

**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

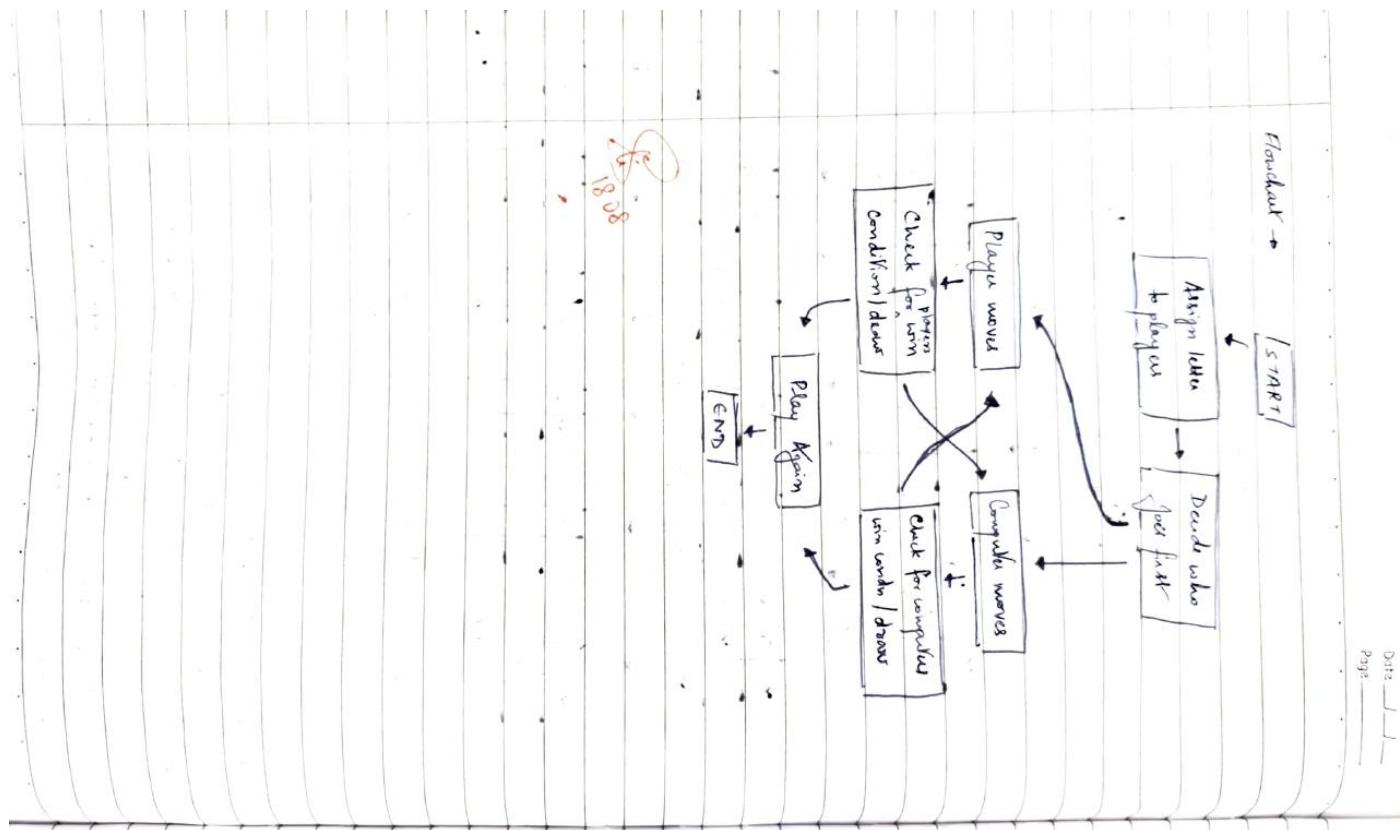
Github Link:

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

Program 1

Implement Tic – Tac – Toe Game

Algorithm:



Week I

Implement TIC TAC TOE game

X	.	0	
0	X	X	Initial instance
		0	

x		0	x	x	0	x		0
0	x	x	0	x	x	0	x	x
x	0		0			0		x

$$\text{Cash} \rightarrow 1+0 = 1 \quad \text{or} \quad 0+\$ = \$ \quad \text{or} \quad \$+\$ = \$$$

Dates
Win (%)

Dear

DRAFT

1

Wim(s)

$\text{win}(x)$

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
                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

```

```

        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): 1
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

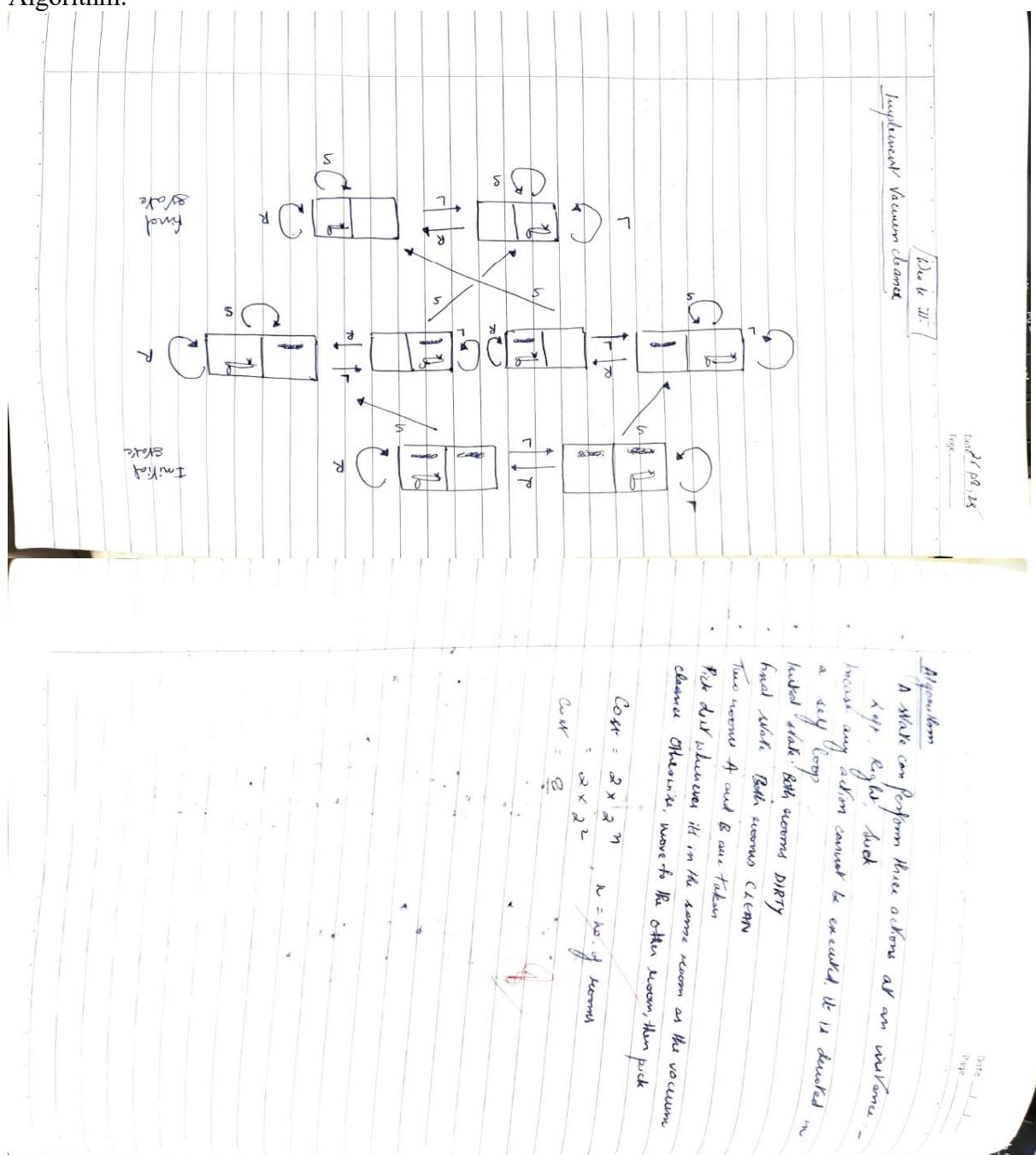
Enter O row (1-3): 1
Enter O col (1-3): 3
Board after O move:
- - O
O O X
X - X

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:

The screenshot shows a code editor window with the file name "main.py" at the top left. At the top right, there are four icons: a copy symbol, a brightness symbol, a share symbol, and a "Run" button. The main area contains the following Python code:

```
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}={status}" 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
```

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

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=Clean | 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

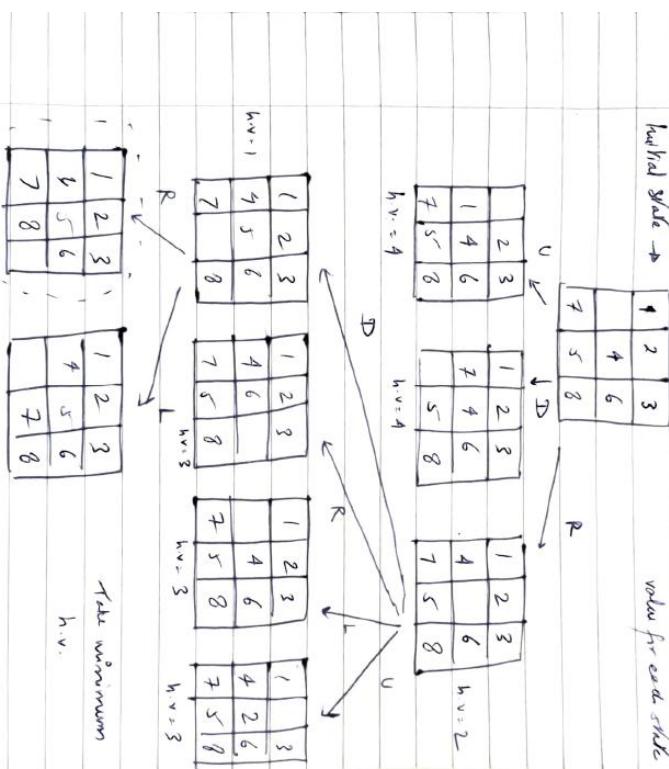
All rooms are clean. Task completed!
Total cost of operation: 4
```

2. BFS with Heuristic

Algorithm:

BFS using Heuristic Approach
→ Using no of misplaced tiles Method

Goal State → $\begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline 7 & 8 & \end{array}$
Calculate heuristic value for each state



Solution (Goal Node)

Properties

SMPT

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

Code:

The screenshot shows a Jupyter Notebook cell with the following code:

```
main.py
```

```
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
21 def get_neighbors(state):
22     neighbors = []
23     x, y = find_blank(state)
24     moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
25
26     for dx, dy in moves:
27         nx, ny = x + dx, y + dy
28         if 0 <= nx < 3 and 0 <= ny < 3:
29             new_state = [row[:] for row in state]
30             new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
31             neighbors.append(new_state)
32
33 def best_first_search(start_state):
34     visited = set()
35     pq = []
36
```

The code implements a Best-First Search algorithm for a 3x3 sliding puzzle. It defines functions for printing the author's name and ID, importing heapq, defining the goal state, calculating the heuristic (number of misplaced tiles), finding the blank tile position, generating neighbors by moving the blank tile, and performing the search starting from a given state.

```
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]

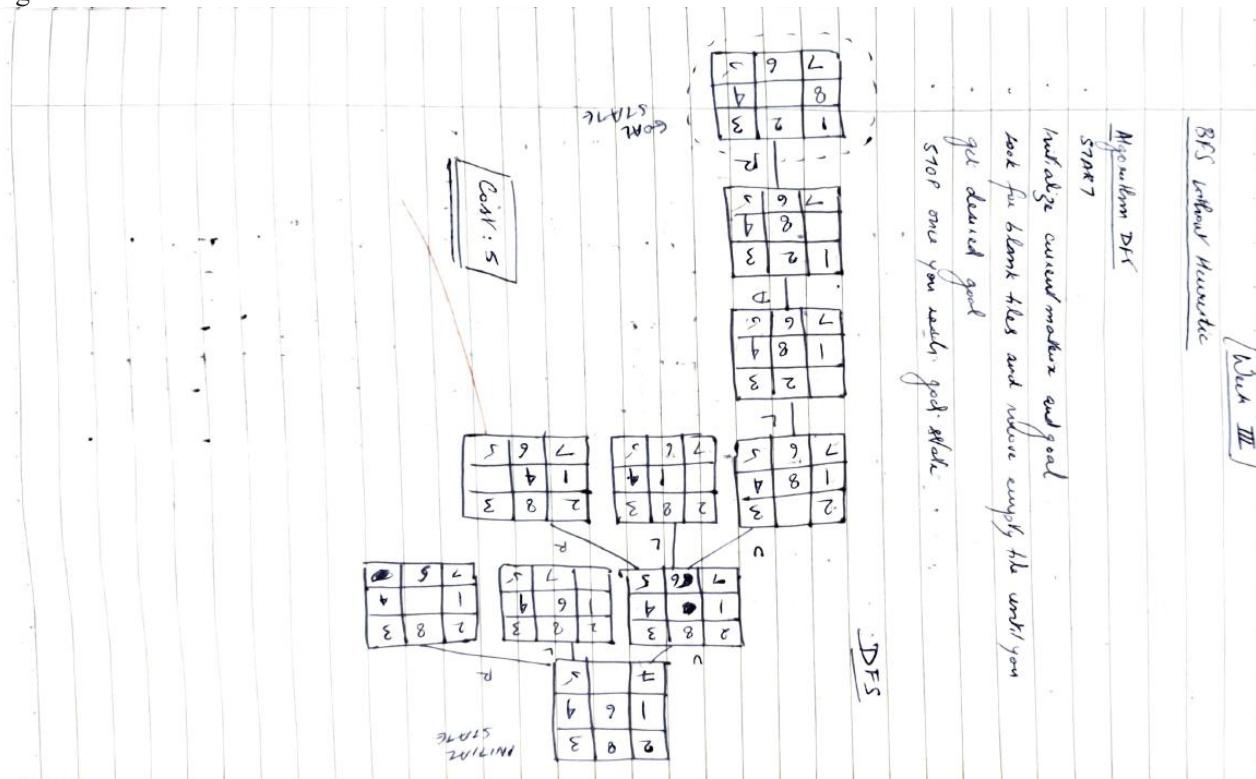
[4, 5, 6]

[7, 8, 0]

==== Code Execution Successful ===

BFS without Heuristic

Algorithm:



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 tlat[i] > tlat[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)

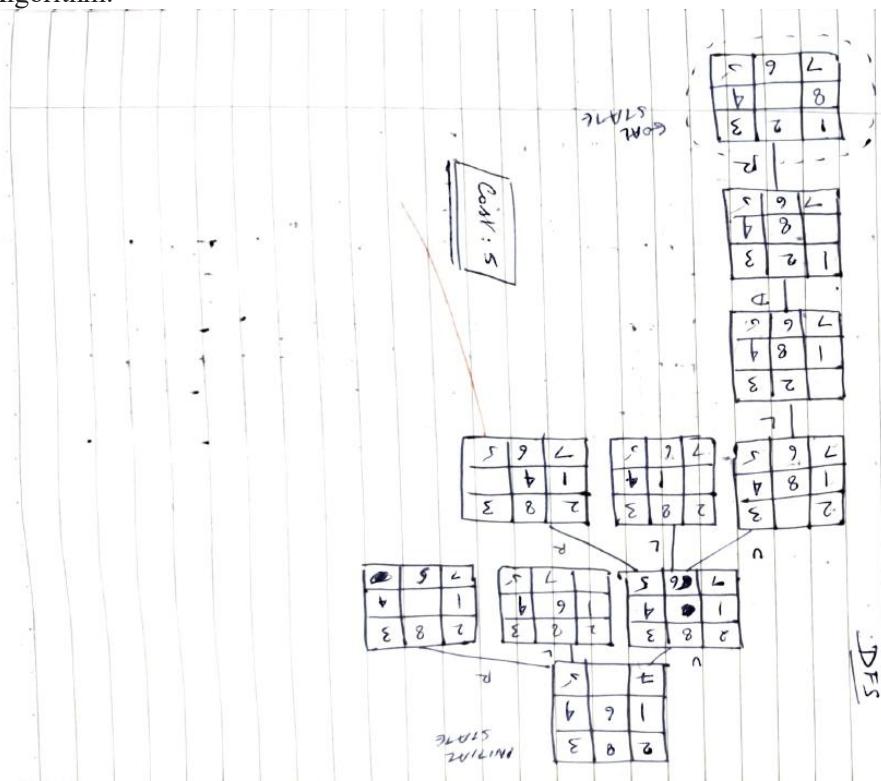
```

Output:

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

DFS

Algorithm:



BFS without heuristic

Week III

Date: 01.07.25
Page: _____

Algorithm DFS

START

Initialize current state and goal

Look for blank tiles and move empty tile until you

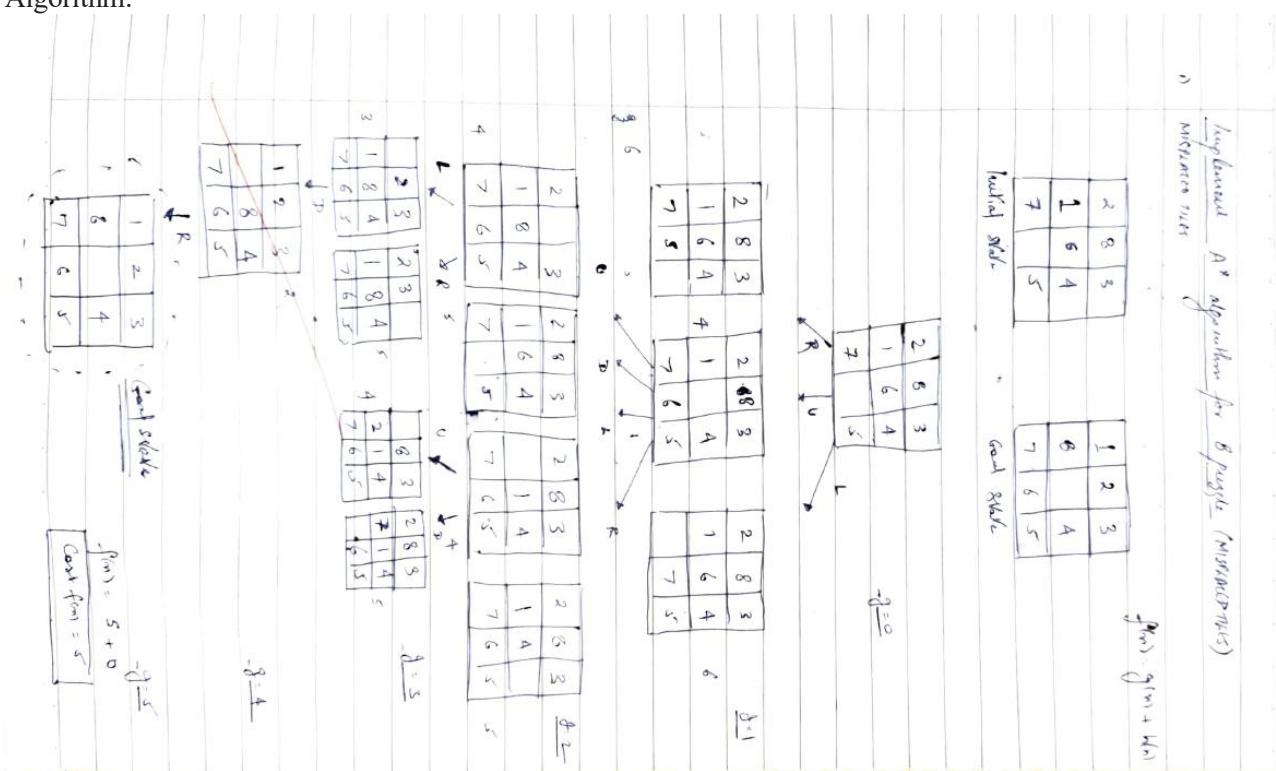
get desired goal

STOP once you reach goal state

DFS

3. Implement A* search algorithm

Algorithm:



2) Manhattan Distance

Initial state

1	5	8
3	2	.
4	6	7

1	5	8
3	2	.
4	6	7

1	5	8
3	2	.
4	6	7

Level - 0

$f_{(0,0)} = g_{(0,0)} + h_{(0,0)}$

Date: 1/1
Page:

L: 1 2 3 4 5 6 7 8

U: 0 1 3 1 1 2 2 2 → 12

D: ~~0 1 3 1 1 2 2 2~~ → ~~14~~

0 1 3 1 1 2 3 3 → 14

Level - II

1	5	.
3	2	8
4	6	7

L ↴ ↴ ↴ ↴ ↴ ↴ ↴ ↴

1	5	.
3	2	8
4	6	7

Level III

12

14

ALGORITHM

START

- Initialize values for current and goal state.
- Consider all moves and pick the state with smallest $f_{(n)}$ value until goal state is achieved.
- Repeat steps if goal state not reached. For manhattan use $m(n)$
- 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 ===
```

Hill Climbing (4 Queen)

Date: 15/09/20
Page _____

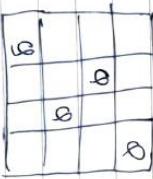
Pseudo Code

- 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 max")
 - d. Otherwise, make the best move and update conflicts
 - If conflicts == 0
problem found
 - Else : No soln

State Space Diagram

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

Conflicts = 2



②

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

Conflicts = 2



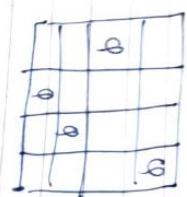
4. Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

③

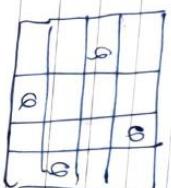
$$x_0, x_1, x_2, x_3 = (-3, 0, 2, 1)$$

$$R_{\Sigma}$$



④

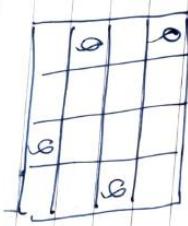
$$x_0, x_1, x_2, x_3 = (2, 0, 3, 0)$$



$$R_{\Sigma}$$

⑤

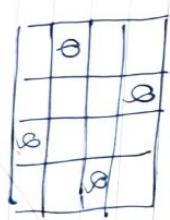
$$\rightarrow (0, 3, 0, 2)$$



$$R_{\Sigma}$$

⑥

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



$$R_{\Sigma}$$

Code & Output:

The screenshot shows a Jupyter Notebook cell with the following code:

```
main.py
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
```

The code is a Python script named `main.py` that implements a hill-climbing algorithm for the N-Queens problem. It includes functions for printing the author's name and ID, calculating the cost of a state (number of conflicts), generating random permutations, generating neighbors by swapping two columns, and performing hill-climbing with restarts.

```
main.py
```

```
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 def format_board(state):
62     n = len(state)
63     lines = []
64     for r in range(n):
65         lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n)))
66     return "\n".join(lines)
67 if __name__ == "__main__":
68     n = 4
69     solution, restarts = hill_climb_with_restarts(n)
70     print("Found solution:", solution)
71     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

Pseudo Code

```
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:

The screenshot shows a Jupyter Notebook cell with the following code:

```
main.py
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
12     return cost
13
14 def get_neighbors(board):
15     neighbors = []
16     n = len(board)
17     for row in range(n):
18         for new_col in range(n):
19             if new_col != board[row]:
20                 new_board = board[:]
21                 new_board[row] = new_col
22                 neighbors.append(new_board)
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:
```

The code implements a hill climbing algorithm to solve the N-Queens problem. It defines functions for calculating the cost of a board configuration, generating neighbors, and performing the search. The main function `hill_climbing` takes the number of queens `n` and the maximum number of restarts `max_restarts` as arguments. It uses a set to store unique solutions found.

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 ===
```

→ 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	α
F	F	F	F	T	F	
F	F	T	T	F	F	
F	T	F	F	T	F	
F	T	T	T	T	T	
T	F	F	T	T	T	
T	F	T	T	F	T	
T	T	F	T	T	T	
T	T	T	T	T	T	T

A, B, C values for which $KB \rightarrow \text{true}$, $\alpha \rightarrow \text{true}$:

$\langle F, T, T \rangle, (T, F, F), (T, T, F), (T, T, T)$

Algorithm

- list all symbols that appear in the KB and α
- for every possible combination of true/false assignments

to these symbols:

- check if KB is true under this assignment. If yes,
- check if α is true under the assignment. If true in any assignment where KB is true and α is false,
- KB does not entail α

Example Case:

$$\alpha = A \rightarrow (\beta \vee c)$$

$$KB = (A \wedge B) \vee \neg C$$

A	B	C	$A \wedge B$	$B \vee C$	KB
T	T	T	T	T	T
T	T	F	F	T	T
T	F	T	F	T	F
F	T	T	F	F	F
F	T	F	F	T	F
F	F	T	F	T	T
F	F	F	F	F	F
F	F	F	F	F	F
F	F	F	F	F	F

$\exists x$ s, t are uninitalized variables

$$a: \neg (s \vee t)$$

$$b: (s \wedge t)$$

$$c: \neg t \vee \neg t$$

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	⊕	F	T
F	F	⊕	F	T

$$q \Rightarrow b \quad (\text{a variable})$$

$$q \not\Rightarrow c \quad (\text{a variable})$$

F	F	F
F	F	F
F	F	T

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

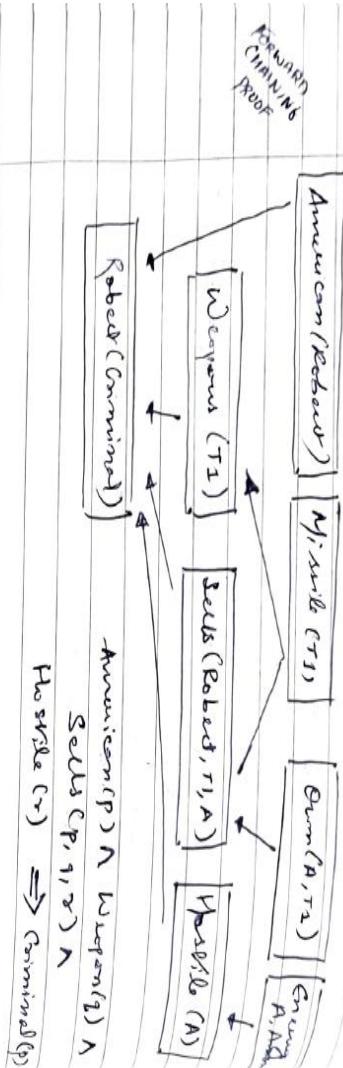
The knowledge base entails alpha.

==== Code Execution Successful ===

First Order Logic

Date: 23/10/2023
Page:

Given
As per the Law, it is a crime for an American
to sell weapons to hostile nations. Consider,
an enemy of America, has some missiles, and
all missiles were sold by Robert, who is an
American citizen. Prove that "Robert is criminal".



Forward Reasoning Algorithm

function FOL-FC-ASK(KB , α) returns a substitution or
input: KB , the knowledge base, a set of first-order
order definite clauses & query, an atomic sentence
to be asked; α , the new sentences inferred on each
iteration

repeat until α is empty

new \leftarrow \perp

for each rule in KB do

$(P_1 \wedge \dots \wedge P_n \Rightarrow Q) \leftarrow \text{STANDARDIZE-VARIABLES } Q \text{ (rule)}$
for each σ such that $\text{SUBST}(\theta, P_1 \wedge \dots \wedge P_n) =$

$\text{SUBST}(\sigma, P_1 \wedge \dots \wedge P_n)$ =

$Q' \leftarrow \text{SUBST}(\sigma, Q)$

7. Implement unification in first order logic

Algorithm:

If φ does not unify with some sentence already
in KB or negt. them

add φ to new

$\varphi \leftarrow$ unify (φ', α)

If φ is not fail then return φ
add new to KB

return false

Unification Algorithm

unify (ψ_1, ψ_2)

• If ψ_1 or ψ_2 is a variable or constant, then :

- If ψ_1 or ψ_2 are identical, then return nil.
- Else if ψ_1 is a variable,
 - Then if ψ_1 occurs in ψ_2 , return failure
 - Else return $\{ \psi_1 / \psi_2 \}$
- Else if ψ_2 is a variable,
 - If ψ_2 occurs in ψ_1 , then return failure.
 - Else return $\{ \psi_1 / \psi_2 \}$.
- Else return FAILURE.

• If the individual predicate symbols in ψ_1 and ψ_2 are
not same, then return FAILURE

• If ψ_1 and ψ_2 have a different no. of arguments,
then return FAILURE

• Set substitution set CSUBST to NIL

• For $i = 1$ to the no. of elements in ψ_1 ,

- a) call unify function with the two elements of ψ_1 and ψ_2 , and put result into S .
- b) If $S = \text{failure}$ then return failure.
- c)
 - a. Apply S to remainder of both ψ_1 & ψ_2
 - b. $\text{SUBST} = \text{APPEND}(S, \text{SUBST})$.

Return SUBST.

Unification Problems (cont'd)

- ① \bullet Unifying $P(Cb, X, fgy(Cz))$ and $P(Cz, fgy, fgy)$

α and $R(Y) \rightarrow X = P(Y)$

$R(Y)$ and $P(Y) \rightarrow Y = P(Y)$

$$\begin{aligned} M_G: & Y = R(b, X, fgy), Y(g(Cz)) \\ & \quad \downarrow \\ & X = R(b, X, fgy) \end{aligned}$$

②

$\leftarrow g(a, f(a)), f(y) \right) \text{ and } \alpha(a, f(a), a, y)$

$R(x, a)$ and $f(R(x, a), y) \rightarrow x = R(b)$

$$\begin{aligned} & R(y) \text{ and } x \Rightarrow x = R(y) = R(a) \\ & \quad \downarrow \\ & x = R(a), y = b \end{aligned}$$

$\leftarrow x = R(a), y = b$

③

$\leftarrow P(R(\alpha)), f(y) \right) \leftarrow P(\alpha, x) \rightarrow$

$$R(\alpha) = x$$

$R(\alpha) = f(y)$ but $P(\alpha)$ and $f(y)$ 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: prime(11) and prime(y)

y = 11

Unifying: knows(John,x) and knows(y,mother(y))

y = John

x = mother(John)

Unifying: knows(John,x) and knows(y,Bill)

y = John

x = Bill

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

Algorithm:

④

$\neg \text{prime}(p) \text{ and prime}(q) \rightarrow$
 $p \rightarrow \text{const } p = 0$

$\text{MAV} : \neg g(1, p)$

⑤

$\text{knows}(\text{John}, x), \text{knows}(y, \text{all}) \rightarrow$

$\text{John} = y$

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

$\text{M6U} : \neg g(1, \text{John}), x \neq \text{mother}(\text{John}) \rightarrow$

⑥ $\neg \text{knows}(\text{John}, n), \text{knows}(y, \text{all})$
 First argument $\text{John} = y$
 Second argument $n = \text{B6U}$

$\text{M6U} : \neg g(1, \text{John}), x \neq \text{B6U} \rightarrow$

Forward Reasoning Problem (cont.)

Rf is a crime for American to sell weapons to hostile nations

lets say p, q and r are variables

$\text{American}(p) \wedge \text{Weapon}(q) \wedge \text{Sells}(p, q, r) \wedge \text{Hostile}(r)$

$\Rightarrow \text{Crimes}(p)$

Country A has some missiles
 $\exists x \text{owns}(A, x) \wedge \text{Missile}(x)$

misleading instantiation, introducing a new constant $r1$:

$\text{owns}(A, r1)$

$\text{missile}(r1)$

All of the missiles were sold to Country A by Robert
 $\forall x \text{missile}(x) \wedge \text{Own}(A, x) \Rightarrow \text{Sells}(\text{Robert}, x, A)$

~~Missiles are weapons~~
 $\text{missile}(x) \Rightarrow \text{Weapon}(x)$

Steps to convert logic statement to CNF

1. Eliminate biconditionals 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. Skolemizing: 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 $\text{Rich}(\bar{x})$ where \bar{x} is a new Skolem constant.

- "Everyone has a best" $\forall x \text{ Person}(x) \rightarrow \exists y \text{ Best}(y) \wedge \text{Has}(x, y) \rightarrow \forall x \text{ Person}(x) \rightarrow \text{Best}(\bar{H}(x)) \wedge \text{Has}(x, \bar{H}(x))$, where \bar{H} is a new symbol

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

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 given 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 $\neg S$ clause (parent clause)
 b. Resolve them together, performing all required
 unifications.
 c. If resolution 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 anyone eats and not killed is food.
- d. Tom eats peanuts and still alive
- e. Hung eats everything that Tom eats
- f. Anyone who is alive implies not killed
- g. Anyone who is not killed implies alive

Prove by resolution that : John likes peanuts

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
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[resolvent_tuple] = ((tuple(ci), tuple(cj)), (lit1, lit2))
29                     print(f"\nStep {step}: Resolving {ci} and {cj} on {lit1} / {lit2} => {resolvent}")
30
31             if resolvent == []:
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, resolvent_tuple)
36
37     return True
```

Code Blame 83 lines (67 loc) · 2.84 KB

```

49
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): $\neg\text{Food}(\text{Peanuts}) \vee \text{Likes}(\text{John}, \text{Peanuts})$

Enter statement 2 (in CNF using v for OR): $\text{Food}(\text{Peanuts})$

Enter the query to prove: $\text{Likes}(\text{John}, \text{Peanuts})$

█ Clauses for Resolution:

1. $[\neg\text{Food}(\text{Peanuts})], [\text{Likes}(\text{John}, \text{Peanuts})]$
2. $[\text{Food}(\text{Peanuts})]$
3. $[\neg\text{Likes}(\text{John}, \text{Peanuts})]$

Step 1: Resolving $[\neg\text{Food}(\text{Peanuts})], [\text{Likes}(\text{John}, \text{Peanuts})]$ and $[\text{Food}(\text{Peanuts})]$ on $\neg\text{Food}(\text{Peanuts}) / \text{Food}(\text{Peanuts}) \Rightarrow [\text{Likes}(\text{John}, \text{Peanuts})]$

Step 2: Resolving $[\neg\text{Food}(\text{Peanuts})], [\text{Likes}(\text{John}, \text{Peanuts})]$ and $[\neg\text{Likes}(\text{John}, \text{Peanuts})]$ on $\text{Likes}(\text{John}, \text{Peanuts}) / \neg\text{Likes}(\text{John}, \text{Peanuts}) \Rightarrow [\neg\text{Food}(\text{Peanuts})]$

Step 3: Resolving $[\neg\text{Food}(\text{Peanuts})], [\text{Likes}(\text{John}, \text{Peanuts})]$ and $[\text{Food}(\text{Peanuts})]$ on $\neg\text{Food}(\text{Peanuts}) / \text{Food}(\text{Peanuts}) \Rightarrow [\text{Likes}(\text{John}, \text{Peanuts})]$

Step 4: Resolving $[\neg\text{Food}(\text{Peanuts})], [\text{Likes}(\text{John}, \text{Peanuts})]$ and $[\neg\text{Likes}(\text{John}, \text{Peanuts})]$ on $\text{Likes}(\text{John}, \text{Peanuts}) / \neg\text{Likes}(\text{John}, \text{Peanuts}) \Rightarrow [\neg\text{Food}(\text{Peanuts})]$

Step 5: Resolving $[\text{Food}(\text{Peanuts})]$ and $[\neg\text{Food}(\text{Peanuts})]$ on $\text{Food}(\text{Peanuts}) / \neg\text{Food}(\text{Peanuts}) \Rightarrow []$

✖ Contradiction (empty clause) found!

✓ Query is PROVED TRUE by Resolution.

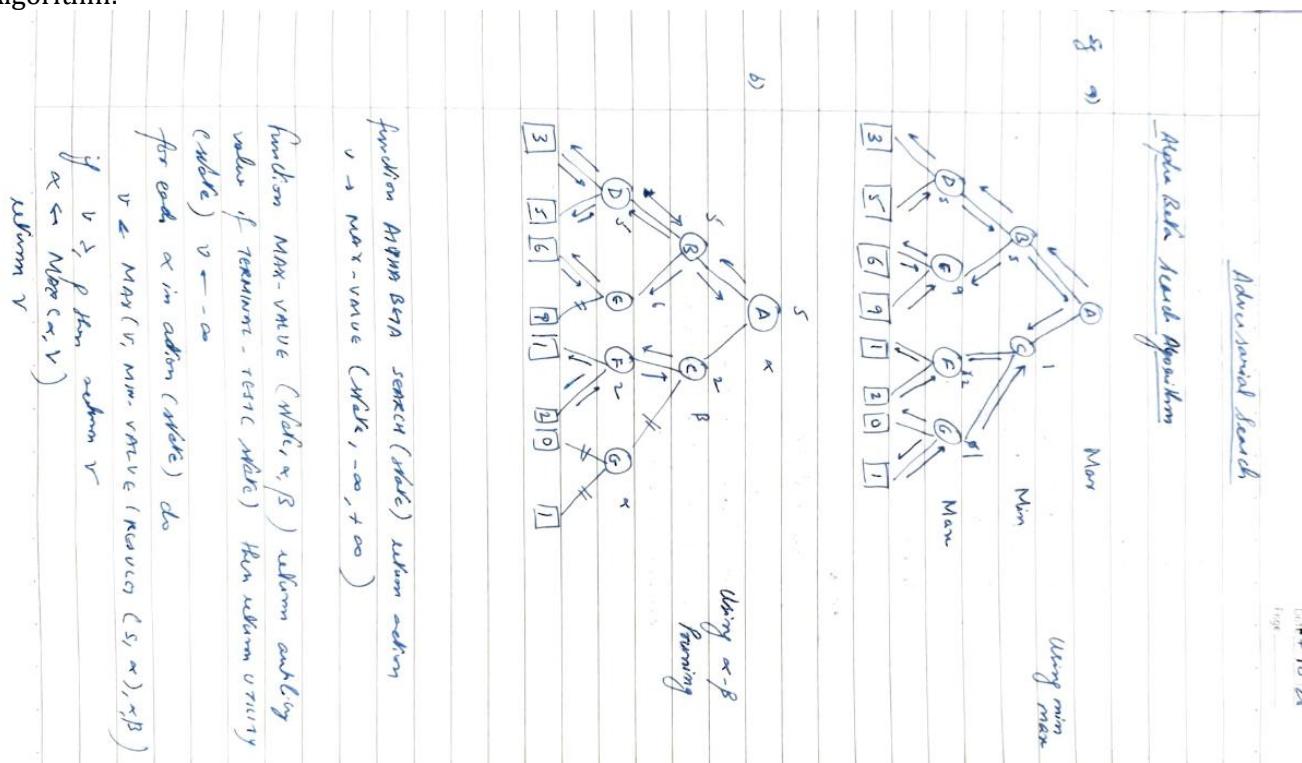
█ Resolution Tree (Trace):

```
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

9. Implement Alpha-Beta Pruning.

Algorithm:



```

function AlphaBetaSearch(state) return action
  v → max-v-value (MAX, -∞, +∞)
  if terminal-test(state) then return v
  for each a in action(state) do
    v ← MAX(v, min-v-value (MIN, α, β), a)
    if v ≥ β then return v
    α ← MAX(α, min-v-value (MIN, α), β)
  return v
  
```

Code & Output:

```
✓ ALPHA_BETA_PRUNING_CODE □

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 +
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[resolvent_tuple] = ((tuple(ci), tuple(cj)), (lit1, lit2))
29 +             print(f"\nStep {step}: Resolving {ci} and {cj} on {lit1} / {lit2} => {resolvent}")
30 +
31 +             if resolvent == []:
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, resolvent_tuple)
36 +             return True
37 +
38 +             new.add(resolvent_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

```

