VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Dhruhi Atykar (1BM23CS091)

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 **Dhruhi Atykar (1BM23CS091),** 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.

Prof.Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title
1	21-08-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent
2	28-08-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm
3	11-09-2025	Implement A* search algorithm
4	09-10-2025	Implement Hill Climbing search algorithm to solve N-Queens problem
5	09-10-2025	Simulated Annealing to Solve 8-Queens problem
6	16-10-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not
7	30-10-2025	Implement unification in first order logic
8	06-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning
9	06-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution
10	13-11-2025	Implement Alpha-Beta Pruning
11	13-11-2025	Conversion to CNF

INDEX

NAME: Dhruhi STD.: CSE SEC.: B ROLL NO.: 091

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	20/8	Tic Tac Toe		
2.	28/8	Vacuum Cleaner		
3.	28/8	8 piece Puzzle		
4.	11/9	IDDFS		
5	9/10	Hill climbing	10	
6.		Annealing.		
7.		8 puzzle A*.	10	7/10/25
8.	16/10	Prepositional Logic.		
9.	30/10	unification Algo.	10	
10.	6/11	First order logic	10	
11.	6/11	CNF	10	
12	13/11	Resolution	10	13/11/2

Github Link: https://github.com/dhruhiatykar/AI-Lab

Program 1

Implement Tic –Tac –Toe Game

Algorithm:

1. Tic Tac Toe Algorithm

Start the game
set first player = O
set second player (AI) = X
if computer ~~selects~~ add +1
if player wins remove -1
if it's a tie then give 0.

Ask player to choose a position
validate the move (empty spot).
Place X.

Computer analyses the possiblemoves
Chooses the move that ~~[illegible]~~ is
best possible to win

After each move check
win : 3 in a row/column/
diagonally.

If all are filled but no 3 in
a row/column/ diagonal then tie.

Example :

Computer is X
Player is O

X	O	X
O	X	

(1.) Player plays at (1,2)

X	O	X
O	X	O

2. Computer plays at (2,0)

X	O	X
O	X	O
X		

→ Computer wins the game.

Case 2:

X	O	X
O	X	

Player plays at (1,2)

X	O	X
O	X	O

computer plays at (2, 3) (Badmove)

X	O	X
O	X	O
	X	

Its a draw.

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))

cost_counter = 0

def minimax(board, is_ai_turn):
global cost_counter
cost_counter += 1

if check_winner(board, 'O'):
return 1 if
check_winner(board, 'X'):
return -1 if
is_draw(board):
return 0

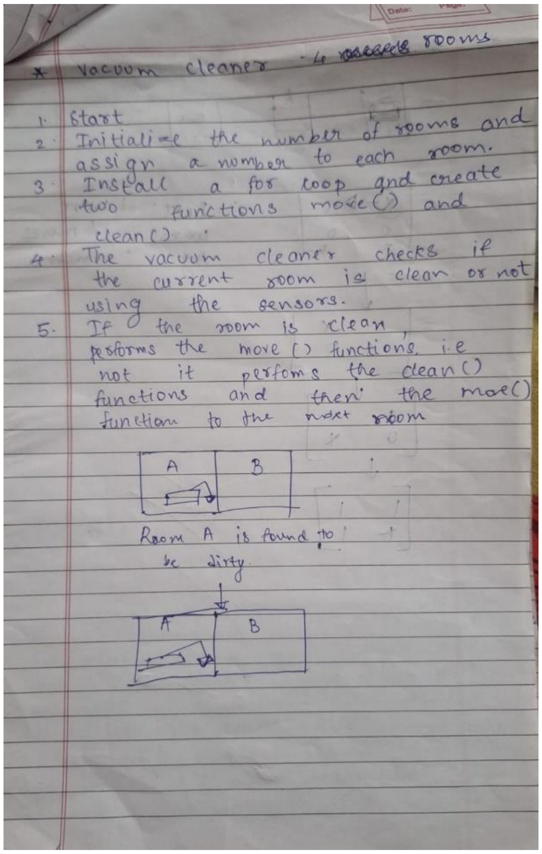
if is_ai_turn:
best_score = -float('inf') for i in
range(3): for j in range(3): if
board[i][j] == '-': board[i][j] = 'O'
score = minimax(board, False)
board[i][j] = '-' best_score =
max(score, best_score) return best_score
else:

```
        best_score = float('inf')
for i in range(3):              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:

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

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

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



        while True:
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



      global cost_counter        cost_counter = 0          cost = minimax(board, True)
print(f"AI evaluation cost from this position: {cost_counter} states examined")
print(f"AI evaluation score from this position: {cost}")

manual_game()
```

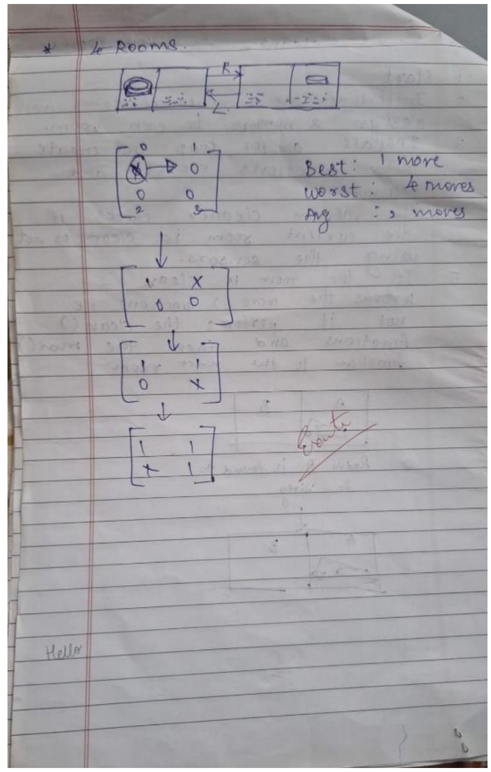
Implement vacuum cleaner agent.

Algorithm:



* Vacuum cleaner - 4 rooms

1. Start
2. Initialize the number of rooms and assign a number to each room.
3. Install a for loop and create two functions move() and clean()
4. The vacuum cleaner checks if the current room is clean or not using the sensors.
5. If the room is clean, performs the move() functions i.e not it perfoms the clean() functions and then the moe() function to the next room



* 4 Rooms.

Code: rooms = {

'A': True,

'B': True,

'C': True,

'D': True

}

\# The agent's current location current_room =

'A' def vacuum_cleaner_agent(): global

current_room print("---Starting Vacuum

Cleaner Agent---") print("Initial state:", rooms)   print("Agent starts in room A.")   #

A set to track visited rooms to avoid loops

visited = set()

   # While there's any dirty room left

while any(rooms.values()):

      # Clean the current room if dirty

   if rooms[current_room]:

         print(f"\nSucking dust in room {current_room}...")

      rooms[current_room] = False

       print(f"Room {current_room} is now clean.")

visited.add(current_room)

      # Decide where to go next based on current location and available dirty rooms

next_room = None   if current_room == 'A':

         options = [room for room in ['B', 'C'] if rooms[room] and room not in visited]

         if options:

while True:

               user_choice = input(f"Do you want to go to room {options[0]} or room {options[-1]}? (Type '{options[0]}' or '{options[-1]}'): ").upper()

if user_choice in options:

next_room = user_choice

```
                    break
else:

                    print("Invalid input. Please choose a valid dirty room.")

          else:

              # Default to B or C if no input needed

for room in ['B', 'C']:                    if rooms[room]

and room not in visited:

                    next_room = room

break

      elif current_room == 'B':             if

rooms['D'] and 'D' not in visited:

              print("Moving to room D.")

next_room = 'D'

       elif rooms['A'] and 'A' not in visited:

             next_room = 'A'        elif

current_room == 'C':           if rooms['D']

and 'D' not in visited:

print("Moving to room D.")

next_room = 'D'            elif rooms['A'] and

'A' not in visited:
```

```
            next_room = 'A'        elif
current_room == 'D':           if rooms['C']
and 'C' not in visited:
print("Moving to room C.")
next_room = 'C'           elif rooms['B'] and
'B' not in visited:

            next_room = 'B'
      # Fallback: find any remaining dirty room not visited yet
if not next_room:

         for room in ['A', 'B', 'C', 'D']:               if
rooms[room] and room not in visited:

               next_room = room
break        if next_room:

         print(f"Moving to room {next_room}.")
   current_room = next_room else:
       # No dirty unvisited rooms left
break    print("\n---Goal State
Achieved---")    print("All rooms are
clean:", rooms)     print("---Agent is
done---") vacuum_cleaner_agent()
```

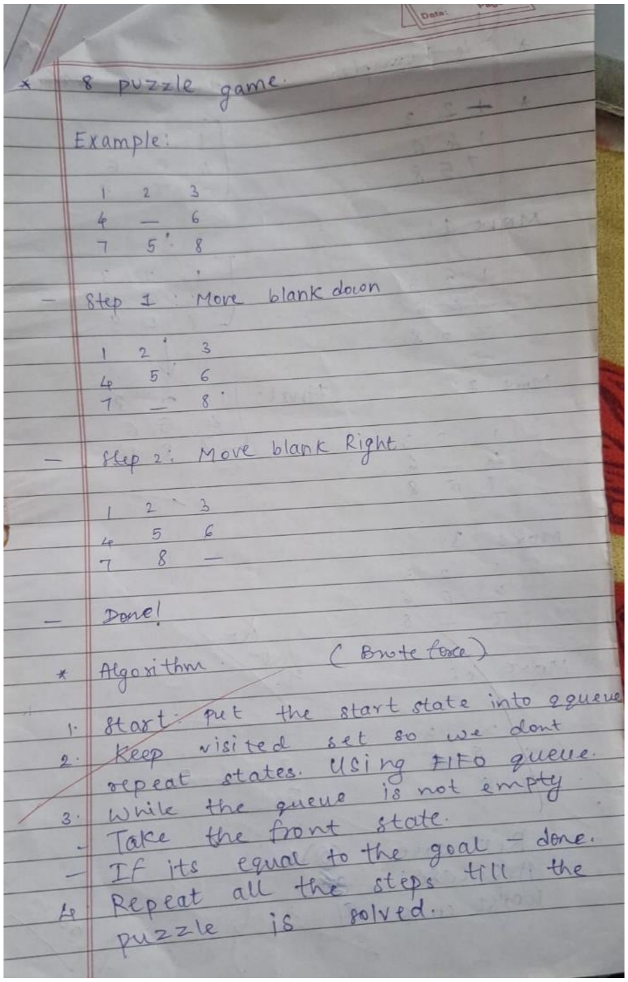
Outputs:

```
---Starting Vacuum Cleaner Agent---
Initial state: {'A': True, 'B': True, 'C': True, 'D': True}
Agent starts in room A.

Sucking dust in room A...
Room A is now clean.
Do you want to go to room B or room C? (Type 'B' or 'C'): C
Moving to room C.

Sucking dust in room C...
Room C is now clean.
Moving to room D.
Moving to room D.

Sucking dust in room D...
Room D is now clean.
Moving to room B.

Sucking dust in room B...
Room B is now clean.

---Goal State Achieved---
All rooms are clean: {'A': False, 'B': False, 'C': False, 'D': False}
---Agent is done---
```

<u>Program 2</u>

Using BFS solve 8 puzzle without heuristic approach.

Algorithm:



* 8 puzzle game.

Example:

1 2 3
4 — 6
7 5 8

— Step 1 : Move blank down

1 2 3
4 5 6
7 — 8

— Step 2 : Move blank Right

1 2 3
4 5 6
7 8 —

— Done!

* Algorithm (Brute force)

1. Start: put the start state into queue
2. Keep visited set so we dont repeat states. Using FIFO queue.
3. While the queue is not empty
- Take the front state.
- If its equal to the goal - done.
4. Repeat all the steps till the puzzle is solved.

Code:

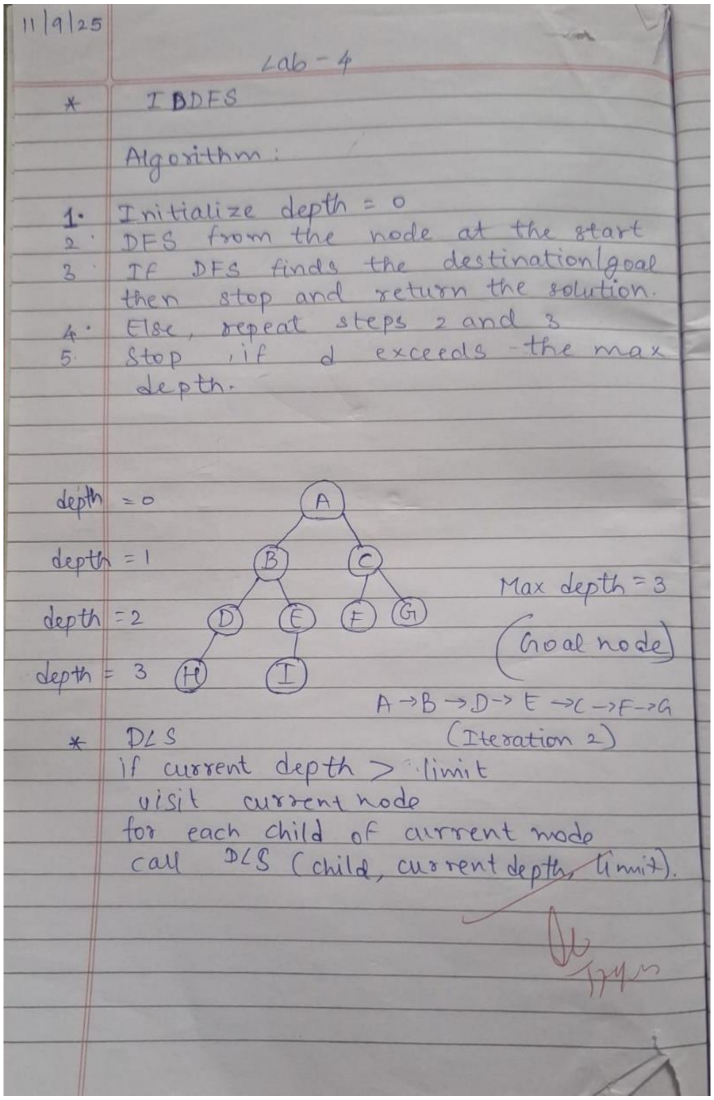
```
from collections import deque
def print_state(state):
for i in range(0, 9, 3):
print(state[i:i+3])
print()

def bfs(start, goal):     queue
= deque([(start, [])])     visited
= set([start])     while queue:
        state, path = queue.popleft()
if state == goal:
            return path + [state]
zero = state.index(0)
moves = []          if zero % 3
> 0:
```

```
        moves.append(zero - 1)
if zero % 3 < 2:
        moves.append(zero + 1)
if zero // 3 > 0:
        moves.append(zero - 3)
if zero // 3 < 2:
        moves.append(zero + 3)
for move in moves:
        new_state = list(state)             new_state[zero], new_state[move] =
new_state[move], new_state[zero]            new_state = tuple(new_state)
if new_state not in visited:                visited.add(new_state)
queue.append((new_state, path + [state]))     return None

def input_state(prompt):     s =
input(prompt).strip().split()
   return tuple(map(int, s))

start = input_state("Enter initial state (9 numbers with 0 for blank): ") goal
= input_state("Enter goal state (9 numbers with 0 for blank): ")

result = bfs(start, goal)
if result:     for step in
result:
print_state(step) else:
   print("No solution found")
```

Output:

```
Enter initial state (9 numbers with 0 for blank): 2 8 3 1 6 4 7 0 5
Enter goal state (9 numbers with 0 for blank): 1 2 3 8 0 4 7 6 5
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

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

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

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

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

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

Using Iterative Deepening DFS solve 8 puzzle without heuristic approach.


11/9/25

Lab - 4

* IDDFS

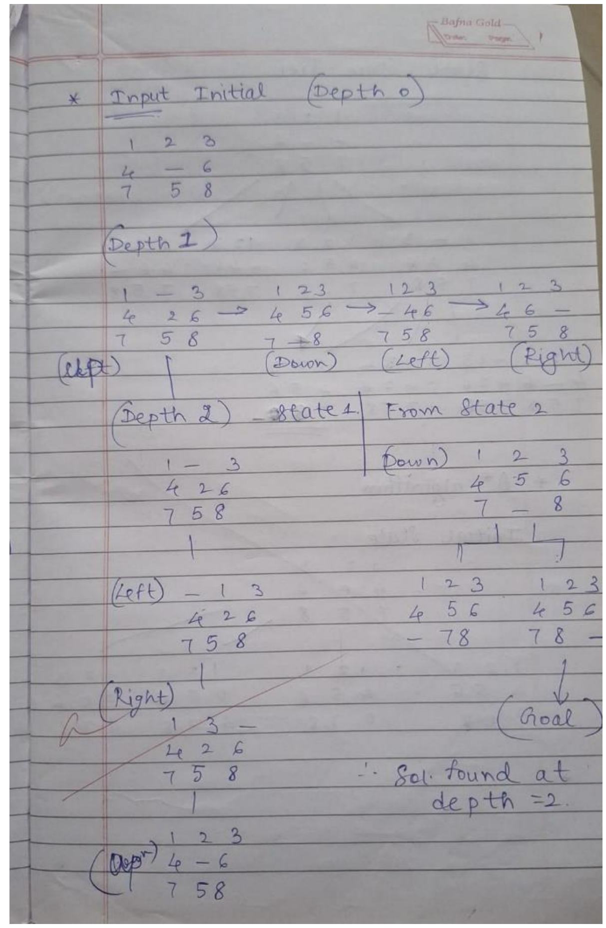
Algorithm:

1. Initialize depth = 0
2. DFS from the node at the start
3. If DFS finds the destination/goal then stop and return the solution.
4. Else, repeat steps 2 and 3
5. Stop, if d exceeds the max depth.

A → B → D → E → C → F → G (Iteration 2)

* DLS

If current depth > limit
visit current node
for each child of current node
call DLS (child, current depth, limit).


* Input Initial (Depth 0)

∴ Sol. found at depth = 2.

Code: class PuzzleState: def __init__(self, board, empty_pos, moves=0, path=None):

```
        self.board = board
self.empty_pos = empty_pos
self.moves = moves
        self.path = path or [board]

    def is_goal(self, goal):
        return self.board == goal

    def get_neighbors(self):        neighbors = []        x, y =
self.empty_pos        directions = [(-1,0),(1,0),(0,-1),(0,1)]  # Up,
Down, Left, Right        for dx, dy in directions:            nx, ny = x +
```

```
dx, y + dy             if 0 <= nx < 3 and 0 <= ny < 3:
new_board = [list(row) for row in self.board]
                # swap empty_pos with target
                new_board[x][y], new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
new_board = tuple(tuple(row) for row in new_board)
neighbors.append(PuzzleState(new_board, (nx, ny), self.moves + 1, self.path + [new_board]))
return neighbors

def dls(state, goal, limit, visited):     if
state.is_goal(goal):         return
state.path     if limit == 0:         return
None     visited.add(state.board)     for
neighbor in state.get_neighbors():
if neighbor.board not in visited:
            result = dls(neighbor, goal, limit - 1, visited)
if result is not None:                return result
visited.remove(state.board)     return None

def iddfs(start, goal):
    depth = 0
while True:
        visited = set()         result = dls(start,
goal, depth, visited)         if result is not
None:
            return result
depth += 1

def print_path(path):
    print(f"Solution Found in {len(path)-1} moves")
for state in path:         for row in state:
            print(" ".join(str(x) if x != 0 else "-" for x in row))
print()

def get_user_board(prompt):
    print(prompt)
board = []     for i
in range(3):
        row = list(map(int, input(f"Row {i+1} (space separated, use 0 for empty): ").strip().split()))
board.append(tuple(row))     return tuple(board)

start_board = get_user_board("Enter the initial state:") goal_board
= get_user_board("Enter the goal state:")
```

```
# Locate empty in start state
empty_pos = None for i in
range(3):     for j in range(3):
if start_board[i][j] == 0:
          empty_pos = (i, j)
break     if empty_pos is not
None:        break

start_state = PuzzleState(start_board, empty_pos)
path = iddfs(start_state, goal_board) if path:
   print_path(path) else:
   print("No solution found.")
```

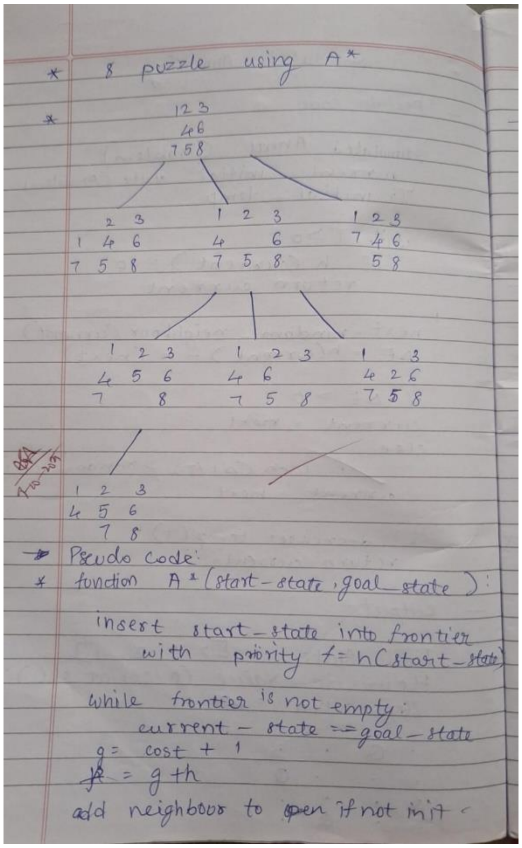
Output:

```
Enter the initial state:
Row 1 (space separated, use 0 for empty): 2 8 3
Row 2 (space separated, use 0 for empty): 1 6 4
Row 3 (space separated, use 0 for empty): 7 0 5
Enter the goal state:
Row 1 (space separated, use 0 for empty): 1 2 3
Row 2 (space separated, use 0 for empty): 8 0 4
Row 3 (space separated, use 0 for empty): 7 6 5
Solution Found in 5 moves
2 8 3
1 6 4
7 - 5

2 8 3
1 - 4
7 6 5

2 - 3
1 8 4
7 6 5

- 2 3
1 8 4
7 6 5

1 2 3
- 8 4
7 6 5

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

Program 3

Apply A* algorithm for misplaced tiles.

Algorithm:



Code: import heapq

```
class PuzzleState:     def __init__(self, board, goal,
parent=None, g=0):
        self.board = board
self.goal = goal         self.parent
= parent
        self.g = g         self.h =
self.misplaced_tiles()         self.f =
self.g + self.h


    def misplaced_tiles(self):
        """Count misplaced tiles (excluding 0)."""
        return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i] != self.goal[i])


    def get_neighbors(self):
        """Generate possible moves by sliding the blank (0)."""
neighbors = []         idx = self.board.index(0)         x, y =
```

```
divmod(idx, 3)   # row, col         moves = [(-1,0),(1,0),(0,-
1),(0,1)]  # up, down, left, right

        for dx, dy in moves:             nx, ny = x+dx, y+dy             if 0 <= nx < 3 and 0 <= ny <
3:                new_idx = nx*3 + ny                 new_board = self.board[:]
new_board[idx], new_board[new_idx] = new_board[new_idx], new_board[idx]
neighbors.append(PuzzleState(new_board, self.goal, self, self.g+1))        return neighbors

    def __lt__(self, other):
        return self.f < other.f  # priority queue uses f value

def reconstruct_path(state):
    path = []
while state:
        path.append(state.board)
state = state.parent
    return path[::-1]

def astar(start, goal):
    start_state = PuzzleState(start, goal)
open_list = []     heapq.heappush(open_list,
start_state)     closed_set = set()

    while open_list:
        current = heapq.heappop(open_list)

        if current.board == goal:
            return reconstruct_path(current)

        closed_set.add(tuple(current.board))

        for neighbor in current.get_neighbors():
if tuple(neighbor.board) in closed_set:
continue             heapq.heappush(open_list,
neighbor)     return None

print("Enter the 8-puzzle START state (use 0 for blank).") start_input =
list(map(int, input("Enter 9 numbers separated by spaces: ").split()))

print("\nEnter the GOAL state (use 0 for blank).") goal_input = list(map(int,
input("Enter 9 numbers separated by spaces: ").split()))
```

```
if len(start_input) != 9 or len(goal_input) != 9:
    print("Invalid input! Please enter exactly 9 numbers for each state.") else:
    solution = astar(start_input, goal_input)

    if solution:
        print("\n Steps to solve:")
for step in solution:             for
i in range(0,9,3):
print(step[i:i+3])
print("------")     else:
        print(" No solution found!")
```

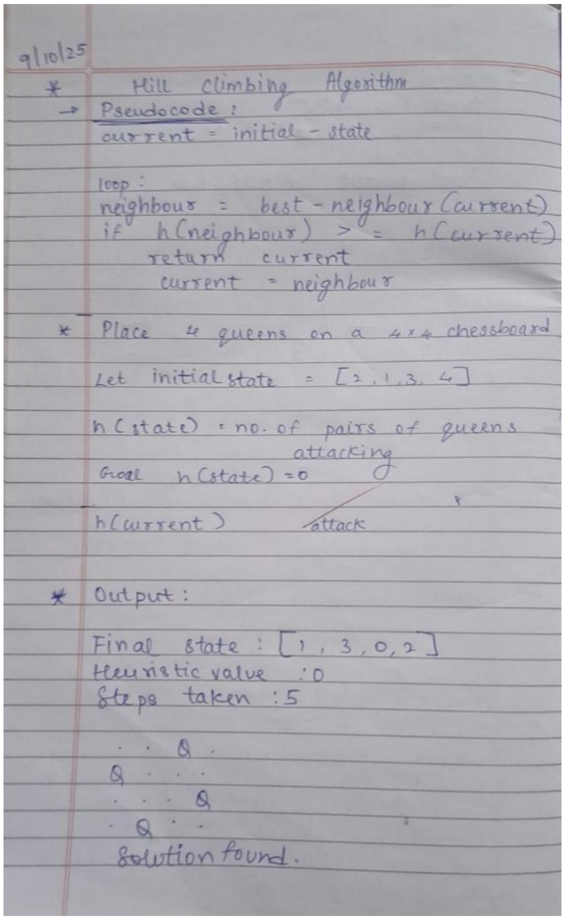
Output:

```
Enter the 8-puzzle START state (use 0 for blank).
Enter 9 numbers separated by spaces: 2 8 3 1 6 4 7 0 5

Enter the GOAL state (use 0 for blank).
Enter 9 numbers separated by spaces: 1 2 3 8 0 4 7 6 5

Steps to solve:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
------
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
------
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
------
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
------
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
------
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
------
```

Program 4

Implement hill climbing search algorithm to solve N-Queens problem.

Algorithm:



9/10/25

* Hill Climbing Algorithm

→ Pseudocode :

current = initial - state

loop :
neighbour = best - neighbour (current)
if h(neighbour) >= h(current)
return current
current = neighbour

* Place 4 queens on a 4x4 chessboard

Let initial state = [2, 1, 3, 4]

h(state) = no. of pairs of queens attacking

Goal h(state) = 0

h(current) ~~attack~~

* Output :

Final state : [1, 3, 0, 2]
Heuristic value : 0
Steps taken : 5

. . Q .
Q . . .
. . . Q
. Q . .
Solution found.

Code:

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

def heuristic(state):
attacks = 0      n = len(state)
for i in range(n):          for j
in range(i + 1, n):
            if state[i] == state[j]:  # same row
                attacks += 1              if abs(state[i] - state[j]) ==
abs(i - j):  # same diagonal                 attacks += 1
    return attacks
```

```
def get_neighbors(state):
    neighbors = []     n = len(state)     for
col in range(n):         for row in
range(n):             if state[col] != row:
new_state = list(state)
new_state[col] = row
neighbors.append(new_state)     return
neighbors

def hill_climbing(start_state):
    current = copy.deepcopy(start_state)
while True:
        current_h = heuristic(current)
if current_h == 0:             return
current, 0

        neighbors = get_neighbors(current)
        neighbor_h = [heuristic(neigh) for neigh in neighbors]

        min_h = min(neighbor_h)
if min_h >= current_h:             #
No improvement possible
return current, current_h

        current = neighbors[neighbor_h.index(min_h)]

def generate_all_states(n):

    states = []     def
backtrack(col=0, state=[]):         if
col == n:
states.append(state.copy())
            return         for row in
range(n):
state.append(row)
backtrack(col+1, state)
            state.pop()
backtrack()
    return states

if __name__ == "__main__":
```

n = 4 all_states =
generate_all_states(n)

print(f"Total initial states for {n} queens: {len(all_states)}\n")

for i, start in enumerate(all_states, start=1):
final_state, cost = hill_climbing(start)
print(f"Initial state {i}: {start}")
print(f"Final state after hill climbing:")
print_board(final_state) print(f"Cost
(heuristic): {cost}") print("="*30)

Output:

```
Total initial states for 4 queens: 256

Initial state 1: [0, 0, 0, 0]
Final state after hill climbing:
. . Q .
Q . . .
. . . Q
. Q . .

Cost (heuristic): 0
==============================
Initial state 2: [0, 0, 0, 1]
Final state after hill climbing:
. Q . .
. . . Q
Q . . .
. . Q .

Cost (heuristic): 0
==============================
Initial state 3: [0, 0, 0, 2]
Final state after hill climbing:
. . Q .
Q . . .
. . . Q
. Q . .

Cost (heuristic): 0
==============================
Initial state 4: [0, 0, 0, 3]
Final state after hill climbing:
. . Q .
Q . . .
. Q . .
. . . Q
```

Program 5

8 Queens Problem using Simulated Annealing Algorithm:

* Stimulated Annealing

— Pseudo code

Stimulated Anneal (Problem)
current - initial- state (Problem)
T= initial - temp

while T>0
if h (current) ==0
return current

next - random - neighbour (current)
ΔE := h(current) - h (next)

if $\Delta E > 0$
current - next
else
~~if~~ each (Δ r/T) > random (0,1)
current - next

T = decrease - temp (T)
return current.

— Output

Final State : [2, 0, 3, 1]
Heuristic Value (0 = perfect sol): 0
Steps taken = 209

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

Code:

```
import random
import math
def cost(state):     attacks = 0     n = len(state)     for i in range(n):
for j in range(i + 1, n):             if state[i] == state[j] or abs(state[i]
- state[j]) == abs(i - j):
                attacks += 1
    return attacks

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

def simulated_annealing(n=8, max_iter=10000):
```

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

    temperature = 100.0
    cooling_rate = 0.95

    best = current[:]
    best_cost = current_cost

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

        neighbor = get_neighbor(current)
neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost
         if delta > 0:
            current, current_cost = neighbor, neighbor_cost
if neighbor_cost < best_cost:
                best, best_cost = neighbor, neighbor_cost
else:
            probability = math.exp(delta / temperature)
if random.random() < probability:
                current, current_cost = neighbor, neighbor_cost

        temperature *= cooling_rate

    return best, best_cost

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

if __name__ == "__main__":
```

n = 8 solution, cost_val = simulated_annealing(n)

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

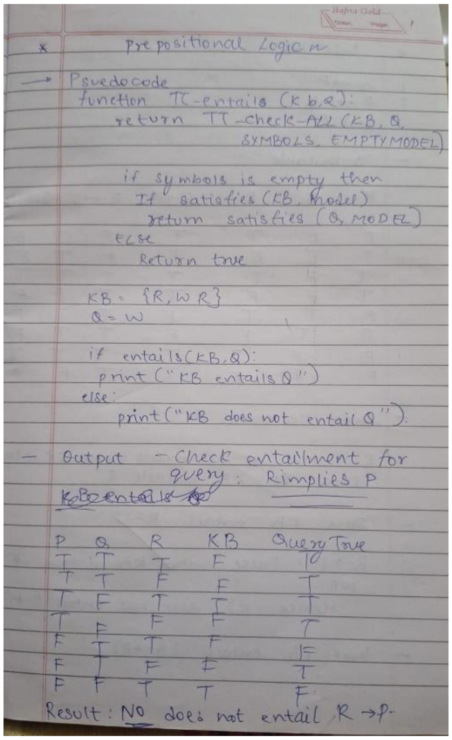
Output:

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

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

Program 6

Implement truth table enumeration algorithm for deciding propositional entailment.

Algorithm:



* Prepositional Logic

→ Psuedocode
function TT-entails (KB,Q):
return TT-check-ALL (KB, Q, SYMBOLS, EMPTYMODEL)

if symbols is empty then
If satisfies (KB, Model)
return satisfies (Q, MODEL)
ELSE
Return true

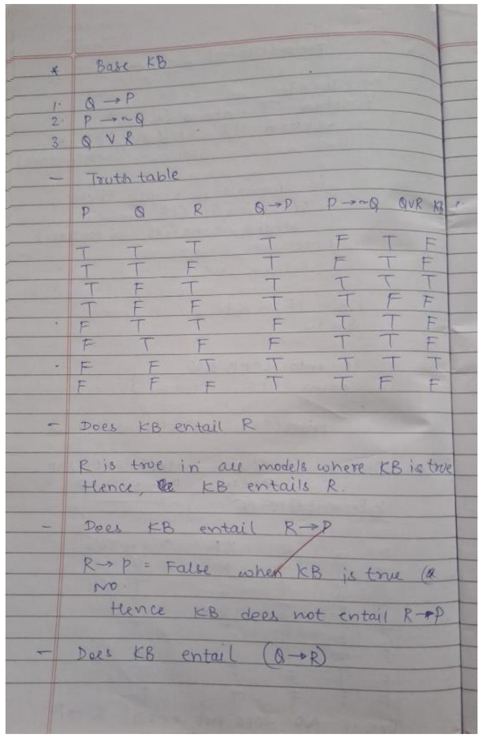
KB = {R, W R}
Q = W

if entails(KB,Q):
print ("KB entails Q")
else:
print ("KB does not entail Q").

– Output – Check entailment for query: R implies P

P	Q	R	KB	Query True
T	T	T	F	T
T	T	F	F	T
T	F	T	T	T
T	F	F	F	T
F	T	T	F	F
F	T	F	F	T
F	F	T	T	F

Result: No does not entail R → P.



* Base KB

1. Q → P
2. P → ~Q
3. Q ∨ R

– Truth table

P	Q	R	Q→P	P→~Q	Q∨R	KB
T	T	T	T	F	T	F
T	T	F	T	F	T	F
T	F	T	T	T	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	F
F	T	F	F	T	T	F
F	F	T	T	T	T	T
F	F	F	T	T	F	F

– Does KB entail R

R is true in all models where KB is true
Hence, KB entails R.

– Does KB entail R → P

R → P = False when KB is true
No.
Hence KB does not entail R → P

– Does KB entail (Q → R)

Code:

```
import pandas as pd from
itertools import product
import re
def tokenize(sentence):
    # Now also tokenize symbols like ∨ (logical OR)     #
Added ∨, ∧, ¬ in the regex to separate them as tokens
token_pattern = r'\w+|[()∨∧¬]'     return
re.findall(token_pattern, sentence)
def pl_true(sentence, model):     tokens =
tokenize(sentence)     logical_ops = {'and',
'or', 'not', '∨', '∧', '¬'}
    evaluated_tokens = []
for token in tokens:
if token == '∨':
            evaluated_tokens.append('or')  # replace symbol with python 'or'
elif token == '∧':
            evaluated_tokens.append('and')  # replace symbol with python 'and'
elif token == '¬':
```

```
            evaluated_tokens.append('not')  # replace symbol with python 'not'
elif token.lower() in logical_ops:
            evaluated_tokens.append(token.lower())
elif token in model:
            evaluated_tokens.append(str(model[token]))
else:
            evaluated_tokens.append(token)
eval_sentence = ' '.join(evaluated_tokens)      try:
        return eval(eval_sentence)
except Exception as e:
        print(f"Error evaluating sentence: {eval_sentence}")
raise e
def tt_entails(kb, alpha, symbols):
    truth_table = []      for model in product([False, True],
repeat=len(symbols)):
        model_dict = dict(zip(symbols, model))
kb_value = pl_true(kb, model_dict)
alpha_value = pl_true(alpha, model_dict)          row
= {
            'A': model_dict.get('A', False),
            'B': model_dict.get('B', False),
            'C': model_dict.get('C', False),
            'A ∨ C': model_dict.get('A', False) or model_dict.get('C', False),
            'B ∨ ¬C': model_dict.get('B', False) or not model_dict.get('C', False),
'KB': kb_value,
            'α': alpha_value
        }
        truth_table.append(row)          if
kb_value and not alpha_value:
            return False, pd.DataFrame(truth_table)
    return True, pd.DataFrame(truth_table)
def get_symbols(kb, alpha):
    return sorted(set(re.findall(r'[A-Z]', kb + alpha)))
 alpha = "A ∨ B"
symbols = get_symbols(kb, alpha) result,
truth_table = tt_entails(kb, alpha, symbols)
def highlight_kb_alpha(row):
if row['KB'] and row['α']:
        return ['background-color: lightgreen' if col in ['KB', 'α'] else '' for col in row.index]
else:
        return ['' for _ in row.index]
```

```
print("Shreya Raj 1BM23CS317") styled_table =
truth_table.style.apply(highlight_kb_alpha, axis=1)
display(styled_table)

if result:
    print("\nKB entails α") else:
    print("\nKB does not entail α")
```

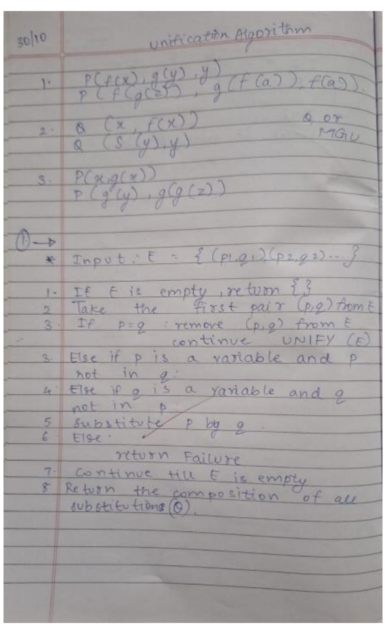
Output:

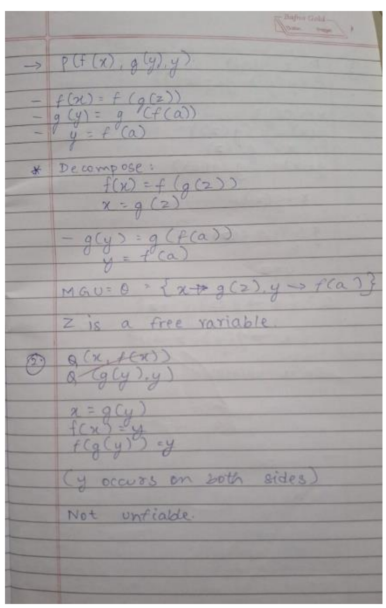
	A	B	C	A ∨ C	B ∨ ¬C	KB	α
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

```
KB entails α
```

Program 7

Implement unification in first order logic.
Algorithm:


30/10
Unification Algorithm
1. P(f(x), g(y), y)
P(f(g(z)), g(f(a)), f(a)).
2. Q(x, f(x))
Q(S(y), y)
Q or MGU
3. P(x, g(x))
P(g(y), g(g(z)))
1.→
* Input: E = {(p1, q1)(p2, q2)...}
1. If E is empty, return {}
2 Take the first pair (p, q) from E
3. If p = q : remove (p, q) from E
continue UNIFY (E)
3. Else if p is a variable and p not in q.
4. Else if q is a variable and q not in p.
5 Substitute p by q.
6 Else:
return Failure
7. Continue till E is empty
8 Return the composition of all substitutions (θ).


→ P(f(x), g(y), y)
- f(x) = f(g(z))
- g(y) = g(f(a))
- y = f(a)
* Decompose:
f(x) = f(g(z))
x = g(z)
- g(y) = g(f(a))
y = f(a)
MGU = θ = {x → g(z), y → f(a)}
Z is a free variable.
2. Q(x, f(x))
Q(g(y), y)
x = g(y)
f(x) = y
f(g(y)) = y
(y occurs on both sides)
Not unfiable.

Code: def unify(x, y, subst=None):

```
if subst is None:
        subst = {}

    # If x or y is a variable or constant
if is_variable(x) or is_constant(x):
if x == y:
            return subst
elif is_variable(x):
            return unify_var(x, y, subst)
elif is_variable(y):
            return unify_var(y, x, subst)
else:
            return None

    # If both x and y are compound expressions
if is_compound(x) and is_compound(y):
if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None         for xi, yi
in zip(x[1], y[1]):             subst =
unify(xi, yi, subst)             if subst
is None:                 return None
return subst
    return None

def is_variable(x):
    return isinstance(x, str) and x.islower() and x.isalpha()

def is_constant(x):
    return isinstance(x, str) and x.isupper() and x.isalpha()

def is_compound(x):     return isinstance(x, tuple) and len(x) == 2 and isinstance(x[0], str)
and isinstance(x[1], list)

def unify_var(var, x, subst):
if var in subst:
        return unify(subst[var], x, subst)
elif x in subst:
        return unify(var, subst[x], subst)
elif occurs_check(var, x, subst):
        return None
else:
        subst[var] = x
```

```
        return subst

def occurs_check(var, x, subst):
if var == x:          return True      elif
is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
elif is_compound(x):
        return any(occurs_check(var, arg, subst) for arg in x[1])
else:
        return False
x = ("P", ["x", "A"])
y = ("P", ["B", "y"])

result = unify(x, y) if
result is not None:
    print("Unification succeeded with substitution:", result) else:
    print("Unification failed.")
```

Output:

```
Unification succeeded with substitution: {'x': 'B', 'y': 'A'}
```

Program 8

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

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

→ Given.
Man (Marcus)
Pompeian (Marcus)

∀x (Pompeian (x) → Roman (x))
∀x(Man (x) → Person(x))
∀x (Roman (x) → Loyal (x))
∀ x (Person (x) → Mortal (x))

Inference

Man (Marcus), Pompeian (Marcus),
Roman (Marcus), Loyal (Marcus),
Person (Marcus), Mortal (Marcus)

~~Asem~~ :
Man (Marcus) → Person(Marcus)
→ Mortal (Marcus).

Query : Mortal (Marcus) proved.

Code: facts = {
    'American(West)': True,
    'Hostile(Nono)': True,
    'Missiles(Nono)': True,
} def rule1(facts):   if facts.get('American(West)', False) and facts.get('Hostile(Nono)', False):
        return 'Criminal(West)'
    return None

```
def rule2(facts):     if facts.get('Missiles(Nono)', False) and
facts.get('Hostile(Nono)', False):
        return 'SoldWeapons(West, Nono)'

def forward_chaining(facts, rules):
    new_facts = facts.copy()
inferred = True     while
inferred:        inferred =
False        for rule in rules:
            result = rule(new_facts)             if
result and result not in new_facts:
                new_facts[result] = True
inferred = True                print(f"New fact
inferred: {result}")
    return new_facts
rules = [rule1, rule2]

inferred_facts = forward_chaining(facts, rules)

print("\nFinal facts:") for
fact in inferred_facts:
    print(fact)
```

Output:

```
New fact inferred: Criminal(West)
New fact inferred: SoldWeapons(West, Nono)

Final facts:
American(West)
Hostile(Nono)
Missiles(Nono)
Criminal(West)
SoldWeapons(West, Nono)
```

Program 9

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

* Resolution

$\forall x : food(x) \rightarrow likes(John, x)$
$food(Apple) \wedge food(vegetables)$
$\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
$eats(Anil, Peanuts) \wedge alive(Anil)$
$\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
$\forall x : \neg killed(x) \rightarrow alive(x)$
$\forall x : alive(x) \rightarrow \neg killed(x)$
$likes(John, Peantusts)$

→

$\neg food(x) \vee likes(John, x)$
$food(Apple)$
$food(vegetables)$
$\neg eats(y, z) \vee killed(y) \vee food(z)$
$eats(Anil, Peanuts)$
$alive(Anil)$
$\neg eats(Anil, w) \vee eats(Harry, w)$
$killed(g) \vee aline(g)$
$\neg alive(k) \vee \neg killed(k)$
$likes(John, Peanuts)$

$\neg likes(John, Peanuts)$ $\quad$ $\neg food(x) \vee likes(John, x)$
$\neg food(peanuts)$ $\quad$ $\neg eats(y, z) \vee killed(y) \vee food(z)$
$\neg eats(y, Peanuts) \vee killed(y)$ $\quad$ $eats(Anil, Peanuts)$ $\quad$ $\{Anil/y\}$
$killed(Anil)$ $\quad$ $alive(Anil)$
$\{\ \}$ Hence, proved.

Algorithm:

1. Eliminate implication ($\rightarrow$)
2. Move negations
3. Standardize variable
4. Move quantifiers to the left
5. Skolemization - remove existential quantifiers by introducing Skolem functions.
6. Dop universal quantifiers
7. Distribute $\vee$ (OR) over $\wedge$ (AND) to get CNF.

Code:

```
from collections import deque
import itertools import copy
import pprint
# ----- Data structures -----
class Var:     def
__init__(self, name):
      self.name = name
def __repr__(self):
      return f"Var({self.name})"
def __eq__(self, other):
      return isinstance(other, Var) and self.name == other.name
def __hash__(self):
      return hash(('Var', self.name))
```

```
class Const:     def
__init__(self, name):
        self.name = name
def __repr__(self):
        return f"Const({self.name})"
def __eq__(self, other):
        return isinstance(other, Const) and self.name == other.name
def __hash__(self):
        return hash(('Const', self.name))

class Func:     def __init__(self,
name, args):
        self.name = name
self.args = args     def
__repr__(self):
        return f"Func({self.name}, {self.args})"
def __eq__(self, other):
        return isinstance(other, Func) and self.name == other.name and self.args == other.args
def __hash__(self):
        return hash(('Func', self.name, tuple(self.args)))

class Literal:
    # predicate_name: str, args: list of Terms, negated: bool
def __init__(self, predicate, args, negated=False):
        self.predicate = predicate
    self.args = tuple(args)
    self.negated = negated def
    negate(self):
        return Literal(self.predicate, list(self.args), not self.negated)
    def __repr__(self):
        sign = "~" if self.negated else ""
        args = ",".join(map(term_to_str, self.args))
return f"{sign}{self.predicate}({args})"     def
__eq__(self, other):
        return (self.predicate, self.args, self.negated) == (other.predicate, other.args, other.negated)
def __hash__(self):
        return hash((self.predicate, self.args, self.negated))

# Clause is frozenset of Literal def
clause_to_str(cl):
    return " OR ".join(map(str, cl)) if cl else "EMPTY"
```

```
def term_to_str(t):     if
isinstance(t, Var):
        return t.name     if
isinstance(t, Const):
        return t.name     if
isinstance(t, Func):
        return f"{t.name}({','.join(term_to_str(a) for a in t.args)})"
return str(t)

# ----- Substitution utilities ----- def
apply_subst_term(term, subst):
if isinstance(term, Var):         if
term in subst:
            return apply_subst_term(subst[term], subst)
else:
            return term     elif
isinstance(term, Const):
        return term     elif
isinstance(term, Func):
        return Func(term.name, [apply_subst_term(a, subst) for a in term.args])
else:
        return term

def apply_subst_literal(lit, subst):
    return Literal(lit.predicate, [apply_subst_term(a, subst) for a in lit.args], lit.negated)

def apply_subst_clause(clause, subst):
    return frozenset(apply_subst_literal(l, subst) for l in clause)

# ----- Unification (Robust, with occurs-check) ----- def
occurs_check(var, term, subst):
    term = apply_subst_term(term, subst)
    if term == var:     return True if
    isinstance(term, Func):
        return any(occurs_check(var, arg, subst) for arg in term.args) return
    False

def unify_terms(x, y, subst):
    # returns updated subst or None on failure
x = apply_subst_term(x, subst)     y =
apply_subst_term(y, subst)
```

```
    if isinstance(x, Var):
if x == y:
            return subst         if
occurs_check(x, y, subst):
            return None
new = subst.copy()
new[x] = y         return
new     if isinstance(y,
Var):
        return unify_terms(y, x, subst)     if isinstance(x, Const) and isinstance(y, Const):          return
subst if x.name == y.name else None     if isinstance(x, Func) and isinstance(y, Func) and x.name ==
y.name and len(x.args) == len(y.args):          for a, b in zip(x.args, y.args):
            subst = unify_terms(a, b, subst)
if subst is None:                  return None
return subst
    return None

def unify_literals(l1, l2):
    # l1 and l2 must have same predicate and opposite polarity for resolution     if
l1.predicate != l2.predicate or l1.negated == l2.negated or len(l1.args) != len(l2.args):
        return None     subst = {}
for a, b in zip(l1.args, l2.args):
        subst = unify_terms(a, b, subst)
if subst is None:             return None
    return subst

# ----- Standardize apart variables (to avoid name clashes) -----
_var_count = 0 def
standardize_apart(clause):
    global _var_count
varmap = {}
new_literals = [] for
lit in clause:
new_args = []     for t
in lit.args:
            new_args.append(_rename_term_vars(t, varmap))
     new_literals.append(Literal(lit.predicate, new_args, lit.negated))     return
     frozenset(new_literals)

def _rename_term_vars(term, varmap):
    global _var_count     if
isinstance(term, Var):         if
term.name not in varmap:
_var_count += 1
```

```
            varmap[term.name] = Var(f"{term.name}_{_var_count}")
return varmap[term.name]     if isinstance(term, Const):
        return term     if
isinstance(term, Func):
        return Func(term.name, [_rename_term_vars(a, varmap) for a in term.args])
return term

# ----- Resolution operation between two clauses ----- def
resolve(ci, cj):
    # returns set of resolvent clauses (frozenset of literals)     resolvents = set()     ci =
standardize_apart(ci)     cj = standardize_apart(cj)     for li in ci:         for lj in cj:             if
li.predicate == lj.predicate and li.negated != lj.negated and len(li.args) == len(lj.args):
                subst = unify_literals(li, lj)
if subst is not None:
                    # build resolvent: (Ci - {li}) U (Cj - {lj}) with subst applied
                    new_clause = set(apply_subst_literal(l, subst) for l in (ci - {li}) | (cj - {lj}))
                    # remove tautologies: a clause containing P and ~P after subst
preds = {}                    taut = False                    for l in new_clause:
                        key = (l.predicate, tuple(map(term_to_str, l.args)))
if key in preds and preds[key] != l.negated:
                            taut = True
break                          preds[key] =
l.negated                       if not taut:
                        resolvents.add(frozenset(new_clause))
return resolvents

# ----- Main resolution loop ----- def fol_resolution(kb_clauses,
query_clause, max_iterations=20000):
    """
    kb_clauses: set/list of clauses (each clause is frozenset of Literal)
    query_clause: single Literal (to be proved), will be negated and added to KB
    Returns True if contradiction (empty clause) is derived.
    """
    # Negate the query and add its literals as separate clauses (each literal is a clause)
negated_query = [query_clause.negate()]     clauses = set(kb_clauses)     for l in
negated_query:
        clauses.add(frozenset([l]))

    new = set()
processed_pairs = set()
    queue = list(clauses)
```

```
    iterations = 0     while True:
pairs = []        clause_list =
list(clauses)         n =
len(clause_list)
        # iterate over all unordered pairs
for i in range(n):             for j in
range(i+1, n):
                pairs.append((clause_list[i], clause_list[j]))

        something_added = False
for (ci, cj) in pairs:             pair_key =
(ci, cj)             if pair_key in
processed_pairs:                 continue
            processed_pairs.add(pair_key)
resolvents = resolve(ci, cj)
iterations += 1             if iterations >
max_iterations:
                return False, "max_iterations_exceeded"
for r in resolvents:                 if len(r) == 0:
                    return True, "Derived empty clause (success)"
if r not in clauses and r not in new:
                    new.add(r)                    something_added = True         if not
something_added:             return False, "No new clauses — failure (KB
does not entail query)"        clauses.update(new)        new = set()

# ----- Helper to create easy constants/vars -----
def C(name): return Const(name) def V(name):
return Var(name)
def F(name, *args): return Func(name, list(args)) def
L(pred, args, neg=False): return Literal(pred, args, neg)

# Build clauses (using variables V('x'), constants C('Anil'), etc.)
x = V('x') y
= V('y')
kb = set()
# 1. ¬Food(x) ∨ Likes(John,x)
kb.add(frozenset([L('Food', [x], neg=True), L('Likes', [C('John'), x], neg=False)]))

# 2a. Food(Apple)
kb.add(frozenset([L('Food', [C('apple')], neg=False)]))
# 2b. Food(vegetable)
kb.add(frozenset([L('Food', [C('vegetable')], neg=False)]))
```

```
# 3. ¬Eats(x,y) ∨ Killed(y) ∨ Food(y)
kb.add(frozenset([L('Eats', [x,y], neg=True), L('Killed', [y], neg=False), L('Food', [y], neg=False)]))

# 4a. Eats(Anil,peanuts)
kb.add(frozenset([L('Eats', [C('Anil'), C('peanuts')], neg=False)]))
# 4b. Alive(Anil)
kb.add(frozenset([L('Alive', [C('Anil')], neg=False)]))

# 5. ¬Eats(Anil,x) ∨ Eats(Harry,x)
kb.add(frozenset([L('Eats', [C('Anil'), x], neg=True), L('Eats', [C('Harry'), x], neg=False)]))

# 6. ¬Alive(x) ∨ ¬Killed(x)
kb.add(frozenset([L('Alive', [x], neg=True), L('Killed', [x], neg=True)]))

# 7. Killed(x) ∨ Alive(x)
kb.add(frozenset([L('Killed', [x], neg=True), L('Alive', [x], neg=False)]))
# Query
query = L('Likes', [C('John'), C('peanuts')], neg=False)

def show_kb(kb):
   print("Knowledge base clauses:")
for c in kb:
       print("  ", clause_to_str(c))
   print()

if __name__ == "__main__":
   print("FOL resolution prover (basic example)\n")     show_kb(kb)
print("Query:", query)     print("Negated query clause will be added to KB and
resolution attempted.\n")     success, info = fol_resolution(kb, query,
max_iterations=20000)     print("Result:", success, "|", info)
```

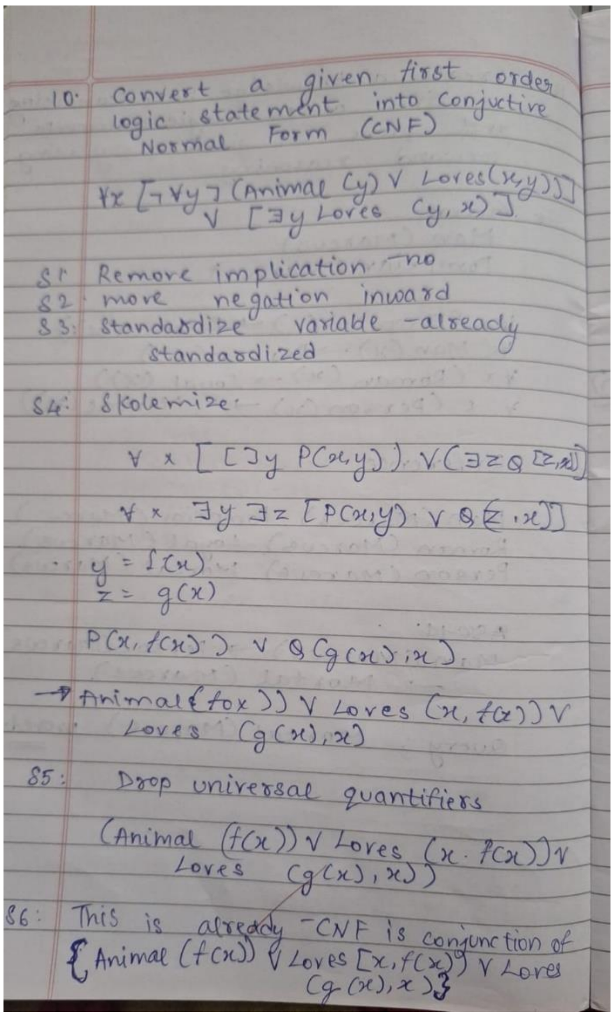
Output:

```
Knowledge base clauses:
   Food(apple)
   Likes(John,x) OR ~Food(x)
   ~Alive(x) OR ~Killed(x)
   ~Eats(x,y) OR Food(y) OR Killed(y)
   Alive(x) OR ~Killed(x)
   Alive(Anil)
   Food(vegetable)
   Eats(Anil,peanuts)
   Eats(Harry,x) OR ~Eats(Anil,x)

Query: Likes(John,peanuts)
Negated query clause will be added to KB and resolution attempted.

Result: True | Derived empty clause (success)
```

Program 11:

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:



10. Convert a given first order logic statement into Conjuctive Normal Form (CNF)

$\forall x\, [\neg \forall y\, \neg\, (Animal(y) \vee Loves(x,y))] \vee [\exists y\, Loves(y, x)]$

S1: Remove implication - no
S2: move negation inward
S3: Standardize variable - already standardized

S4: Skolemize:

$\forall x\, [(\exists y\, P(x,y)) \vee (\exists z\, Q(z,x))]$

$\forall x\, \exists y\, \exists z\, [P(x,y) \vee Q(z,x)]$

$y = f(x)$
$z = g(x)$

$P(x, f(x)) \vee Q(g(x), x)$

$\rightarrow Animal(f(x)) \vee Loves(x, f(x)) \vee Loves(g(x), x)$

S5: Drop universal quantifiers

$(Animal(f(x)) \vee Loves(x, f(x)) \vee Loves(g(x), x))$

S6: This is already CNF is conjunction of $\{Animal(f(x)) \vee Loves[x, f(x)] \vee Loves(g(x), x)\}$

Code:

```
import re
def getAttributes(string):
    expr = r'\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = r'[A-Za-z~]+\([A-Za-z,]+\)'
```

```
    return re.findall(expr, string)
def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '[' in string
    string = string.replace('~[', '')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(s):
        if c == 'V':
            s[i] = '^'
        elif c == '^':
            s[i] = 'V'
    string = ''.join(s)
    string = string.replace('~~', '')
    return f'[{string}]' if flag else string
def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
    statements = re.findall(r'\[\[[^]]+\]\]', statement)
    for s in statements:
        statement = statement.replace(s, s[1:-1])
    for predicate in getPredicates(statement):
        attributes = getAttributes(predicate)
        if ''.join(attributes).islower():
            statement = statement.replace(predicate, predicate)
        else:
            aL = [a for a in attributes if a.islower()]
            aU = [a for a in attributes if not a.islower()][0] if attributes else ''
            if aU:
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if len(aL) else
match[1]})')
    return statement
def clean_output(expr):
    # Remove multiple brackets and redundant negations
    expr = expr.replace('~~', '')
    while '[[' in expr or ']]' in expr:
        expr = expr.replace('[[', '[').replace(']]', ']')
    expr = expr.strip('[] ')
    # Remove redundant outer brackets like [(p | q)] -> p | q
    if expr.startswith('(') and expr.endswith(')'):
        expr = expr[1:-1]
    # Replace internal redundant patterns
```

```
    expr = re.sub(r'\s+', ' ', expr)
    return expr
def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^[' + statement[i+1:] + '=>' +
statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = r'\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + 'V' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement

    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
        statement = ''.join(statement)

    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[∀', '[~∀')
    statement = statement.replace('~[∃', '[~∃')

    expr = r'(~[∀V∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))

    expr = r'~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement
```

```
def main():
    print("=== FOL to CNF Converter (Simplified Output) ===")
    print("Supports ∀, ∃, ~, &, |, >>, <=>, brackets [] () {}")
    print("Example 1: ∀x[~∀y~(Animal(y) | Loves(x,y)) | ∃y Loves(y,x)]")
    print("Example 2: ~(p >> q) | (r & s)")
    print("------------------------------------------------------------")

    fol = input("Enter FOL formula: ").strip()
    try:
        result = Skolemization(fol_to_cnf(fol))
        print("\nSimplified CNF form:")
        print(clean_output(result))
    except Exception as e:
        print("\nError: Could not parse the formula.")
        print("Details:", e)
if __name__ == "__main__":
    main()
```

Output:

```
nt\Desktop\cnf.py
=== FOL to CNF Converter (Simplified Output) ===
Supports ∀, ∃, ~, &, |, >>, <=>, brackets [] () {}
Example 1: ∀x[~∀y~(Animal(y) | Loves(x,y)) | ∃y Loves(y,x)]
Example 2: ~(p >> q) | (r & s)
------------------------------------------------------------
Enter FOL formula: ∀x[~∀y~(Animal(y)∨Loves(x,y))]∨[∃y Loves(y,x)]


Simplified CNF form:
Animal(y)∨Loves(x,y))]∨[ Loves(y,x
```