

# **VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

“JnanaSangama”, Belgaum -590014, Karnataka.



## **LAB REPORT on**

## **Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**Shivansh Aswal (1BM23CS315)**

*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 **Shivansh Aswal (1BM23CS315)** 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

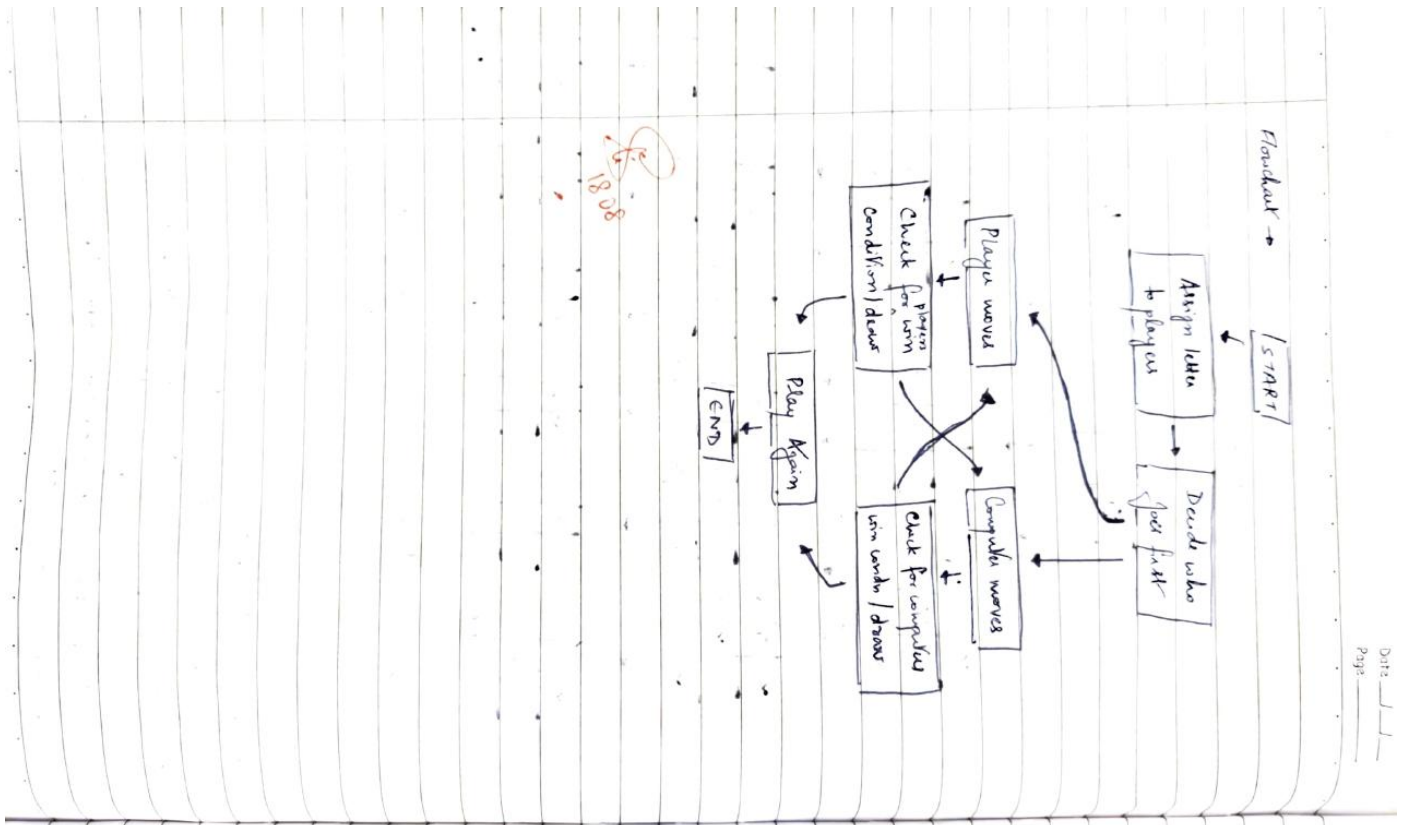
Github Link:

<https://github.com/ShivanshAswal/AI-LAB>

## Program 1

Implement Tic - Tac - Toe Game

Algorithm:



Week I

Implement TIC TAC TOE game

X		O
O	X	X
	O	

Initial Instance

X		O
O	X	X
X	O	

X	X	O
O	X	X
	O	

X		O
O	X	X
	O	X

Cost →

$1+0=1$

$0+1=1$

$1+1=2$

X	O	O
O	X	X
X	O	

X		O
O	X	X
X	O	O

X	X	O
O	X	X
O	O	O

X	X	O
O	X	X
O	O	O

X	O	O
O	X	X
O	O	X

X		O
O	X	X
O	O	X

X	O	O
O	X	X
X	O	X

X	X	O
O	X	X
X	O	O

X	X	O
O	X	X
X	O	O

X	X	O
O	X	X
O	O	X

X	O	O
O	X	X
X	O	X

X	X	O
O	X	X
O	O	X

~~Draw~~  
Win(X)

Draw

Draw

~~Draw~~  
Win(X)

Win(X)

Win(X)

ALGORITHM

- Assign X and O to two players (one each)
- Decide who starts the game
- Alternatively place assigned symbol until one player wins or game ends in a draw
- Win Condition: one whole row / column / diagonal filled with respective character ('O' or 'X')



Code:

```
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):
```

```

    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
            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

```

```

        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:

```
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

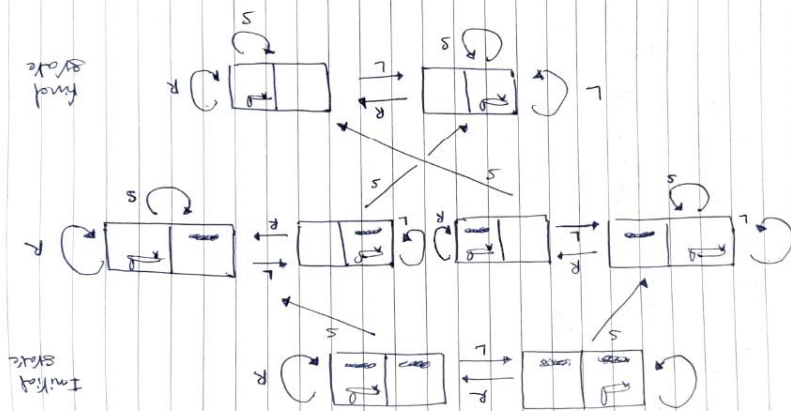
Enter O row (1-3): 1
Enter O col (1-3): 3
Board after O move:
- - O
O O X
X - X

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:







### Algorithm

A state can perform three actions at an instance -  
 Left, Right, Suck  
 Hence any action cannot be executed, it is denoted in a self-loop  
 Initial state: Both rooms DIRTY  
 Final state: Both rooms CLEAN  
 Two vacuums A and B are taken  
 Pick dust whenever it is in the same room as the vacuum cleaner. Otherwise, move to the other room, then pick

$$\text{Cost} = 2 \times 2^m \quad m = \text{no. of rooms}$$

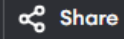
$$\text{Cost} = 2 \times 2^2$$

Code:

```
main.py    Share  Run

1 print("Shivansh Aswal / 1BM23CS315")
2 def vacuum_cleaner():
3     while True:
4         try:
5             n = int(input("Enter the number of rooms (n): "))
6             if n < 1:
7                 print("Number of rooms must be at least 1.")
8                 continue
9             break
10        except ValueError:
11            print("Please enter a valid integer.")
12
13    rooms_labels = [chr(ord('A') + i) for i in range(n)]
14
15    rooms = {label: 1 for label in rooms_labels}
16
17    vacuum_position = rooms_labels[0]
18
19    cost_function = 2 * (2 ** n)
20    print(f"\nCost function value for {n} rooms: {cost_function}\n")
21
22    total_cost = 0
23
24    print(f"Initial state: All rooms {[r for r in rooms_labels]} are Dirty.")
25    print(f"Vacuum cleaner starts at Room {vacuum_position}.\n")
26
27    while True:
28        room_status = ", ".join([f"Room {r}={'Dirty' if status else 'Clean'}" for r, status
29                                in rooms.items()])
30        print(f"Current location: Room {vacuum_position} | Status: {room_status} | Total
31            cost: {total_cost}")
32
33        if all(status == 0 for status in rooms.values()):
34            print("\nAll rooms are clean. Task completed!")
35            print(f"Total cost of operation: {total_cost}")
36            break
```

main.py



Run

```
36     action = input("Choose action - Move vacuum (M), Clean current room (C), Quit (Q):")
37         .strip().upper()
38
39     if action == 'Q':
40         print("Exiting simulation.")
41         break
42
43     elif action == 'M':
44         possible_rooms = [r for r in rooms_labels if r != vacuum_position]
45         print("Rooms you can move to:", ", ".join(possible_rooms))
46
47         new_room = input("Enter room to move vacuum to: ").strip().upper()
48
49         if new_room not in possible_rooms:
50             print(f"Invalid room. Choose from {'', '.join(possible_rooms)}")
51         else:
52             vacuum_position = new_room
53             total_cost += 1
54             print(f"Vacuum moved to Room {vacuum_position} (Cost +1).")
55
56     elif action == 'C':
57         if rooms[vacuum_position] == 1:
58             rooms[vacuum_position] = 0
59             total_cost += 1
60             print(f"Room {vacuum_position} cleaned (Cost +1).")
61         else:
62             print(f"Room {vacuum_position} is already clean.")
63
64     else:
65         print("Invalid action. Please enter M, C, or Q.")
66
67     print()
68
69 if __name__ == "__main__":
70     vacuum_cleaner()
```

Output:

```
Shivansh Aswal / 1BM23CS315
Enter the number of rooms (n): 2

Cost function value for 2 rooms: 8

Initial state: All rooms ['A', 'B'] are Dirty.
Vacuum cleaner starts at Room A.

Current location: Room A | Status: Room A=Dirty, Room B=Dirty | Total cost: 0
Choose action - Move vacuum (M), Clean current room (C), Quit (Q): M
Rooms you can move to: B
Enter room to move vacuum to: B
Vacuum moved to Room B (Cost +1).

Current location: Room B | Status: Room A=Dirty, Room B=Dirty | Total cost: 1
Choose action - Move vacuum (M), Clean current room (C), Quit (Q): C
Room B cleaned (Cost +1).

Current location: Room B | Status: Room A=Dirty, Room B=Clean | Total cost: 2
Choose action - Move vacuum (M), Clean current room (C), Quit (Q): M
Rooms you can move to: A
Enter room to move vacuum to: A
Vacuum moved to Room A (Cost +1).

Current location: Room A | Status: Room A=Dirty, Room B=Clean | Total cost: 3
Choose action - Move vacuum (M), Clean current room (C), Quit (Q): C
Room A cleaned (Cost +1).

Current location: Room A | Status: Room A=Clean, Room B=Clean | Total cost: 4

All rooms are clean. Task completed!
Total cost of operation: 4
```

## 2. BFS with Heuristic Algorithm:

BFS using Heuristic Approach

Goal state →

1	2	3
4	5	6
7	8	

→ Using no of misplaced tiles Method

Calculate heuristic value for each state

Initial state →

1	2	3
4		6
7	5	8



Solution (Goal state)





Algorithm

Step 1:





- Initial and goal state are given
- Misplaced tiles are the tiles not in correct position
- Calculate all possible next states based on direction
- While counting no of misplaced tiles in each state
- Heuristic value = tiles which are not in place



Code:

```
main.py    Share  Run

1 print("Shivansh Aswal / 1BM23CS315")
2 import heapq
3 goal_state = [[1, 2, 3],
4               [4, 5, 6],
5               [7, 8, 0]]
6
7 def heuristic(state):
8     misplaced = 0
9     for i in range(3):
10        for j in range(3):
11            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
12                misplaced += 1
13    return misplaced
14
15 def find_blank(state):
16     for i in range(3):
17         for j in range(3):
18             if state[i][j] == 0:
19                 return i, j
20 def get_neighbors(state):
21     neighbors = []
22     x, y = find_blank(state)
23     moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
24
25     for dx, dy in moves:
26         nx, ny = x + dx, y + dy
27         if 0 <= nx < 3 and 0 <= ny < 3:
28             new_state = [row[:] for row in state]
29             new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
30             neighbors.append(new_state)
31     return neighbors
32
33 def best_first_search(start_state):
34     visited = set()
35     pq = []
36
```

```
main.py    Share  Run
```

```
37     heapq.heappush(pq, (heuristic(start_state), start_state, []))
38
39     while pq:
40         h, current, path = heapq.heappop(pq)
41         state_tuple = tuple(tuple(row) for row in current)
42         if state_tuple in visited:
43             continue
44         visited.add(state_tuple)
45         if current == goal_state:
46             return path + [current]
47         for neighbor in get_neighbors(current):
48             if tuple(tuple(row) for row in neighbor) not in visited:
49                 heapq.heappush(pq, (heuristic(neighbor), neighbor, path + [current]))
50     return None
51
52 def print_state(state):
53     for row in state:
54         print(row)
55     print()
56
57 if __name__ == "__main__":
58     start_state = [[1, 2, 3],
59                   [0, 4, 6],
60                   [7, 5, 8]]
61
62     print("Initial State:")
63     print_state(start_state)
64     solution = best_first_search(start_state)
65
66     if solution:
67         print("State Path:")
68         for step in solution:
69             print_state(step)
70     else:
71         print("No solution found!")
72
```

Output:

### Output

Shivansh Aswal / 1BM23CS315

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]

=== Code Execution Successful ===

Algorithm:

BFS without heuristic

Black III

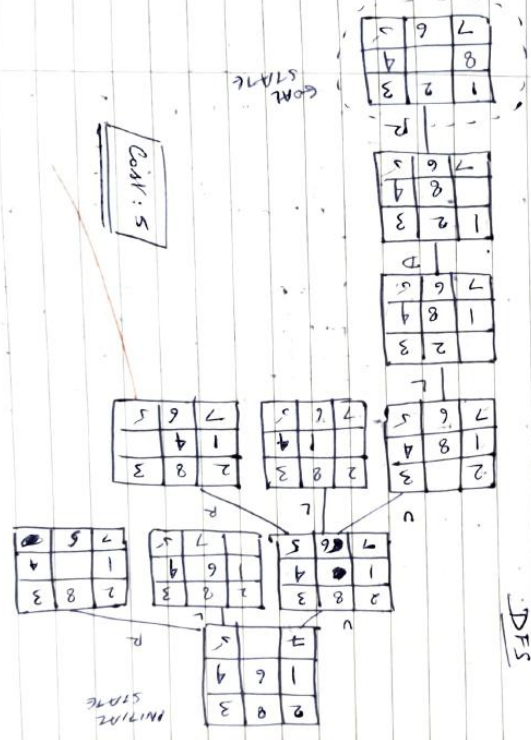
Agave americana Dr

57AR7

bridge current matters and goal

took for blank files and move empty file until you

STOP once you reach god. & stay

DFS



Code:

```
▶ print("Shivansh")
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)]

    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]
    inversions = 0
    for i in range(len(flat)):
        for j in range(i + 1, len(flat)):
```

```

        if t1at[1] > t1at[j]:
            inversions += 1
    return inversions % 2 == 0
def bfs(start):
    if not is_solvable(start):
        return None

    queue = deque([(start, [])])
    visited = set()
    while queue:
        current, path = queue.popleft()
        if current == goal_state:
            return path + [current]
        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

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)

```

Output:

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





### 3. Implement A\* search algorithm

Algorithm:

1) Impatiens A<sup>2</sup> sp. unknown for 8 purple (MISSISSIPPIANS)  
MISSISSIPPIANS

d) Misplacement fault

2	3	4
1	5	4
7		5

1	2	3
6		4
7	6	5

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

A 3x3 grid with the following numbers:

2	6	3
1	6	4
7		5

Arrows labeled R, U, and L point to the first, second, and third columns respectively.

2

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

1.8

[illegible]

82

2	3	
8	4	
6	5	
7		

2	3	
8	4	
6	5	
7		

$$\underline{d = 3}$$

2	8	3
7	1	4
6	5	

2	8	3
7	1	4
6	5	

$$\underline{d = 3}$$

1	2	3
8	4	
6	5	
7		

$$\frac{-8}{-4}$$

1	2	3
6		4
7	6	5

$$\frac{1}{12}$$

15

Cost firm = 5



2)

Manhattan RoutineDate: / /  
Page: /

Initial state

1	5	8
3	2	.
4	6	7

Goal state

1	2	3
4	5	6
7	8	.

 $f(n) = g(n) + m(n)$ Goal  
State

1	5	3
3	2	.
4	6	7

L

U

D

1	5	8
3	2	6
4	6	7

Goal

1 2 3 4 5 6 7 8

L: 0 2 3 1 1 2 2 3 → 14

U: 6 1 3 1 1 2 2 2 → 12

D: ~~0 2 3 1 1 2 2 2~~ → 14

0 1 3 1 1 2 3 3 → 14

Level - II

1	.	5
3	2	6
4	6	7

1 2 3 4 5 6 7 8  
0 1 3 1 2 2 2 2  
15

1	5
3	2
4	6

1	2	5
3	.	8
4	6	7

Level III

14

12

Algorithm

START

Calculate values for initial and goal state

Consider all moves and pick the state with smallest

 $f(n) = g(n) + h(n)$  value with goal state is achieved

Repeat steps if goal state not reached. Termination case reached

END



Output:

```
ShivanshAswal / 1BM23CS315
Initial State:
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)

Goal State:
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

Final cost (g(n) + m(n)): 2
Solution found! Moves: ['R', 'R']
Move: R
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

Move: R
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

=== Code Execution Successful ===
```

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

Algorithm:

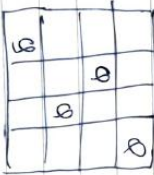
##### Hill Climbing (4 Queen)

###### Pseudo Code

- Start with random placement of 4 queens on board
- Calculate no. of conflicts (queens attacking each other)
- Repeat:
  - a. Look at all possible moves by moving one queen in its row to different column
  - b. Choose the move that reduces the no. of conflict the most
  - c. If no better move is found, stop (local best)
  - d. Otherwise, make the best move and update conflicts
- If conflicts = 0 = 0  
   solutions found
- Else: No soln

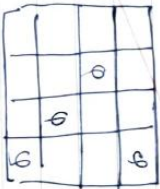
###### State space Diagram

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



Goal,  $h = 0$

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



$h = 3$

③

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

	q		q
	q	q	

$$h=1$$

④

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

	q		q
	q	q	

$$h=0$$

⑤

$$\rightarrow (0, 3, 0, 2)$$

	q		q
	q	q	

$$h=1$$





⑥

$$\rightarrow (1, 3, 0, 2)$$

		q	
	q		q

$$h=0$$

Code & Output:

```
main.py    Share  Run

1 print("Shivansh Aswal / 1BM23CS315")
2 import random
3 import math
4
5 def compute_cost(state):
6     """Count diagonal conflicts for a permutation-state (one queen per row & column)."""
7     conflicts = 0
8     n = len(state)
9     for i in range(n):
10         for j in range(i + 1, n):
11             if abs(state[i] - state[j]) == abs(i - j):
12                 conflicts += 1
13     return conflicts
14
15 def random_permutation(n):
16     arr = list(range(n))
17     random.shuffle(arr)
18     return arr
19
20 def neighbors_by_swaps(state):
21     """All neighbors obtained by swapping two columns (keeps permutation property)."""
22     n = len(state)
23     for i in range(n - 1):
24         for j in range(i + 1, n):
25             nb = state.copy()
26             nb[i], nb[j] = nb[j], nb[i]
27             yield nb
28
29 def hill_climb_with_restarts(n, max_restarts=None):
30     """Hill climbing on permutations with random restart on plateau (no revisits)."""
31     visited = set()
32     total_states = math.factorial(n)
33     restarts = 0
34
35     while True:
36         # pick a random unvisited start permutation
```

```
main.py ⌵ ☰ ☼ 🔗 Share Run
```

```
37         raise RuntimeError("All states visited – giving up (no solution found).")
38     state = random_permutation(n)
39     while tuple(state) in visited:
40         state = random_permutation(n)
41     visited.add(tuple(state))
42     while True:
43         cost = compute_cost(state)
44         if cost == 0:
45             return state, restarts
46         best_neighbor = None
47         best_cost = float("inf")
48         for nb in neighbors_by_swaps(state):
49             c = compute_cost(nb)
50             if c < best_cost:
51                 best_cost = c
52                 best_neighbor = nb
53         if best_cost < cost:
54             state = best_neighbor
55             visited.add(tuple(state))
56         else:
57             restarts += 1
58             if max_restarts is not None and restarts >= max_restarts:
59                 raise RuntimeError(f"Stopped after {restarts} restarts (no solution
60                                     found).")
61             break
62     def format_board(state):
63         n = len(state)
64         lines = []
65         for r in range(n):
66             lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n)))
67         return "\n".join(lines)
68     if __name__ == "__main__":
69         n = 4
70         solution, restarts = hill_climb_with_restarts(n)
71         print("Found solution:", solution)
72         print(format_board(solution))
```

## Output

Shivansh Aswal / 1BM23CS315

Found solution: [1, 3, 0, 2]

- - Q -

Q - - -

- - - Q

- Q - -

=== Code Execution Successful ===

## 5. Simulated Annealing to Solve 8-Queens problem

Algorithm:

### Simulated Annealing





#### Pseudocode

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

8-23-21

18 09 21  
Log

Code & Output:

```
main.py    Share  Run

1 print("Shivansh Aswal / 1BM23CS315")
2 import random
3
4 def calculate_cost(board):
5     cost = 0
6     n = len(board)
7     for i in range(n):
8         for j in range(i + 1, n):
9             if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
10                 cost += 1
11     return cost
12
13 def get_neighbors(board):
14     neighbors = []
15     n = len(board)
16     for row in range(n):
17         for new_col in range(n):
18             if new_col != board[row]:
19                 new_board = board[:]
20                 new_board[row] = new_col
21                 neighbors.append(new_board)
22     return neighbors
23
24 def hill_climbing(n, max_restarts=100):
25     solutions = set()
26
27     for restart in range(max_restarts):
28         current_board = [random.randint(0, n-1) for _ in range(n)]
29         current_cost = calculate_cost(current_board)
30
31         while current_cost > 0:
32             neighbors = get_neighbors(current_board)
33             next_board = None
34             next_cost = current_cost
35
36             for neighbor in neighbors:
```

## Output

Shivansh Aswal / 1BM23CS315

Best position found: [6, 2, 0, 5, 7, 4, 1, 3]

Number of non-attacking pairs: 28

Board:

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

=== Code Execution Successful ===

# Propositional Logic

→ truth table for connectives

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \leftrightarrow Q$
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

→ Enumeration Method,  $\alpha = A \vee B$ ,  $KB = (A \vee C) \wedge (B \vee \neg C)$

A	B	C	$A \vee C$	$B \vee \neg C$	KB	$\alpha$
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

A, B, C values for which  $KB \rightarrow \alpha$  true,  $\alpha \rightarrow$  true:  
 $\neg (F, T, T)$ ,  $(T, F, F)$ ,  $(T, T, F)$ ,  $(T, T, T)$  y

## Algorithm

- List all symbols that appear in the KB and  $\alpha$
- For every possible combination of true/false assignments to these symbols:
- Check if KB is true under this assignment. If yes, check if  $\alpha$  is true under the assignment. If there is any assignment where KB is true and  $\alpha$  is false, KB does not entail  $\alpha$

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

Algorithm:

Sample case:  $KB = (A \wedge B) \vee \neg C$   
 $\alpha = A \rightarrow (B \vee C)$

	A	B	C	$A \wedge B$	$B \vee C$	KB	$\alpha$
$\rightarrow$	T	T	T	T	T	(T)	(T)
$\rightarrow$	T	T	F	T	T	(T)	(T)
	T	F	T	F	T	F	(T)
	T	F	F	F	F	F	T
	F	T	T	F	T	T	F
$\rightarrow$	F	T	F	F	T	(T)	(T)
	F	F	T	F	T	(T)	(T)
$\rightarrow$	F	F	F	F	F	F	T

eg 8 S, T are individual variables

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

b:  $(S \wedge T)$

c:  $T \vee \neg T$

Truth table ① a entails b

② a entails b





S	T	$\neg (S \vee T)$	b	c
T	T	F	T	T
T	F	F	F	T
F	T	F	F	T
F	F	T	F	T

a  $\Rightarrow$  b (entails)

a  $\not\Rightarrow$  c (entails)



Code & Output:

```
main.py    Share  Run

1 print("Shivansh Aswal / 1BM23CS315")
2 import itertools
3
4 def evaluate_formula(formula, truth_assignment):
5     eval_formula = formula
6     for symbol, value in truth_assignment.items():
7         eval_formula = eval_formula.replace(symbol, str(value))
8     return eval(eval_formula)
9
10 def generate_truth_table(variables):
11     return list(itertools.product([False, True], repeat=len(variables)))
12
13 def is_entailed(KB_formula, alpha_formula, variables):
14     truth_combinations = generate_truth_table(variables)
15     print(f"{' ' .join(variables)} | KB Result | Alpha Result")
16     print("-" * (len(variables) * 2 + 15))
17     for combination in truth_combinations:
18         truth_assignment = dict(zip(variables, combination))
19         KB_value = evaluate_formula(KB_formula, truth_assignment)
20         alpha_value = evaluate_formula(alpha_formula, truth_assignment)
21         result_str = " ".join(["T" if value else "F" for value in combination])
22         print(f"{result_str} | {'T' if KB_value else 'F'} | {'T' if alpha_value else 'F'}")
23     if KB_value and not alpha_value:
24         return False
25     return True
26
27 KB = "(A or C) and (B or not C)"
28 alpha = "A or B"
29 variables = ['A', 'B', 'C']
30
31 if is_entailed(KB, alpha, variables):
32     print("\nThe knowledge base entails alpha.")
33 else:
34     print("\nThe knowledge base does not entail alpha.")
```

## Output

Shivansh Aswal / 1BM23CS315

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.

=== Code Execution Successful ===

## 7. Implement unification in first order logic

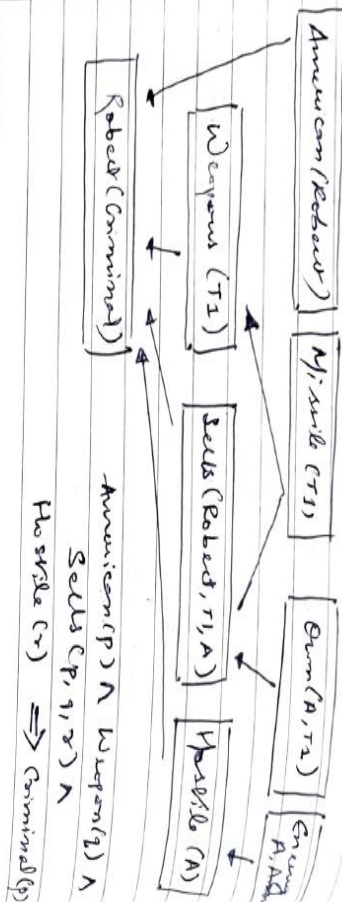
Algorithm:

First Order Logic

Date: 13/10/20  
Page: \_\_\_\_\_

Given: As per the case, it is a crime for an American to sell weapons to hostile nations. Country A is a country of America, has some missiles, and all missiles were sold by Robert, who is an American citizen. From this, "Robert is a criminal".

PROOF  
CHAINING



Forward Reasoning Algorithm

function FOL-FC-ASK(KB,  $\gamma$ ) returns a subgoal/KB or input: KB, the knowledge base, a set of subgoals defined clauses  $\gamma$ , given, an atomic sentence local variables: new, the new sentences inferred on each iteration

repeat until new is empty  
new  $\leftarrow \{\}$

for each rule in KB do

$(\gamma_1 \wedge \dots \wedge \gamma_n \Rightarrow \gamma) \leftarrow \text{STANDARDIZE-VARIABLES}(\text{rule})$   
for each  $\theta$  such that  $\text{MATCH-SUBST}(\theta, \gamma_1 \wedge \dots \wedge \gamma_n) =$

$\text{SUBST}(\theta, \gamma_1 \wedge \dots \wedge \gamma_n)$   
do some  $\gamma_1', \dots, \gamma_n'$  in KB  
 $\gamma' \leftarrow \text{SUBST}(\theta, \gamma)$

if  $q$  does not unify with some sentence already in KB or neither then  
 add  $q$  to new  
 $\Phi \leftarrow \text{unify}(q, \infty)$   
 if  $\Phi$  is not fail then return  $\Phi$   
 add new to KB  
 return false

### Unification Algorithm unify( $\psi_1, \psi_2$ )

- 58. If  $\psi_1$  or  $\psi_2$  is a variable or constant, then:
  - a) If  $\psi_1$  or  $\psi_2$  are identical, then return nil.
  - b) Else if  $\psi_1$  is a variable,
    - a. then if  $\psi_1$  occurs in  $\psi_2$ , return FAILURE
    - b. Else return  $\tau(\psi_2 / \psi_1)$
  - c) Else if  $\psi_2$  is a variable,
    - a. If  $\psi_2$  occurs in  $\psi_1$ , then return FAILURE,
    - b. Else return  $\tau(\psi_1 / \psi_2)$
  - d) Else return FAILURE

- If the individual predicate symbols in  $\psi_1$  and  $\psi_2$  are not same, then return FAILURE
- If  $\psi_1$  and  $\psi_2$  have a different no. of arguments, then return FAILURE
- Set substitution set (subst) to nil.
- For  $i = 1$  to the no. of elements in  $\psi_1$ ,

- a) call leafy function with the element of  $\psi_1$ , and the element of  $\psi_2$ , and push result into  $S$ .
- b) if  $S = \text{failure}$  then return failure.
- c) if  $S \neq \text{NIL}$  then do,
  - a. Apply  $S$  to remainder of both  $L1$  &  $L2$ .
  - b.  $\text{STACK} = \text{APPEND}(S, \text{SUBST})$ .

Return SUBST.

### Unification Problems (continued)

1. Unifying:  $p(b, x, f(g(z)))$  and  $p(c, f(y), f(y))$   
~~Unification fails~~  $b$  and  $c$ ,  $b$  is constant,  $c$  is variable

2. Unifying  $z = L$

$$x \text{ and } R(y) \rightarrow x = p(y)$$

$$R(g(z)) \text{ and } p(y) \Rightarrow y(z) = g$$

$$\text{MGU: } \sigma = \{z/b, x/g(y), y(g(z))\}$$

3.  $\sigma(g(a, g(x, a), f(y)))$  and  $\sigma(x, g(f(a), a), x)$   
 $a$  and  $b$  match  $x$   
 $R(x, a)$  and  $g(R(a), a) \rightarrow x = R(a)$   
 $p(y)$  and  $x \Rightarrow x = R(y) = R(a)$   
 $\sigma = \{x/R(a), y/b\}$

4.  $\sigma(p(R(a)), g(y), f(x, x))$   
 $R(a) = x$   
 $g(y) = x$   
 $R(x) = g(y)$  but  $p(a)$  and  $g(y)$  are different so no unification

## Code & Output:

```
1  import re
2  from collections import namedtuple
3
4  Var = namedtuple('Var', ['name'])
5  Const = namedtuple('Const', ['name'])
6  Func = namedtuple('Func', ['name', 'args'])
7
8  def parse(s):
9      s = s.strip()
10     if '(' in s:
11         n, rest = s[:s.index('(')], s[s.index('(')+1:-1]
12         args = []
13         depth = 0; current = []
14         for c in rest + ',':
15             if c == ',' and depth == 0:
16                 args.append(''.join(current).strip())
17                 current = []
18             else:
19                 if c == '(': depth += 1
20                 elif c == ')': depth -= 1
21                 current.append(c)
22         return Func(n, [parse(a) for a in args])
23     if re.fullmatch(r'[a-z][a-z0-9]*', s): return Var(s)
24     return Const(s)
25
26 def occurs(v, x, s):
27     x = subst(x, s)
28     if v == x: return True
29     if isinstance(x, Func):
30         return any(occurs(v, a, s) for a in x.args)
31     return False
32
33 def subst(t, s):
34     while isinstance(t, Var) and t.name in s:
35         t = s[t.name]
36     if isinstance(t, Func):
```

```

37         return Func(t.name, [subst(a, s) for a in t.args])
38     return t
39
40 def unify(t1, t2, s=None):
41     if s is None: s = {}
42     t1, t2 = subst(t1, s), subst(t2, s)
43     if t1 == t2: return s
44     if isinstance(t1, Var):
45         if occurs(t1, t2, s): return None
46         s[t1.name] = t2
47         return s
48     if isinstance(t2, Var):
49         if occurs(t2, t1, s): return None
50         s[t2.name] = t1
51         return s
52     if isinstance(t1, Func) and isinstance(t2, Func):
53         if t1.name != t2.name or len(t1.args) != len(t2.args): return None
54         for a1, a2 in zip(t1.args, t2.args):
55             s = unify(a1, a2, s)
56             if s is None: return None
57         return s
58     if isinstance(t1, Const) and isinstance(t2, Const) and t1.name == t2.name:
59         return s
60     return None
61
62 def to_str(t):
63     if isinstance(t, Var) or isinstance(t, Const):
64         return t.name
65     return f"{t.name}({','.join(to_str(a) for a in t.args)})"
66
67 def show_subs(s):
68     if s is None:
69         print("Unification failed.")
70     elif not s:
71         print("No substitution needed.")

```

```

72         else:
73             for k,v in s.items():
74                 print(f"{k} = {to_str(v)}")
75     print("Name:Shivansh Aswal\nUSN:1BM23CS315\n\n")
76     tests = [
77         ("p(b,X,f(g(Z)))", "p(z,f(Y),f(Y))"),
78         ("Q(a,g(x,a),f(y))", "Q(a,g(f(b),a),x)"),
79         ("p(f(a),g(Y))", "p(X,X)"),
80         ("prime(11)", "prime(y)"),
81         ("knows(John,x)", "knows(y,mother(y))"),
82         ("knows(John,x)", "knows(y,Bill)")
83     ]
84
85     for e1, e2 in tests:
86         print(f"Unifying: {e1} and {e2}")
87         s = unify(parse(e1), parse(e2))
88         show_subs(s)
89         print('-'*40)

```

Name:Shivansh Aswal

USN:1BM23CS315

Unifying:  $p(b,X,f(g(Z)))$  and  $p(z,f(Y),f(Y))$

Unification failed.

-----  
Unifying:  $Q(a,g(x,a),f(y))$  and  $Q(a,g(f(b),a),x)$

$x = f(b)$

$y = b$

-----  
Unifying:  $p(f(a),g(Y))$  and  $p(X,X)$

Unification failed.

-----  
Unifying:  $\text{prime}(11)$  and  $\text{prime}(y)$

$y = 11$

-----  
Unifying:  $\text{knows}(\text{John},x)$  and  $\text{knows}(y,\text{mother}(y))$

$y = \text{John}$

$x = \text{mother}(\text{John})$

-----  
Unifying:  $\text{knows}(\text{John},x)$  and  $\text{knows}(y,\text{Bill})$

$y = \text{John}$

$x = \text{Bill}$





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

Algorithm:

- ④  $x \text{ prime}(p)$  and  $\text{prime}(x)$   
 $\forall \text{ count } f = 0$   
 $\text{MGR} : x \text{ g}(11)$
  - ⑤  $x \text{ knows}(\text{John}, x)$ ,  $\text{knows}(y, \text{Bill})$   
 $\text{John} = y$   
 $x \text{ mother}(y)$   
 $\text{MGR} : x \text{ g}(\text{John}, x \mid \text{mother}(\text{John}))$
  - ⑥  $x \text{ knows}(\text{John}, x)$ ,  $\text{knows}(y, \text{Bill})$   
 $\text{First argument John} = y$   
 $\text{Second argument } x = \text{Bill}$   
 $\text{MGR} = x \text{ g}(\text{John}, x \mid \text{Bill})$
- Forward Reasoning Problem (over)  
 It is a crime for American to sell weapons to hostile nations
- Let's say  $p, q$  and  $x$  are variables  
 $\text{American}(p) \wedge \text{Weapon}(q) \wedge \text{Sells}(p, q, x) \wedge \text{Hostile}(x)$   
 $\Rightarrow \text{Criminal}(p)$   
 Country A has some missiles  
 $\exists x \text{ Guns}(A, x) \wedge \text{Missile}(x)$
- Goal: find instantiation, introducing a new constant  $\tau$ :  
 $\text{Guns}(A, \tau)$   
 $\text{Missile}(\tau)$   
 All of the missiles were sold to country A by Robert  
 $\forall x \text{ Missile}(x) \wedge \text{Guns}(A, x) \Rightarrow \text{Sells}(\text{Robert}, x, A)$   
~~Missiles are weapons~~  
 $\text{Missile}(x) \Rightarrow \text{Weapon}(x)$





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

Algorithm:

### Resolution in FOL

24/10/25

Steps to convert logic statement to CNF

1. Eliminate biconditional and 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,  $\exists x \text{ Rich}(x)$  becomes  $\text{Rich}(c_1)$  where  $c_1$  is a new Skolem constant.

• "Everyone has a heart"  $\forall x \text{ Person}(x) \rightarrow \exists y \text{ Heart}(y) \wedge \text{Has}(x, y)$  becomes  $\forall x \text{ Person}(x) \rightarrow \text{Heart}(f_1(x)) \wedge \text{Has}(x, f_1(x))$ , where  $f_1$  is a new symbol.

5. Drop universal quantifiers  
(For instance  $\forall x \text{ Person}(x)$  becomes  $\text{Person}(x)$ )



6 Distribute  $\neg$  over  $\vee$ :

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

Basic steps for proving a conclusion S from premises

1. Convert all sentences to CNF
2. Negate conclusion S and convert result to CNF
3. Add negated conclusion S to the premises clauses
4. Repeat until contradiction or no progress is made:
  - a. Select 2 clauses (parent clauses)
  - b. Resolve them together, performing all required unifications.
  - c. If resultant is the empty clause, a contradiction has been found
  - d. If not, add resultant to the premises

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

Given the KB or Premises

- a. John likes all kind of food
- b. Apple and vegetables are food
- c. anything organic eats and not killed is food.
- d. Animal eats premises and still alive
- e. Henry eats everything that animal eats
- f. Anyone who is alive implies not killed
- g. Anybody who is not killed implies alive

Prove by resolution that: John likes premises





## Code & Output:

```
Code Blame 83 lines (67 loc) · 2.84 KB

1  def negate(literal):
2      return literal[1:] if literal.startswith("~") else "~" + literal
3
4  def resolve(ci, cj):
5      resolvents = []
6      for lit1 in ci:
7          for lit2 in cj:
8              if lit1 == negate(lit2):
9                  new_clause = list(set(ci + cj))
10                 new_clause.remove(lit1)
11                 new_clause.remove(lit2)
12                 resolvents.append((new_clause, lit1, lit2))
13
14     return resolvents
15
16 def resolution_algorithm(clauses):
17     new = set()
18     tree = {}
19     step = 0
20
21     while True:
22         pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i+1, len(clauses))]
23         for (ci, cj) in pairs:
24             results = resolve(ci, cj)
25             for (resolvent, lit1, lit2) in results:
26                 step += 1
27                 resolvent_tuple = tuple(sorted(resolvent))
28                 if resolvent_tuple not in tree:
29                     tree[resolvent_tuple] = ((tuple(ci), tuple(cj)), (lit1, lit2))
30                     print(f"\nStep {step}: Resolving {ci} and {cj} on {lit1} / {lit2} ⇒ {resolvent}")
31
32             if resolvent == []:
33                 print("\n🚫 Contradiction (empty clause) found!")
34                 print("✅ Query is PROVED TRUE by Resolution.\n")
35                 print("🌲 Resolution Tree (Trace):\n")
36                 print_tree(tree, resolvent_tuple)
37                 return True
```

Code Blame 83 lines (67 loc) · 2.84 KB

```
50         if clause not in tree:
51             print(indent + "Clause: " + str(list(clause)))
52             return
53         parents, resolved = tree[clause]
54         lit1, lit2 = resolved
55         print(indent + f"Clause: {list(clause)} (resolved {lit1}/{lit2})")
56         print(indent + " └─ From:")
57         helper(parents[0], indent + " | ")
58         print(indent + " └─ And:")
59         helper(parents[1], indent + " ")
60
61     helper(final_clause)
62
63     print("🔴 FIRST ORDER LOGIC RESOLUTION SYSTEM (with Tree)")
64     print("-----")
65
66     n = int(input("Enter number of statements in the Knowledge Base: "))
67     kb = []
68     for i in range(n):
69         stmt = input(f"Enter statement {i+1} (in CNF using v for OR): ")
70         kb.append(stmt)
71
72     query = input("\nEnter the query to prove: ")
73
74     negated_query = "~" + query if not query.startswith("~") else query[1:]
75
76     clauses = [stmt.replace(" ", "").split("v") for stmt in kb]
77     clauses.append([negated_query])
78
79     print("\n🟡 Clauses for Resolution:")
80     for i, c in enumerate(clauses, 1):
81         print(f"{i}. {c}")
82
83     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 v for OR): ~Food(Peanuts) v Likes(John,Peanuts)
Enter statement 2 (in CNF using v 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) ⇒ []

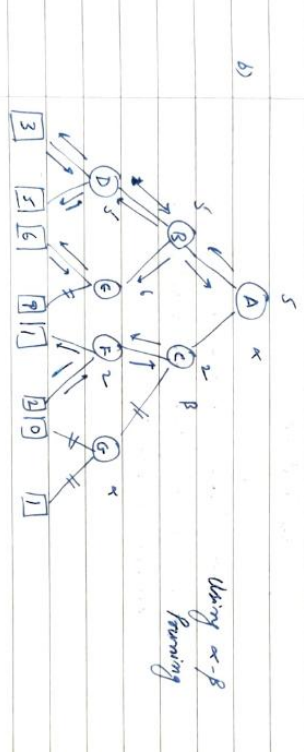
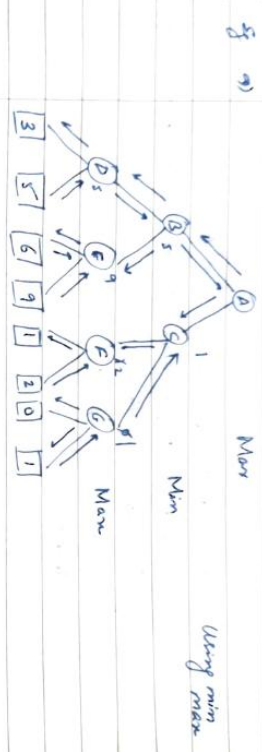
🚩 Contradiction (empty clause) found!
✅ Query is PROVED TRUE by Resolution.

🌲 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

```

# Advanced Search

## Alpha Beta Search



## 9. Implement Alpha-Beta Pruning.

Algorithm:

```

function ALPHA BETA SEARCH (node) return action
    v ← MAX-VALUE (node, -∞, +∞)

function MAX-VALUE (node, α, β) return action
    value ← TERMINAL-TEST (node) then return v
    v ← -∞
    for each a in action (node) do
        v ← MAX(v, MIN-VALUE (node(a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
    return v
    
```

## Code & Output:

```

1 + def negate(literal):
2 +     return literal[1:] if literal.startswith("~") else "~" + literal
3 +
4 + def resolve(ci, cj):
5 +     resolvents = []
6 +     for lit1 in ci:
7 +         for lit2 in cj:
8 +             if lit1 == negate(lit2):
9 +                 new_clause = list(set(ci + cj))
10 +                 new_clause.remove(lit1)
11 +                 new_clause.remove(lit2)
12 +                 resolvents.append((new_clause, lit1, lit2))
13 +     return resolvents
14 +
15 + def resolution_algorithm(clauses):
16 +     new = set()
17 +     tree = {}
18 +     step = 0
19 +
20 +     while True:
21 +         pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i+1, len(clauses))]
22 +         for (ci, cj) in pairs:
23 +             results = resolve(ci, cj)
24 +             for (resolvent, lit1, lit2) in results:
25 +                 step += 1
26 +                 resolvent_tuple = tuple(sorted(resolvent))
27 +                 if resolvent_tuple not in tree:
```

```

28 +         tree[resolver_tuple] = ((tuple(ci), tuple(cj)), (lit1, lit2))
29 +         print(f"\nStep {step}: Resolving {ci} and {cj} on {lit1} / {lit2} ⇒ {resolver}")
30 +
31 +         if resolver == []:
32 +             print("\n🚫 Contradiction (empty clause) found!")
33 +             print("✅ Query is PROVED TRUE by Resolution.\n")
34 +             print("🌲 Resolution Tree (Trace):\n")
35 +             print_tree(tree, resolver_tuple)
36 +             return True
37 +
38 +         new.add(resolver_tuple)
39 +
40 +         if new.issubset(set(map(tuple, clauses))):
41 +             print("\n⚠️ No contradiction found. Query CANNOT be proved from the Knowledge Base.")
42 +             return False
43 +
44 +         for clause in new:
45 +             if list(clause) not in clauses:
46 +                 clauses.append(list(clause))
47 +
48 +     def print_tree(tree, final_clause):
49 +         def helper(clause, indent=""):
50 +             if clause not in tree:
51 +                 print(indent + "Clause: " + str(list(clause)))
52 +                 return
53 +             parents, resolved = tree[clause]
54 +             lit1, lit2 = resolved
55 +             print(indent + f"Clause: {list(clause)} (resolved {lit1}/{lit2})")

```

```

56 +         print(indent + " └─ From:")
57 +         helper(parents[0], indent + " | ")
58 +         print(indent + " └─ And:")
59 +         helper(parents[1], indent + " ")
60 +
61 +     helper(final_clause)
62 +
63 + print("🔴 FIRST ORDER LOGIC RESOLUTION SYSTEM (with Tree)")
64 + print("-----")
65 +
66 + n = int(input("Enter number of statements in the Knowledge Base: "))
67 + kb = []
68 + for i in range(n):
69 +     stmt = input(f"Enter statement {i+1} (in CNF using v for OR): ")
70 +     kb.append(stmt)
71 +
72 + query = input("\nEnter the query to prove: ")
73 +
74 + negated_query = "~" + query if not query.startswith("~") else query[1:]
75 +
76 + clauses = [stmt.replace(" ", "").split("v") for stmt in kb]
77 + clauses.append([negated_query])
78 +
79 + print("\n🟡 Clauses for Resolution:")
80 + for i, c in enumerate(clauses, 1):
81 +     print(f"{i}. {c}")
82 +
83 + resolution_algorithm(clauses)

```

```

🔗 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 ( $\alpha \geq \beta$ )
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 ( $\alpha \geq \beta$ )
MAX: Depth=0, Node=0, Alpha=5, Beta=inf

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

```

