

**VISVESVARAYA TECHNOLOGICAL  
UNIVERSITY**

"JnanaSangama", Belgaum -590014, Karnataka.



**LAB REPORT  
on**

**Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**Antra V (1BM23CS044)**

*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 **Antra V (1BM23CS044)**, who is bona fide 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.

Dr. K. R. Mamatha Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE
--	--

## Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	20-8-2025	Implement Tic – Tac – Toe Game Implement vacuum cleaner agent	5-12
2	28-8-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	12-20
3	3-9-2025	Implement A* search algorithm	21-27
4	10-9-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	28-29
5	17-9-2025	Simulated Annealing to Solve 8-Queens problem	30-23
6	24-9-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	34-38
7	8-10-2025	Implement unification in first order logic	39-40
8	15-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	40-42
9	29-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	42-45
10	12-11-2025	Implement Alpha-Beta Pruning. Convert given FOL to CNF	45-54

Github Link:

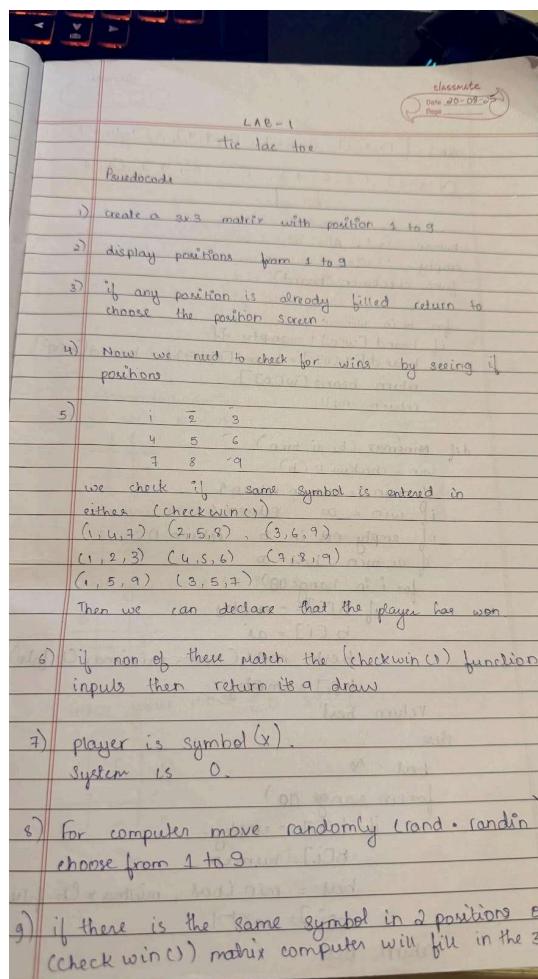
[https://github.com/Anthra044/AI\\_LAB](https://github.com/Anthra044/AI_LAB)

## Program 1

Implement Tic – Tac – Toe Game

Implement vacuum cleaner agent

Algorithm:



```

classmate
Date 40-09-25
Page 1

LAB-1
tie lac line

Pseudocode

def checkwin(board):
    win = [[1, 2, 3], [4, 5, 6], [7, 8, 9], [1, 4, 7], [2, 5, 8], [3, 6, 9], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
    human = 'x', ai = 'o'
    empty = '_'
    for w in win:
        if board[w[0]] == empty and board[w[1]] == board[w[2]]:
            return board[w[0]]
    return null

def minimax(b, ai_turn):
    win = checkwin(b)
    if win == human:
        return 1
    if win == ai:
        return -1
    if empty not in b:
        return 0
    if ai_turn:
        best = -infinity
        for i in range(9):
            if b[i] == empty:
                b[i] = ai
                best = max(best, minimax(b, False))
                b[i] = empty
        return best
    else:
        best = infinity
        for i in range(9):
            if b[i] == empty:
                b[i] = human
                best = min(best, minimax(b, True))
                b[i] = empty
        return best
  
```

```

def ai_move(b):
    score = -99
    move = -1
    for i in range(9):
        if b[i] == ' ':
            b[i] = 'A'
            best = minimax(b, False)
            b[i] = ' '
            if best > score:
                score = best
                move = i
    return move.

Welcome to Tic Tac Toe !!
You are X , AI is O .

```

Handwritten Tic Tac Toe board:


enter your move (0-8) : 5

Handwritten Tic Tac Toe board after move 5:

	X	

enter your move (0-8) : 8

Handwritten Tic Tac Toe board after moves 5 and 8:

O		O
	X	
X		

enter your move (0-8) : 6

Handwritten Tic Tac Toe board after moves 5, 8, and 6:

O	O	O
	X	
X		

AI won.

LAB-1  
Vacuum cleaner

pseudo code

```

def vacuum_cleaner(room_array):
    for room in room_array:
        label, status = room[0], room[1].lower()
        room[1] = "cleaning"
        print("currently in room", label, "is", status)
        if status == "dirty":
            print("room", label, "is already clean")
            print("moving to next room...")
        rooms = ['A', 'dirty'], ('B', 'clean'), ('C', 'clean'),
        ('D', 'clean'), ('A', 'clean')
    vacuum_cleaner(rooms)

```

Output

currently in Room A  
Room A cleaned  
moving to next room

currently in Room B  
Room B cleaned  
moving to next room

currently in Room C  
Room C cleaned  
moving next room

currently in Room D  
Room D cleaned  
moving next room

Code:

Tic tac toe game:-

import math

```

def print_board(board):
    print("\n")
    for row in board:

```

```
print(" | ".join(row))
    print("-" * 9)
    print("\n")

def check_winner(board, player):
    for row in board:
        if all(cell == player for cell in row):
            return True

    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True

    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True

    return False

def is_full(board):
    return all(cell != '' for row in board for cell in row)

def get_human_move(board):
    while True:
        try:
            move = input("Enter your move (row and column: 1 1): ")
            row, col = map(int, move.split())
            row -= 1
            col -= 1

            if row not in range(3) or col not in range(3):
                print("Please enter values between 1 and 3.")
                continue

            if board[row][col] != '':
                print("That cell is already taken. Try again.")
                continue

            return row, col
        except ValueError:
            print("Invalid input. Enter row and column numbers separated by a space.")
```

```

def minimax(board, depth, is_maximizing, human, computer):
    if check_winner(board, computer):
        return 1
    elif check_winner(board, human):
        return -1
    elif is_full(board):
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == '':
                    board[i][j] = computer
                    score = minimax(board, depth + 1, False, human, computer)
                    board[i][j] = ''
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == '':
                    board[i][j] = human
                    score = minimax(board, depth + 1, True, human, computer)
                    board[i][j] = ''
                    best_score = min(score, best_score)
        return best_score

def get_best_move(board, human, computer):
    best_score = -math.inf
    move = None

    for i in range(3):
        for j in range(3):
            if board[i][j] == '':
                board[i][j] = computer
                score = minimax(board, 0, False, human, computer)
                board[i][j] = ''
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move

def play_game():

```

```

board = [[' ' for _ in range(3)] for _ in range(3)]
human = 'X'
computer = 'O'
current_player = human # Human starts first

print("Welcome to Tic-Tac-Toe (You vs Unbeatable Computer)!")
print_board(board)

while True:
    if current_player == human:
        row, col = get_human_move(board)
    else:
        print("Computer is making a smart move...")
        row, col = get_best_move(board, human, computer)

    board[row][col] = current_player
    print_board(board)

    if check_winner(board, current_player):
        if current_player == human:
            print("🎉 You win!")
        else:
            print("💻 Computer wins!")
        break

    if is_full(board):
        print("It's a draw!")
        break

# Switch turns
current_player = computer if current_player == human else human

if name == "__main__":
    play_game()

```

Vacuum cleaner agent:-

```

def display_state(vacuum_location, rooms):
    print("\nCurrent State:")
    print(f'Vacuum is in Room {vacuum_location}')
    print(f'Room states: Left = {rooms['R']} | Right = {rooms['L']}')

def vacuum_simulation():
    # Initial setup

```

```

rooms = {
    'L': input("Is Left room dirty or clean? (Dirty/Clean): ").strip().capitalize(),
    'R': input("Is Right room dirty or clean? (Dirty/Clean): ").strip().capitalize()
}

vacuum_location = input("Where should the vacuum start? (L/R): ").strip().upper()
if vacuum_location not in ['R', 'L']:
    print("Invalid input. Defaulting to 'L'")
    vacuum_location = 'L'
print("\n--- Vacuum Cleaner Simulation Started ---")

while True:

    display_state(vacuum_location, rooms)

    # Check if both rooms are clean
    if rooms['L'] == 'Clean' and rooms['R'] == 'Clean':
        print(" ✅ All rooms are clean! Job done.")
        break

    action = input("Enter action (left / right / pick / exit): ").strip().lower()
    if action == 'exit':
        print("Simulation ended by user.")
        break

    if action == 'pick':
        if rooms[vacuum_location] == 'Dirty':
            print(f"❑ Picking dust in Room {vacuum_location}")
            rooms[vacuum_location] = 'Clean'
        else:
            print(f"Room {vacuum_location} is already clean.")
    elif action == 'left':
        vacuum_location = 'L' # Moving to right if user says left
        print("Vacuum moves to Room R (user chose 'left')")
    elif action == 'right':
        vacuum_location = 'R' # Moving to left if user says right
        print("Vacuum moves to Room L (user chose 'right')")
    else:
        print(" ❌ Invalid action. Try again.")

    print("Simulation complete.")

if name____ == "__main__":
    vacuum_simulation()

```

**Output:-**

```

IDLE Shell 3.13.7
File Edit Shell Debug Options Window Help
-----
Enter your move (row and column: 1 1): 2 1
O | X | 
X | X | 
- | O | 
-----
Computer is making a smart move...
O | X | 
X | X | O 
- | O | 
-----
Enter your move (row and column: 1 1): 3 1
O | X | 
X | X | O 
X | O | 
-----
Computer is making a smart move...
O | X | O 
X | X | O 
X | O | 
-----
Enter your move (row and column: 1 1): 3 3
O | X | O 
X | X | O 
X | O | X 
-----
>>> it's a draw!

```

```

* Is Left room dirty or clean? (Dirty/Clean): dirty
Is Right room dirty or clean? (Dirty/Clean): dirty
Where should the vacuum start? (L/R): L

--- Vacuum Cleaner Simulation Started ---

Current State:
Vacuum is in Room L
Room states: Left = Dirty | Right = Dirty
Enter action (left / right / pick / exit): right
Vacuum moves to Room R (user chose 'right')

Current State:
Vacuum is in Room R
Room states: Left = Dirty | Right = Dirty
Enter action (left / right / pick / exit):
✖ Invalid action. Try again.

Current State:
Vacuum is in Room R
Room states: Left = Dirty | Right = Dirty
Enter action (left / right / pick / exit): pick
✓ Picking dust in Room R

Current State:
Vacuum is in Room R
Room states: Left = Clean | Right = Dirty
Enter action (left / right / pick / exit):

```

## Program 2

Solve 8 puzzle problems

Implement Iterative deepening search algorithm  
Implement Depth first search algorithm

Algorithm:

LAB - 2

Implement 8 - puzzle problem ?

Algorithm

Solve 8 Puzzle (start, state)

1. define goal state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
2. Define function Heuristic (state) :

$n \leftarrow 0$

for each tile  $t$  in state

- if  $t \neq 0$ :
- $(x_1, y_1) \leftarrow$  position of  $t$  in current time
- Define Manhattan + misplaced + total tile dis to goal count of incorrect tiles.

3. start with initial board, moves = 0
4. Use a priority queue to store states by:

  - priority = moves + heuristics

5. Track visited boards in a set
6. while queue not empty
  - Remove state with lowest priority
  - If it removes the goal, return the path
  - add state to visited
  - add all valid neighbor state to queue (if not visited)

If no solution, return "No solution"

ENG

import collections

def misplaced\_tiles(current\_state, goal\_state):
 count = 0
 for i in range(9):
 if current\_state[i] != 0 and current\_state[i] != goal\_state[i]:
 count += 1
 return count

def manhattan\_distance(current\_state, goal\_state):
 distance = 0
 goal\_positions = [7, 2, 1, 0, 3, 4, 5, 6]
 for i, tile in enumerate(goal\_state):
 if tile != 0:
 goal\_position[tile] = (i // 3, i % 3)
 for i, tile in enumerate(current\_state):
 if tile != 0:
 current\_row, current\_col = i // 3, i % 3
 goal\_row, goal\_col = goal\_position[tile]
 distance += abs(current\_row - goal\_row) +
 abs(current\_col - goal\_col)
 return distance

def greedy\_best\_first\_search(initial\_state, heuristic\_func):
 open\_list = [heuristic\_func(initial\_state), initial\_state, [initial\_state]]
 visited = {tuple(initial\_state)}
 while open\_list:
 best\_node = min(open\_list, key=lambda node: node[0])
 open\_list.remove(best\_node)
 h\_value, current\_state, path = best\_node
 if current\_state == goal\_state:
 return path
 for i in range(9):
 if current\_state[i] == 0:
 for j in range(i+1, 9):
 if current\_state[j] != 0:
 new\_state = current\_state.copy()
 new\_state[i], new\_state[j] = new\_state[j], new\_state[i]
 if tuple(new\_state) not in visited:
 open\_list.append([heuristic\_func(new\_state), new\_state, path + [j]])

for action in get\_possible\_actions(current\_state):  
 next\_state = apply\_action(current\_state, action)  
 if tuple(next\_state) not in visited:  
 visited.add(tuple(next\_state))  
 h\_value = heuristic\_func(next\_state, goal\_state)  
 open\_list.append((h\_value, next\_state))  
return None.

classmate  
Date 3/9  
Page 1

LAB-21

### IDDFS

pseudocode

```

def IDDFS (root ,goal ,graph ,max_depth):
    def ddfs (node, depth, path):
        if node == goal:
            return path
        if depth == 0:
            return None
        for child in graph.get (node, []):
            result = ddfs (child, depth-1, path + [node])
            if result:
                return result
        for l in range (1,max_depth+1):
            result = ddfs (root, l, [root])
            if result:
                return result
    return None

```

```

graph TD
    A((A)) --> B((B))
    A((A)) --> C((C))
    B((B)) --> D((D))
    B((B)) --> E((E))
    C((C)) --> F((F))
    C((C)) --> G((G))
    D((D)) --> I((I))
    E((E)) --> J((J))
    F((F)) --> K((K))
    G((G)) --> K((K))

```

A

C → F → K

Code:

IDS:-

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

```
MOVES = {
```

```
    'UP': -3,
```

```
    'DOWN': 3,
```

```
    'LEFT': -1,
```

```
    'RIGHT': 1
```

```
}
```

```
def is_valid_move(pos, move):
```

```
    if move == 'UP' and pos < 3:
```

```
        return False
```

```
    if move == 'DOWN' and pos > 5:
```

```
        return False
```

```
    if move == 'LEFT' and pos % 3 == 0:
```

```
        return False
```

```
    if move == 'RIGHT' and (pos + 1) % 3 == 0:
```

```
        return False
```

```
    return True
```

```
def move_tile(state, move):
```

```
    new_state = list(state)
```

```
    idx = new_state.index(0)
```

```
    if not is_valid_move(idx, move):
```

```
        return None
```

```
    swap_idx = idx + MOVES[move]
```

```
    new_state[idx], new_state[swap_idx] = new_state[swap_idx], new_state[idx]
```

```
    return tuple(new_state)
```

```
def get_successors(state):
```

```
    successors = []
```

```
    for move in MOVES:
```

```
        new_state = move_tile(state, move)
```

```
        if new_state:
```

```
            successors.append(new_state)
```

```
    return successors
```

```
def dls(state, depth):
```

```
    stack = [(state,
```

```
        [])] visited = set()
```

```
    while stack:
```

```
        curr_state, path = stack.pop()
```

```

if curr_state in visited:
    continue
visited.add(curr_state)

if curr_state == GOAL_STATE:
    return path + [curr_state]

if len(path) >= depth:
    continue

for succ in get_successors(curr_state):
    stack.append((succ, path + [curr_state]))

return None

def ids(start_state, max_depth=50):
    for depth in range(max_depth + 1):
        result = dls(start_state, depth)
        if result:
            return result
    return None

def print_path(path):
    print("Number of steps:", len(path) - 1)
    for state in path:
        for i in range(0, 9, 3):
            print(state[i:i+3])
    print()

def get_user_input():
    print("Enter the initial 8-puzzle state row by row.")
    print("Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3")
    user_values = []

    while len(user_values) < 9:
        try:
            row_input = input(f"Row {len(user_values)//3 + 1}: ").strip()
            row = list(map(int, row_input.split()))
            if len(row) != 3:
                print("Please enter exactly 3 numbers.")
                continue
            user_values.extend(row)
        except ValueError:
            print("Invalid input. Please enter numbers only.")

    if sorted(user_values) != list(range(9)):
        print("The puzzle must contain all digits from 0 to 8 exactly once.\nLet's try again.")

```

```

    return get_user_input()

    print("\n ✅ Puzzle input
accepted!\n") return tuple(user_values)

if name____ == "__main__":
    start_state = get_user_input()
    print("==== Solving 8-puzzle with IDS ===")
    solution = ids(start_state)
    if solution:
        print_path(solution)
    else:
        print("No solution found within depth
limit.") DFS:-
```

```

from collections import deque

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

MOVES = {
    'UP': -3,
    'DOWN': 3,
    'LEFT': -1,
    'RIGHT': 1
}

def is_valid_move(pos, move):
    if move == 'UP' and pos < 3:
        return False
    if move == 'DOWN' and pos > 5:
        return False
    if move == 'LEFT' and pos % 3 == 0:
        return False
    if move == 'RIGHT' and (pos + 1) % 3 == 0:
        return False
    return True

def move_tile(state, move):
    new_state = list(state)
    idx = new_state.index(0)
    if not is_valid_move(idx, move):
        return None
    swap_idx = idx + MOVES[move]
    new_state[idx], new_state[swap_idx] = new_state[swap_idx], new_state[idx]
    return tuple(new_state)
```

```

def get_successors(state):
    successors = []
    for move in MOVES:
        new_state = move_tile(state, move)
        if new_state:
            successors.append(new_state)
    return successors

def bfs(start_state):
    queue = deque([(start_state, [])])
    visited = set()

    while queue:
        state, path = queue.popleft()
        if state in visited:
            continue
        visited.add(state)

        if state == GOAL_STATE:
            return path + [state]

        for succ in get_successors(state):
            queue.append((succ, path + [state]))

    return None

def print_path(path):
    print("Number of steps:", len(path) - 1)
    for state in path:
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print()

def get_user_input():
    print("Enter the initial 8-puzzle state row by row.")
    print("Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3")
    user_values = []

    while len(user_values) < 9:
        try:
            row_input = input(f"Row {len(user_values)//3 + 1}: ").strip()
            row = list(map(int, row_input.split()))
            if len(row) != 3:
                print("Please enter exactly 3 numbers.")
                continue
            user_values.extend(row)
        except ValueError:
            pass

```

```

print(" Invalid input. Please enter numbers only.")

if sorted(user_values) != list(range(9)):
    print("The puzzle must contain all digits from 0 to 8 exactly once.")
    return get_user_input()

print("\n Puzzle input accepted!\n")
return tuple(user_values)

if name == "__main__":
    start_state = get_user_input()
    print("== Solving 8-puzzle with BFS ==")
    result = bfs(start_state)
    if result:
        print_path(result)
    else:
        print("No solution found.")

```

### Output:-

**Output**

```

^ Enter the initial 8-puzzle state row by row.
Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 0 8

✓ Puzzle input accepted!

== Solving 8-puzzle with IDS ==
Number of steps: 1
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

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

== Code Execution Successful ==

```

**Output**

```

^ Enter the initial 8-puzzle state row by row.
Use digits 0-8 exactly once (0 is the blank). Example input for one row: 1 2 3
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 0 8

✓ Puzzle input accepted!

== Solving 8-puzzle with BFS ==
Number of steps: 1
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

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

== Code Execution Successful ==

```

### Program 3

Solve 8 puzzle problems  
Implement A\* search algorithm  
Algorithm:

Pseudocode

```

A* (start state, goal)
open = priority queue with start
parent [start] = null
g [start] = 0
f [start] = g [start] + h [start]

while open not empty:
    current = node in open with smallest f
    if current == goal:
        return path from parent
    remove current from open for each
    neighbour of current:
        if neighbour not visited or
        parent [neighbour] != temp g
        temp g = g [current] + 1
        parent [neighbour] = temp g
        f [neighbour] = temp g + h [neighbour]
        Add neighbour to open with priority
        (neighbour)
    temp g < g [neighbour]
return "NO solution".

```

-	A	1
B	2	-
3	G	3

5	6	3
1	2	8
4	7	C

$\begin{array}{ c c c } \hline 5 & 6 & 3 \\ \hline 1 & 2 & 8 \\ \hline 4 & 7 & C \\ \hline \end{array}$	$f(n) = g(n) + h(n)$ $g(v) = 1$ $h(v) = 1$ $f(v) = 1 + 1 = 2$	$\begin{array}{ c c c } \hline 1 & 2 & 3 \\ \hline 4 & 5 & C \\ \hline 7 & 8 & 1 \\ \hline \end{array}$
$\begin{array}{ c c c } \hline 5 & 6 & 3 \\ \hline 2 & 8 & 3 \\ \hline 1 & 4 & 7 \\ \hline \end{array}$	$14 - 1 = 13$	$\begin{array}{ c c c } \hline 1 & 2 & 3 \\ \hline 8 & 7 & 2 \\ \hline 6 & 5 & 4 \\ \hline \end{array}$
$\begin{array}{ c c c } \hline 5 & 6 & 3 \\ \hline 2 & 3 & 8 \\ \hline 1 & 4 & 7 \\ \hline \end{array}$	$13 - 1 = 12$	$\begin{array}{ c c c } \hline 1 & 2 & 3 \\ \hline 6 & 8 & 4 \\ \hline 5 & 4 & 7 \\ \hline \end{array}$
$\begin{array}{ c c c } \hline 5 & 8 & 6 \\ \hline 3 & 7 & - \\ \hline 2 & 1 & 4 \\ \hline \end{array}$	$12 - 1 = 11$	$\begin{array}{ c c c } \hline 5 & 6 & 1 \\ \hline 4 & 8 & 2 \\ \hline 7 & 3 & - \\ \hline \end{array}$
$\begin{array}{ c c c } \hline 5 & 8 & 6 \\ \hline 7 & 4 & - \\ \hline - & - & - \\ \hline \end{array}$	$11 - 1 = 10$	$\begin{array}{ c c c } \hline 5 & 6 & 1 \\ \hline 4 & 8 & 1 \\ \hline - & - & - \\ \hline \end{array}$

$g(n) = 1$ $h(n) = 5$ $f(n) = 1 + 5 = 6$	$p(n) = g(n) + \theta h(n)$ $\theta = 1$ $p(n) = 1 + 5 = 6$	<b>Create</b> Time Complexity: $O(bd)$ Space Complexity: $O(b^d)$ $1000 \times 3 \times 3 = 27000$
$g(n) = 1$ $h(n) = 5$ $f(n) = 1 + 5 = 6$	$g(n) = 1$ $h(n) = 5$ $f(n) = 1 + 5 = 6$	DFS: 105.003476 BFS: 0.0022 seconds IDDFS: 55.0078 seconds A*: 0.0003 seconds
$g(n) = 1$ $h(n) = 3$ $f(n) = 1 + 3 = 4$	$g(n) = 1$ $h(n) = 3$ $f(n) = 1 + 3 = 4$	out of all comparison made A* is the most suitable.
$g(n) = 1$ $h(n) = 2$ $f(n) = 1 + 2 = 3$	$g(n) = 1$ $h(n) = 2$ $f(n) = 1 + 2 = 3$	
$g(n) = 1$ $h(n) = 1$ $f(n) = 1 + 1 = 2$	$g(n) = 1$ $h(n) = 1$ $f(n) = 1 + 1 = 2$	

Code:-

```
# A* Search Algorithm Implementation in Python

from queue import PriorityQueue

def a_star_search(graph, heuristic, start, goal):
    # Priority queue to store (f_score, node,
    # path) pq = PriorityQueue()
    pq.put((0, start, [start]))

    # Dictionary to store g_scores (cost from start)
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0

    while not pq.empty():
        f, current, path = pq.get()

        # Goal check
        if current == goal:
            print("Path found:", ' → '.join(path))
            print("Total cost:", g_score[goal])
            return

        # Explore neighbors
        for neighbor, cost in graph[current]:
            tentative_g = g_score[current] + cost

            if tentative_g < g_score[neighbor]:
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic[neighbor]
                pq.put((f_score, neighbor, path + [neighbor]))
    print("No path found!")

# Example usage
if __name__ == "__main__":
    # Define the graph as adjacency
    list # Each node: [(neighbor, cost),
    ...] graph = {
        'A': [('B', 1), ('C', 3)],
        'B': [('D', 1), ('E', 5)],
        'C': [('F', 2)],
        'D': [('G', 3)],
        'E': [('G', 1)],
        'F': [('G', 5)],
        'G': []
    }
```

```
}
```

```
# Define heuristic values (estimated cost to reach goal)
heuristic = {
    'A': 7,
    'B': 6,
    'C': 5,
    'D': 4,
    'E': 2,
    'F': 1,
    'G': 0
}

start_node = 'A'
goal_node = 'G'

print("A* Search Algorithm")
print("-----")
print(f'Finding path from {start_node} → {goal_node}\n')

a_star_search(graph, heuristic, start_node, goal_node)
```

Output:-

```
A* Search Algorithm
-----
Finding path from A → G

Path found: A → B → D → G
Total cost: 5

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

## Program 4

Implement Hill Climbing search algorithm to solve N-Queens

problem Algorithm:-

```

hill climb → return the state as local maximum
current ← make Node (problem .. initial state)
loop do
    neighbour ← a highest valued successor of current
    if neighbour.value ≤ current.value then return current
    current ← neighbour

```

```

for i in range(n) for j in range(n):
    def best_neighbour(state):
        neighbor = []
        for col in range(n):
            for row in range(n):
                if row == state.col[i]:
                    neighbor.append(state(neighbors), neighbor))
        return min(neighbor, key=lambda x: x[0])
    def hill_climbing():
        state = [random.randrange(n) for i in range(n)]
        current_cost = cost(state)
        while True:
            next_cost, next_state = best_neighbour(state)
            if next_cost ≥ current_cost:
                break
            state, current_cost = next_state, next_cost
        return state, current_cost
    def print_board(state):
        for r in range(n):
            print(" ".join(["Q" if i == state[r] else " " for i in range(n)]))

```

3> Repeat until solution found or temperature is low:

- Generate a neighbour by moving one queen to a different row
- calculate cost difference  $\Delta$  between neighbour and current state
- if  $\Delta < 0$  (better), accept neighbor  
Else, accept with probability  $e^{(-\Delta/T)}$ .
- cool down:  $T \leftarrow T \alpha$ .

> Return final state.

Code:-

```

import random

def generate_board(N):
    """Generate a random board: board[i] = row of queen in column i"""
    return [random.randint(0, N-1) for _ in range(N)]

def compute_heuristic(board):
    """Compute number of pairs of queens attacking each other"""
    h = 0
    N = len(board)
    for i in range(N):
        for j in range(i+1, N):
            if board[i] == board[j]:          # same row
                h += 1
            elif abs(board[i] - board[j]) == j - i: # same diagonal
                h += 1
    return h

def get_neighbors(board):
    """Generate all neighbors by moving one queen in its column"""
    neighbors = []
    N = len(board)
    for col in range(N):
        for row in
range(N):
            if board[col] != row:
                new_board = list(board)
                new_board[col] = row
                neighbors.append(new_board)
    return neighbors

def hill_climbing(N, max_restarts=100):
    for restart in range(max_restarts):
        board = generate_board(N)
        steps = 0
        while True:
            h = compute_heuristic(board)
            if h == 0:
                print(f"Solution found in {steps} steps after {restart} restarts!")

```

return board

```
neighbors = get_neighbors(board)
h_values = [compute_heuristic(nb) for nb in neighbors]
min_h = min(h_values)
if min_h >= h: # no improvement
    break      # local maxima, do random restart
# move to the neighbor with minimum
heuristic board =
    neighbors[h_values.index(min_h)] steps += 1
print("No solution found")
return None

# Example usage
N = 8
solution = hill_climbing(N)
if solution:
    print("Board (column: row):",
solution) Output:-
```

```
Solution found in 3 steps after 10 restarts!
Board (column: row): [3, 7, 0, 4, 6, 1, 5, 2]

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

## Program 5

### Simulated Annealing

To Solve 8-Queens problem

Algorithm:-

```

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):
            conflicts += 1
return conflicts

def random_neighbour(state):
    neighbour = state.copy()
    col = random.randint(0, N-1)
    new_row = random.randint(0, N-1)
    if new_row >= neighbour[col]:
        new_row += 1
    neighbour[col] = new_row
    return neighbour

def simulated_annealing():
    T0 = 5.0, alpha = 0.995, Tmin = 1e-6, max_iters = 5000
    state = [random.randint(0, N-1) for _ in range(N)]
    current_cost = cost(state)
    T = T0
    it = 0
    while T > Tmin and it < max_iters and current_cost
        neighbour = random_neighbour(state)

```

```

row = 0
for c in range(N):
    row += "Q " if Matrix[c] == r else ". "
print(row)
print()

solution = simulated_annealing()
print("Final state (col->row):", solution)
print("cost:", cost(solution))
print("inBoard:")
print_board(solution)

Output:
final state (col->row): [2, 0, 3, 1]
cost 0
Board
. Q ..
- - . Q
Q .. .
. . Q .

```

8/10

if N = 6

Final state (col->row): [0: 4, 1, 4, 2, 3)

cost : 2

Q

Code:-

```

import random
import math

def generate_board(N):
    """Generate a random board: board[i] = row of queen in column i"""
    return [random.randint(0, N-1) for _ in range(N)]

def compute_heuristic(board):
    """Compute number of pairs of queens attacking each other"""
    h = 0
    N = len(board)
    for i in range(N):

```

```

for j in range(i+1, N):
    if board[i] == board[j]:           # same row
        h += 1
    elif abs(board[i] - board[j]) == j - i: # same diagonal
        h += 1
return h

def get_random_neighbor(board):
    """Generate a neighbor by moving one queen in its column to a random row"""
    N = len(board)
    col = random.randint(0, N-1)
    row = random.randint(0, N-1)
    while board[col] == row:
        row = random.randint(0, N-1)
    new_board = list(board)
    new_board[col] = row
    return new_board

def simulated_annealing(N, max_steps=100000, initial_temp=1000, cooling_rate=0.99):
    board = generate_board(N)
    current_h = compute_heuristic(board)
    T = initial_temp

    for step in range(max_steps):
        if current_h == 0:
            print(f"Solution found in {step} steps!")
            return board
        neighbor = get_random_neighbor(board)
        neighbor_h = compute_heuristic(neighbor)
        delta_h = neighbor_h - current_h

        if delta_h < 0:
            # Better neighbor, move to it
            board = neighbor
            current_h = neighbor_h
        else:
            # Worse neighbor, move with probability e^(-ΔH/T)
            probability = math.exp(-delta_h / T)
            if random.random() < probability:
                board = neighbor
                current_h = neighbor_h

```

```
# Cool down the temperature
T *= cooling_rate

print("No solution found")
return None

# Example usage
N = 8
solution = simulated_annealing(N)
if solution:
    print("Board (column: row):", solution)
```

Output:-

```
Solution found in 671 steps!
Board (column: row): [6, 3, 1, 7, 5, 0, 2, 4]

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

## Program 6

### Propositional Logic

Algorithm:-

<p>eg:-</p> <p>if it is raining the ground gets wet  <math>P \rightarrow Q</math></p> <p>if the ground is wet, the grass is slippery  <math>Q \rightarrow R</math></p> <p>It is raining      Is the grass slippery? (<math>\alpha</math>)      P: Rainning      Q: Ground is wet      R: Grass is slippery.</p> <p>Knowledge base (KB)</p> <ol style="list-style-type: none"> <li>YES Yes <math>Q \rightarrow P</math></li> <li>NO NO <math>\bar{P} \rightarrow \neg Q</math></li> <li>YES Yes <math>Q \vee R</math></li> </ol> <p>Query (<math>\alpha</math>): R      check KB <math>\vdash \alpha</math>.</p> <p>Truth table Enumeration:</p>	
---	--

F T F			F T F	T	T	T	T
F F T			F F T	T	T	T	F
F F F			F F F	T	T	F	F
{ propositional logic }							
P	Q	$\neg Q$	P $\wedge$ Q		P $\vee$ Q		P $\Rightarrow$ Q
F	F	T	T		F		T
F	T	F	T		F		F
T	F	F	F		F		F
T	T	F	T		T		T

Algorithm:

Func TT-Entails (KB,  $\alpha$ ) returns true or false  
 inputs: KB the knowledge base,  $\alpha$   
 sentence in propositional logic of  
 the query, a sentence in propositional logic

Symbol ← a list to propositional  
 symbol in KB and  $\alpha$   
 return TT-checkall (KB,  $\alpha$ , symbol's)

Code:-

```
from itertools import product
```

```
# Define the propositional variables
variables = ['P', 'Q', 'R']
```

```
# Define all possible truth assignments
assignments = list(product([True, False], repeat=len(variables)))
```

```
# Function to evaluate the KB sentences
def evaluate_KB(P, Q, R):
```

```

s1 = (not Q) or P      # Q -> P
s2 = (not P) or (not Q) # P ->
¬Q s3 = Q or R      # Q ∨ R
KB_true = s1 and s2 and s3
return KB_true, s1, s2, s3

# Function to evaluate a statement
def evaluate_statement(statement, P, Q, R):
    # statement is a lambda function
    return statement(P, Q, R)

# Statements to check entailment
statements = {
    "R": lambda P,Q,R: R,
    "R -> P": lambda P,Q,R: (not R) or P,
    "Q -> R": lambda P,Q,R: (not Q) or R
}

# Loop through all assignments
print(f'{P}^{5} {Q}^{5} {R}^{5} {KB}^{5} {R'}^{5} {R->P}^{7} {Q->R}^{7}")
for values in assignments:
    P, Q, R = values
    KB_true, s1, s2, s3 = evaluate_KB(P, Q, R)
    R_val = evaluate_statement(statements["R"], P, Q, R)
    RtoP_val = evaluate_statement(statements["R -> P"], P, Q, R)
    QtoR_val = evaluate_statement(statements["Q -> R"], P, Q, R)
    print(f'{P}!s^{5} {Q}!s^{5} {R}!s^{5} {KB_true}!s^{5} {R_val}!s^{5} {RtoP_val}!s^{7} {QtoR_val}!s^{7}")

) # Check entailment

def
check_entailment(statement_func):
    for values in assignments:
        P,Q,R = values
        KB_true, _, _, _ = evaluate_KB(P,Q,R)
        if KB_true and not statement_func(P,Q,R):
            return False
    return True

print("\nEntailment results:")
for name, func in statements.items():

```

```
print(f"KB entails {name}? {check_entailment(func)}")
```

returns (check-AU(kB,  $\alpha$ , rest, model)  $\vee P \wedge P = \text{true}$ )  
and  
TT - check AU(kB, rest model)  $\vee P \wedge P = \text{true}$

P	Q	R	$Q \rightarrow P$	$P \rightarrow \neg Q$	$Q \vee R$	$R \rightarrow P$	$Q \wedge R$
T	T	T	T	F	T	T	T
T	T	F	T	F	T	F	F
T	F	T	F	T	T	T	T
T	F	F	F	T	F	T	F
F	T	T	F	T	F	F	T
F	T	F	F	T	F	F	F
F	F	T	T	F	T	F	T
F	F	F	T	T	T	T	T

Q5) Does KB entail R  
 Does KB entail  $R \rightarrow R$   
 Does KB entail  $Q \rightarrow R$

all values in KB which are true are true in R  $\rightarrow$  F  
 Some values in KB are not true in R  $\rightarrow$  F

Query  $\alpha$ : AVB

Since all values in KB are true  $\alpha$  in KB entails  $\alpha$ .

A	B	C	AVC	$B \vee C$	KB   $\alpha$
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	T	T
T	T	F	F	F	T
T	T	T	T	T	T

Output:-

```
P   Q   R   KB   R   R->P   Q->R
True True True False True  True  True
True True False False False True  False
True False True True  True  True  True
True False False False False True  True
False True True False True  False  True
False True False False False True  False
False True True False False True  False
False False False False False True  True

Entailment results:
KB entails R? True
KB entails R -> P? False
KB entails Q -> R? True

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

## Program 7

Implement unification in first order logic

Algorithm:-

Handwritten notes for the unification algorithm:

- (1)  $P(f(g(x)), g(f(a)), f(a))$
- (2)  $Q(x, f(x))$
- (3)  $Q(f(y), y)$
- (4)  $H(x, g(x))$
- (5)  $H(g(y), g(g(z)))$

Step 1:  $f(x) \rightarrow f(g(x))$  (using  $x = g(x)$ )  
 $g(y) \rightarrow g(f(a))$  (using  $y = f(a)$ )  
 $y \rightarrow f(a)$  (using  $y = f(a)$ )

unification successful:  $\theta = \{x = g(x), y = f(a)\}$

a)  $Q(x, f(x))$   
 $Q(f(y), y)$  (using  $x = g(x)$  and  $y = f(a)$ )  
 $x \rightarrow f(y) \rightarrow \text{Unification Failed}$   
 $f(x) \rightarrow y$   
 Here put (1)  
 $f(f(y)) \rightarrow y$  (using  $y = f(a)$ )

$y \rightarrow b(f(y))$  is cyclic.  $y$  appears in its own replacement.

Second:  
 $(ax) \rightarrow y$        $y = f(x) \rightarrow \text{Unification Failed}$

Step 2:  $x \rightarrow g(y)$        $x = g(x)$   
 $g(x) \rightarrow g(g(y))$   
 $g(g(y)) \rightarrow g(g(z))$   
 $g(g(y)) \rightarrow g(x) \rightarrow y = z$   
 $y = z$  (using  $y = f(a)$ )

unification successful:  $\theta = \{x = g(x), y = f(a), z = f(a)\}$

Algorithm:  
 unify  $P(x, y, a)$  &  $P(b, z, a)$

First argument -  $x \neq b$ ,  $x$  is variable.  
 $b$  is constant  
 Substitute  $\{x/b\}$  apply it to  $P(b, y, a)$  and  $P(b, z, a)$

Second argument -  $y \neq z$ . Both are variables  
 Substitute  $\{y/z\}$   
 Apply it to  $P(b, z, a)$  and  $P(b, z, a)$

Third argument -  $a \neq a$  both are identical & correct  
 unification succeeded.

Code:-

```
# -----
# Unification Algorithm in First-Order Logic
# -----
```

class Term:

```
def __init__(self, name, args=None):
    self.name = name
    self.args = args or []
```

```
def is_variable(self):
```

```

        return self.args == [] and self.name[0].isupper()

def is_constant(self):
    return self.args == [] and self.name[0].islower()

def repr_(self): if
    not self.args:
        return self.name
    else:
        return f'{self.name}({", ".join(map(str, self.args))})'

# Occurs check: prevents infinite recursion like X = f(X)
def occurs_check(var, term, subs):
    if var == term:
        return True
    elif term.is_variable() and term.name in subs:
        return occurs_check(var, subs[term.name], subs)
    elif term.args:
        return any(occurs_check(var, t, subs) for t in term.args)
    return False

# Apply substitution to a term
def apply(subs, term):
    if term.is_variable() and term.name in subs:
        return apply(subs, subs[term.name])
    elif term.args:
        return Term(term.name, [apply(subs, t) for t in term.args])
    else:
        return term

# Unification function
def unify(x, y, subs=None):
    if subs is None:
        subs = {}

    x = apply(subs, x)
    y = apply(subs, y)

```

```

if x == y:
    return subs
elif x.is_variable():
    if occurs_check(x, y, subs):
        return None
    subs[x.name] = y
    return subs
elif y.is_variable():
    if occurs_check(y, x, subs):
        return None
    subs[y.name] = x
    return subs
elif x.name == y.name and len(x.args) == len(y.args):
    for a, b in zip(x.args, y.args):
        subs = unify(a, b, subs)
    if subs is None:
        return None
    return subs
else:
    return None

# ----- Example Test Cases -----
if name____ == "_main__":
    tests = [
        ("Unify X with a", Term("X"), Term("a")),
        ("Unify f(X,b) with f(a,Y)", Term("f", [Term("X"), Term("b")]), Term("f", [Term("a"),
Term("Y")])),
        ("Unify f(X) with X (occurs check fail)", Term("f", [Term("X")]), Term("X")),
        ("Unify g(X,h(Y)) with g(h(Z),h(a))", Term("g", [Term("X"), Term("h", [Term("Y")])]),
Term("g", [Term("h", [Term("Z")]), Term("h", [Term("a")])])),
        ("Unify p(X,X) with p(a,b) (should fail)", Term("p", [Term("X"), Term("X")]), Term("p",
[Term("a"), Term("b")])),
    ]
    for desc, t1, t2 in tests:
        print("Test:", desc)
        result = unify(t1, t2)
        if result is None:
            print(" ❌ Unification failed")

```

```
else:  
    print(" ✓  
        Substitution:") for k, v in  
        result.items():  
            print(f"    {k} -> {v}")  
    print()
```

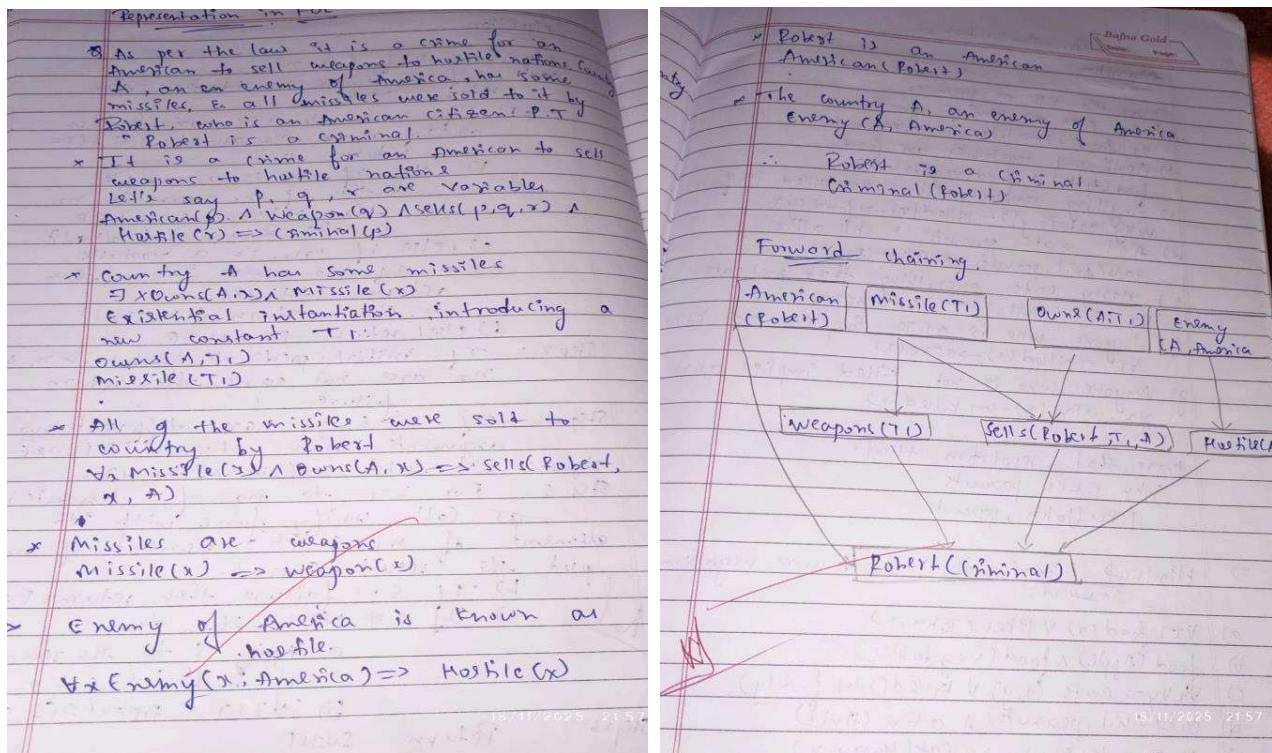
Output:-

```
Test: Unify X with a  
✓ Substitution:  
X -> a  
  
Test: Unify f(X,b) with f(a,Y)  
✓ Substitution:  
X -> a  
Y -> b  
  
Test: Unify f(X) with X (occurs check fail)  
✓ Substitution:  
X -> f(X)  
  
Test: Unify g(X,h(Y)) with g(h(Z),h(a))  
✓ Substitution:  
X -> h(Z)  
Y -> a  
  
Test: Unify p(X,X) with p(a,b) (should fail)  
✗ Unification failed  
  
==== Code Execution Successful ===
```

## Program 8

### Forward Reasoning Algorithm

Algorithm:-



Code:-

```

# Facts as tuples (Predicate, Arguments)
facts = [
    ('American', 'Robert'),
    ('Enemy', 'CountryA', 'America'),
    ('Sells', 'Robert', 'Missile1', 'CountryA'),
    ('Sells', 'Robert', 'Missile2', 'CountryA')
]

# Rule: American sells weapons to hostile nation -> Criminal
def infer_criminal(facts):
    new_facts = []
    for fact in facts:
        if fact[0] == 'American':
            person = fact[1]
            for sell in facts:
                if sell[0] == 'Sells' and sell[1] == person:
                    weapon, country = sell[2], sell[3]
                    new_facts.append((('Criminal',), (person,)))
    return new_facts

```

```
for enemy in facts:  
    if enemy[0] == 'Enemy' and enemy[1] == country and enemy[2] == 'America':  
        criminal_fact = ('Criminal', person)  
        if criminal_fact not in facts and criminal_fact not in new_facts:  
            new_facts.append(criminal_fact)  
return new_facts  
  
# Forward chaining  
while True:  
    new_facts = infer_criminal(facts)  
    if not new_facts:  
        break  
    facts.extend(new_facts)  
  
# Result  
for f in facts:  
    print(f)
```

Output:-

```
('American', 'Robert')  
('Enemy', 'CountryA', 'America')  
('Sells', 'Robert', 'Missile1', 'CountryA')  
('Sells', 'Robert', 'Missile2', 'CountryA')  
('Criminal', 'Robert')  
  
==== Code Execution Successful ===
```

## Program 9

Resolution in

FOL Algorithm:-

```

clause1, clause2 = select clause(kB)
resolvent = resolve(clause1, clause2)
if resolvent == empty clause:
    return true
if resolvent not in kB:
    kB = kB ∪ {resolvent}
if no resolvable pairs remain:
    return false

function select clauses(kB):
    return arbitrary pair(kB)

function resolve (clauses1 clauses2):
    unifier = unify (clauses1, clauses2)
    return resolve clauses with unifier (clauses1, clauses2, unifier)
else:
    return null

function unify (clause1, clause2):
    return most general unifier (clause1, clause2)

function resolve clauses with unifier (clauses, clauses, unif):
    return combined clause (clauses1, clauses2, unif)

```

Query Q(0) can be inferred. true

Code:-

```
def resolve(ci, cj):
```

Try resolving two clauses.

Returns a list of possible resolvents.

```
resolvents = []
```

for literal in ci:

```
# complement literal
```

```
comp = "-" + literal if literal[0] != "-" else literal[1:]
```

if comp in cj:

```

# resolvent = (ci ∪ cj) − {literal, comp}

new_clause = (ci - {literal}) | (cj - {comp})

resolvents.append(new_clause)

return resolvents

```

```

def resolution(clauses):
    Main resolution algorithm.

    clauses: list of sets, e.g. [{"A","B"}, {"-A"}]

    new = set()

```

```

while True:
    pairs = [(clauses[i], clauses[j])

        for i in range(len(clauses))
        for j in range(i + 1, len(clauses))]

```

```

for (ci, cj) in pairs:
    resolvents = resolve(ci, cj)

    for r in resolvents:
        print(f"Resolving {ci} and {cj} → {r}")

```

```

if len(r) == 0: # Empty clause found
    print("\n✖ Empty clause produced: CONTRADICTION")

```

```

        return True

    new.add(frozenset(r))

# Stop if no new clauses

if new.issubset(set(map(frozenset, clauses))):
    print("\nNo new clauses. Cannot derive contradiction.")

    return False

# Add new clauses

for c in new:
    if set(c) not in clauses:
        clauses.append(set(c))

```

Input :

```

clauses = [
    {"A", "B"},    # A ∨ B
    {"¬A"},       # ¬A
    {"¬B"}        # ¬B
]

```

```
resolution(clauses)
```



Output:-

```
Resolving {'B', 'A'} and {'-A'} → {'B'}
Resolving {'B', 'A'} and {'-B'} → {'A'}
Resolving {'B', 'A'} and {'-A'} → {'B'}
Resolving {'B', 'A'} and {'-B'} → {'A'}
Resolving {'-A'} and {'A'} → set()
```

✖ Empty clause produced: CONTRADICTION

## Program 10

Alpha beta pruning

Algorithm:-

```

function Alpha-Beta-Search(state):
    v ← max-value(state, -∞, +∞)
    return The actions in Actions(state) with value v
    if terminal-test(state):
        return utility(state)

function max-value(state, α, β) returns a utility-value v
    if terminal-test(state):
        return utility(state)
    v ← -∞
    for each a in Actions(state):
        v ← max(v, result(a))
        if v ≥ β then return v
        α ← max(α, v)
    return v

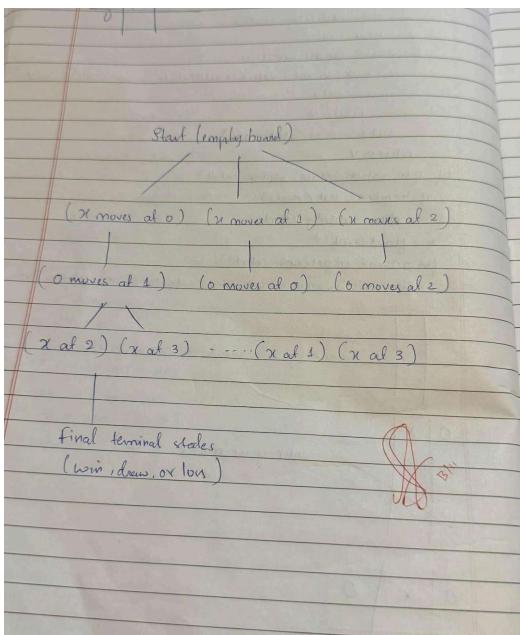
function min-value(state, α, β) returns a utility-value v
    if terminal-test(state):
        return utility(state)
    v ← +∞
    for each a in Actions(state):
        v ← min(v, result(a))
        if v ≤ α then return v
        β ← min(β, v)
    return v
  
```

```

for action in actions(state):
    v = max(v, min-value(result(state, action), alpha, beta))
    if v ≥ beta:
        return v
    alpha = max(alpha, v)
return v

def min-value(state, alpha, beta):
    if terminal-test(state):
        return utility(state)
    v = float('inf')
    for action in actions(state):
        v = min(v, max-value(result(state, action), alpha, beta))

    enter your move ( O )
    | X |
    enter your move ( O )
    | O | X
    |   |   |
  
```



Code:-

```
# Alpha-Beta Pruning in Python for tic tac toe

import math

import random

# ----- TIC TAC TOE CLASS -----
class TicTacToe:
    def __init__(self):
        self.board = [" "] * 9

    def print_board(self):
        b = self.board
        print("\n")
        print(f" {b[0]} | {b[1]} | {b[2]} ")
        print(" ---+---+---")
        print(f" {b[3]} | {b[4]} | {b[5]} ")
        print(" ---+---+---")
        print(f" {b[6]} | {b[7]} | {b[8]} ")
        print("\n")

    def available_moves(self):
        return [i for i, v in enumerate(self.board) if v == " "]

    def make_move(self, move, player):
        self.board[move] = player

    def undo_move(self, move):
        self.board[move] = " "

    def winner(self):
        winning_patterns = [
            [0,1,2], [3,4,5], [6,7,8], # rows
            [0,3,6], [1,4,7], [2,5,8], # columns
            [0,4,8], [2,4,6] # diagonals
        ]
        for pattern in winning_patterns:
            if self.board[pattern[0]] == self.board[pattern[1]] == self.board[pattern[2]] != " ":
                return self.board[pattern[0]]
        return None
```

```

[0,4,8], [2,4,6]      # diagonals
]
for a,b,c in winning_patterns:
    if self.board[a] == self.board[b] == self.board[c] != " ":
        return self.board[a]
return None

def is_full(self):
    return " " not in self.board

def game_over(self):
    return self.winner() is not None or self.is_full()

# ----- ALPHA-BETA MINIMAX -----
def minimax(game, depth, alpha, beta, maximizing):
    winner = game.winner()

    if winner == "X": return 10 - depth
    if winner == "O": return depth - 10
    if game.is_full(): return 0

    if maximizing:
        max_eval = -math.inf
        for move in game.available_moves():
            game.make_move(move, "X")
            eval = minimax(game, depth + 1, alpha, beta, False)
            game.undo_move(move)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha: break
        return max_eval
    else:
        min_eval = math.inf
        for move in game.available_moves():

```

```

game.make_move(move, "O")
eval = minimax(game, depth + 1, alpha, beta, True)
game.undo_move(move)
min_eval = min(min_eval, eval)
beta = min(beta, eval)
if beta <= alpha: break
return min_eval

def best_move(game, player):
    best_val = -math.inf if player == "X" else math.inf
    best_move = None

    for move in game.available_moves():
        game.make_move(move, player)
        move_val = minimax(game, 0, -math.inf, math.inf, player == "O")
        game.undo_move(move)

        if player == "X" and move_val > best_val:
            best_val = move_val
            best_move = move
        elif player == "O" and move_val < best_val:
            best_val = move_val
            best_move = move

    return best_move

# ----- COMPUTER VS COMPUTER -----
def computer_vs_computer():
    game = TicTacToe()
    player = "X"

    print("Computer vs Computer (Alpha-Beta Pruning)\n")
    game.print_board()

```

```
while not game.game_over():
    move = best_move(game, player)
    game.make_move(move, player)
    print(f"Player {player} plays at {move}")
    game.print_board()
    player = "O" if player == "X" else "X"

winner = game.winner()

if winner:
    print(f"Winner: {winner}")
else:
    print("It's a draw!")

# Run the match
computer_vs_computer()
```

Output:-

The terminal window shows a game of Tic-Tac-Toe between two AI players. Player O starts at position 7, followed by Player X at position 8. Both players make moves at positions 7 and 8 respectively, resulting in a draw.

```
Player O plays at 7

x | x | o
---+---+---
o | o | x
---+---+---
x | o |

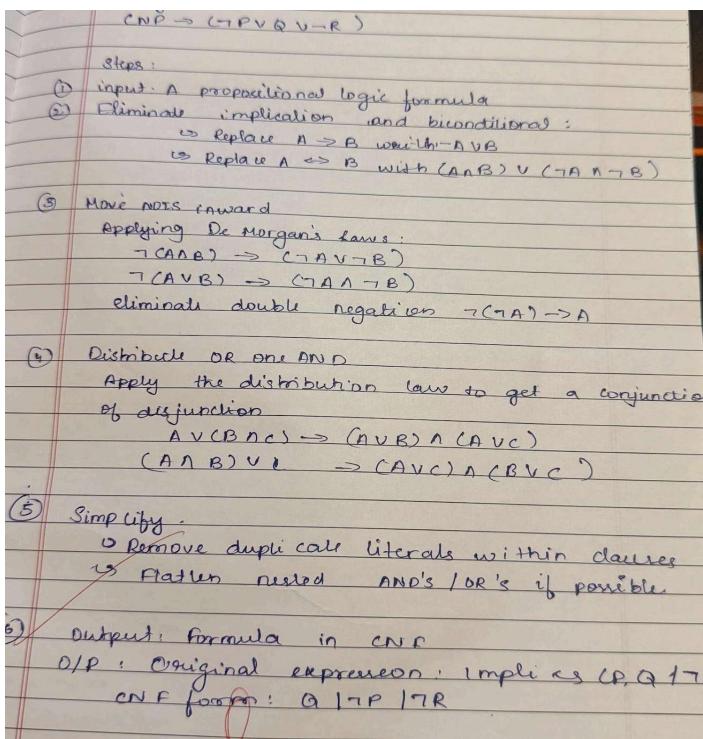
Player X plays at 8

x | x | o
---+---+---
o | o | x
---+---+---
x | o | x

It's a draw!
```

## Program 10

Conversion of a given FOL to CNF



Code:-

```
import re
```

```
# ----- STEP 1: Remove Implications -----
```

```
def remove_implications(expr):
```

```
    expr = expr.replace("->", "=>")
```

```
    # A => B becomes  $\neg A \vee B$ 
```

```
    while "=>" in expr:
```

```
        expr = re.sub(r"(.*)=>(.*)",

```

```
                    r"(\neg\1) | \2", expr)
```

```
    return expr
```

```
# ----- STEP 2: Move NOT inward -----
```

```
def move_not_inward(expr):
```

```
    #  $\sim\sim A \rightarrow A$ 
```

```
    expr = expr.replace("~~", "")
```

```
# DeMorgan: ~(A & B) → (~A | ~B)
expr = expr.replace("~, "~")
expr = expr.replace("&", ") & (")
expr = expr.replace("|", ") | (")
return expr
```

# ----- STEP 3: Standardize variables -----

```
def standardize_variables(expr):
    # replace x, y, z with unique names
    counter = 0
    def repl(match):
        nonlocal
        counter
        counter += 1
        return "v" + str(counter)
    return re.sub(r"\b[a-z]\b", repl, expr)
```

# ----- STEP 4: Skolemization -----

```
def skolemize(expr):
```

```
count = 1
while "∃" in expr:
    expr = re.sub(r"∃[a-z]\s*", f"Sk{count} ", expr, count=1)
    count += 1
return expr
```

# ----- STEP 5: Drop universal quantifiers -----

```
def drop_universal(expr):
    expr = expr.replace("∀", "")
    return expr
```

# ----- STEP 6: Distribute OR over AND -----

```
def distribute(expr):
```

```
expr = expr.replace(" & ", ") & (")
expr = expr.replace(" | ", ") | (")
return "(" + expr + ")"
```

```
def fol_to_cnf(expr):
    print("\nOriginal:", expr)
```

```
expr = remove_implications(expr)
print("After removing implications:", expr)
```

```
expr = move_not_inward(expr)
print("After moving NOT inwards:", expr)
```

```
expr = standardize_variables(expr)
print("After standardizing variables:", expr)
```

```
expr = skolemize(expr)
print("After skolemization:", expr)
```

```
expr = drop_universal(expr)
print("After dropping universals:",
      expr)
```

```
expr = distribute(expr)
print("Final CNF:", expr)
```

```
return expr
expr = "∀x ( P(x) -> ∃y Q(y,x) )"
```

```
fol_to_cnf(expr)
```

Output:

```
Original:  $\forall x (\ P(x) \rightarrow \exists y Q(y,x) )$ 
After removing implications:  $(\sim(\forall x (\ P(x) )) \mid (\ \exists y Q(y,x) ))$ 
After moving NOT inwards:  $(\sim\forall x (\ P(x) )) \mid (\ (\exists y Q(y,x) ))$ 
After standardizing variables:  $(\sim\forall v1 (\ P(v2) )) \mid (\ (\exists v3 Q(v4,v5) ))$ 
After skolemization:  $(\sim\forall v1 (\ P(v2) )) \mid (\ (\forall Sk1 \exists Q(v4,v5) ))$ 
After dropping universals:  $(\sim v1 (\ P(v2) )) \mid (\ (\forall Sk1 \exists Q(v4,v5) ))$ 
Final CNF:  $((\sim v1 (\ P(v2) )) \mid ((\forall Sk1 \exists Q(v4,v5) )))$ 
```