

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

Sl. No.	Date	Experiment Title	Page No.
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	5
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	15
3	14-10-2024	Implement A* search algorithm	27
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	36
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	41
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	44
7	2-12-2024	Implement unification in first order logic	51
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	57
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	62
10	16-12-2024	Implement Alpha-Beta Pruning.	71

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
2	25/8/25	VACUUM CLEANER	10	25-8-
3a	25/8/25	BFS without heuristic	10	25-
3b	25/8/25	BFS with heuristic	10	25-
3c	25/8/25	Iterative Deeply DFS	10	10
4	8/9/25	A* search, Ori placed	10	8
5	15/9/25	Naveen's way Hill climbing	10	15
6	15/9/25	Simulated Annealing	10	88
7	22/9/25	Propositional Logic	88	..151.
8	6/10/25	Unification	22/9/25	
9	13/10/25	Foloder logic	22/9/25	88
10	.	Resolution in FOL	10	10
11		The Alpha-beta Algo	31/10/25	

Circular

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)

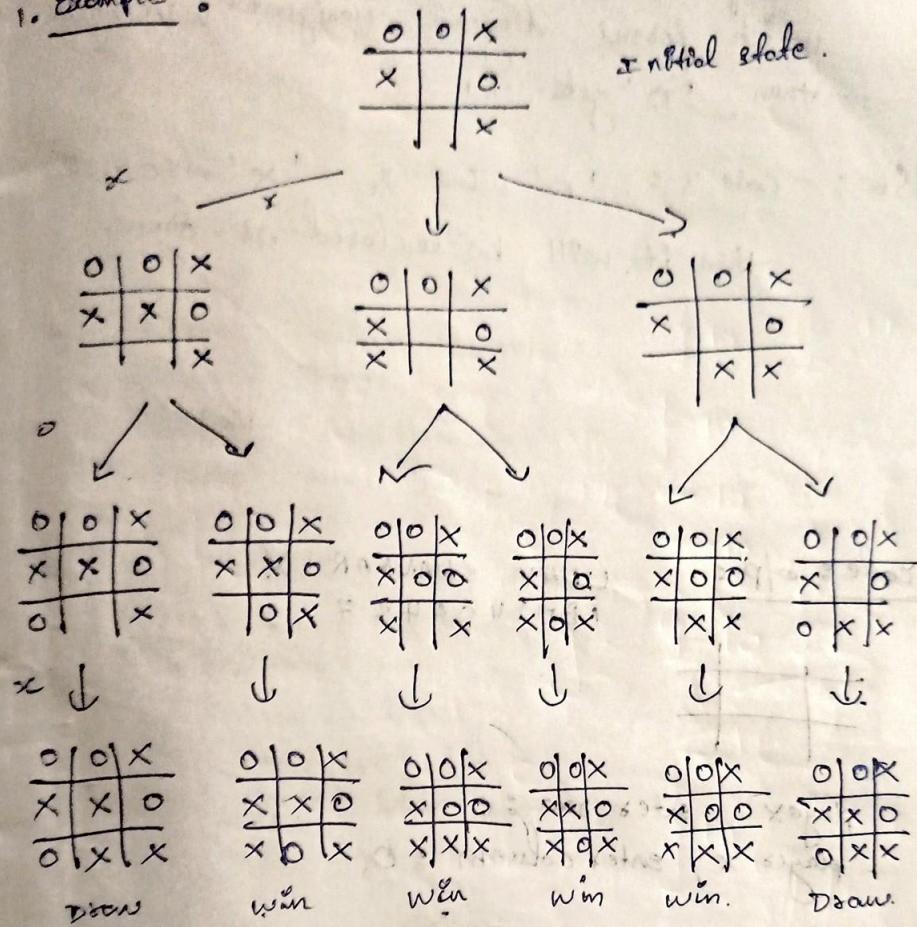
            if check_win(board, player):
                print(f"Player {player} wins!")
                break
            elif all([cell != " " for row in board for cell in row]):
                print("It's a tie!")
                break
            else:
                turn += 1
        else:
            print("That 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.

Step 1 : Start Accepting the input from the user whether 'X' or 'O'

Step 2 : or we can start from the initial state

Step 3 : if it forms same letter 'X' or 'O' diagonally, vertically, horizontally then that player wins.

Step 4 : Case 1 : played + wins (X)

If 'X' is placed horizontally, vertically or diagonally then 'X' gets +1

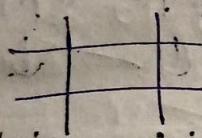
S5 : case 2 : 'O' wins (player 2)

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

S6 : case 3 : 'O' wins & 'X' also wins.
then it will be declared as draw.

✓
draw

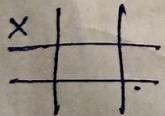
code o/p :



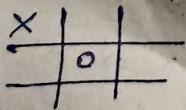
UDAY SHANKAR Y.

(BM24CSE427).

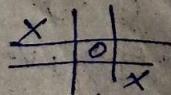
player X, enter row : 0.
player X, enter column : 0.



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



player X, enter row : 2.
player X, enter column : 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("\nCost: {cost}")
print(rooms)
```

```
Uday Shankar Y
IBM24CS427
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 D^{ot})
- * move to \leftarrow left
- * move to \rightarrow right
- * move to \uparrow up
- * move to \downarrow down.

S3 : check-Dirt() \rightarrow true

clean_Room() \rightarrow true
check All room cleaned() \rightarrow false

S4 : move to right

check_Dot() \rightarrow true

clean_Room() \rightarrow true
check All rooms cleaned() \rightarrow false

S5 : move to down.

check_Dirt() \rightarrow true..

clean_Room() \rightarrow true.

check All rooms cleaned() \rightarrow false.

S6 : move left

check_Dot() \rightarrow false.

check All rooms cleaned \rightarrow true

end program (stop)

if check Dot \rightarrow true

Then only check clean_Room()

O/P : Name: UDAY SHANKAR Y

USN : IBM 24C8427

Current room : A

Room status : { 'Room A': 0, 'Room B': 0 }

Total cost so far : 0

Enter action (clean / move / clean and move)

: move

Moving to room B

Current Room : Room B

Room status : { 'Room A': 0, 'Room B': 0 }

Total cost so far : 0

Enter action (clean / move / clean and move)

: clean

Cleaning Room B - - -

Room B is clean
now

All doors are clean ! & simulation finished

Total cost of cleaning : 4

8/8

varic

* 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): bfs

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 a puzzle without heuristic

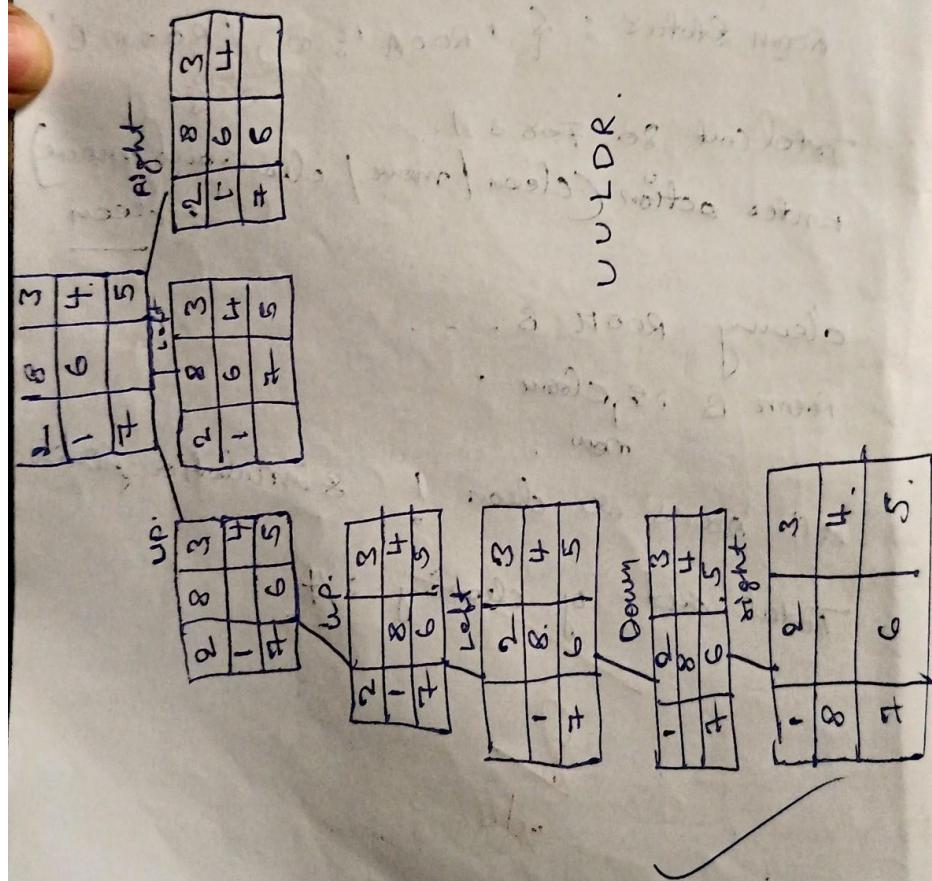
DFS.
BFS.

2	8	3
1	6	4
7		5

initial

1	2	3
8		4
7	6	5

goal



Step 1

S1: take input the initial state.

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

S3: if you got goal state stop.

For DFS

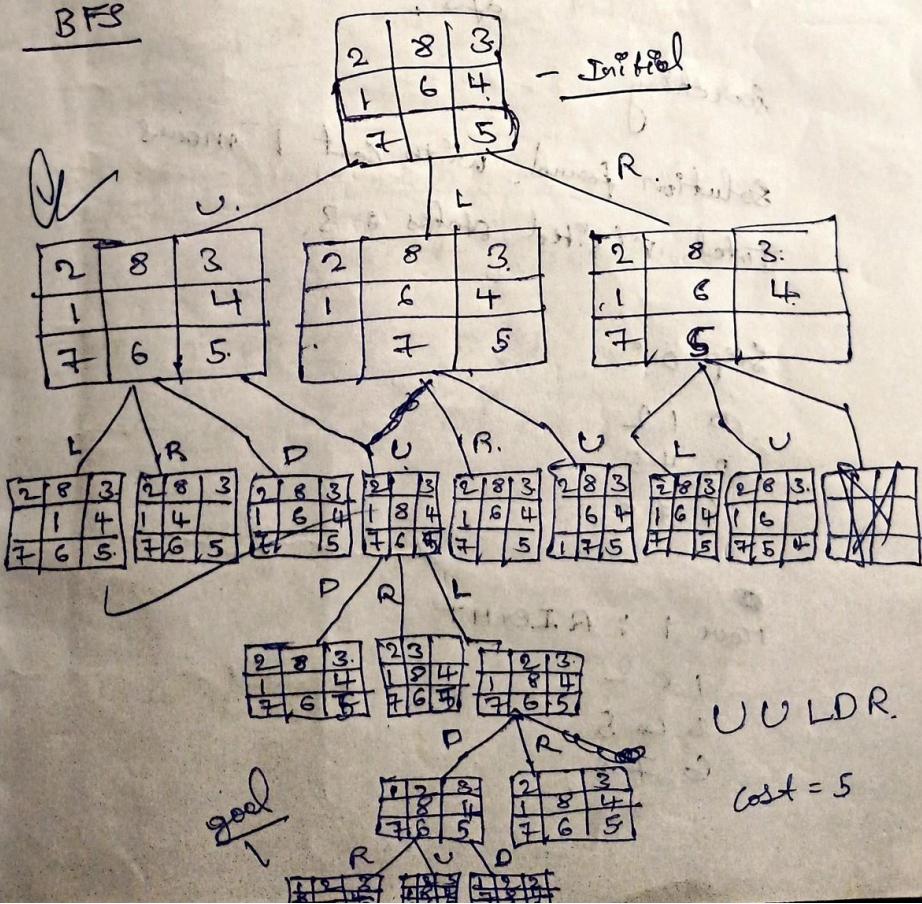
first traverse in left most to get goal state.

For BFS

traverse level by level to reach goal state.

S5: repeat until you reach goal.

BFS



O/P :-

UDAY SHANKAR Y.

ADM 24CS427

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

1 0 1 2

3 4 5

6 7 8

Enter the goal state

1 0 2

3 4 5

6 7 8

Enter search method (bt8 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 limit $d=0$

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 get solution.

$$\begin{array}{l}
 \text{dp} = 0 + p \\
 \text{dp} = 1 + q \\
 \text{dp} = 0 + r \\
 \text{dp} = 1 + s
 \end{array}$$

dp:

1 2 3

4 0 5

6 7 8

↓

1 2 3

4 5 0

6 7 8

↙ ↘

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

2 0 0 + 2 0

1 2 3 + 1 0 3

4 7 5 + 4 2 3

6 0 8 + 6 7 8

1 2 3

0 4 5

6 7 8

1 2 3

4 5 0

6 7 8

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 = heapq.heappop(frontier)
        if current.state == goal:
            break
        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(current.state)

                new_board[current.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos], new_board[current.zero_pos]
                new_state = PuzzleState(tuple(new_board), goal, current.path + move, current.cost + 1)

                if new_state.board not in explored:
                    heapq.heappush(frontier, new_state)
                    parent_map[new_state.board] = current.state
                    move_map[new_state.board] = move
                    explored.add(new_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.
```

App. by A* Algorithm

replaced
+led.

2	8	3
1	6	4
7		5

I

Manhattan
Distance

1	2	3
8		4
7	6	5

F.

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

2	8	3
1	6	4
7		5

$$g = 0.$$

2	8	3
1	6	4
7	5	

2	8	3
1	6	4
7	6	5

2	8	3
1	6	4
7	5	

$$1+2+0+0+0+0 \\ +0+1$$

$$h = 5$$

$$1+2+4+5+6+8 \\ +1+0+0+0+0+0+2$$

$$h = 4$$

$$1+2+3+4+5+6+8 \\ +1+0+0+0+1+1+2$$

$$h = 6$$

2	8	3
1	4	
7	6	5

$$1+2+3+4+5+6+8 \\ +1+0+1+0+0+0+2$$

$$= 5$$

2	3	
1	8	4
7	6	5

$$1+2+3+4+5+6+8 \\ +0+0+0+0+0+0+1$$

$$= 3$$

2	8	3
1	6	4
7		5

$$1+2+3+4+5+6+8 \\ +1+0+0+0+0+0+2$$

$$= 5$$

R | L | D

2	3	
1	8	4
7	6	5

$$h = 4$$

R | L | D

2	8	3
1	8	4
7	6	5

$$1+2+3+4+5+6+8 \\ +1+0+0+0+0+0+2$$

$$= 5$$

R

Möbius Distance
replaced tiles.

1	5	8
3	2	.
4	6	7

I

1	2	3
4	5	6
7	8	.

F

1	5	8
3	2	.
4	6	7

$$0 + 1 + 3 + 1 + 1 + 2 \\ + 2 + 3$$

$$h = 8$$

1	5	8
3	2	.
4	6	7

1	5	8
3	2	.
4	6	7

1	5	8
3	2	7
4	6	6

$$1 + 1 + 1 = 8$$

$$1 + 1 + 1 + 1$$

$$1 + 0$$

$$0 + 1 + 3 + 1 + 1 + 2 + 2 + 2 = 14$$

$$h = 14$$

$$h = 8$$

$$0 + 2 + 3 + 1 + 1 + 2 + 2 + 2 = 14$$

$$2 + 3$$

$$h = 8$$

$$14$$

$$h = 12$$

$$L$$

$$D$$

$$1$$

$$5$$

$$3$$

$$2$$

$$8$$

$$4$$

$$6$$

$$7$$

$$1$$

$$5$$

$$3$$

$$2$$

$$7$$

$$4$$

$$6$$

$$7$$

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

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

Algorithm for manhattan Distance

Start : initialize start node, with $g=0$,
 calculate $h = \text{sum of manhattan distances}$
 of tiles goal, $f=g+h$ put it in open list.

open list

Select : pick node with lowest f
 from open list if goal node.

Expand :- generate child states for each
 child $g = \text{parent } g + 1$

$h = \text{sum of manhattan distance}$

1	2	3	4	5	6	7	8	9
3	1	5	2	4	6	7	8	9
2	4	6	8	0	*	5	7	9
7	8	9	6	4	2	1	3	5

Op:

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
7	6

Step 3 :

2	3
1	8 4
7	6 5

Step 4 :

1	2 3
8	4
7	6 5

Step 5 :

1	2 3
8	4
7	6 5

Algorithm of misplaced field

* sum of manhattan of misplaced just + 1
for heuristic.

O/P:

step by step solution.

Step 0 : Initial.

2	8 3
1	6 4
7	5

Step 1 : Moves : U

2	8 3
1	4
7	6 5

Step 2 : Moves : UV

2	0 3
1	8 4
7	6 5

Step 3 : Rows : UUL

0 2 3

1 8 4

7 6 5

Step 4 : Rows : UULD

1 2 3

8 4

7 6 5

Step 5 : Rows : UULDD.

1 2 3

8 4

12 6 5 } without row 10 and 11.

row 12 and 13.

Cost = 5,

U
15 9

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:

Enter the number of queens (N): 4

==== Restart 1 ====

Initial state (heuristic: 3):

```

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

Step 1: (heuristic: 1)

```
. . Q .
```

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

Reached local minimum at step 1, heuristic: 1
+ No solution in this attempt (local minimum).

==== Restart 2 ====

Initial state (heuristic: 3):

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

Step 1: (heuristic: 1)

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

Reached local minimum at step 1, heuristic: 1
+ No solution in this attempt (local minimum).

==== Restart 3 ====

Initial state (heuristic: 2):

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

Step 1: (heuristic: 1)

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

Step 2: (heuristic: 0)

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

Reached local minimum at step 2, heuristic: 0
Solution found after 3 restart(s):

. Q ..

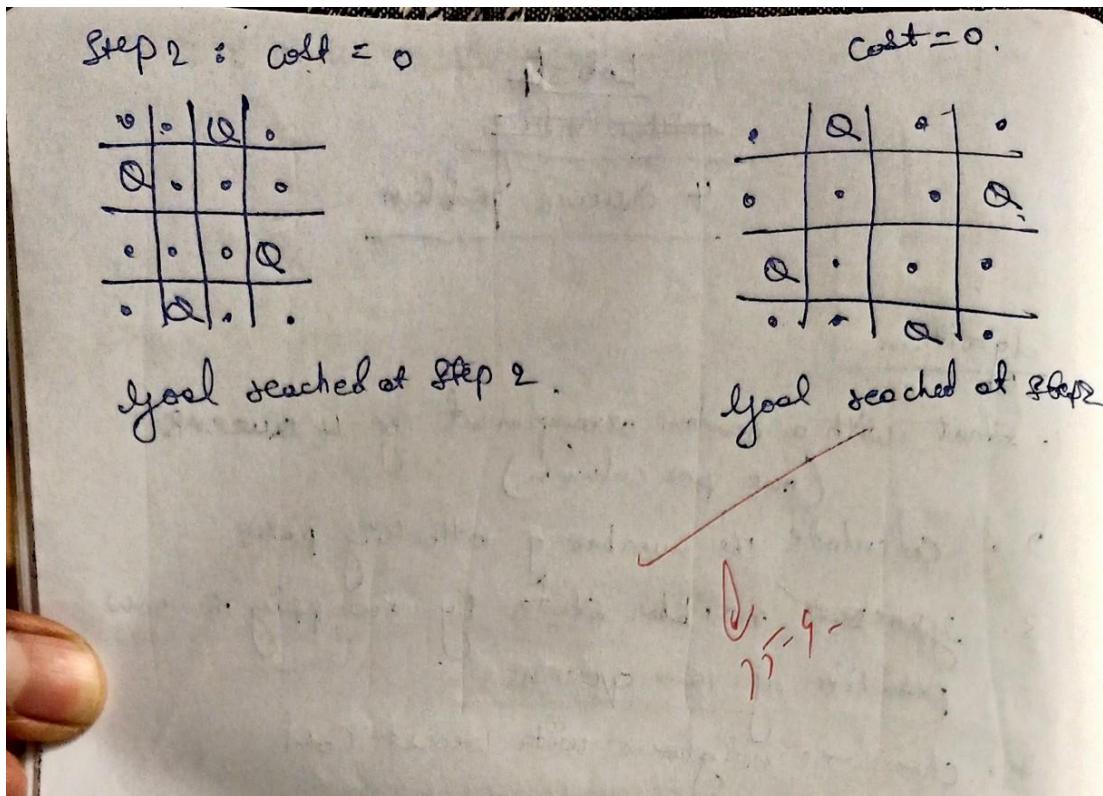
... Q

Q ...

.. Q .

1BM24CS427

Uday Shankar Y



Lab-5
With 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. Repeat steps 2-5 until no better neighbor.
7. If cost = 0, solution is found - otherwise local maximum is reached.

Q.P:

Initial Board

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

cost = 2.

Q	.	.	.
.	-	Q	0
0	Q	.	.
0	0	0	Q

Cost = 2.

Step 1 : cost = 1

.	.	.	Q
Q	.	.	0
.	0	Q	.
0	Q	0	0

Q	0	0	0
0	0	0	Q
0	Q	0	0
0	0	Q	0

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:

```

Enter number of queens (N): 8
Solution found at step 675 Position format:
[ 3 0 4 7 5 2 6 1 ]
Heuristic: 0 1BM24CS427
Uday shankar Y

```

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.energy - next.energy  
    if Δ E > 0 then  
        current ← next  
    else  
        current ← next with probability  $e^{\frac{-\Delta E}{T}}$   
end if  
end while  
return current
```

Output:

The best position found is : [5, 1, 4, 7, 5, 9, 2]
The number of queens that are not attacking
each other is : 5

88
1 5 9 2 4

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):
        return Not(self)

    def and_(self, other):
        return And(self, other)

    def or_(self, other):
        return Or(self, other)

    def rshift(self, other):
        return Implication(self, other)

    def eq(self, other):
        return Biconditional(self, other)

    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:

Knowledge Base: (P | (Q & P))
Query: (Q | P)

P | Q | KB | Query

False | False | False | False
False | True | False | True
True | False | True | True
True | True | True | True

Does KB entail Query? True

propositional logic on Knowledge Based

week-6

Semantics

Truth Table for connectives.

P	$\neg P$	$\rightarrow P$	$P \wedge Q$	$P \vee Q$	$P \Leftrightarrow Q$
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 $KB \models Q$

A	B	C	$\neg C$	$A \vee C$	$B \vee \neg C$	KB	Q
F	F	F	T	F	T	F	F
F	F	T	F	T	F	F	T
F	T	F	T	F	T	T	T
F	T	T	F	T	T	T	T
T	F	F	T	T	F	F	T
T	F	T	F	T	T	T	T
T	T	F	T	T	F	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 & Q :

Ex:- A, B, C

② Try every possibility.

- Each symbol can be true or false
- So we test all combination.

③ Check $\models KB$

for each combination, see if $\models KB$ is true

④ check $\models Q$

- If $\models KB$ is true, then $\models Q$ 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 \rightarrow KB entails α .
- * if in any case KB is true, but false \rightarrow KB does not entail α .

Q/F %.

KB = NOT
Enter query (alpha) : T.

Truth table

not	KB	alpha
T	F	F
F	T	F
F	F	T
F	F	F

~~KB Entails Alpha : NO.~~

~~default is false~~

~~SPD
2/19/25~~

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:
    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:
        if term1 in substitutions:
            return unify(substitutions[term1], term2,
                        substitutions)
        elif occurs_check(term1, term2):
            raise UnificationError(f"Occurs check fails: {term1} "
                                   f"in {term2}")
        else:
            substitutions[term1] = term2
            return unify(term1, term2, substitutions)
    # If term1 is a compound
    # (function term)
    else:
        # term2 must be a variable
        # or a compound
        # ...
        # Implementation details omitted for brevity
```

```

    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:

```
Most General Unifier (MGU):
{'Z': 'b', 'X': ('f', 'Y'),
 'Y': ('g', 'Z')}
```

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 { (ψ_2 / ψ_1) }.

c) Else if ψ_2 is a variable,

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

b. Else return { (ψ_1 / ψ_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 σ_1 (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 result
into σ_1 .

- b) If $\delta = \text{failure}$ then returns failure.
- c) If $\delta \neq \text{NIL}$ then do,
- Apply δ to the remainder of both L1 & L2
 - $\text{SUBST} \leftarrow \text{APPEND}(\delta, \text{SUBST})$

step. 6: Return SUBST

Solve the following.

Q) Find most general unifier (mgu) of
 $\{\alpha(a, g(x, a), f(y)) \text{ and } \alpha(a, g(f(b), a), x)\}$.

$$\Rightarrow \theta = x/f(b)$$

unify $\{\alpha(a, g(f(b), a), f(y))\}$

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

unify $\{\theta(\alpha(a, g(f(b), a), x)), \alpha(a, g(f(b), a), x)\}$

2) unify $\{\text{prime}(x) \text{ and prime}(y)\}$

$$\theta = x/y.$$

unify $\{\text{prime}(y) \text{ and prime}(y)\}$

3) ~~unify $\{\text{knows}(\text{John}, x), \text{knows}(y, \text{Bill})\}$~~

$$\theta = y \mid \text{John}$$

~~$\text{knows}(\text{John}, x), \text{knows}(\text{John}, \text{Bill})$~~

$$\theta = x \mid \text{Bill}$$

~~$\text{knows}(\text{John}, \text{Bill}), \text{knows}(\text{John}, \text{Bill})$~~

4 find new of $\{P(f(a), g(y)), P(x, z)\}$
 $\theta = f(a) \rightarrow x$

$\{P(x, g(x)), P(x, z)\}$

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)

Final Facts:

Weapon(T1)
Criminal(Robert)
Hostile(A)
Sells(Robert, T1, A)
Missile(T1)
Enemy(A, America)
Owns(A, T1)
American(Robert)

First order logic

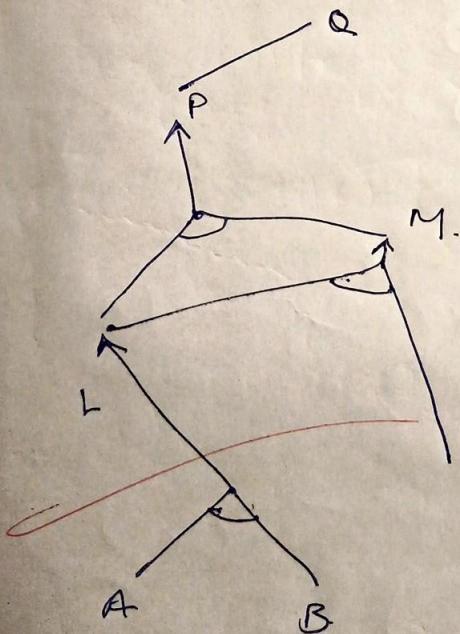
Lab-8

principles

$$\begin{array}{l} P \Rightarrow Q \\ L \wedge M \Rightarrow P \\ B \wedge L \Rightarrow M \\ A \wedge P \Rightarrow L \\ A \wedge B \Rightarrow L \end{array} \quad \left. \begin{array}{l} \\ \\ \\ \\ \end{array} \right\} \text{Rules.}$$

A Y-
B B facts.

Prove Q.



function FOL-FC-ASK (KB, α) return a substitute
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 $\leftarrow \emptyset$

for each rule in KB do

$(P_1 \wedge \dots \wedge P_n \Rightarrow q) \leftarrow \text{standardize-variables (rule)}$

for each θ such that $\text{SUBST}(\theta, P_1 \wedge \dots \wedge P_n) = \text{SUBST}(\theta, P'_1 \wedge \dots \wedge P'_n)$

for some P'_1, \dots, P'_n in KB .

$q' \leftarrow \text{SUBST}(\theta, q)$

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

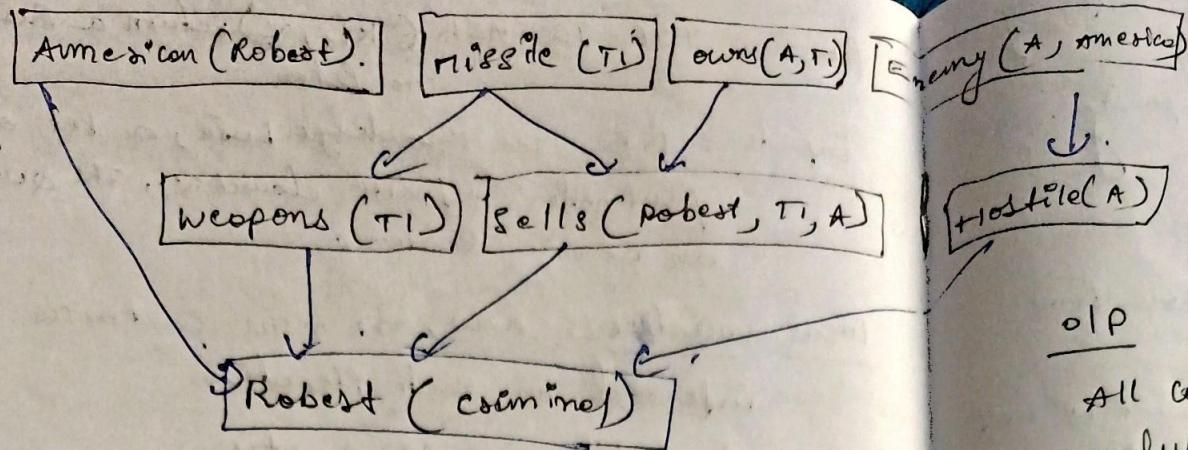
add q' to new

$\phi \leftarrow \text{unify}(q', \alpha)$

if ϕ is not fail then return ϕ

add new to KB

return false.



1. American (P) \wedge weapon (y) \wedge sells (x, y, z) \wedge hostile (z) \Rightarrow criminal (x).

~~13/12/25~~
2. \exists x missile (x) \wedge owns (A, x)
Essential instantiation, introduce a new constant $T1$:
owns ($A, T1$).
missile ($T1$).

~~13/12/25~~
3. All of the missiles ~~were~~ were sold to country A by Robert.
 ~~$\forall x$ missile (x) \wedge owns (A, x) \Rightarrow sells (Robert, x, π).~~

~~13/12/25~~
4. Missiles are weapons.
missile (x) \Rightarrow weapon (x).

~~13/12/25~~
5. Enemy of America is known as
 ~~$\forall x$ Enemy ($x, America$) \Rightarrow hostile (x)~~

~~13/12/25~~

enemy (A, America)

↓

Hostile (A)

Robert is an American
American (Robert)

olp

All conditions met Robert is criminal.
conclusion: Robert is criminal.

88
13/12/25

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.")


```

```

Uday shankar Y(1BM@$CS427)
Enter number of premises: 4
Enter clause 1 (e.g., A, ¬B, C): A
Enter clause 2 (e.g., A, ¬B, C): ¬A
Enter clause 3 (e.g., A, ¬B, C): -B
Enter clause 4 (e.g., A, ¬B, C): c
Enter conclusion (e.g., C or ¬C): c

--- Initial Clauses ---
C1: {A}
C2: {-B}
C3: {c}
C4: {-A}
C5 (¬Conclusion): {¬c}


```

```
--- Resolution Steps ---
C6 = RESOLVE(C3, C5 ( $\neg$ Conclusion) ) -> set()
```

```
Empty clause derived! (C6)
```

```
--- Resolution Tree ---
```

```
Conclusion can be proven from the premises.
```

Resolution in FOL

1. Eliminate ~~substitution~~ 2 implications
- Eliminate \hookrightarrow replacing $\alpha \Rightarrow p$ with $(\alpha \cup B) \wedge (\neg \alpha \Rightarrow p)$
 - Eliminate \Rightarrow replacing $\alpha \Rightarrow B$ with $\neg \alpha \vee B$

2. Now \rightarrow for models.

$$\begin{aligned} \neg(\forall x p) &\equiv \exists x \neg p \\ \neg(\exists x p) &\equiv \forall x \neg p \\ \neg(\exists x \vee B) &\equiv \neg \exists x \wedge \neg B \\ \neg(\exists x \vee B) &\equiv \neg \exists x \vee \neg B \\ \neg \neg \exists x &\equiv \exists x \end{aligned}$$

3. Standardize variables apart by saying they
each quantifies 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 simple inference rule which
can efficiently operate on the conjunctive normal
form or clauses form

4. Generalize: each constant of variable is replaced by a symbol constant or symbol. function of the enclosing universally quantified variables

• for instance, $\exists x P(x) \rightarrow h(x)$ becomes $(\forall x) h_1$ where h_1 new constant

5. Drop universal quantifiers

• $\forall person(x) \rightarrow person(x)$

6. Distribute \wedge over \vee :

• $(\exists x A \wedge B) \vee C \rightarrow (\exists x A \vee C) \wedge (\exists x B \vee C)$

Algorithm for Resolution

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

call ExpelSet, in sol :

1. convert all sentences to CNF

2. negate conclusion & to premise clause

3. add negated conclusion & to the premise clauses

4. repeat until contradiction or no progress
in method

a. select 2 clauses call them precl & claud

b. resolve them together, performing all required unifications

c. if resultant is the empty clause, a contradiction has been found (\therefore , & follows

from the premises.

- d. if not, add refuted to the premises
if not succeed in step m, we have proved the conclusion.

Representation of FOL

- a. $\forall x : \text{food}(x) \rightarrow \text{Likes}(\text{John}, x)$
- b. $\text{food}(\text{apple}) \wedge \text{food}(\text{vegetables})$
- c. $\forall x \forall y : \text{eats}(x, y) \wedge \neg \text{killed}(x) \rightarrow \text{Food}(y)$
- d. $\text{eats}(\text{Anil}, \text{peanuts}) \wedge \text{alive}(\text{Anil})$
- e. $\forall x : \text{eats}(\text{Anil}, x) \rightarrow \text{eats}(\text{Hosday}, x)$
- f. $\forall x : \neg \text{killed}(x) \rightarrow \text{alive}(x)$
- g. $\forall x : \text{alive}(x) \rightarrow \neg \text{killed}$
- h. $\text{Likes}(\text{John}, \text{peanuts})$

Proof by resolution

Eliminate implication.

- a. $\forall x : \text{food}(x) \rightarrow \text{Likes}(\text{John}, x)$
- b. ~~$\text{food}(\text{apple}) \wedge \text{food}(\text{vegetables})$~~
- c. $\forall x \forall y : \text{eats}(x, y) \wedge \neg \text{killed}(x) \rightarrow \text{Food}(y)$
- d. $\text{eats}(\text{Anil}, \text{peanuts}) \wedge \text{alive}(\text{Anil})$
- e. $\forall x : \text{eats}(\text{Anil}, x) \rightarrow \text{eats}(\text{Hosday}, x)$
- f. $\forall x : \neg \text{killed}(x) \rightarrow \text{alive}(x)$

g. $\forall x \rightarrow \text{alive}(x) \rightarrow \exists y \text{ killed}(x)$

h. likes (John, peanuts)

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

a. $\forall x \rightarrow \text{food}(x) \vee \text{likes}(\text{John}, x)$

b. food (apple) \wedge food (vegetables)

c. $\forall x \forall y \rightarrow [\text{eats}(x, y) \wedge \neg \text{killed}(x)]$

$\vee \text{food}(y)$

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

e. $\forall x \rightarrow \text{eats}(\text{Anil}, x) \vee \text{eats}(\text{HARRY}, x)$

f. $\forall x \rightarrow [\neg \text{killed}(x)] \vee \text{alive}(x)$

g. $\forall x \rightarrow \text{alive}(x) \vee \neg \text{killed}(x)$

h. likes (John, peanuts)

eliminate implicator $\alpha \rightarrow \beta$ with $\rightarrow \alpha \vee \beta$

a. $\forall x \rightarrow \text{food}(x) \vee \text{likes}(\text{John}, x)$

b. food (apple) \wedge food (vegetables)

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

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

e. $\forall x \rightarrow \text{eats}(\text{Anil}, x) \vee \text{eats}(\text{HARRY}, x)$

f. $\forall x \rightarrow [\neg \text{killed}(x)] \vee \text{alive}(x)$

g. $\forall x \rightarrow \text{alive}(x) \vee \neg \text{killed}(x)$

h. likes (John, peanuts)

more negation in words.

- a. $\forall x \rightarrow \text{food}(x) \vee \text{likes}(\text{John}, x)$
- b. $\text{food}(\text{apple}) \wedge \text{food}(\text{vegetables})$
- c. $\forall x \forall y \rightarrow \text{eats}(x, y) \vee \text{killed}(x) \vee$
 $\text{eats}(\text{Anil}, \text{peanut}) \wedge \text{alive}(\text{Anil})$
- d. $\forall x \rightarrow \text{eats}(\text{Anil}, x) \vee \text{eats}(\text{Hobby}, x)$
- e. $\forall x \text{ killed}(x) \vee \text{alive}(x)$
- f. $\forall x \rightarrow \text{alive}(x) \vee \rightarrow \text{killed}(x)$

same variables or standardise variables

- a. $\forall x \rightarrow \text{food}(x) \vee \text{likes}(\text{John}, x)$
- b. $\text{food}(\text{apple}) \wedge \text{food}(\text{vegetables})$
- c. $\forall y \forall z \rightarrow \text{eats}(y, z) \vee \text{liked}(y) \vee$
 $\text{food}(z)$
- d. $\text{eats}(\text{Anil}, \text{peanuts}) \wedge \text{alive}(\text{Anil})$
- e. $\forall w \rightarrow \text{eats}(\text{Anil}, w) \vee \text{eats}(\text{Hobby}, w)$
- f. $\forall g \text{ killed}(g) \vee \text{alive}(g)$
- g. $\forall h \rightarrow \text{alive}(h) \vee \rightarrow \text{killed}(h)$
- h. $\text{likes}(\text{John}, \text{peanuts})$

Deep universal quantifier

- a. $\rightarrow \text{food}(x) \vee \text{likes}(\text{John}, x)$
- b. $\text{food}(\text{apple})$
- c. $\text{food}(\text{vegetables})$
- d. $\rightarrow \text{eats}(y, z) \vee \text{killed}(y) \vee \text{food}(z)$

c) eats (ampl, peanuts)

f) olive (mii)

g) ~ eats (ampl, w) v eats (hotdog, w)

h) killed (g) v olive (g)

i) ~ olive (x) v ~ killed (x)

j) likes (john, peanuts)

o/p:

→ likes (john, peanuts)

→ foot (x) v likes (john, x)

{ x (peanut)

→ food (peanuts)

→ eats (y, z) v killed (y) v foot (z)

{ peanuts (z)

→ eats (y, peanuts) v killed (y)

eats

{ amii, peanut
{ amii, ly3

→ foot (x) v (x)

→ killed (amii)

→ foot (x) v

→ killed (x)

→ foot (x) v

→ killed (amii)

{ amii, x }

→ olive (amii)

→ olive (amii)

→ olive (amii)

{ y, hence peanut.

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
}

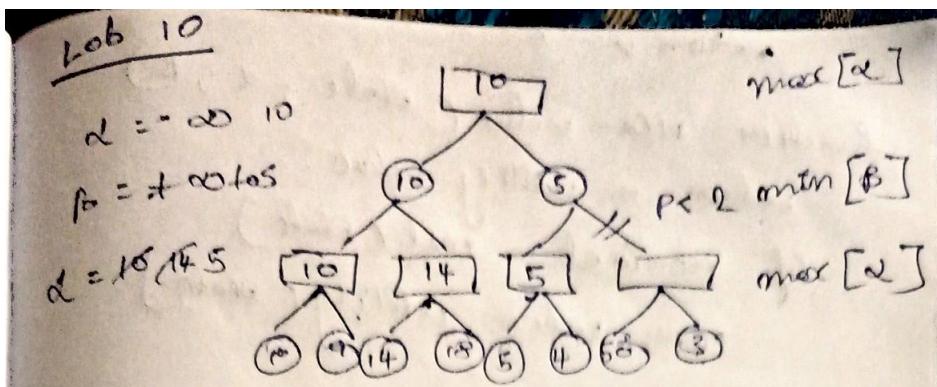
best_value = alpha_beta(
    'A', depth=3, alpha=-math.inf, beta=math.inf,
    maximizingPlayer=True, game_tree=game_tree
)

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

```

Output:

Best value for maximizer: 3



Algorithm

(stage)

```

    alphaBetaSearch(state)
        return actions(state)
        v ← MAX-VALUE(state, -∞, +∞)
    } v foot(z)
        return the action in actions(state)
        with value n
    
```

(init, parent)
 { init / y }

```

        function MAX-VALUE(state, α, β)
            returns a utility value :
            if Terminal-test(state)
                then return utility(state)
            do
                v ← -∞
                for each a in actions(state)
                    v ← MAX(v, MIN-VALUE(result
                        (s, a), α, β))
                if v ≥ β
                    then return v
                α ← MAX(α, v)
    
```

~ killed(a)

{ final / x }

return v
function max-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

$v \leftarrow \min(v, \text{max-value}(\text{result},$
 $(s, a), \alpha, \beta))$

$B \leftarrow \min(B, v)$

return v .

Op:

Alpha-Beta pruning process.

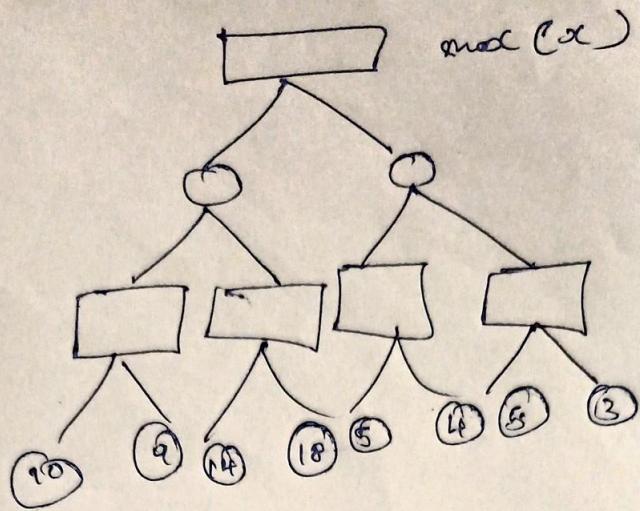
pruned at depth 2, node 1, $\alpha = 14, \beta = 10$

pruned at depth 1, node 1, $\alpha = 10, \beta = 7$

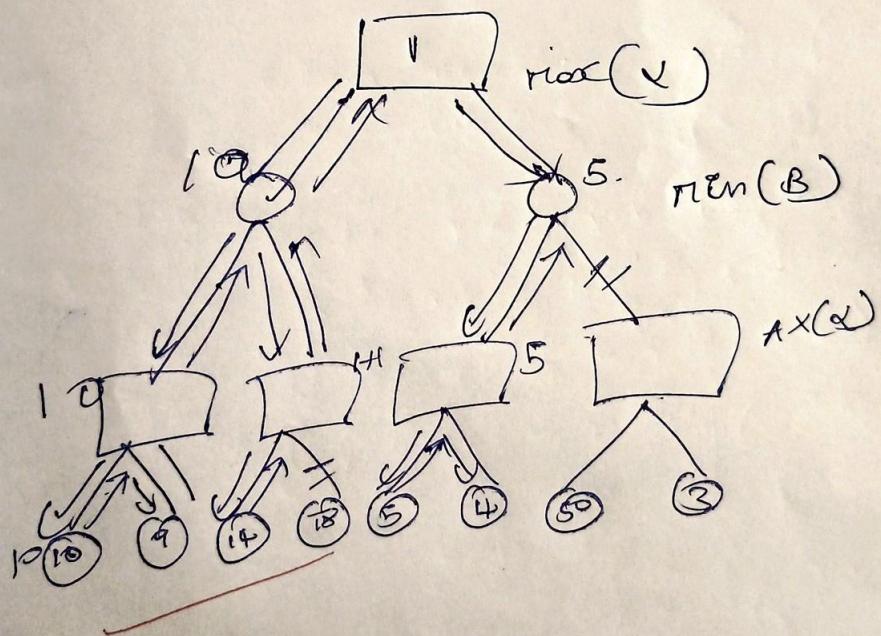
optimal value (root node): 10

problem

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



Solution



DUO
SIT 10/13