6 7 8
```

Using BFS solve 8 puzzle without heuristic

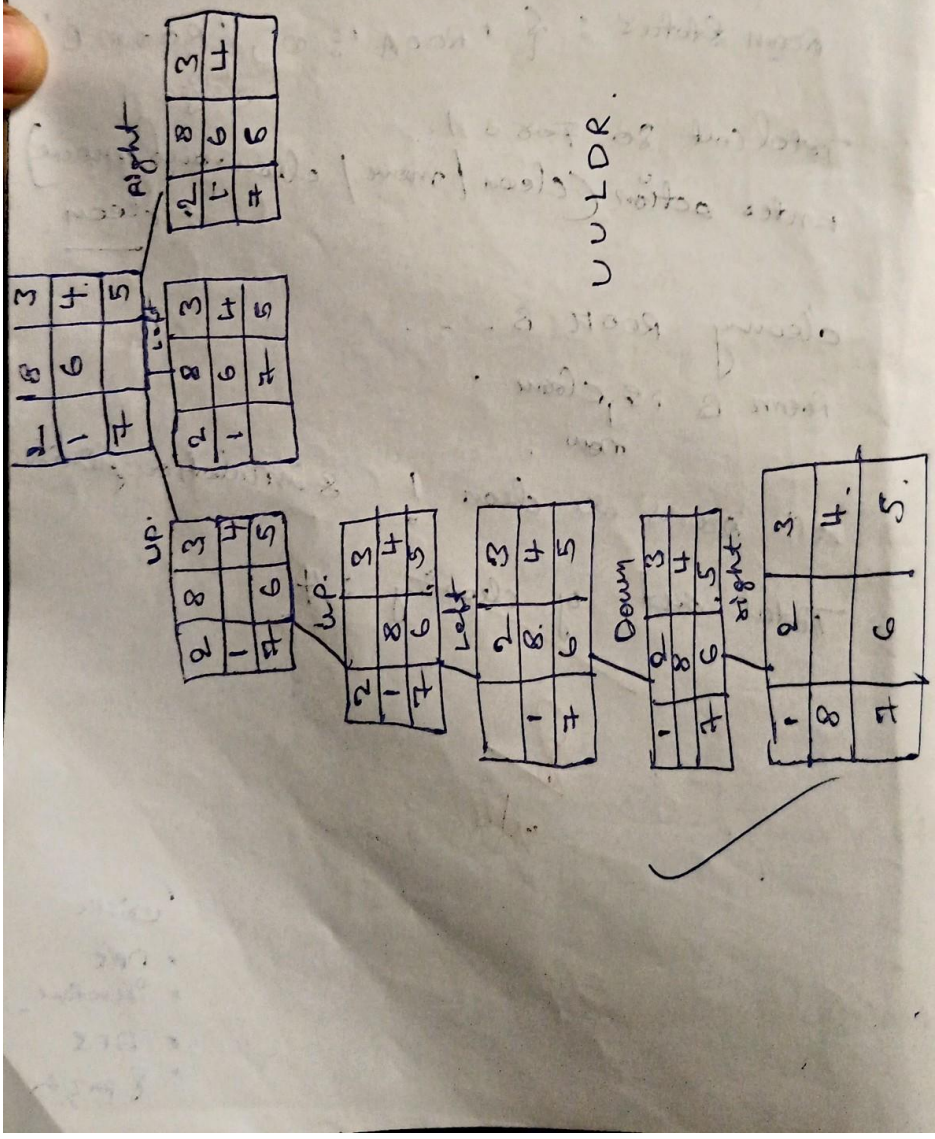
DFS.
BFS.



initial



goal



Steps

S1: take input the initial state

S2: move the empty Book swap it to left, right, up down to reach goal state.

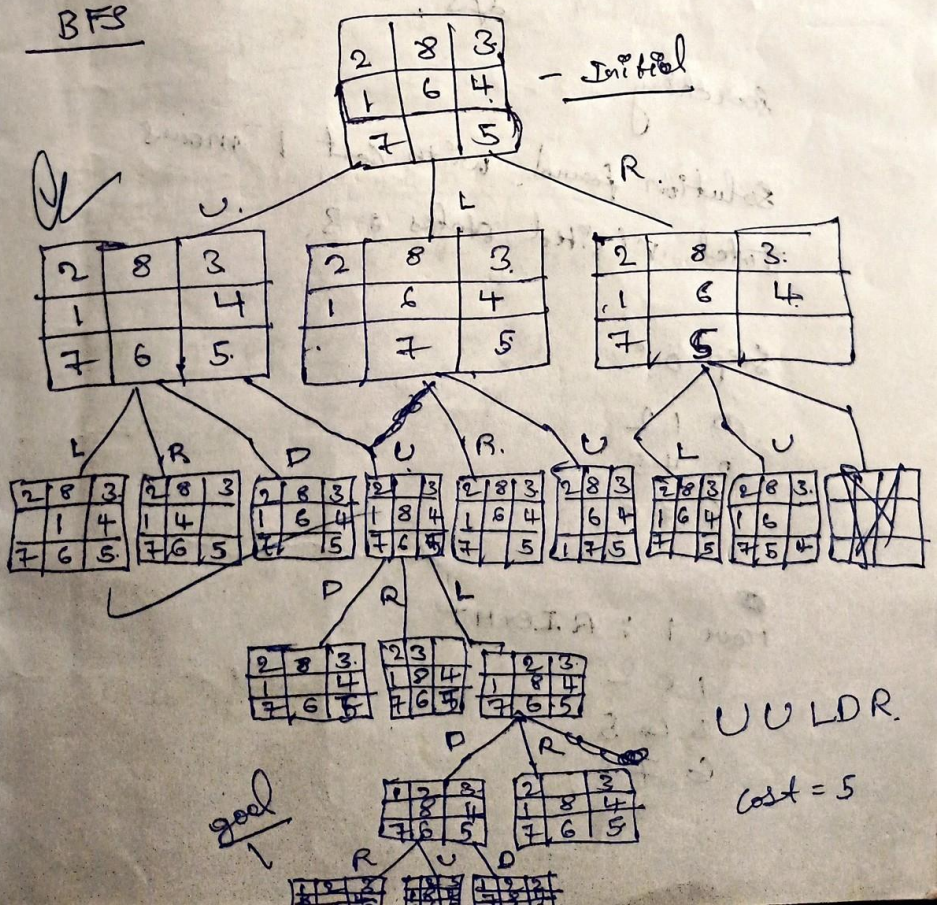
S3: if you got goal state stop.

S4: For DFS
first traverse in left most to get goal state

For BFS.
Traverse level by level to reach goal state.

S5: repeate until you reach goal.

BFS



O/P :

UDAY SHANKAR V.

1BM24CS427

Enter the initial state (3 rows, 3 numbers each, use 0 for blank) :

1 0 2
3 4 5
6 7 8.

Enter the goal state

1 0 2
3 4 5
6 7 8.

Enter search method (bfs or dfs):

dfs.

Searching....

Solution found with cost 1 moves.

Total visited states : 3.

Step 0 :

1 0 2
3 4 5
6 7 8

Move 1 : RIGHT

1 0 2
3 4 5
6 7 8.

A. Iterative Deepening Search goal_state = '123456780'

```
        moves = { 'U': -3,
        'D': 3,
        'L': -1,
        'R': 1
        }

        invalid_moves = {
        0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
        3: ['L'],       5: ['R'],
        6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
        }



MAX_VISITED_DISPLAY = 10
NUM_INTERMEDIATE_STATES = 3
MAX_DEPTH_LIMIT = 50

def print_state(state):
    for row in state:
        print(' '.join(str(x) for x in row))
    print()

def is_goal(state, goal_state):
    return state == goal_state

def find_zero(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_zero(state)
    directions = [(1,0), (-1,0), (0,1), (0,-1)]
    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]
            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 is_solvable(state):
    flat = [num for row in state for num in row if num != 0]
    inv_count = 0
```

```
    for i in range(len(flat)):
        for j in range(i + 1, len(flat)):
            if flat[i] > flat[j]:
                inv_count += 1
    return inv_count % 2 == 0

def dls(current_state, goal_state, depth_limit, path, visited, visited_states_display):
    """
    Depth-Limited Search helper for IDDFS.
    Returns:
        path if goal found else None
    """
    if len(path) - 1 > depth_limit:
        return None

    visited_states_display.append(current_state)
    if len(visited_states_display) <= MAX_VISITED_DISPLAY:
        print(f"Visited state #{len(visited_states_display)}:")
        print_state(current_state)

    if is_goal(current_state, goal_state):
        return path

    for neighbor in reversed(get_neighbors(current_state)):
        neighbor_tuple = tuple(tuple(row) for row in neighbor)
        if neighbor_tuple not in visited:
            visited.add(neighbor_tuple)
            result = dls(neighbor, goal_state, depth_limit, path + [neighbor], visited, visited_states_display)
            if result is not None:
                return result
            # Backtrack visited set for other paths:
            visited.remove(neighbor_tuple)
    return None

def iddfs(start_state, goal_state, max_depth=MAX_DEPTH_LIMIT):
    """
    Iterative Deepening DFS:
    Tries DFS with increasing depth limits until goal found or max depth exceeded.
    """
    print("Starting Iterative Deepening DFS traversal...\n")

    for depth in range(max_depth + 1):
        print(f"Trying depth limit: {depth}")
        visited_states_display = []
        visited = set()
        visited.add(tuple(tuple(row) for row in start_state))
        path = dls(start_state, goal_state, depth, [start_state], visited, visited_states_display)
        if path is not None:
            print(f"\nGoal reached at depth {depth}!")
            print(f"Total visited states in last iteration: {len(visited_states_display)}")
            return path
```

```
        print(f"No solution found at depth {depth}\n")
    print(f"No solution found within max depth limit {max_depth}")
    return None

def read_state(name):
    print(f"Enter the {name} state, row by row (use space-separated numbers, 0 for empty):")
    state = []
    for _ in range(3):
        row = input().strip().split()
        if len(row) != 3:
            raise ValueError("Each row must have exactly 3 numbers.")
        row = list(map(int, row))
        state.append(row)
    return state

# --- Main Execution ---

initial_state = read_state("initial")
goal_state = read_state("goal")

if not (is_solvable(initial_state) == is_solvable(goal_state)):
    print("The puzzle is unsolvable.")
    exit()

solution_path = iddfs(initial_state, goal_state)

if solution_path:
    cost = len(solution_path) - 1
    print(f"\nSolution found with cost: {cost}\n")
    print("Solution path:")

    total_steps = len(solution_path) - 1  # number of moves
    print("Initial State:")
    print_state(solution_path[0])

    if total_steps > 1:
        step_indices = list(range(1, total_steps))
        if len(step_indices) > NUM_INTERMEDIATE_STATES:
            interval = len(step_indices) // (NUM_INTERMEDIATE_STATES + 1)
            selected_indices = [step_indices[i * interval] for i in range(1, NUM_INTERMEDIATE_STATES + 1)]
        else:
            selected_indices = step_indices

        for idx in selected_indices:
            print(f"Intermediate State (Step {idx}):")
            print_state(solution_path[idx])

    print("Final State:")
    print_state(solution_path[-1])
else:
    print("No solution found")
```

```
for i, move in enumerate(result, 1):
 current_state = move_tile(current_state, move) print(f"Move {i}: {move}") print_state(current_state)
 else:
 print("No solution exists for the given start state or max depth reached.")
 else:
 print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

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

Name: Uday Shankar Y
USN: 1BM24CS427

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

2 8 3
1 _ 4
7 6 5

2 _ 3
1 8 4
7 6 5

_ 2 3
1 8 4
7 6 5

1 2 3
_ 8 4
7 6 5

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

IDS

Algorithm

Step 1 : Start

Step 2 : Set depth until d=0

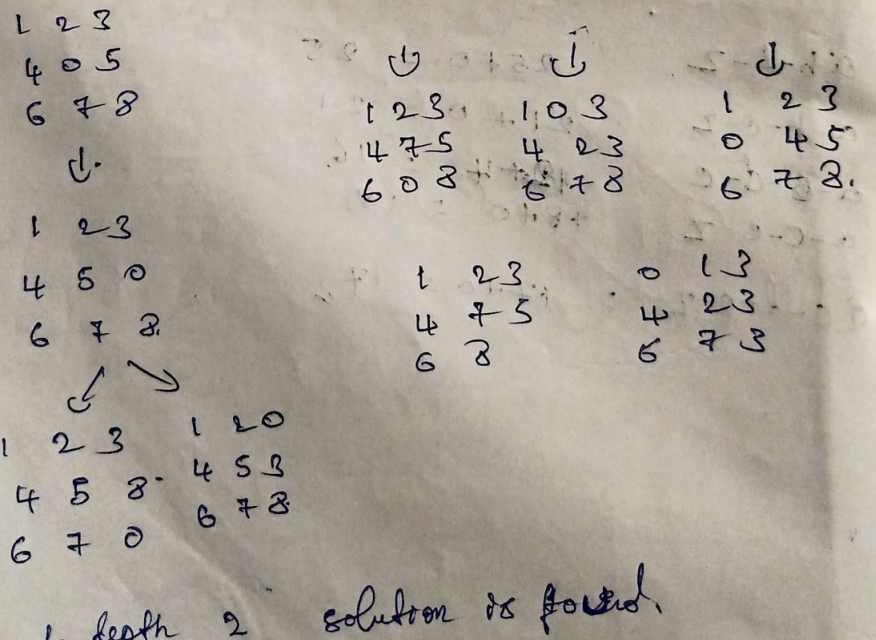
Step 3 : perform Depth Limited DFS. with current depth limited.

Step 4 : Explore search tree to depth only if goal is formed return to path.

Step 5 : If no solution is formed at depth d, Increment by 1.

Step 6 : repeat step 3 until you got solution.

dp :



at depth 2 solution is found.

Program 3

Implement A* search algorithm

Code:
MisplaceType

```
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: misplaced tiles \n")
        print("uday shankar y\n1BM24CS427")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])  # Moves taken to reach this step
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)  # Pause 1 second between steps for clarity
    else:
        print("No solution found.")
```

OutPut :

```
Step-by-step solution: misplaced tiles

uday shankar y
1BM24CS427
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
```

b.Manhattan distance.

```
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:Uday Shankar Y")
print("USN:1BM24CS427")
print("No solution found.")
```

Output :

```
Name:Uday Shankar Y
USN:1BM24CS427
No solution found.
```

Apply A* Algorithm

Misplaced tiles

Manhattan Distance

2	8	3
1	6	4
7		5

I

1	2	3
8		4
7	6	5

F

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

2	8	3
1	6	4
7		5

$g = 0$.

R. U L.

2	8	3
1	6	4
7	5	

1+1+0+0+0+0
+0+1

$h = 5$

2	8	3
1		4
7	6	5

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

$h = 4$

2	8	3
1	6	4
	7	5

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

$h = 6$.

R U L D

2	8	3
1	4	
7	6	5

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

$= 5$.

2		3
1	8	4
7	6	5

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

$= 3$.

2	8	3
	1	4
7	6	5

1 2 8 4 5 6 7 8
2+1+0+0+0+0+0+2,

$= 5$.

2	8	3
1	6	4
7		5

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

$= 5$

R L D

2	3	
1	8	4
7	6	5

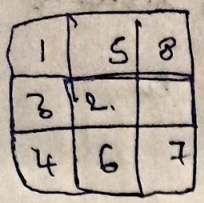
$h = 4$

	2	3
1	8	4
7	6	5

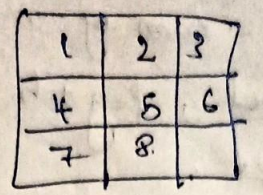
2	8	3
1		4
7	6	5

D

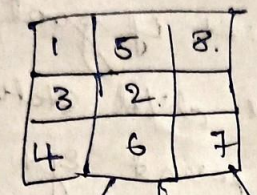
Manhattan Distance
misplaced tiles.



I



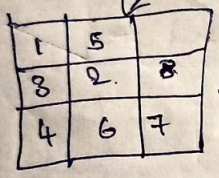
F

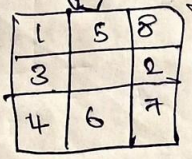


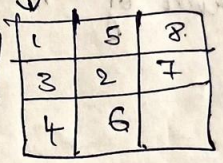
0 + 1 + 3 + 1 + 1 + 2 + 2 + 3

h = 8

U L D







1 + 1 + = 8

1 + 1 + 1 + 1

1 + 1

0 + 1 + 3 + 1 + 1 + 2 + 2 + 2.

h = 12

h = 8

0 + 2 + 3 + 1 + 1 + 2 + 2 + 3

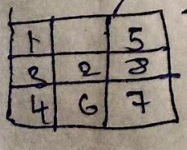
L = 14

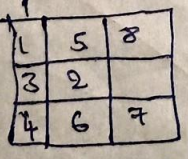
h = 8

0 + 1 + 3 + 1 + 1 + 1 + 2 + 3 + 3

14.

L D





0 + 1 + 3 + 1 + 2 + 2 + 2 + 2 = 13

0 + 1 + 3 + 1 + 1 + 2 + 2 + 3 = 13

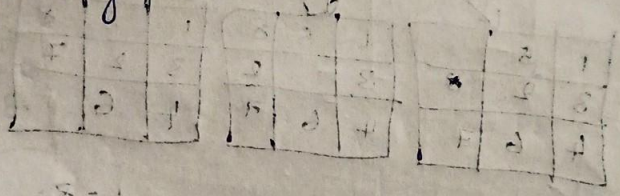
Algorithm for manhatten Distance

Start : Initialize start node, with g = 0, calculate h = sum of manhatten 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 g + 1

h = sum of manhatten distance

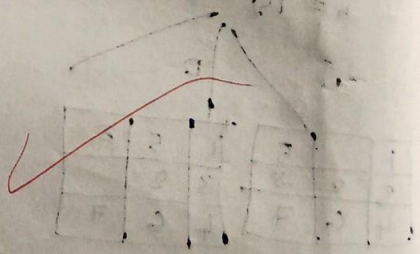


o/p :

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 8 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
```

Algorithm of misplaced tiles

* same as manhatten of misplaced just +1 for heuristic.

o/p :

step by step solution.

step 0 : moves.

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

step 1 : moves : 0

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

step 2 : moves : 00

```
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
7	6	5

Step 5 : Moves : UULDD.

1	2	3
8		4
7	6	5

Cost = 5 "

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

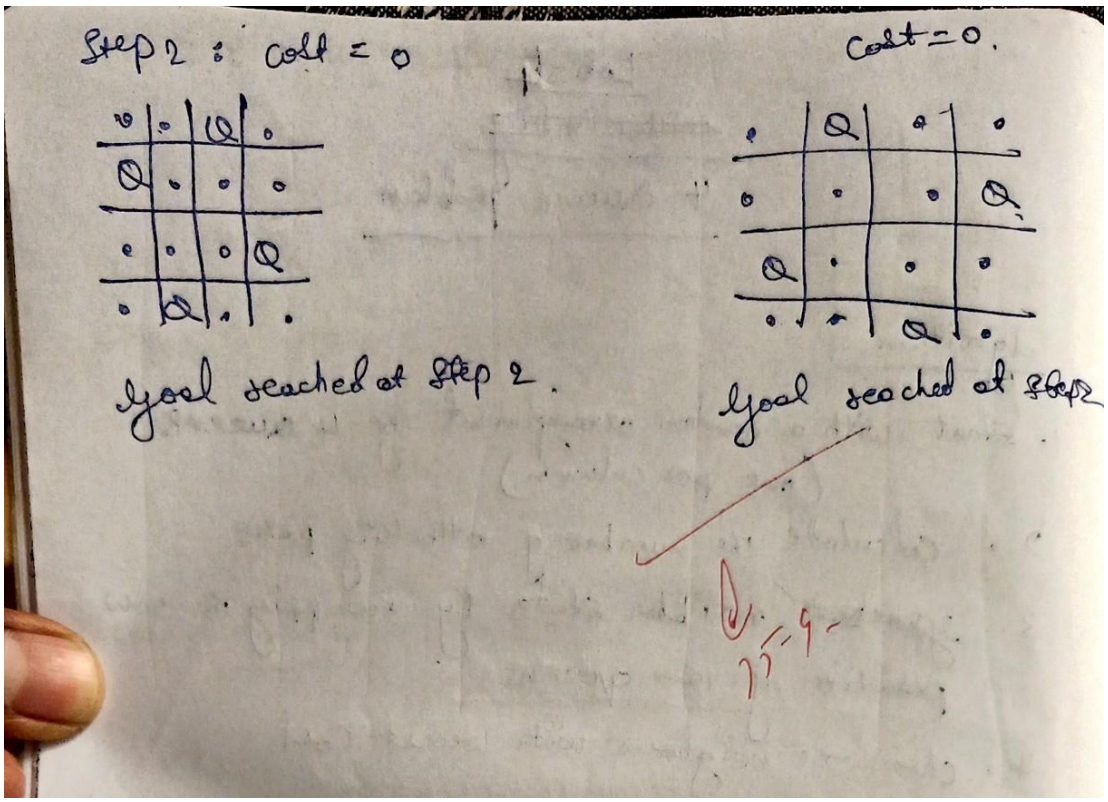
Algorithm:

```
import random
import time

def print_board(state):
"""Prints the chessboard for a given state.""" n = len(state)
for row in range(n): line = ""
for col in range(n):
if state[col] == row: line += "Q "
else:
line += ". " print(line)
print()

def compute_heuristic(state):
"""Computes the number of attacking pairs of queens.""" h = 0
n = len(state) for i in range(n):
for j in range(i + 1, n):
if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j): h += 1
return h

def get_neighbors(state):
"""Generates all possible neighbors by moving one queen in its column.""" neighbors = []
n = len(state)
for col in range(n): for row in range(n):
if row != state[col]: neighbor = list(state) neighbor[col] = row
neighbors.append(neighbor) return neighbors

def hill_climb_verbose(initial_state, step_delay=0.5):
"""Hill climbing algorithm with verbose output at each step.""" current = initial_state
current_h = compute_heuristic(current)

step = 0

print(f"Initial state (heuristic: {current_h}):") print_board(current)
time.sleep(step_delay)

while True:
```

```
neighbors = get_neighbors(current) next_state = None
next_h = current_h

for neighbor in neighbors:
h = compute_heuristic(neighbor) if h < next_h:
next_state = neighbor next_h = h

if next_h >= current_h:
print(f"Reached local minimum at step {step}, heuristic: {current_h}") return current, current_h

current = next_state current_h = next_h step += 1
print(f"Step {step}: (heuristic: {current_h})") print_board(current)
time.sleep(step_delay)

def solve_n_queens_verbose(n, max_restarts=1000):
"""Solves N-Queens problem using hill climbing with restarts and verbose output.""" for attempt in
range(max_restarts):
print(f"\n=== Restart {attempt + 1} ===\n")
initial_state = [random.randint(0, n - 1) for _ in range(n)] solution, h = hill_climb_verbose(initial_state)
if h == 0:
print(f" ■  Solution found after {attempt + 1} restart(s):") print_board(solution)
return solution else:
print(f"+ No solution in this attempt (local minimum).\n") print("Failed to find a solution after max
restarts.")

return None

# --- Run the algorithm ---
if   name                    == "  main  ":
N = int(input("Enter the number of queens (N): ")) solve_n_queens_verbose(N)
print("1BM24CS427 Uday Shankar Y")
```

Output:

```
Hill Climbing
Uday Shankar Y
1BM24CS427
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.
```



Lab - 4

~~Hill climbing~~

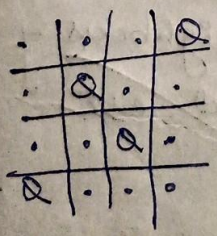
4 Queens problem

algorithm:

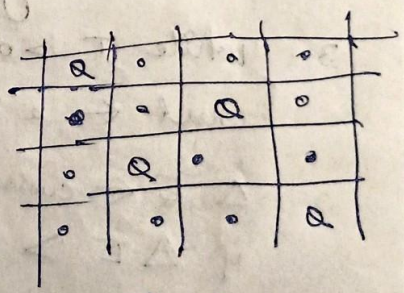
1. Start with a random arrangement of 4 queens (one per column)
2. Calculate the number of attacking pairs
3. Generate neighbor states by swapping the row position of two queens
4. Choose the neighbour with lowest cost
5. If the neighbor with is better (lower cost), move to that state.
6. Repeate steps 2 - 5 until no better neighbor exist.
7. If cost = 0, solution is found. otherwise local maximum is reached.

o/p:

Initial Board

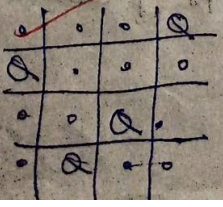


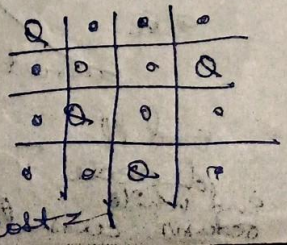
cost: 2.



cost = 2.

step 1 : ~~cost = 1~~





cost = 1

Program 5

Simulated Annealing to Solve 8-Queens problem

Code:

```
import random import math

def compute_heuristic(state): """Number of attacking pairs.""" h = 0
n = len(state) for i in range(n):
for j in range(i + 1, n):
if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

h += 1
return h

def random_neighbor(state):
"""Returns a neighbor by randomly changing one queen's row.""" n = len(state)
neighbor = state[:]
col = random.randint(0, n - 1) old_row = neighbor[col]
new_row = random.choice([r for r in range(n) if r != old_row]) neighbor[col] = new_row
return neighbor

def dual_simulated_annealing(n, max_iter=10000, initial_temp=100.0, cooling_rate=0.99):
"""Simulated Annealing with dual acceptance strategy."""
current = [random.randint(0, n - 1) for _ in range(n)] current_h = compute_heuristic(current)
temperature = initial_temp

for step in range(max_iter): if current_h == 0:
print(f" Solution found at step {step}") return current

neighbor = random_neighbor(current) neighbor_h = compute_heuristic(neighbor) delta = neighbor_h
- current_h

if delta < 0:
current = neighbor current_h = neighbor_h
else:
# Dual acceptance: standard + small chance of higher uphill move probability = math.exp(-delta /
temperature)
if random.random() < probability: current = neighbor
current_h = neighbor_h

temperature *= cooling_rate
if temperature < 1e-5: # Restart if stuck temperature = initial_temp
```

```
current = [random.randint(0, n - 1) for _ in range(n)] current_h = compute_heuristic(current)

print("+ Failed to find solution within max iterations.") return None

# --- Run the algorithm ---
if   name                    == "  main  ":
N = int(input("Enter number of queens (N): ")) solution = dual_simulated_annealing(N)

if solution:
print("Position format:")
print("[", " ".join(str(x) for x in solution), "]") print("Heuristic:", compute_heuristic(solution))
print("1BM24CS427 Uday Shankar Y")
```

Output:

```
Final board with conflicts = 1
Name:Uday Shankar Y
USN:1BM24CS427

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

Failed to find a solution.
```

Lab-5

Simulated Annealing

Algorithm

1. current ← initial state
2. T ← a large positive value
3. while T > 0 do
   next ← a random neighbour of current
   Δ E ← current.cost - next.cost
   if Δ E > 0 then
      current ← next
   else
      current ← next with probability $p = \frac{\Delta E}{e^T}$
   end if
   decrease T
end while
return current

output:

The best position found is : [5, 1, 3, 4, 7, 5, 0, 2]

The number of queens that are not attacking each other is : 5

15/9/24

Program 6

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

Code:

```
from itertools import product

# ----------------- Propositional Logic Symbols ------------------ class Symbol:
def  init (self, name): self.name = name

def  invert (self): # ~P return Not(self)

def  and (self, other): # P & Q return And(self, other)

def  or (self, other): # P | Q return Or(self, other)

def  rshift (self, other): # P >> Q (implication) return Or(Not(self), other)

def   eq  (self, other): # P == Q (biconditional) return And(Or(Not(self), other), Or(Not(other), self))

def eval(self, model): return model[self.name]

def symbols(self): return {self.name}

def  repr (self): return self.name

class Not:
def  init (self, operand): self.operand = operand

def eval(self, model):
return not self.operand.eval(model)

def symbols(self):
return self.operand.symbols()
```

```
    def   repr  (self):
    return f"~{self.operand}"

    def  invert (self): # allow ~~A return Not(self)


class And:
    def _init_(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) and self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def __repr__(self):
        return f"({self.left} & {self.right})"

    def _invert_(self): # allow ~(A & B) return
        Not(self)


class Or:
    def _init_(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) or self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def __repr__(self):
        return f"({self.left} | {self.right})"

    def _invert_(self): # allow ~(A | B) return
        Not(self)
```

```
tt_entails(kb, alpha, show_table=False):
    symbols = sorted(list(kb.symbols() | alpha.symbols()))
    if show_table:
        print_truth_table(kb, alpha, symbols)
    return tt_check_all(kb, alpha, symbols, {})


def tt_check_all(kb, alpha, symbols, model):
    if not symbols: # all symbols assigned
        if kb.eval(model): # KB is true
            return alpha.eval(model)
        else:
            return True # if KB is false, entailment holds
    else:
        P, rest = symbols[0], symbols[1:]

        model_true = model.copy()
        model_true[P] = True
        result_true = tt_check_all(kb, alpha, rest, model_true)

        model_false = model.copy()
        model_false[P] = False
        result_false = tt_check_all(kb, alpha, rest, model_false)

        return result_true and result_false


# ---------------------- Truth Table Printer ----------------------
def print_truth_table(kb, alpha, symbols):
    header = symbols + ["KB", "Query"]
    print(" | ".join(f"{h:^5}" for h in header))
    print("-" * (7 * len(header)))

    for values in product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        kb_val = kb.eval(model)
        alpha_val = alpha.eval(model)
        row = [str(model[s]) for s in symbols] + [str(kb_val), str(alpha_val)]
        print(" | ".join(f"{r:^5}" for r in row))
```

```
    print()
C = Symbol("C")
T = Symbol("T")


c = T | ~T

# KB: P → Q
kb1 = ~ (S|T)
# Query: Q
alpha1 = S & T


print("Knowledge Base:", kb1)
print("Query:", alpha1)
print()
result = tt_entails(kb1, alpha1, show_table=True)
print("Does KB entail Query?", result)
```

Output:

```
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 on Knowledge Based — Week-6

Semantics.

Truth table for connectives.

P	Q	¬P	P∧Q	P∨Q	P⇒Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

If both are same then only true.

Propositional Inference : ⊨ enumeration method

Checking that $KB \models \alpha$

A	B	C	$\neg C$	$A \vee C$	$B \vee \neg C$	KB	α
F	F	F	T	F	T	F	F
F	F	T	F	T	F	F	F
F	T	F	T	F	T	F	F
F	T	T	F	T	T	T	T
T	F	F	T	T	T	T	T
T	F	T	F	T	F	F	T
T	T	F	T	T	T	T	T
T	T	T	F	T	T	T	T

Truth-Table Enumeration Algorithm

(1) List all variables.
- Find all the symbols that appear in KB & α.

Ex:- A, B, C

(2) Try every possibility.
- Each symbol can be True or False
- So we test all combination.

(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 don't 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 false → KB does not entail α

o/p :

KB : NOT
Enter query (alpha) : T

Truth table

Not	T	KB	alpha
T	F	T	T
T	F	T	F
F	T	F	T
F	F	F	F

KB Entails Alpha : No.

Result : False.

SRS
22/9/25

Program 7

Implement unification in first order logic

Code:

```
class
   UnificationError(
   Exception): pass

def occurs_check(var, term):

   """Check if a variable occurs in a term (to prevent infinite
   recursion)""" if var == term:
      return True
   if isinstance(term, tuple): # Term is a compound
      (function term) return any(occurs_check(var,
      subterm) for subterm in term)
   return False

def unify(term1, term2, substitutions=None):
   """Try to unify two terms, return the MGU (Most
   General Unifier)""" if substitutions is None:
      substitutions = {}

   # If both terms are equal, no further
   substitution is needed if term1 == term2:
      return substitutions

   # If term1 is a variable, we substitute
   it with term2 elif isinstance(term1,
   str) and term1.isupper():
      # If term1 is already
      substituted, recurse if
      term1 in substitutions:
         return unify(substitutions[term1], term2,
      substitutions) elif occurs_check(term1,
      term2):
         raise UnificationError(f"Occurs check fails: {term1}
      in {term2}") else:
         substitutions[term
```

```
        1] = term2 return
        substitutions

    # If term2 is a variable, we substitute
    it with term1 elif isinstance(term2,
    str) and term2.isupper():
        # If term2 is already
        substituted, recurse if
        term2 in substitutions:
            return unify(term1, substitutions[term2],
        substitutions) elif occurs_check(term2,
        term1):
            raise UnificationError(f"Occurs check fails: {term2}
        in {term1}") else:
            substitutions[term
            2] = term1 return
            substitutions

    # If both terms are compound (i.e., functions), unify their
    parts recursively elif isinstance(term1, tuple) and
    isinstance(term2, tuple):
        # Ensure that both terms have the same "functor" and number of arguments

        # if len(term1) != len(term2):
        #     raise UnificationError(f"Function arity mismatch: {term1} vs {term2}")

        for subterm1, subterm2 in zip(term1, term2):
            substitutions = unify(subterm1, subterm2,

        substitutions) return substitutions

    else:
        raise UnificationError(f"Cannot unify: {term1} with {term2}")

# Define the terms
as tuples term1 =
('p', 'b', 'X', ('f',
('g', 'Z')))
term2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

try:
```

```
    # Find the MGU
    result = unify(term1,
    term2) print("Most
    General Unifier
    (MGU):") print(result)
except
    UnificationError
    as e:
    print(f"Unification
    failed: {e}")
finally:
```

Output:

```
Uday Shankar Y
usn : 1BM24CS427
No unifier found.
```

Unification

Algorithm : 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 in ψ_1 and ψ_2 are not same, then return FAILURE

Step 3 : If ψ_1 and ψ_2 have a different number of arguments, then return FAILURE.

Step. 4 : Set Substitution set (SUBST) to NIL

Step 5 : For $i = 1$ to the number of elements in

a) call unify function with the i^{th} element of ψ_1 and i^{th} element of ψ_2, put the results into S.

b) if S = failure then returns failure.

c) If S ≠ NIL then do,

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

b. SUBST = APPEND (S, SUBST)

step. 6. Return SUBST

Solve the following:

1) Find most general unifier (mgu) of

{Q(a, g(x,a), f(y)) and

Q(a, g(f(b), a), x)}.

→ θ = x/f(b)

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

θ = f(y)/x.

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

2) unify { prime(11) and prime(y) }

θ = 11/y.

unify {prime(y) and prime(y)}

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

θ = y/John

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

θ = x/Bill

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

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

$\theta = f(a) / x$

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

unification fails

Program 8

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

```
facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Missile(T1)",
    "Owns(A, T1)"
}


rules = [
    ("Enemy(x, America)", "Hostile(x)"),
    ("Missile(x)", "Weapon(x)"),
    ("Missile(x) ∧ Owns(A, x)", "Sells(Robert, x, A)"),
    ("American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r)", "Criminal(p)")
]

def apply_rules(facts):
    new_facts = set()


    if "Enemy(A, America)" in facts and "Hostile(A)" not in facts:
        print("Step 1: Enemy(A, America) → Hostile(A)")
        new_facts.add("Hostile(A)")


    if "Missile(T1)" in facts and "Weapon(T1)" not in facts:
        print("Step 2: Missile(T1) → Weapon(T1)")
        new_facts.add("Weapon(T1)")


    if "Missile(T1)" in facts and "Owns(A, T1)" in facts and "Sells(Robert, T1, A)"
not in facts:
        print("Step 3: Missile(T1) ∧ Owns(A, T1) → Sells(Robert, T1, A)")
        new_facts.add("Sells(Robert, T1, A)")

    if {"American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)", "Hostile(A)"} <=
facts and "Criminal(Robert)" not in facts:
        print("Step 4: American(Robert) ∧ Weapon(T1) ∧ Sells(Robert, T1, A) ∧
Hostile(A) → Criminal(Robert)")
        new_facts.add("Criminal(Robert)")

    return new_facts


step = 1
while True:
    new_facts = apply_rules(facts)
    if not new_facts:
        break
    facts |= new_facts
    step += 1
```

```
print("\n✅ Final
Facts:") for fact in
facts:
    print(fact)
```

```
Step 1: Enemy(A, America) → Hostile(A)
Step 2: Missile(T1) → Weapon(T1)
Step 3: Missile(T1) ⋀ Owns(A, T1) → Sells(Robert, T1, A)
Step 4: American(Robert) ⋀ Weapon(T1) ⋀ Sells(Robert, T1, A) ⋀ Hostile(A) →
Criminal(Robert)
```

Output:

```
Uday Shankar Y
1BM24CS427
Step 1: Enemy(A, America) → Hostile(A)
Step 2: Missile(T1) → Weapon(T1)
Step 3: Missile(T1) ∧ Owns(A, T1) → Sells(Robert, T1, A)
Step 4: American(Robert) ∧ Weapon(T1) ∧ Sells(Robert, T1, A) ∧ Hostile(A) → Criminal(Robert)

✅ Final Facts:
Owns(A, T1)
Hostile(A)
Criminal(Robert)
Enemy(A, America)
Sells(Robert, T1, A)
Missile(T1)
Weapon(T1)
American(Robert)
```

First order Logic

Lab-8

premises

$P \Rightarrow Q$.

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$.

$A \wedge P \Rightarrow L$.

$A \wedge B \Rightarrow L$

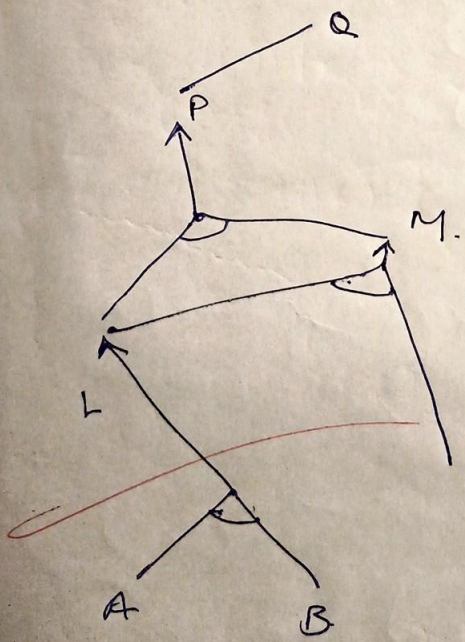
} Rules.

A

B

} facts.

Prove Q.



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

(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

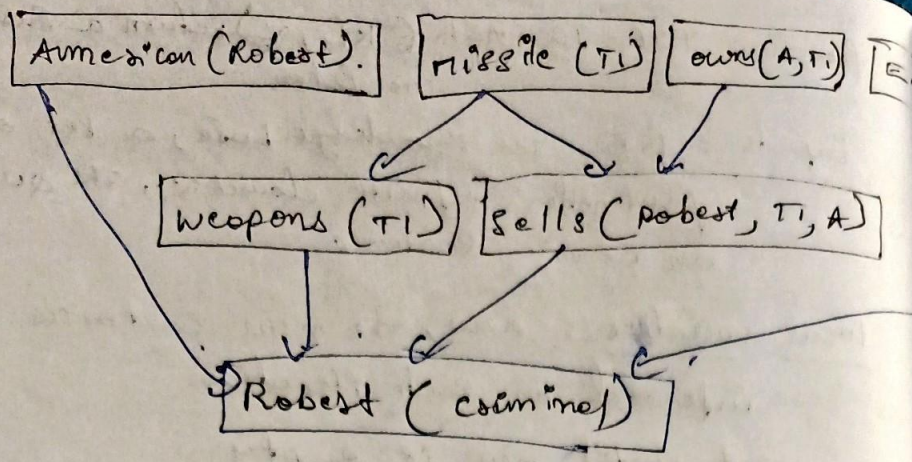
add q' to new

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 (A, x)

Essential instantiation, introducing a new constant T1:

owns (A, T1).

missile (T1).

3. All of the missiles were sold to country A by Robert.

∀ x missile (x) ∧ owns (A, x) ⇒ sells (Robert, x, A).

4. Missiles are weapons.

missile (x) ⇒ weapon (x)

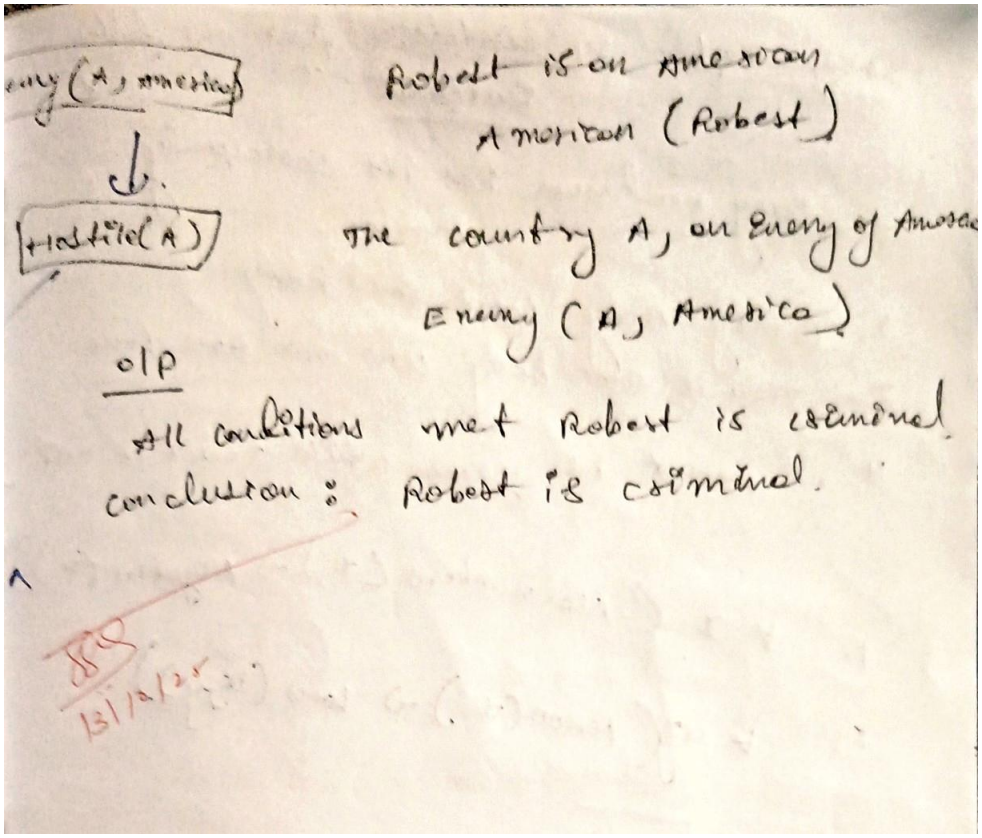
5. Enemy of America is known as hostile.

∀ x Enemy (x, America) ⇒ hostile (x)

olp

All cou
conclus

13/12/25


enemy (A, America)
Hostile(A)
Robert is an American
American (Robert)
The country A, an Enemy of America
Enemy (A, America)
o/p
All conditions met Robert is criminal
conclusion : Robert is criminal.
13/2/25

Program 9

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

```
class Literal:
    def __init__(self, name, negated=False):
        self.name = name
        self.negated = negated

    def __repr__(self):
        return f"¬{self.name}" if self.negated else self.name

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

    def __hash__(self):
        return hash((self.name, self.negated))
print('Suhas B P (1BM23CS345)')
def convert_to_cnf(sentence):
    return sentence  # Placeholder for CNF conversion logic

def negate_literal(literal):
    return Literal(literal.name, not literal.negated)

def resolve(clause1, clause2):
    resolvents = []
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1.name == literal2.name and literal1.negated !=
literal2.negated:
                new_clause = (set(clause1) - {literal1}) | (set(clause2) -
{literal2})
                resolvents.append(frozenset(new_clause))
    return resolvents

def prove_conclusion(premises, conclusion):
    cnf_premises = [convert_to_cnf(p) for p in premises]
    cnf_conclusion = convert_to_cnf(conclusion)
    negated_conclusion = frozenset({negate_literal(lit) for lit in cnf_conclusion})

    clauses = set(frozenset(p) for p in cnf_premises)
    clause_list = list(clauses)
    id_map = {}
    parents = {}
    clause_id = 1
```

```
    for c in clause_list:
        id_map[c] = f"C{clause_id}"
        clause_id += 1

    id_map[negated_conclusion] = f"C{clause_id} (¬Conclusion)"
    clauses.add(negated_conclusion)
    clause_list.append(negated_conclusion)
    clause_id += 1

    print("\n--- Initial Clauses ---")
    for c in clause_list:
        print(f"{id_map[c]}: {set(c)}")

    print("\n--- Resolution Steps ---")

    while True:
        new_clauses = set()
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                for r in resolvents:
                    if r not in id_map:
                        id_map[r] = f"C{clause_id}"
                        parents[r] = (id_map[clause_list[i]],
id_map[clause_list[j]])
                        clause_id += 1

                    print(f"{id_map[r]} = RESOLVE({id_map[clause_list[i]]},
{id_map[clause_list[j]]}) -> {set(r)}")

                    if not r:
                        print(f"\nEmpty clause derived! ({id_map[r]})")
                        print("\n--- Resolution Tree ---")
                        print_resolution_tree(parents, id_map, r)
                        return True

                    new_clauses.add(r)

        if new_clauses.issubset(clauses):
            print("\nNo new clauses can be derived.")
            return False

        clauses |= new_clauses
        clause_list = list(clauses)

def print_resolution_tree(parents, id_map, empty_clause):
    """Recursively print resolution tree leading to the empty clause."""
    def recurse(clause):
        if clause not in parents:
            print(f" {id_map[clause]}: {set(clause)}")
```

```
            return
        left, right = parents[clause]
        print(f" {id_map[clause]} derived from {left} and {right}")
        for parent_clause, parent_name in zip([k for k, v in id_map.items() if v in
[left, right]], [left, right]):
            recurse(parent_clause)

def parse_literal(lit_str):
    lit_str = lit_str.strip()
    if lit_str.startswith("¬") or lit_str.startswith("~"):
        return Literal(lit_str[1:], True)
    return Literal(lit_str, False)

def get_user_input():
    premises = []
    num_premises = int(input("Enter number of premises: "))
    for i in range(num_premises):
        clause_str = input(f"Enter clause {i+1} (e.g., A, ¬B, C): ")
        literals = {parse_literal(l) for l in clause_str.split(",")}
        premises.append(literals)

    conclusion_str = input("Enter conclusion (e.g., C or ¬C): ")
    conclusion = {parse_literal(l) for l in conclusion_str.split(",")}
    return premises, conclusion

if __name__ == "__main__":
    premises, conclusion = get_user_input()

    if prove_conclusion(premises, conclusion):
        print("\n Conclusion can be proven from the premises.")
    else:
        print("\n Conclusion cannot be proven from the premises.")
```

Output :

```
Domain: [-3, -2, -1, 0, 1, 2, 3, 4, 5]
Statement 1: ∀x (is_even(x) → is_positive(x)) = False
Statement 2: ∃x (is_even(x) ∧ ¬is_positive(x)) = True
Code By : Uday Shankar Y : 1BM24CS427
```

Resolution in FOL

1. Eliminate bicondition & implications
   - Eliminate ⟷ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - Eliminate ⇒ replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

2. Move ¬ towards atomic
   - $\neg(\forall x\, p) \equiv \exists x\, \neg p$
   - $\neg(\exists x\, p) \equiv \forall x\, \neg p$
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
   - $\neg\neg\alpha \equiv \alpha$

3. Standardize variables apart by renaming them each quantifier should use different variable

Resolution is theorem proving technique that proceeds by building refutation proofs, i.e proofs by contradiction.

Resolution is used, if there are various statements are given and we need to prove a conclusion of these statements unification is key concept of proofs by resolution

Resolution is a single inference rule which can efficiently operate on the conjuctive normal form or clausal form

4. Skolemize: Each existential variable is replaced by a Skolem constant or Skolem function of the enclosing universally quantified variables

- for instance, $\exists x \; R(x, h(x))$ becomes (h_1) where h_1 new constant

5. Drop universal quantifiers
   - $\forall$ person(x) becomes person(x)

6. Distribute $\wedge$ over $\vee$:
   - $(\alpha \wedge \beta) \vee \gamma = (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Algorithm for Resolution

Basic steps: for proving a conclusion S given premises, premise, ---, premise n

call Explained in FOL):

1. Convert all sentences to CNF
2. Negate conclusion S & to premise clauses
3. Add negated conclusion S to the premise clauses
4. Repeat until contradiction or no progress in matrix
   a. Select 2 clauses call them parent clauses
   b. Resolve them together, performing all required unifications
   c. if resolvent is the Empty clause, a contradiction has been found (i.e., S follows

from the premises.

d. If not, add resolvent to the premises

If not succeed in step m, we have proved the conclusion.

Representation of FOL

a. $\forall x$: food(x) → likes(John, x)

b food(Apple) ∧ food(vegetables)

c $\forall x \forall y$: eats(x, y) ∧ ¬ killed(x) → food(y)

d eats(Anil, peanuts) ∧ alive(Anil)

e $\forall x$: eats(Anil, x) → eats(Harry, x)

f $\forall x$: killed(x) → alive(x)

g $\forall x$: alive(x) → ¬ killed

h likes(John, peanuts)

proof by resolution

Eliminate Implication.

a. $\forall x$: food(x) → likes(John, x)

b. food(Apple) ∧ food(vegetables)

c. $\forall x \forall y$: eats(x, y) ∧ ¬ killed(x) → food(y)

d. eats(Anil, peanuts) ∧ alive(Anil)

e - $\forall x$: eats(Anil, x) → eats(Harry, x)

f $\forall x$ ¬ killed(x) → alive(x)

g $\forall x : alive(x) \Rightarrow \neg killed(x)$

h- likes (John, peanuts)

$\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$.

a. $\forall x \neg food(x) \vee likes(John, x)$

b food (Apple) $\wedge$ food (vegetables)

c. $\forall x \forall y \neg [eats(x, y) \wedge \neg killed(x)]$ $\vee food(y)$

d eats (Anil, peanuts) $\wedge$ anil (Anil)

e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$

f. $\forall x \neg [\neg killed(x)] \vee alive(x)$

g $\forall x \neg alive(x) \vee \neg killed(x)$

h likes (John, peanuts)

Eliminate implication $\alpha \rightarrow \beta$ with $\neg \alpha \vee \beta$

a. $\forall x \neg food(x) \vee likes(John, x)$

b food (Apple) $\wedge$ food (vegetables)

c $\forall x \forall y \neg [eats(x, y) \wedge \neg killed(x)] \vee$ food (y)

d. eats (Anil, peanuts) $\wedge$ alive (Anil)

e $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$

f. $\forall x \neg [killed(x)] \vee alive(x)$

g $\forall x \neg alive(x) \vee \neg killed(x)$

h likes (John, peanuts)

Prove negation in words.

a. $\forall x \neg food(x) \vee likes(John, x)$

b. food (Apple) $\wedge$ food (vegetables)

c. $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$

d. eats (Anil, peanut) $\wedge$ alive (Anil)

e. $\forall x \neg eats(Anil, x) \vee eats(Hoosy, x)$

f. $\forall x$ killed (x) $\vee$ alive (x)

g. $\forall x \neg alive(x) \vee \neg killed(x)$

Rename variables or standardise variables

a. $\forall x \neg food(x) \vee likes(John, x)$

b. food (Apple) $\wedge$ food (vegetables)

c. $\forall y \forall z \neg eats(y, z) \vee killed(y) \vee food(z)$

d. Eats (Anil, peanuts) $\wedge$ alive (Anil)

e. $\forall w \neg eats(Anil, w) \vee eats(Hoosy, w)$

f. $\forall g$ killed (g) $\vee$ alive (g)

g. $\forall k \neg alive(k) \vee \neg killed(k)$

h. likes (John, peanuts)

Drop universal quantifier

a. $\neg food(x) \vee likes(John, x)$

b. food (apple)

c. food (vegetables)

d. $\neg eats(y, z) \vee killed(y) \vee food(z)$

e. eats (Anil, peanuts).

f. alive (Anil)

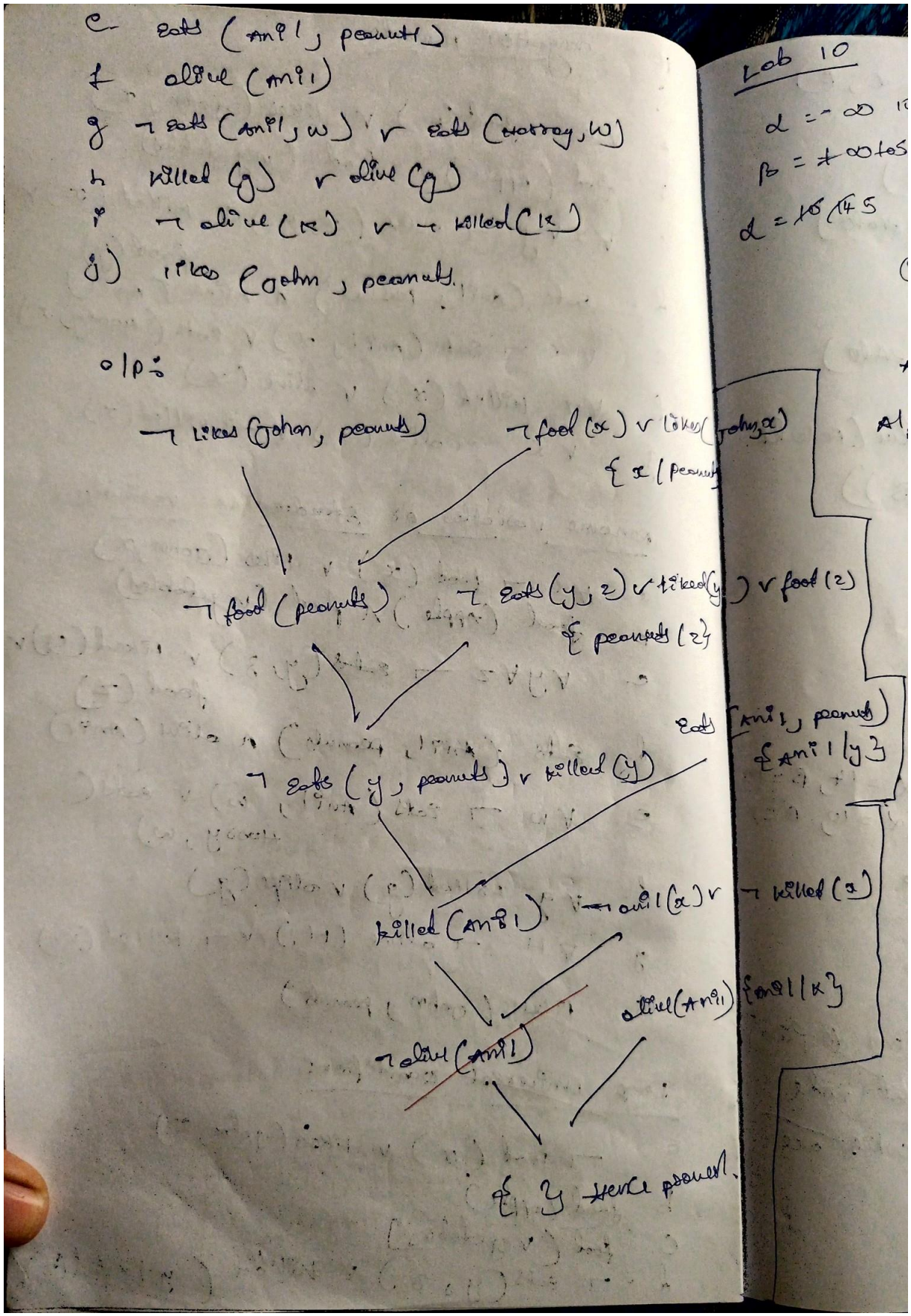
g. ¬ eats (Anil, w) ∨ eats (Harry, w)

h. killed (g) ∨ alive (g)

i. ¬ alive (k) ∨ ¬ killed (k)

j) likes (John, peanuts.

o/p:



Lab 10

α = −∞ to

β = +∞ to 5

α = 10 (14 5

Program 10

Implement Alpha-Beta Pruning.

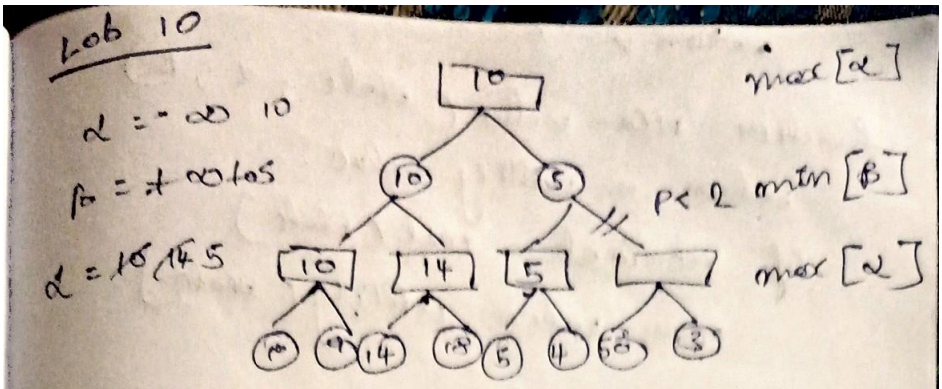
```
import math

def alpha_beta(node, depth, alpha, beta, maximizingPlayer, game_tree):


    if depth == 0 or isinstance(game_tree[node], int):
        return game_tree[node]

    if maximizingPlayer:
        maxEval = -math.inf

        for child in game_tree[node]:
            eval = alpha_beta(child, depth - 1, alpha, beta, False,
game_tree)

            maxEval = max(maxEval, eval)

            alpha = max(alpha, eval)

            if beta <= alpha:
                break

        return maxEval

    else:
        minEval = math.inf

        for child in game_tree[node]:
            eval = alpha_beta(child, depth - 1, alpha, beta, True, game_tree)

            minEval = min(minEval, eval)

            beta = min(beta, eval)

            if beta <= alpha:
                break
```

```
        return minEval


game_tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': 3,
    'E': 5,
    'F': 2,
    'G': 9
}

best_value = alpha_beta(
    'A', depth=3, alpha=-math.inf, beta=math.inf,
    maximizingPlayer=True, game_tree=game_tree
)

print("Best value for maximizer:", best_value)
```

output:

```
Enter leaf values separated by space:  -10 8 4 3 -2 -7

====================
       GAME TREE
====================

        (MAX)
    (MIN)        (MIN)
  (MAX)    (MAX)    (MAX)

Leaf Values:
[-10, 8, 4, 3, -2, -7]

====================
 Alpha-Beta Result
====================
Root Value = -2
====================

Code By : Uday Shankar Y : 1BM24CS427
```

Lab 10



Algorithm

Alpha Beta Search (state)
return an action
$v \leftarrow$ MAX-VALUE (state, $-\infty, +\infty$)

return the action in Actions(state)
with value v

function MAX-value (state, α, β)
returns a utility value
if Terminal-Test (state)
then return utility (state)

$v \leftarrow -\infty$

for each a in Actions (state)
do

$v \leftarrow$ MAX (v, MIN-value (Result (s, a), α, β))

if $v \geq \beta$
then return v
$\alpha \leftarrow$ MAX (α, v)

return v

function MIN-VALUE (state, α, β)
return a utility value.
if TERMINAL-TEST (state)
then return utility (state)

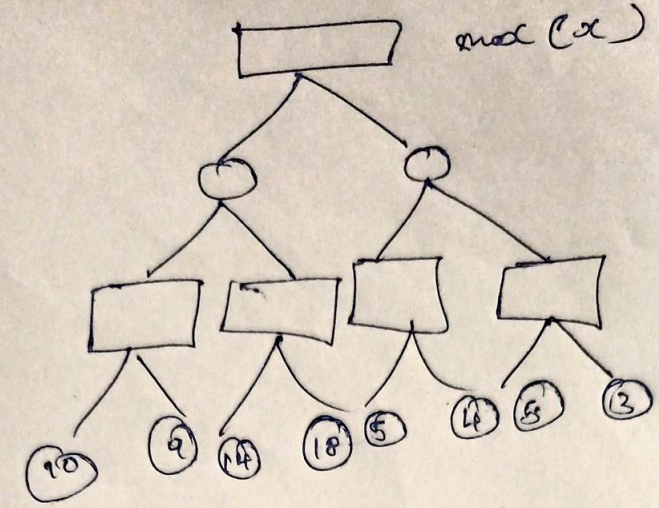
$v \leftarrow +\infty$
for each a in ACTIONS (state)
do
$v \leftarrow \min(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$
$\beta \leftarrow \min(\beta, v)$
return v.

o/p:

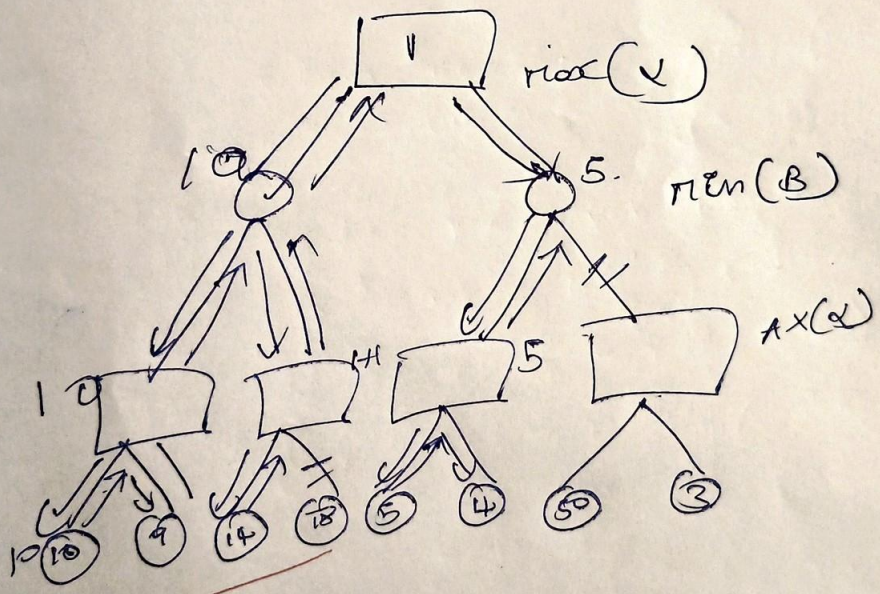
Alpha - beta pruning process.
pruned at depth 2, node 1, $\alpha = 14$, $\beta = 10$
pruned at depth 1, node 1, $\alpha = 10$, $\beta = \infty$
optimal value (root node): 10

problems

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



Ans
5 | 10 | 5