VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Siddhant Sawhney (1BM23CS327)

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 **Siddhant Sawhney (1BM23CS327)** 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.

Surabhi S 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	4-12
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	13-24
3	14-10-2024	Implement A* search algorithm	25-29
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	30-33
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	34-36
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	37-40
7	2-12-2024	Implement unification in first order logic	41-48
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	49-51
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	52-59
10	16-12-2024	Implement Alpha-Beta Pruning.	60-62

Github Link:
https://github.com/siddhantsawhney327/AI-

Program 1
Implement Tic –Tac –Toe Game

Algorithm:

Tic Tac Toe

18/8

Self initial assumption:-

Code:

Siddhant Sawhney 1BM23CS327 TTT.py

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

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

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):
            return True
        if all(board[j][i] == player for j in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def is_draw(board):
    return all(board[i][j] != '-' for i in range(3) for j in range(3))

def minimax(board, is_ai_turn):
    if check_winner(board, 'O'):  # AI win
        return 1
    if check_winner(board, 'X'):  # Player win
        return -1
    if is_draw(board):
        return 0

    if is_ai_turn:
        best_score = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == '-':
                    board[i][j] = 'O'
                    score = minimax(board, False)
                    board[i][j] = '-'
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(3):
```

Siddhant Sawhney 1BM23CS327 TTT.py

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
        else:
            best_score = float('inf')
            for i in range(3):
                for j in range(3):
                    if board[i][j] == '-':
                        board[i][j] = 'X'
                        score = minimax(board, True)
                        board[i][j] = '-'
                        best_score = min(score, best_score)
            return best_score

    def manual_game():
        board = [['-' for _ in range(3)] for _ in range(3)]
        print("Initial Board:")
        print_board(board)

        while True:
            # Input X move
            while True:
                try:
                    x_row = int(input("Enter X row (1-3): ")) - 1
                    x_col = int(input("Enter X col (1-3): ")) - 1
                    if board[x_row][x_col] == '-':
                        board[x_row][x_col] = 'X'
                        break
                    else:
                        print("Cell occupied!")
                except:
                    print("Invalid input!")

            print("Board after X move:")
            print_board(board)

            if check_winner(board, 'X'):
                print("X wins!")
                break
            if is_draw(board):
                print("Draw!")
                break


            while True:
```

Siddhant Sawhney 1BM23CS327 TTT.py

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
            except:
                print("Invalid input!")

        print("Board after X move:")
        print_board(board)

        if check_winner(board, 'X'):
            print("X wins!")
            break
        if is_draw(board):
            print("Draw!")
            break


        while True:
            try:
                o_row = int(input("Enter O row (1-3): ")) - 1
                o_col = int(input("Enter O col (1-3): ")) - 1
                if board[o_row][o_col] == '-':
                    board[o_row][o_col] = 'O'
                    break
                else:
                    print("Cell occupied!")
            except:
                print("Invalid input!")

        print("Board after O move:")
        print_board(board)

        if check_winner(board, 'O'):
            print("O wins!")
            break
        if is_draw(board):
            print("Draw!")
            break


        cost = minimax(board, True)
        print(f"AI evaluation cost from this position: {cost}")

manual_game()
```

Output:

Siddhant Sawhney 1BM23CS327 TTT.py

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
            cost = minimax(board, True)
            print(f"AI evaluation cost from this position: {cost}")

    manual_game()
```

```
Initial Board:
- - -
- - -
- - -

Enter X row (1-3): 3
Enter X col (1-3): 3
Board after X move:
- - -
- - -
- - X

Enter O row (1-3): 2
Enter O col (1-3): 2
Board after O move:
- - -
- O -
- - X

AI evaluation cost from this position: 0
Enter X row (1-3): 3
Enter X col (1-3): 1
Board after X move:
- - -
- O -
X - X

Enter O row (1-3): 2
Enter O col (1-3): 1
Board after O move:
- - -
O O -
X - X

AI evaluation cost from this position: 1
Enter X row (1-3): 2
Enter X col (1-3): 3
Board after X move:
- - -
O O X
X - X

Enter O row (1-3): 1
Enter O col (1-3): 3
Board after O move:
- - O
O O X
X - X

AI evaluation cost from this position: 0
Enter X row (1-3): 3
Enter X col (1-3): 2
Board after X move:
- - O
O O X
X X X

X wins!
```

Implement vacuum cleaner agent

Algorithm:

Code:

```python
def vacuum_cleaner():
    # Get input for room states
    state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    loc = input("Enter location (A or B): ").upper()

    cost = 0  # To count number of actions

    # Logic based on location
    if loc == 'A':
        if state_A == 1:
            print("Room A is dirty. Cleaning...")
            state_A = 0
            cost += 1
        else:
            print("Room A already clean.")
        print("Moving vacuum right")
        cost += 1
        if state_B == 1:
            print("Cleaned B.")
            state_B = 0
            cost += 1
        else:
            print("B already clean.")
    elif loc == 'B':
        if state_B == 1:
            print("Room B is dirty. Cleaning...")
            state_B = 0
            cost += 1
        else:
            print("Room B already clean.")
        print("Moving vacuum left")
        cost += 1
        if state_A == 1:
            print("Cleaned A.")
            state_A = 0
            cost += 1
        else:
            print("A already clean.")
    else:
        print("Invalid location entered.")
        return

    print(f"Cost: {cost}")
    print({'A': state_A, 'B': state_B})


# Run the simulation
vacuum_cleaner()
```

Output:

SIDDHANT SAWHNEY 1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
    print(f"Cost: {cost}")
    print({'A': state_A, 'B': state_B})


# Run the vacuum cleaner simulation
vacuum_cleaner()
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

SIDDHANT SAWHNEY 1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
    print(f"Cost: {cost}")
    print({'A': state_A, 'B': state_B})


# Run the vacuum cleaner simulation
vacuum_cleaner()
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Moving vacuum right
Cleaned B.
Cost: 2
{'A': 0, 'B': 0}
```

2. BFS wirh Heuristic

Algorithm:

Algo:-

1.) Start

2.) Initial & goal state are given

3.) misplaced tiles → tiles not in correct position

4.) Generate all possible next states based on direction

5.) Count misplaced tiles in each new state

6.) Heuristic value → how many not in place
pick the minimum and repeat steps until reach goal

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

25/7/25

Code:

SIDDHANT SAWHNEY 1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```python
import heapq

# Define goal state
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]   # 0 represents blank

# Heuristic function: number of misplaced tiles
def heuristic(state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced

# Find position of blank tile (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Generate next possible states
def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)

    # Possible moves: Up, Down, Left, Right
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Copy current state
            new_state = [row[:] for row in state]
            # Swap blank with neighbor
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

# Best First Search
def best_first_search(start_state):
    visited = set()
    pq = []

    # Priority queue with (heuristic value, state, path)
```

SIDDHANT SAWHNEY 1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```python
    # Priority queue with (heuristic value, state, path)
    heapq.heappush(pq, (heuristic(start_state), start_state, []))

    while pq:
        h, current, path = heapq.heappop(pq)

        # Convert state to tuple for visited check
        state_tuple = tuple(tuple(row) for row in current)
        if state_tuple in visited:
            continue
        visited.add(state_tuple)

        # Check goal
        if current == goal_state:
            return path + [current]

        # Expand neighbors
        for neighbor in get_neighbors(current):
            if tuple(tuple(row) for row in neighbor) not in visited:
                heapq.heappush(pq, (heuristic(neighbor), neighbor, path + [current]))

    return None  # No solution

# Print board nicely
def print_state(state):
    for row in state:
        print(row)
    print()

# Example usage
if __name__ == "__main__":
    # Initial state from your diagram
    start_state = [[1, 2, 3],
                   [0, 4, 6],
                   [7, 5, 8]]

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

    solution = best_first_search(start_state)

    if solution:
        print("Solution Path:")
        for step in solution:
            print_state(step)
    else:
        print("No solution found!")
```

Output:

SIDDHANT SAWHNEY 1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
        print("Solution Path:")
        for step in solution:
            print_state(step)
    else:
        print("No solution found!")
```

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

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

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

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

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

BFS wirhout Heuristic

Algorithm:

→ Algo BFS

↳ Write correctly defined goal state
↳ Take input in string
↳ BFS
↳ Move accordingly to valid moves & choose least misplaced tile for next BFS.
↳ Push each value, state, path taken into queue.
↳ Choose the least visited branch and move ahead
↳ If start state = goal state, calculate the number of moves.

Code:

1BM23CS327 bfs w/o heuristic.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```python
from collections import deque

goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
def to_tuple(state):
    return tuple([num for row in state for num in row])

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

def get_neighbors(state):
    x, y = find_blank(state)
    moves = []
    directions = [(0,1),(0,-1),(1,0),(-1,0)]  # Right, Left, Down, Up
    for dx, dy in directions:
        nx, ny = x+dx, y+dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            moves.append(new_state)
    return moves

def is_solvable(state):
    flat = [num for row in state for num in row if num != 0]  # ignore 0
    inversions = 0
    for i in range(len(flat)):
        for j in range(i+1, len(flat)):
            if flat[i] > flat[j]:
                inversions += 1
    return inversions % 2 == 0


def bfs(start):
    if not is_solvable(start):
        return None  # unsolvable puzzle

    queue = deque([(start, [])])  # (current_state, path)
    visited = set()

    while queue:
        current, path = queue.popleft()
```

1BM23CS327 bfs w/o heuristic.ipynb Saving...

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```python
        if current == goal_state:
            return path + [current]  # solution found

        visited.add(to_tuple(current))

        for neighbor in get_neighbors(current):
            if to_tuple(neighbor) not in visited:
                queue.append((neighbor, path + [current]))

    return None  # no solution


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

solution = bfs(initial_state)

print("BFS Solution Path:")
if solution is None:
    print("No solution exists for this puzzle.")
else:
    for step in solution:
        for row in step:
            print(row)
        print()
```

Output:

```
BFS Solution Path:
No solution exists for this puzzle.
```

DFS

Algorithm:

⇒ Algo DFS

↳ start & get goal state

↳ Take input in string form

↳ DFS

↳ move all to valid moves and choose best tile

↳ add each state to recursion

↳ If start state = final goal state

return path

DFS Soln

2	8	3
1	6	4
7		5

↓ Up

Left ✗

Right ✗

2	8	3
1		4
7	6	5

✗

✗

↓ Down

2		3
1	8	4
7	6	5

↓ Right

	2	3
1	8	4
7	6	5

↓ Up

1	2	3
	8	4
7	6	5

↓ Left

1	2	3
8		4
7	6	5

3. Implement A* search algorithm

Algorithm:

L
U
2 8 3
0 1 4
7 6 5
2 0 3
1 8 4
7 6 5
g=3
U
R
D
0 8 3
2 1 4
7 6 5
2 8 3
1 0 4
7 6 5
2 8 3
7 1 4
0 6 5
L
R
D
0 2 3
1 8 4
7 6 5
2 0 3
1 8 4
7 6 5
2 8 3
1 0 4
7 6 5
3
f(n)=6
3
f(n)=6
4
f(n)=7
2
f(n)=5
3
f(n)=6
3
f(n)=6
g=4
D
R
1 2 3
0 8 4
7 6 5
2 0 3
1 8 4
7 6 5
1
f(n)=5
U
D
R
1 2 3
8 0 4
7 6 5
g=5
goal
h(n)=0
g(n)=5
f(n)=5

⇒ Manhattan Dist.

1 5 8
3 2 □
4 6 7

I

L U D

1 5 8
3 2 □
4 6 7

1 5 8
3 2 7
4 6 □

1 2 3 4 5 6 7 8
0 1 2 1 1 2 3 3
= 13

1 5 8
3 2 □
4 6 7

1 2 3 4 5 6 7 8
0 2 3 1 1 2 2 3
= 14

1 2 3 4 5 6 7 8
0 1 3 1 1 2 2 2
= 12

①

L D

1 □ 5
3 2 8
4 6 7

1 5 8
3 2 □
4 6 7

X

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

13 + 2 = 15

②

L D R

□ 1 5
3 2 8
4 6 7

1 2 5
3 □ 8
4 6 7

1 5 □
3 2 8
4 6 7

=14 =12 +3=15

1 2 3 4 5 6 7 8

3+1
+2 +2 +2 +2

③

(4)

1	2	5
3	□	8
4	6	7

L D R

1	2	5
□	3	8
4	6	7

15

2
2
1
2
2
10

1	2	5
3	6	8
4	□	7

15

2
3
1
1
2
9

1	2	5
3	8	□
4	6	7

15

2
3
1
2
2
10

8/9

A* Algo

1) A* search evaluates nodes by combining g(n), the cost to reach node and h(n) to get from node to goal

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

3) In manhattan, calculated by add minimum moves

Code & Output:

1BM23CS327tilesandmanhattan.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
# 8-Puzzle heuristics

# Goal state for reference
goal_state = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]   # 0 represents the blank tile
]

# Heuristic 1: Number of tiles out of place
def misplaced_tiles(state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced


# Heuristic 2: Manhattan Distance
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                # Find the goal position of the tile
                goal_x = (value - 1) // 3
                goal_y = (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance


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

print("Misplaced Tiles:", misplaced_tiles(initial_state))
print("Manhattan Distance:", manhattan_distance(initial_state))
```

```
Misplaced Tiles: 6
Manhattan Distance: 9
```

4. Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Hill Climbing for 4 Queens

Pseudocode

1.) Start with a random placement of 4 queens on the board

2.) Calculate the no. of conflicts

3.) Repeat:

a.) Look at all possible moves by moving one queen in its row to a different column

b.) Choose that reduce conflicts

c.) If no better found then stop, or make best move & update

4.) If conflicts == 0

Solⁿ found

5.) Else:

No solⁿ

State Space

1.) $x_0 = 3, x_1 = 1, x_2 = 2, x_3 = 0$

cost, h = 2

2.) $x_0 = 3, x_1 = 1, x_2 = 2, x_3 = 3$

$h = 2 + 1 = 3$

3.) $x_0 = 3, x_1 = 0, x_2 = 2, x_3 = 1$

$h = 1$

4.) $x_0 = 2, x_1 = 0, x_2 = 3, x_3 = 1$

$h = 0$

5.) $x_0 = 0, x_1 = 3, x_2 = 0, x_3 = 2$

$h = 1$

6.) $x_0 = 1, x_1 = 3, x_2 = 0, x_3 = 2$

$h = 0$

Code & Output:

Week_5_Siddhant_Sawhney_1BM23CS327_Hill_Climbing.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
import random
import math

def compute_cost(state):
    """Count diagonal conflicts for a permutation-state (one queen per row & column)."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def random_permutation(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr

def neighbors_by_swaps(state):
    """All neighbors obtained by swapping two columns (keeps permutation property)."""
    n = len(state)
    for i in range(n - 1):
        for j in range(i + 1, n):
            nb = state.copy()
            nb[i], nb[j] = nb[j], nb[i]
            yield nb

def hill_climb_with_restarts(n, max_restarts=None):
    """Hill climbing on permutations with random restart on plateau (no revisits)."""
    visited = set()
    total_states = math.factorial(n)
    restarts = 0

    while True:
        # pick a random unvisited start permutation
        if len(visited) >= total_states:
            raise RuntimeError("All states visited — giving up (no solution found).")

        state = random_permutation(n)
        while tuple(state) in visited:
            state = random_permutation(n)
        visited.add(tuple(state))
```

Week_5_Siddhant_Sawhney_1BM23CS327_Hill_Climbing.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
        # climb from this start
        while True:
            cost = compute_cost(state)
            if cost == 0:
                return state, restarts

            # find best neighbor (swap-based neighbors)
            best_neighbor = None
            best_cost = float("inf")
            for nb in neighbors_by_swaps(state):
                c = compute_cost(nb)
                if c < best_cost:
                    best_cost = c
                    best_neighbor = nb

            # if strictly better, move; otherwise it's a plateau/local optimum -> restart
            if best_cost < cost:
                state = best_neighbor
                visited.add(tuple(state))
            else:
                # plateau or local optimum -> restart
                restarts += 1
                if max_restarts is not None and restarts >= max_restarts:
                    raise RuntimeError(f"Stopped after {restarts} restarts (no solution found).")
                break  # go pick a new unvisited start

def format_board(state):
    n = len(state)
    lines = []
    for r in range(n):
        lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n)))
    return "\n".join(lines)

if __name__ == "__main__":
    n = 4
    print("Siddhant Sawhney - 1BM23CS327")
    solution, restarts = hill_climb_with_restarts(n)
    print("Found solution:", solution)
    print(format_board(solution))
```

```
Siddhant Sawhney - 1BM23CS327
Found solution: [2, 0, 3, 1]
- Q - -
- - - Q
Q - - -
- - Q -
```

5. Simulated Annealing to Solve 8-Queens problem

Algorithm:

Simulate Annealing

Pseudocode

current ← initial state

T ← a large positive value

while T > 0 do

next ← a random neighbour

ΔE ← current-cost − next-cost

if ΔE > 0 then

current ← next

else

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

end if

decrease T

end while

return current

Code & Output:

Week_5_Siddhant_Sawhney_1BM23CS327_Simulated_Annealing.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
import random
import math

def cost(state):

    attacks = 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):
                attacks += 1
    return attacks

def get_neighbor(state):

    neighbor = state[:]
    i, j = random.sample(range(len(state)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

def simulated_annealing(n=8, max_iter=10000, temp=100.0, cooling=0.95):

    current = list(range(n))
    random.shuffle(current)
    current_cost = cost(current)

    temperature = temp
    cooling_rate = cooling

    best = current[:]
    best_cost = current_cost

    for _ in range(max_iter):
        if temperature <= 0 or best_cost == 0:
            break

        neighbor = get_neighbor(current)
        neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost

        if delta > 0 or random.random() < math.exp(delta / temperature):
            current, current_cost = neighbor, neighbor_cost
            if neighbor_cost < best_cost:
                best, best_cost = neighbor[:], neighbor_cost

        temperature *= cooling_rate

    return best, best_cost

def print_board(state):

    n = len(state)
    for row in range(n):
        line = " ".join("Q" if state[col] == row else "." for col in range(n))
        print(line)
    print()


n = 8
solution, cost_val = simulated_annealing(n, max_iter=20000)
print("Siddhant Sawhney - 1BM23CS327")
print("Best position found:", solution)
print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
print("\nBoard:")
print_board(solution)
```

Week_5_Siddhant_Sawhney_1BM23CS327_Simulated_Annealing.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
print("\nBoard:")
print_board(solution)
```

```
Siddhant Sawhney - 1BM23CS327
Best position found: [4, 6, 0, 3, 1, 7, 5, 2]
Number of non-attacking pairs: 28

Board:
. . Q . . . . .
. . . . Q . . .
. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
```

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

Algorithm:

Propositional logic

- Truth tables for connectives

P	Q	~P	P ∧ Q	P ∨ Q	P ⇒ Q
F	F	T	F	F	T
F	T	T	F	T	F
T	F	F	F	T	F
T	T	F	T	T	T

- Propositional Inference : Enumeration Method

Ex: $\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \sim C)$

Checking that $KB = \alpha$

A	B	C	A ∨ C	B ∨ ~C	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

New KB

$A \to B$, $B \to C$

$\neg A \vee B$ ~~$A \vee B$~~ $A \vee C$

A	B	C	$A \to B$	$B \to C$	KB	α $(A \to C)$
F	F	F	T	T	T	T
F	F	T	T	T	T	T
F	T	F	T	F	F	T
F	T	T	T	T	T	T
T	F	F	F	T	F	F
T	F	T	F	T	F	T
T	T	F	F	F	F	F
T	T	T	T	T	T	T

Algo:-

↳ Collect Symbols

look at KB and α and list all proportional symbols

↳ Enumerate models

For n symbols, list all 2^n assignments of F/T. For A, B, C $3 \Rightarrow 2^3$

↳ Evaluate KB and α in each

↳ Evaluate each clause in KB: here compute $A \to B$ & $B \to C$

↳ KB is true in that model if both clauses are true

↳ Evaluate α $(A \to C)$ in that model

↳ Collect KB-true models

keep the subset of models where KB evaluates true

↳ Check α in those models

↳ If α is true in every KB-true model, KB entails α

↳ If atleast one KB true has α = false, it fails

S, T as variables

a : $\neg(S \vee T)$

b : $(S \wedge T)$

c : $\neg S \vee \neg T$

write TT

i) a entails B

2.i) a entails C

S	T	a $\neg(S \vee T)$	b	c	$a \to b$	$a \to c$
			f		f	f
T	T	f	T	T	f	f
T	f	f	f	T	f	f
f	T	f	f	T	T	T
f	f	T	f	T		

~~a entails B are True all false~~
~~rest false~~

~~a entails C all True~~

22.09

Code & Output:

Week_6_Siddhant_Sawhney_1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
import itertools
print("Siddhant Sawhney - 1BM23CS327")
def evaluate_formula(formula, truth_assignment):
    eval_formula = formula
    for symbol, value in truth_assignment.items():
        eval_formula = eval_formula.replace(symbol, str(value))
    return eval(eval_formula)

def generate_truth_table(variables):
    return list(itertools.product([False, True], repeat=len(variables)))

def is_entailed(KB_formula, alpha_formula, variables):
    truth_combinations = generate_truth_table(variables)
    print(f"{' '.join(variables)} | KB Result | Alpha Result")
    print("-" * (len(variables) * 2 + 15))
    for combination in truth_combinations:
        truth_assignment = dict(zip(variables, combination))
        KB_value = evaluate_formula(KB_formula, truth_assignment)
        alpha_value = evaluate_formula(alpha_formula, truth_assignment)
        result_str = " ".join(["T" if value else "F" for value in combination])
        print(f"{result_str} | {'T' if KB_value else 'F'}         | {'T' if alpha_v
        if KB_value and not alpha_value:
            return False
    return True

KB = "(A or C) and (B or not C)"
alpha = "A or B"
variables = ['A', 'B', 'C']

if is_entailed(KB, alpha, variables):
    print("\nThe knowledge base entails alpha.")
else:
    print("\nThe knowledge base does not entail alpha.")
```

```
Siddhant Sawhney - 1BM23CS327
A B C | KB Result | Alpha Result
--------------------
F F F | F         | F
F F T | F         | F
F T F | F         | T
F T T | T         | T
T F F | T         | T
T F T | F         | T
T T F | T         | T
T T T | T         | T

The knowledge base entails alpha.
```

7. Implement unification in first order logic

Algorithm:

LAB 7 : Unification

Algo : 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, 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 diff no. of argument, then return failure

Step 4 : Set substitution set (SUBST) to NIL

Step 5 : For i = 1 to the no. of element in ψ_1 :

a) Call unify func with the ith element of ψ_1 and ith element of ψ_2 and put the result into S.

b) If S = failure then return failure

c) If S ≠ NIL then do,

a) apply S to the remainder to both L1 and L2

b) Subst = Append (S, subst)

Step 6 : Return Subst

1.) MGU of $\{ p(b, x, f(g(z))), p(z, f(y), f(y)) \}$

compare predicates :- both p are same

Compare first arguments b & z

$\rightarrow z = b$

Compare second arg x & $f(y)$

$\rightarrow x = f(y)$

Compare third arg $f(g(z))$ and $f(y)$

since both start with f, unify

$g(z) = y \rightarrow y = g(z)$

MGU $\{ z \mapsto b, x \mapsto f(g(z)), y \mapsto g(z) \}$

2.) Find MGU of $\{ p(f(a), g(y)), p(x, x) \}$

Predicate names, both p same

Arguments must unify :-

~~First : $f(a)$ & $x \rightarrow x = f(a)$~~

~~Second : $g(y)$ & $x \rightarrow g(y) = x = f(a)$~~

Since g & f are different functional symbols,

$g(y) = f(a)$ cant be unified

Result : No unifier exists.

3. Unify {prime(11), prime(y)}

Arg: 11 & y → y = 11

MGU: {y ↦ 11}

4. Unify {knows(John, x), knows(y, mother(y))}

first args: John & y → y = John

Second: x & mother(y) → with subst^n y = John,

x = mother(John)

MGU:

{y ↦ John, x ↦ mother(John)}

5.) Unify {knows(John, x), knows(y, Bill)}

first args: John & y → y = John

Second: x & Bill → x = Bill

MGU:

{y ↦ John, x ↦ Bill}

Code & Output:

Lab_7_Unification_1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
import json

# --- Helper Functions for Term Manipulation ---

def is_variable(term):
    """Checks if a term is a variable (a single capital letter string)."""
    return isinstance(term, str) and len(term) == 1 and 'A' <= term[0] <= 'Z'

def occurs_check(variable, term, sigma):
    """
    Checks if 'variable' occurs anywhere in 'term' under the current substitution 'sigma'.
    This prevents infinite recursion (e.g., unifying X with f(X)).
    """
    term = apply_substitution(term, sigma) # Check the substituted term

    if term == variable:
        return True

    # If the term is a list (function/predicate), check its arguments recursively
    if isinstance(term, list):
        for arg in term[1:]:
            if occurs_check(variable, arg, sigma):
                return True

    return False

def apply_substitution(term, sigma):
    """
    Applies the current substitution 'sigma' to a 'term' recursively.
    """
    if is_variable(term):
        # If the variable is bound in sigma, apply the binding
        if term in sigma:
            # Recursively apply the rest of the substitutions to the binding's value
            # This is critical for chains like X/f(Y), Y/a -> X/f(a)
            return apply_substitution(sigma[term], sigma)
        return term

    if isinstance(term, list):
        # Apply substitution to the arguments of the function/predicate
        new_term = [term[0]] # Keep the function/predicate symbol
        for arg in term[1:]:
            new_term.append(apply_substitution(arg, sigma))
        return new_term

    # Term is a constant or an unhandled type, return as is
    return term
```

Lab_7_Unification_1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
def term_to_string(term):
    """
    Converts the internal list representation of a term into standard logic notation string.
    e.g., ['f', 'Y'] -> "f(Y)"
    """
    if isinstance(term, str):
        return term

    if isinstance(term, list):
        # Term is a function or predicate
        symbol = term[0]
        args = [term_to_string(arg) for arg in term[1:]]
        return f"{symbol}({', '.join(args)})"

    return str(term)


# --- Main Unification Function ---

def unify(term1, term2):
    """
    Implements the Unification Algorithm to find the MGU for term1 and term2.
    Returns the MGU as a dictionary or None if unification fails.
    """
    # Initialize the substitution set (MGU)
    sigma = {}

    # Initialize the list of pairs to resolve (the difference set)
    diff_set = [[term1, term2]]

    print(f"--- Unification Process Started ---")
    print(f"Initial Terms:")
    print(f"L1: {term_to_string(term1)}")
    print(f"L2: {term_to_string(term2)}")
    print("-" * 35)

    while diff_set:
        # Pop the current pair of terms to unify
        t1, t2 = diff_set.pop(0)

        # 1. Apply the current MGU to the terms before comparison
        t1_prime = apply_substitution(t1, sigma)
        t2_prime = apply_substitution(t2, sigma)

        print(f"Attempting to unify: {term_to_string(t1_prime)} vs {term_to_string(t2_prime)}")
```

Lab_7_Unification_1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
        # 2. Check if terms are identical
        if t1_prime == t2_prime:
            print(f"  -> Identical. Current MGU: {term_to_string(sigma)}")
            continue

        # 3. Handle Variable-Term unification
        if is_variable(t1_prime):
            var, term = t1_prime, t2_prime
        elif is_variable(t2_prime):
            var, term = t2_prime, t1_prime
        else:
            var, term = None, None

        if var:
            # Check if term is a variable, and if so, don't bind V/V
            if is_variable(term):
                print(f"  -> Both are variables. Skipping {var} / {term}")
                # Ensure they are added back if not identical (which is caught by step 2).
                # If V1 != V2, we add V1/V2 or V2/V1 to sigma. Since step 2 handles V/V, this means V1 != V2 here.
                if var != term:
                    sigma[var] = term
                    print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU: {term_to_string(sigma)}")
            # Occurs Check: Fail if the variable occurs in the term it's being bound to
            elif occurs_check(var, term, sigma):
                print(f"  -> OCCURS CHECK FAILURE: Variable {var} occurs in {term_to_string(term)}")
                return None

            # Create a new substitution {var / term}
            else:
                sigma[var] = term
                print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU: {term_to_string(sigma)}")

        # 4. Handle Complex Term (Function/Predicate) unification
        elif isinstance(t1_prime, list) and isinstance(t2_prime, list):
            # Check functor/predicate symbol and arity (number of arguments)
            if t1_prime[0] != t2_prime[0] or len(t1_prime) != len(t2_prime):
                print(f"  -> FUNCTOR/ARITY MISMATCH: {t1_prime[0]} != {t2_prime[0]} or arity mismatch.")
                return None

            # Add corresponding arguments to the difference set
            # Start from index 1 (after the symbol)
            for arg1, arg2 in zip(t1_prime[1:], t2_prime[1:]):
                diff_set.append([arg1, arg2])
            print(f"  -> Complex terms matched. Adding arguments to difference set.")

        # 5. Handle Constant-Constant or other mismatches (Fail)
        else:
            print(f"  -> TYPE/CONSTANT MISMATCH: {term_to_string(t1_prime)} and {term_to_string(t2_prime)} cannot be unified.")
```

Lab_7_Unification_1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
        else:
            print(f"  -> TYPE/CONSTANT MISMATCH: {term_to_string(t1_prime)} and {term_to_string(t2_prime)} cannot be unified.")
            return None

    print("-" * 35)
    print("--- Unification Successful ---")

    # Final cleanup to ensure all bindings are fully resolved
    final_mgu = {k: apply_substitution(v, sigma) for k, v in sigma.items()}
    return final_mgu

# --- Define the Input Terms ---

# L1 = Q(a, g(X, a), f(Y))
literal1 = ['Q', 'a', ['g', 'X', 'a'], ['f', 'Y']]

# L2 = Q(a, g(f(b), a), X)
literal2 = ['Q', 'a', ['g', ['f', 'b'], 'a'], 'X']

# --- Run the Unification ---

mgu_result = unify(literal1, literal2)

if mgu_result is not None:
    print("\n[ Final MGU Result Siddhant Sawhney 1BM23CS327 ]")

    # Format the final MGU for display using the new helper function
    clean_mgu = {k: term_to_string(v) for k, v in mgu_result.items()}
    final_output = ', '.join([f"{k} / {v}" for k, v in clean_mgu.items()])
    print(f"Final MGU: {{ {final_output} }}")

    # --- Verification ---
    print("\n[ Verification ]")
    unified_l1 = apply_substitution(literal1, mgu_result)
    unified_l2 = apply_substitution(literal2, mgu_result)

    print(f"L1 after MGU: {term_to_string(unified_l1)}")
    print(f"L2 after MGU: {term_to_string(unified_l2)}")

    if unified_l1 == unified_l2:
        print("-> SUCCESS: L1 and L2 are identical after applying the MGU.")
    else:
        print("-> ERROR: Unification failed verification.")
else:
    print("\nUnification FAILED.")
```

Lab_7_Unification_1BM23CS327.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
        else:
            print("-> ERROR: Unification failed verification.")
    else:
        print("\nUnification FAILED.")
```

```
--- Unification Process Started ---
Initial Terms:
L1: Q(a, g(X, a), f(Y))
L2: Q(a, g(f(b), a), X)
----------------------------------
Attempting to unify: Q(a, g(X, a), f(Y)) vs Q(a, g(f(b), a), X)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {}
Attempting to unify: g(X, a) vs g(f(b), a)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: f(Y) vs X
  -> Variable binding added: X / f(Y). New MGU: {'X': ['f', 'Y']}
Attempting to unify: f(Y) vs f(b)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {'X': ['f', 'Y']}
Attempting to unify: Y vs b
  -> Variable binding added: Y / b. New MGU: {'X': ['f', 'Y'], 'Y': 'b'}
----------------------------------
--- Unification Successful ---

[ Final MGU Result Siddhant Sawhney 1BM23CS327 ]
Final MGU: { X / f(b), Y / b }

[ Verification ]
L1 after MGU: Q(a, g(f(b), a), f(b))
L2 after MGU: Q(a, g(f(b), a), f(b))
-> SUCCESS: L1 and L2 are identical after applying the MGU.
```

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

Algorithm:

Week 8 First Order Logic

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

Premises Conclusion

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
} Rules

A
B
} Facts

Prove Q

Forward Chaining - First Order Logic - Solved Example

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

Prove that "West is criminal."

Rule 1: $\forall x, y, z$ American$(x) \wedge$ Weapon$(y) \wedge$ Sells$(x, y, z) \wedge$ Hostile(z)
$\Rightarrow$ Criminal(x)

Rule 2: $\forall x$ Missile$(x) \wedge$ Owns$(\text{Nono}, x) \Rightarrow$ Sells$(\text{West}, x, \text{Nono})$

Rule 3: $\forall x$ Enemy$(x, \text{America}) \Rightarrow$ Hostile(x)

Rule 4: $\forall x$ Missile$(x) \Rightarrow$ Weapon(x)

5: American(West)

6: Enemy(Nono, America)

7.) Owns (nono, M1) and
8.) Missile (M1)

Algo:-

On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arg with new ones that are unused.

function FOL-FC-Ask(KB, α) returns a subst^n or false
inputs: KB, 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, p'1 ∧ ... ∧ p'n)
for some p'1, ... p'n in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentence already
in KB or new then
add q' to new
φ ← UNIFY(q', α)
if φ is not fail then return φ
add new to KB
~~return false~~

13.10

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

Algorithm:

LAB - 9 RESOLUTION FOL

Steps to convert logic statement to CNF

1.) Eliminate biconditionals & implications:

- Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
- Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

2.) Move $\neg$ inwards:

- $\neg(\forall x\, p) \equiv \exists x\, \neg p$,
- $\neg(\exists x\, p) \equiv \forall x\, \neg p$,
- $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$,
- $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$,
- $\neg\neg\alpha \equiv \alpha$

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

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

- For instance, $\exists x\, Rich(x)$ becomes $Rich(G1)$ where $G1$ is a new Skolem constant
- "Everyone has a heart" $\forall x\, Person(x)$

$\Rightarrow \exists y\ Heart(y) \wedge Has(x,y)$

becomes $\forall x\ Person(x) \Rightarrow Heart(H(x)) \wedge Has(x, H(x))$,

where H is a new ~~funct~~ symbol

5.) Drop universal quantifiers

for instance, $\forall x\ Person(x)$ becomes $Person(x)$

6.) Distribute $\wedge$ over $\vee$:

- $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Resolution in FOL

1.) Convert all sentences to CNF

2.) Negate conclusion S & convert result to CNF

3.) Add negated conclusion S to the premise clauses

4.) Repeat until contradiction or no progress is made:

a.) Select 2 clauses (call them parent clauses)

b.) Resolve them together, performing all required unifications

c.) If resolvent is empty clause, a contradiction has been found (i.e., S follows from the premises)

d.) If not, add resolvent to the premises

If we succeed in step 4, we have proved the conclusion

Rep in FOL

1.) John likes all kind of food
2.) Apple & vegetables are food
3.) Anything anyone eats and not killed is food
4.) Anil eats peanuts and still alive
5.) Harry eats everything that Anil eats
6.) Anyone who is alive implies not killed
7.) Anyone who is not killed implies alive

PT ⇒ h.) John likes peanuts

a.) $\forall x: food(x) \rightarrow likes(John, x)$
b.) $food(Apple) \wedge food(vegetables)$
c.) $\forall x \forall y: eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$
d.) $eats(Anil, Peanuts) \wedge alive(Anil)$
e.) $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$
f.) $\forall x: killed(x) \rightarrow \neg alive(x)$
g.) $\forall x: alive(x) \rightarrow \neg killed(x)$
h.) $likes(John, Peanuts)$

Proof by Resolution
Eliminate implication

a) $\forall x: food(x) \rightarrow likes(John, x)$
b) $food(Apple) \wedge food(vegetables)$
c) $\forall x \forall y: eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$
d) $eats(Anil, Peanuts) \wedge alive(Anil)$
e) $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$
f) $\forall x: killed(x) \rightarrow \neg alive(x)$
g) $\forall x: alive(x) \rightarrow \neg killed(x)$
h) $likes(John, Peanuts)$

a.) $\forall x \neg food(x) \vee likes(John, x)$
b.) $food(Apple) \wedge food(vegetables)$
c.) $\forall x \forall y \neg[eats(x,y) \wedge killed(x)] \vee food(y)$
d.) $eats(Anil, Peanuts) \wedge alive(Anil)$
e.) $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f.) $\forall x \neg[killed(x)] \vee \neg alive(x)$
g.) $\forall x \neg alive(x) \vee \neg killed(x)$
h.) $likes(John, Peanuts)$

move negation inwards

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.) $eats(Anil, Peanuts) \wedge alive(Anil)$

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

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

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

h.) $likes(John, Peanuts)$

Standardize variables

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

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

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

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

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

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

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

h.) $likes(John, Peanuts)$

Drop Univers

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

b.) $food(apple)$

c.) $food(vegetables)$

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

e.) $eats(Anil, Peanut)$

f.) $alive(Anil)$

g.) $\neg eats(Anil, w) \vee eats(Harry, w)$

h.) $killed(g) \vee alive(g)$

i.) $\neg alive(k) \vee \neg killed(k)$

j.) $likes(John, Peanuts)$

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.) $\neg eats(Anil, w) \vee eats(Harry, w)$

h.) $killed(g) \vee alive(g)$

i.) $\neg alive(k) \vee \neg killed(k)$

j.) $likes(John, Peanuts)$

Code & Output:

LAB_9_Resolution_by_FOL_1BM23CS327.ipynb Save in GitHub to keep changes

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
def negate(literal):
    """Return negation of a literal."""
    return literal[1:] if literal.startswith("~") else "~" + literal

def resolve(ci, cj):
    """Resolve two clauses and return resolvents + which literals were resolved."""
    resolvents = []
    for lit1 in ci:
        for lit2 in cj:
            if lit1 == negate(lit2):
                new_clause = list(set(ci + cj))
                new_clause.remove(lit1)
                new_clause.remove(lit2)
                resolvents.append((new_clause, lit1, lit2))
    return resolvents

def resolution_algorithm(clauses):
    """Main Resolution Loop — with Resolution Tree"""
    new = set()
    tree = {}
    step = 0

    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i+1, len(clauses))]
        for (ci, cj) in pairs:
            results = resolve(ci, cj)
            for (resolvent, lit1, lit2) in results:
                step += 1
                resolvent_tuple = tuple(sorted(resolvent))
                if resolvent_tuple not in tree:
                    tree[resolvent_tuple] = ((tuple(ci), tuple(cj)), (lit1, lit2))
                print(f"\nStep {step}: Resolving {ci} and {cj} on {lit1} / {lit2} ⇒ {resolvent}")

                if resolvent == []:
                    print("\n🎉 Contradiction (empty clause) found!")
                    print("✅ Query is PROVED TRUE by Resolution.\n")
                    print("🌳 Resolution Tree (Trace):\n")
                    print_tree(tree, resolvent_tuple)
                    return True

                new.add(resolvent_tuple)

        if new.issubset(set(map(tuple, clauses))):
            print("\n⚠️ No contradiction found. Query CANNOT be proved from the Knowledge Base.")
            return False

        for clause in new:
```

LAB_9_Resolution_by_FOL_1BM23CS327.ipynb Save in GitHub to keep changes

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
            for clause in new:
                if list(clause) not in clauses:
                    clauses.append(list(clause))

    def print_tree(tree, final_clause):
        """Recursively print the resolution tree."""
        def helper(clause, indent=""):
            if clause not in tree:
                print(indent + "Clause: " + str(list(clause)))
                return
            parents, resolved = tree[clause]
            lit1, lit2 = resolved
            print(indent + f"Clause: {list(clause)}  (resolved {lit1}/{lit2})")
            print(indent + " ├─ From:")
            helper(parents[0], indent + " │  ")
            print(indent + " └─ And:")
            helper(parents[1], indent + "    ")

        helper(final_clause)



print("🧠 FIRST ORDER LOGIC RESOLUTION SYSTEM (with Tree)")
print("--------------------------------------------------")


n = int(input("Enter number of statements in the Knowledge Base: "))
kb = []
for i in range(n):
    stmt = input(f"Enter statement {i+1} (in CNF using ∨ for OR): ")
    kb.append(stmt)


query = input("\nEnter the query to prove: ")


negated_query = "~" + query if not query.startswith("~") else query[1:]


clauses = [stmt.replace(" ", "").split("∨") for stmt in kb]
clauses.append([negated_query])


print("\n📘 Clauses for Resolution:")
for i, c in enumerate(clauses, 1):
    print(f"{i}. {c}")


resolution_algorithm(clauses)
```

LAB_9_Resolution_by_FOL_1BM23CS327.ipynb Save in GitHub to keep changes

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
for i, c in enumerate(clauses, 1):
    print(f"{i}. {c}")


resolution_algorithm(clauses)
```

```
🧠 FIRST ORDER LOGIC RESOLUTION SYSTEM (with Tree)
--------------------------------------------------
Enter number of statements in the Knowledge Base: 2
Enter statement 1 (in CNF using ∨ for OR): ~Food(Peanuts) ∨ Likes(John,Peanuts)
Enter statement 2 (in CNF using ∨ for OR): Food(Peanuts)

Enter the query to prove: Likes(John,Peanuts)

📜 Clauses for Resolution:
1. ['~Food(Peanuts)', 'Likes(John,Peanuts)']
2. ['Food(Peanuts)']
3. ['~Likes(John,Peanuts)']

Step 1: Resolving ['~Food(Peanuts)', 'Likes(John,Peanuts)'] and ['Food(Peanuts)'] on ~Food(Peanuts) / Food(Peanuts) ⇒ ['Likes(John,Peanuts)']

Step 2: Resolving ['~Food(Peanuts)', 'Likes(John,Peanuts)'] and ['~Likes(John,Peanuts)'] on Likes(John,Peanuts) / ~Likes(John,Peanuts) ⇒ ['~Food(Peanuts)']

Step 3: Resolving ['~Food(Peanuts)', 'Likes(John,Peanuts)'] and ['Food(Peanuts)'] on ~Food(Peanuts) / Food(Peanuts) ⇒ ['Likes(John,Peanuts)']

Step 4: Resolving ['~Food(Peanuts)', 'Likes(John,Peanuts)'] and ['~Likes(John,Peanuts)'] on Likes(John,Peanuts) / ~Likes(John,Peanuts) ⇒ ['~Food(Peanuts)']

Step 5: Resolving ['Food(Peanuts)'] and ['~Food(Peanuts)'] on Food(Peanuts) / ~Food(Peanuts) ⇒ []

🎉 Contradiction (empty clause) found!
✅ Query is PROVED TRUE by Resolution.

🌳 Resolution Tree (Trace):

Clause: []  (resolved Food(Peanuts)/~Food(Peanuts))
 ├─ From:
 │  Clause: ['Food(Peanuts)']
 └─ And:
    Clause: ['~Food(Peanuts)']  (resolved Likes(John,Peanuts)/~Likes(John,Peanuts))
     ├─ From:
     │  Clause: ['~Food(Peanuts)', 'Likes(John,Peanuts)']
     └─ And:
        Clause: ['~Likes(John,Peanuts)']
True
```

10. Implement Alpha-Beta Pruning.

Algorithm:

LAB-10

Alpha-Beta Pruning

1) α-β proposes to find the optimal path without looking at every node in the game tree

2) Maintaining α and β bound during the calculation

3a) In both Min & Max node, we return when α ≥ β which compares with its parent node only

4) Both α-β cut-off give same path

5a) α-β gives optimal solution as it takes less time to get the value for the root node.

```
function ALPHA-BETA-SEARCH (state) returns an action
  v ← MAX-VALUE(state, -∞, +∞)
  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 ← -∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)  return v
```

Code & Output:

LAB_10_Adversarial_Search_1BM23CS327.ipynb Save in GitHub to keep changes

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
def alpha_beta(node, depth, alpha, beta, isMax, values, maxDepth):
    print(f"Depth={depth}, Node={node}, Alpha={alpha}, Beta={beta}")
    if depth == maxDepth:

        first_leaf_index = (2 ** maxDepth) - 1
        leaf_idx = node - first_leaf_index
        return values[leaf_idx]

    if isMax:
        best = float('-inf')
        for i in range(2):  # two children (left and right)
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, False, values, maxDepth)
            best = max(best, val)
            alpha = max(alpha, best)
            print(f"MAX: Depth={depth}, Node={node}, Alpha={alpha}, Beta={beta}")

            if beta <= alpha:
                print(f"🚫 PRUNED at MAX node {node} (α ≥ β)")
                break
        return best
    else:
        best = float('inf')
        for i in range(2):
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, True, values, maxDepth)
            best = min(best, val)
            beta = min(beta, best)
            print(f"MIN: Depth={depth}, Node={node}, Alpha={alpha}, Beta={beta}")

            if beta <= alpha:
                print(f"🚫 PRUNED at MIN node {node} (α ≥ β)")
                break
        return best



print("🧠 ALPHA-BETA PRUNING — Interactive Demo")
print("========================================\n")


maxDepth = int(input("Enter maximum depth of the game tree: "))

num_leaves = 2 ** maxDepth
print(f"For depth {maxDepth}, the tree will have {num_leaves} leaf nodes.\n")
```

LAB_10_Adversarial_Search_1BM23CS327.ipynb Save in GitHub to keep changes

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all Copy to Drive

```python
values = []
print("Enter the leaf node values from LEFT to RIGHT:")
for i in range(num_leaves):
    val = int(input(f"Value of leaf {i + 1}: "))
    values.append(val)

print("\n🔄 Running Alpha-Beta pruning...\n")
result = alpha_beta(0, 0, float('-inf'), float('inf'), True, values, maxDepth)


print("\n✅ Final Result:")
print(f"Value of the root node (best achievable for MAX): {result}")
```

```
🧠 ALPHA-BETA PRUNING - Interactive Demo
========================================

Enter maximum depth of the game tree: 3
For depth 3, the tree will have 8 leaf nodes.

Enter the leaf node values from LEFT to RIGHT:
Value of leaf 1: 3
Value of leaf 2: 5
Value of leaf 3: 6
Value of leaf 4: 9
Value of leaf 5: 1
Value of leaf 6: 22
Value of leaf 7: 0
Value of leaf 8: -1

🔄 Running Alpha-Beta pruning...

MAX: Depth=2, Node=3, Alpha=3, Beta=inf
MAX: Depth=2, Node=3, Alpha=5, Beta=inf
MIN: Depth=1, Node=1, Alpha=-inf, Beta=5
MAX: Depth=2, Node=4, Alpha=6, Beta=5
🚫 PRUNED at MAX node 4 (α ≥ β)
MIN: Depth=1, Node=1, Alpha=-inf, Beta=5
MAX: Depth=0, Node=0, Alpha=5, Beta=inf
MAX: Depth=2, Node=5, Alpha=5, Beta=inf
MAX: Depth=2, Node=5, Alpha=22, Beta=inf
MIN: Depth=1, Node=2, Alpha=5, Beta=22
MAX: Depth=2, Node=6, Alpha=5, Beta=22
MAX: Depth=2, Node=6, Alpha=5, Beta=22
MIN: Depth=1, Node=2, Alpha=5, Beta=0
🚫 PRUNED at MIN node 2 (α ≥ β)
MAX: Depth=0, Node=0, Alpha=5, Beta=inf

✅ Final Result:
Value of the root node (best achievable for MAX): 5
```