VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Ayush Ranjan (1BM23CS058)

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 **Ayush Ranjan (1BM23CS058),** 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.

Sandhya A Kulkarni Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/AyushRanjan-58/AI

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner
agent

Algorithm for Tic-Tac-Toe:

Lab – 1.

TIC - TAC -TOE

Algorithm

1) Start.

2) Print the board

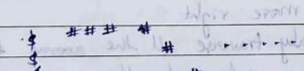
```
for i in range (3)
    for j in range (3)
    print (board [i][j])
Initializing the board.
for i from 0 to 3
  for j from 0 to 3
    board [i][j] = '-'
```

3) Taking input from the user, Ask for the user to select a position in 3×3 region.

4) Mark that region with symbol "$". And take the symbol "#" for the the opponents marking.

5) Traverse the 3×3 matrix to find the empty spaces present in the board such that the player cannot overwrite the symbol.

6) Check for the winning Conditions, i.e, there can be 8 possible winning Conditions.



So, check these winning Condition. If it is $ → Then player 1 wins.
if it is # → Then player 2 wins.

Code for Tic-Tac-Toe:

```
board={1:' ',2:' ',3:' ',

    4:' ',5:' ',6:' ',

    7:' ',8:' ',9:' '

}

def printBoard(board):

  print(board[1]+'|'+board[2]+'|'+board[3]

  ) print('-+-+-')

  print(board[4] + '|' + board[5] + '|' +

  board[6]) print('-+-+-')

  print(board[7] + '|' + board[8] + '|' +

  board[9]) print('\n')


def spaceFree(pos):

  if(board[pos]==' '):

     return True
```

```
    else:
      return False


def checkWin():
  if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
      return True
  elif(board[4]==board[5] and board[4]==board[6] and board[4]!='
      '): return True
  elif(board[7]==board[8] and board[7]==board[9] and board[7]!='
      '): return True
  elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
      return True
  elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
      return True
  elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
      return True
  elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
      return True
  elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
      return True
  else:
      return False


def checkMoveForWin(move):
```

```
    if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
        return True
    elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
        return True
    elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False


def checkDraw():
    for key in
        board.keys(): if
        (board[key]==' '):
            return False
```

return True

```python
def insertLetter(letter, position):
  if (spaceFree(position)):
      board[position] = letter
      printBoard(board)


      if (checkDraw()):
          print('Draw!')
      elif (checkWin()):
          if (letter == 'X'):
              print('Bot wins!')
          else:
              print('You win!')
      return


  else:
      print('Position taken, please pick a different position.')
      position = int(input('Enter new position: '))
      insertLetter(letter, position)
      return


player = 'O'
bot ='X'
```

```
def playerMove():
  position=int(input('Enter position for
  O:')) insertLetter(player, position)
  return

def compMove():
  bestScore=-100
  0 bestMove=0
  for key in
     board.keys(): if
     (board[key]==' '):
        board[key]=bot
        score = minimax(board, False)
        board[key] = ' '
        if (score > bestScore):
           bestScore = score
           bestMove = key

  insertLetter(bot, bestMove)
  return
def minimax(board,
  isMaximizing): if
  (checkMoveForWin(bot)):
```

```
    return 1
elif (checkMoveForWin(player)):
```

```
    return -1
elif (checkDraw()):
    return 0

if isMaximizing:
    bestScore = -1000

    for key in board.keys():
        if board[key] == ' ':
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if (score > bestScore):
                bestScore = score
    return bestScore
else:
    bestScore = 1000

    for key in board.keys():
        if board[key] == ' ':
            board[key] = player
            score = minimax(board, True)
            board[key] = ' '
            if (score < bestScore):
```

```
            bestScore = score

    return bestScore


while not checkWin():

  compMove()

  playerMove()
```

Algorithm for Vacuum Cleaner Agent:

Vacuum Cleaner.

1) Start.

2) Initialize the room A, B, C, D as 0.
i.e, A = 0, B = 0, C = 0, D = 0. 0 represents the room is dirty, 1 represents room is clean.

3) Start the robot from room A.
If Room[A] = 0
Then robot ← clean.
The robot cleans the room.
Update 0 ← 1
move right.

4) When 1 is returned, then it should move to the next room. So, similarly,
if Room[B] = 0
Then Robot ← clean,
status → 1.
move right

5) If Room is already clean, then update the status and return that ; move to next room.
if Room[C] = 1
Return 1.
move right.

6) Similarly traverse all the rooms A, B, C, D until the status of all rooms = 1.
Room[A] = Room[B] = Room[C] = Room[D] = 1

7) Return All rooms are clean.

```
def vacuum_world():

    goal_state = {'A': '0', 'B':

    '0'}

    cost = 0

    location_input = input("Enter Location of Vacuum: ")

    status_input = input("Enter status of " + location_input + ": ")

    status_input_complement = input("Enter status of other room: ")


    print("Initial Location Condition: " + str(goal_state))


    if location_input == 'A':


        print("Vacuum is placed in Location A")

        if status_input == '1':

            print("Location A is Dirty.")


            goal_state['A'] = '0'

            cost += 1

            print("Cost for CLEANING A: " + str(cost))

            print("Location A has been Cleaned.")


            if status_input_complement == '1':


                print("Location B is Dirty.")
```

```
print("Moving right to Location B.")
```

```
        cost += 1
        print("COST for moving RIGHT: " + str(cost))


        goal_state['B'] = '0'
        cost += 1
        print("COST for SUCK: " + str(cost))
        print("Location B has been
        Cleaned.")
    else:
        print("Location B is already clean. No action.")
else:
    print("Location A is already
    clean.") if status_input_complement
    == '1':
        print("Location B is Dirty.")
        print("Moving right to Location B.")
        cost += 1
        print("COST for moving RIGHT: " + str(cost))


        goal_state['B'] = '0'
        cost += 1
        print("Cost for SUCK: " + str(cost))
        print("Location B has been Cleaned.")
    else:
```

```
print("Location B is already clean. No action.")
```

```
else:
    print("Vacuum is placed in Location B")

    if status_input == '1':
        print("Location B is Dirty.")

        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING B: " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':

            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1
            print("COST for moving LEFT: " + str(cost))

            goal_state['A'] = '0'
            cost += 1
            print("COST for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean. No action.")
```

```
        else:
            print("Location B is already clean.")
            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving left to Location A.")
                cost += 1
                print("COST for moving LEFT: " + str(cost))

                goal_state['A'] = '0'
                cost += 1
                print("Cost for SUCK: " + str(cost))
                print("Location A has been Cleaned.")
            else:
                print("Location A is already clean. No action.")

    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

Program 2 :

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

8 – puzzle usig DFS :

Algorithm :

3/9/25.

Bafna Gold
Date: Page:

Lab - 2.

8 - Puzzle Game.

Algorithm.

1) Start. 1-D list.

2) Initially, we represent the puzzle state using 0, 1, 2, 3, 4, 5, 6, 7, 8.
where 0 ← blank tile.

3) There will be a 3×3 grid and we need to arrange in ordered way.

0	1	2
3	4	5
6	7	8

4) Using Manhattan distance, we need to calculate the minimum distance to shift the displaced tile to it's original place.

i.e.,

6	1	2
4	0	5
3	7	8

So, 6 should move 2 places down & 3 should move 1 place up

```
def manhattan_distance (state):
dist = 0
for i ← v (state),
  if v == 0:
     Continue.
     goal_index = v - 1.
```

```
N = 3

class PuzzleState:

    def _init_(self, board, x, y, depth):

        self.board = board

        self.x = x

        self.y = y

        self.depth =

        depth

row = [0, 0, -1, 1]

col = [-1, 1, 0, 0]

def is_goal_state(board):

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

    return board == goal

def is_valid(x, y):

    return 0 <= x < N and 0 <= y <

N def print_board(board):

    for row in board:

        print(' '.join(map(str, row)))

    print("          ")

def solve_puzzle_dfs(start, x,

    y): stack = []

    visited = set()

    stack.append(PuzzleState(start, x, y,

    0)) visited.add(tuple(map(tuple, start)))
```

```
    while stack:
        curr = stack.pop()
        print(f'Depth: {curr.depth}')
        print_board(curr.board)
        if is_goal_state(curr.board):
            print(f'Goal state reached at depth {curr.depth}')
            return
        for i in range(4):
            new_x = curr.x + row[i]
            new_y = curr.y + col[i]
            if is_valid(new_x, new_y):
                new_board = [row[:] for row in curr.board]
                new_board[curr.x][curr.y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[curr.x][curr.y]
                board_tuple = tuple(map(tuple, new_board))
                if board_tuple not in visited:
                    visited.add(board_tuple)
                    stack.append(PuzzleState(new_board, new_x, new_y, curr.depth + 1))
    print('No solution found (DFS Brute Force reached depth limit)')
if _name___ == '_main_':
    start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]
    x, y = 1, 1
    print('Initial State:')
    print_board(start)
```

solve_puzzle_dfs(start, x,

y) 8-puzzle for IDS :

Pseudocode.

1) Start.

2) Initialize A* function,

function A-star (start, goal):
  open-set = priority queue().
  open-set.push(start, priority = heuristics(start))

3) While open-set is non-empty,
  current = open-set.pop().

  if curr = goal:
    return solution-path

4) for neighbour in neighbour(current):
(g)   tentative-g = g-score [current] + 1.

5) If neighbour not in g-score,

  g-score [neighbour] = tentative-g.
    f = tentative-g + heuristic(neighbour).

  open-set.push(neighbour, priority = f).

  return no solution

6) Now arrange all the popped elements in the order way to get solution

```
N = 3

class PuzzleState:

    def _init_(self, board, x, y, depth):

        self.board = board

        self.x = x

        self.y = y

self.depth = depth


row_moves = [0, 0, -1, 1]

col_moves = [-1, 1, 0, 0]

def is_goal_state(board):

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

    return board == goal

def is_valid(x, y):

    return 0 <= x < N and 0 <= y <

N def print_board(board):

    for r in board:

        print(' '.join(map(str, r)))

    print("          ")

def dfs_with_depth_limit(start, x, y, depth_limit):

    stack = []
```

```
visited = set()
stack.append(PuzzleState(start, x, y,
0)) visited.add(tuple(map(tuple, start)))
while stack:
    curr = stack.pop()
    print(f'Depth: {curr.depth}')
    print_board(curr.board)

    if is_goal_state(curr.board):

        print(f'Goal state reached at depth {curr.depth}')
        return True

    if curr.depth ==
        depth_limit: continue

    for i in range(4):
        new_x = curr.x + row_moves[i]
        new_y = curr.y + col_moves[i]

        if is_valid(new_x, new_y):
            new_board = [row[:] for row in curr.board]
                new_board[curr.x][curr.y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[curr.x][curr.y]
```

```
            board_tuple = tuple(map(tuple, new_board))

            if board_tuple not in visited:

                visited.add(board_tuple)

                stack.append(PuzzleState(new_board, new_x, new_y, curr.depth + 1))

    return False

def iterative_deepening_search(start, x, y, max_depth=50):

    for depth in range(max_depth):

        print(f"Trying depth limit: {depth}")
        found = dfs_with_depth_limit(start, x, y, depth)
        if found:
      print(f"Solved at depth {depth}!")

return

    print("No solution found within max depth

limit.") if _name___== '_main_':

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

    x, y = 1, 1

    print('Initial State:')

    print_board(start)

    iterative_deepening_search(start, x, y)
```

Pseudocode.

1) function IDDFS (root, goal):
   depth = 0.
   found = DLS (root, goal, depth).
   return true.

loop → while true.

2) function DLS (node, goal, limit):
   if node is null,
   return false.

   if node == goal:
   return true

   if limit == 0:
   return false.

3) for each child in children (node)
   if DLS (child, goal, limit - 1) == true:
   return true.
   return false

Path → A → C → F → K.

4) max - depth (src) {
   if (src == null) {
   return 0
   }
   L = max-depth (src . left);
   R = max - depth (src . right).

Problem 3:

Implement A* search algorithm

Algorithm:

Algorithm

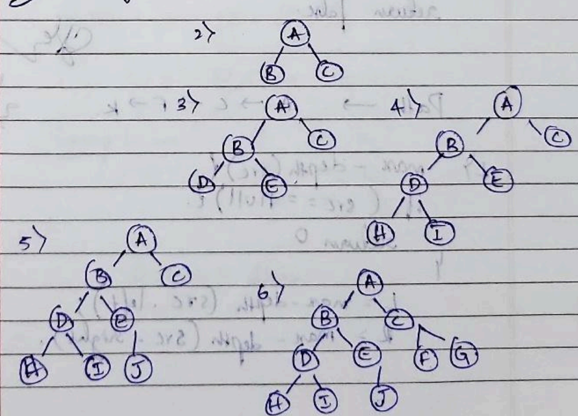
Consider,

1) bool DLS (src, target, limit):
   if (src == target)
   return true.

2) If not, then in,
   for src ← i:
   if (DLS (i, target, limit))
   return true.
   return false.

3) bool IDDFS (src, target, max-deep).
   for limit 0 to max-depth.
   if DLS (src, target, max-limit).
   return false.
   return 1 + max (l, r).

Tracking : 1) A



Code :

```
from copy import deepcopy

import heapq
```

```
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]
DIRECTIONS =
    { 'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}
def print_state(state):
    for row in state:
        print(row)
    print('-' * 10)
def state_to_tuple(state):
    return tuple(tuple(row) for row in state)
def find_zero(state):
    for i in range(3):
        for j in
        range(3):
            if state[i][j] == 0:
                return i, j
```

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

```
    x, y = find_zero(state)
    dx, dy = DIRECTIONS[direction]
    nx, ny = x + dx, y + dy
    if 0 <= nx < 3 and 0 <= ny <
        3: new_state =
        deepcopy(state)
        new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
        return new_state
    return None
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in
        range(3):
            value = state[i][j]
            if value != 0:
                goal_x = (value - 1) // 3
                goal_y = (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance
def a_star(start_state, goal_state):
    open_list = []
    g_score = {state_to_tuple(start_state): 0}
    f_score = {state_to_tuple(start_state): manhattan_distance(start_state)}
```

```
heapq.heappush(open_list, (f_score[state_to_tuple(start_state)], start_state, []))

visited = set()
```

```
    iteration = 0
    print("\nStarting A* Search...\n")
    while open_list:
        iteration += 1
        _, current_state, path = heapq.heappop(open_list)
        print(f"Iteration {iteration}:")
        print_state(current_state)
                print(f"g(n): {len(path)}, h(n): {manhattan_distance(current_state)}, f(n): {len(path) +
manhattan_distance(current_state)}")
        state_key = state_to_tuple(current_state)
        if state_key in visited:
            continue
        visited.add(state_key)
        if current_state == goal_state:
            print("Goal state reached!\n")
            return path + [current_state]
        for direction in DIRECTIONS.keys():
            new_state = move(current_state, direction)
            if new_state:
                new_key =
                state_to_tuple(new_state) if
                new_key not in visited:
                    new_g = len(path) + 1
                    new_f = new_g + manhattan_distance(new_state)
                    heapq.heappush(open_list, (new_f, new_state, path + [current_state]))
```

```
    print("No solution found.")
    return None

if __name__ == "__main__":
    print("Enter the initial 3x3 puzzle state (use 0 for the blank):")
    initial_state = []
    for i in range(3):
        row = input(f"Row {i+1} (space-separated): ").strip().split()
        initial_state.append([int(num) for num in row])
    solution_path = a_star(initial_state, GOAL_STATE)

    if solution_path:
        print("Solution Path (step-by-step):")
        for idx, state in enumerate(solution_path):
            print(f"Step {idx}:")
            print_state(state)
        print(f"Puzzle Solved in {len(solution_path) - 1} moves!")
    else:
        print("Could not find a solution.")
```

Problem 4:

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

8/10/25.

Hill Climbing Algorithm.

1) Start with a random board i.e, one queen per column, random row.

2) Then we need to compute heuristics. ($h(n)$). i.e, the number of attacking queen.

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

where: $h(n)$ = attacking queen.

$g(n)$ = number of iterations / steps.

3) find / get best neighbour after generating the neighbours.

calculate attack :

n = len (board)

for i from 0 to n

board [i] == board [j]

attacks += 1.

return attack.

4) Evaluate the neighbours value to move to next position.

i.e,

if h < best_h :

best_h = h.

best_board = best_h

else

return h.

Code :

```
import random

def calculate_conflicts(state):

    conflicts = 0
```

```
    N = len(state)
    for i in
    range(N):
        for j in range(i + 1, N):
            if state[i] ==
            state[j]:
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def get_neighbors(state):
    neighbors = []
    N = len(state)
    for col in range(N):
        for row in
        range(N):
            if state[col] != row:
                new_state = state.copy()
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors
def print_board(state):
    N = len(state)
    board = [["." for _ in range(N)] for _ in range(N)]
```

```
for col in range(N):
    board[state[col]][col] = "Q"
for row in board:
```

```
        print(" ".join(row))
    print()


def hill_climbing_nqueens(N=4):
    current_state = [random.randint(0, N - 1) for _ in range(N)]
    current_cost = calculate_conflicts(current_state)
    print_board(current_state)
    while True:
        if current_cost == 0:
            return current_state
        neighbors = get_neighbors(current_state)
        best_neighbor = min(neighbors, key=calculate_conflicts)
        best_cost = calculate_conflicts(best_neighbor)
        if best_cost >= current_cost:
            return current_state
        else:
            current_state, current_cost = best_neighbor, best_cost
            print_board(current_state)
solution = hill_climbing_nqueens(4)
print("Final Solution:", solution)
print("Conflicts:", calculate_conflicts(solution))
```

Problem 5:

Simulated Annealing to Solve 8-Queens problem

Algorithm:

8/10/2025.

Simulated Annealing.

1) Start.

2) Initialize current_state & initial state.
Check for the temperature T.

Simulated annealing().
current ← initial_state.
T ← a large positive value.

3) while T > 0
do,
next ← a random neighbour of current.
ΔE ← current.cost − next.cost.

4) if ΔE > 0
current ← next.
else
current ← next with probability = p. $e^{\Delta E/T}$.

decrease T.

return current.

5) End / terminate.

Code :

```
import random

import math

def random_state(n=8):

    """Generate a random board: list of row positions for each column."""

    return [random.randint(0, n - 1) for _ in range(n)]
```

```
def conflicts(state):
    """
    Number of attacking pairs of
    queens. Lower is better. A solution
    has 0. """
    h = 0
    n = len(state)
    for i in
    range(n):
        for j in range(i + 1, n):
            if state[i] ==
            state[j]:
                h += 1
            if abs(state[i] - state[j]) == abs(i -
               j): h += 1
    return h
def
    random_neighbor(state):
    """
    Create a neighbor by moving a queen in one random
    column to a random row.
    """
    n = len(state)
    new_state = state.copy()
    col = random.randint(0, n - 1)
```

```
row = random.randint(0, n - 1)

new_state[col] = row
```

```
    return new_state
def simulated_annealing(max_steps=100000, n=8):
    current = random_state(n)
    current_cost = conflicts(current)
    T = 1.0
    cooling = 0.0001
    for step in
        range(max_steps): if
        current_cost == 0:
            return current, step
        T = max(T * math.exp(-cooling * step),
        0.0001) next_state =
        random_neighbor(current) next_cost =
        conflicts(next_state)
        delta = current_cost - next_cost
        if delta > 0 or random.random() < math.exp(delta /
            T): current = next_state
            current_cost =
    next_cost return None,
    max_steps
solution, steps =
simulated_annealing() if solution:
    print(f"Solution found in {steps}
    steps:") print("State:", solution)
    print("Conflicts:", conflicts(solution))
```

```
else:
    print("No solution found.")
```

Problem 6:

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

Algorithm :

Bafna Gold
Date: Page:

15/10/25.

LAB - 6.

Create a knowledge Base (KB) using propositional logic & show that the given query entails the KB (or) not.

1) Start.

2) We need to Convert all sentences in KB to CNF i.e, the normal form.
function TT - Entails? (KB, α) symbols ← (KB, α)
return true / false.

3) Input should be the set of propositional sentences i.e, KB, α, the query & a sentence in propositional logic.

4) Negate the query α and convert $\neg\alpha$ to CNF, add $\neg\alpha$ to the KB.
function TT-Check-All (KB, α, symbols, model)
return true / false.
if Empty? (symbols) then,
if PL-True? (KB, model)
return PL-True? (α, model).
else
return true.

5) else do
P ← First (symbols).
rest ← REST (symbols).
return (TT-Check-All (KB, α, rest, model)
P = true.

Code :

```
import itertools

class Formula:

    def _init_(self, symbols, expr):

        self.symbols = set(symbols)
```

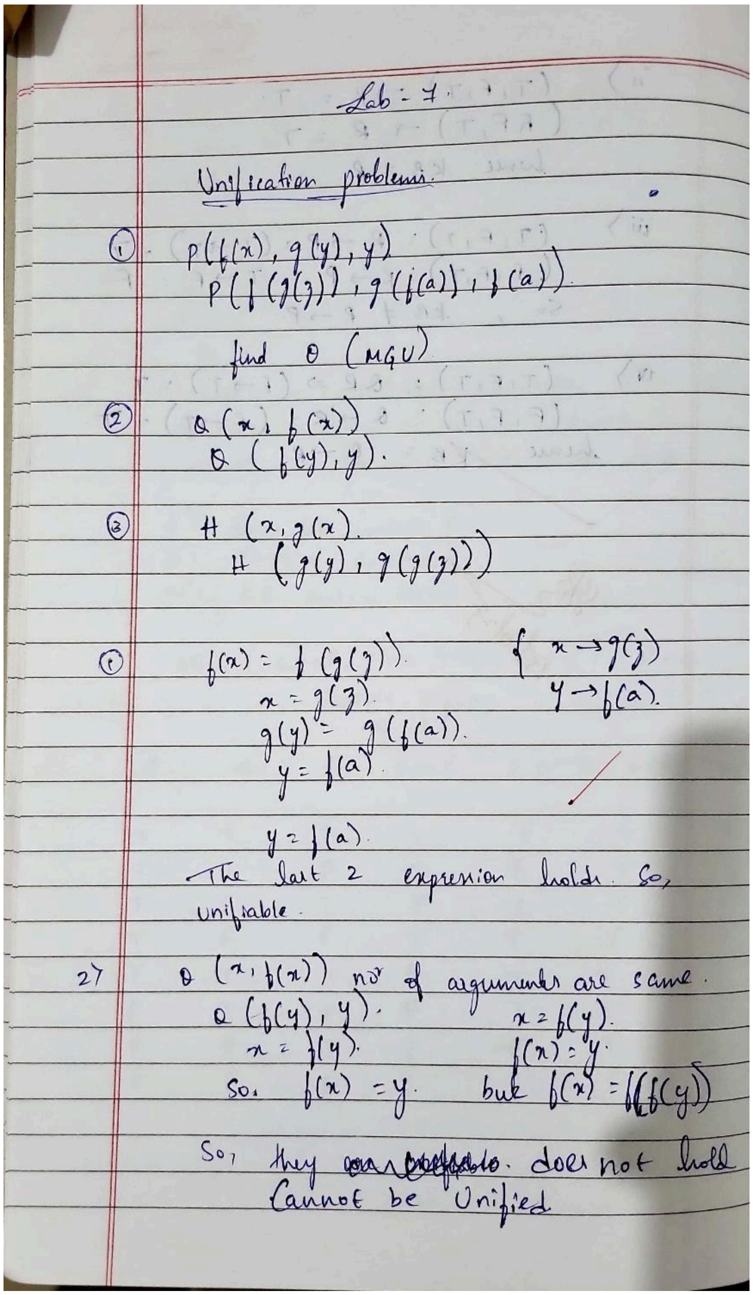
```
        self.expr = expr
    def evaluate(self, model):
        return
        self.expr(model)
def get_all_symbols(kb, query):
    symbols = set()
    for f in kb + [query]:
        symbols |=
        f.symbols
    return sorted(symbols)
def entails(kb, query):
    symbols = get_all_symbols(kb, query)
    for values in itertools.product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if all(f.evaluate(model) for f in kb):
            if not query.evaluate(model):
                print("Counterexample found:", model)
                return False
    return True
R_implies_W = Formula({"R", "W"}, lambda m: (not m["R"]) or m["W"])
S_implies_W = Formula({"S", "W"}, lambda m: (not m["S"]) or m["W"])
W_implies_L = Formula({"W", "L"}, lambda m: (not m["W"]) or m["L"])
C_implies_R = Formula({"C", "R"}, lambda m: (not m["C"]) or m["R"])
S_or_C = Formula({"S", "C"}, lambda m: m["S"] or m["C"])
```

```python
S_equiv_D = Formula({"S", "D"}, lambda m: m["S"] == m["D"])


Query_L = Formula({"L"}, lambda m: m["L"])



KB = [
    R_implies_W,
    S_implies_W,
    W_implies_L,
    C_implies_R,
    S_or_C,
    S_equiv_D
]
result = entails(KB, Query_L)
print("\nDoes KB entail L (grass is slippery)? →", result)
```

Program 7:

Implement unification in first order logic

Algorithm:



Lab : 7.

Unification problems.

① $P(f(x), g(y), y)$
$P(f(g(z)), g(f(a)), f(a))$

find θ (MGU)

② $Q(x, f(x))$
$Q(f(y), y)$.

③ $H(x, g(x))$.
$H(g(y), g(g(z)))$

① $f(x) = f(g(z))$. $\{ x \to g(z)$
$x = g(z)$. $y \to f(a)$.
$g(y) = g(f(a))$.
$y = f(a)$.

$y = f(a)$.
The last 2 expression holds. So, unifiable.

2) $Q(x, f(x))$ no of arguments are same.
$Q(f(y), y)$. $x = f(y)$.
$x = f(y)$. $f(x) = y$.
So, $f(x) = y$. but $f(x) = f(f(y))$

So, they ~~are unifiable~~. does not hold
Cannot be Unified

Code :

```
def occurs_check(var,
    expr): if var == expr:
```

```
    return True
elif isinstance(expr, list):
```

```
        return any(occurs_check(var, subexpr) for subexpr in expr)
    return False



def unify(x, y,
    subst=None): if subst is
    None:
        subst = {}

    if isinstance(x, str) and
        x.islower(): if x in subst:
            return unify(subst[x], y,
        subst) elif occurs_check(x, y):
            return None
        else:
            subst[x] = y
            return subst

    elif isinstance(y, str) and
        y.islower(): if y in subst:
            return unify(x, subst[y], subst)
        elif occurs_check(y, x):
            return None
        else:
```

subst[y] = x

```
        return subst

    elif x == y:
        return subst

    elif isinstance(x, list) and isinstance(y,
        list): if len(x) != len(y):
            return None
        for xi, yi in zip(x, y):
            subst = unify(xi, yi,
            subst) if subst is None:
                return None
        return subst

    else:
        return None




expr1 = ["Knows", "John", "x"]
expr2 = ["Knows", "y", "Mary"]

print("Expression 1:", expr1)
print("Expression 2:", expr2)
```

```
result = unify(expr1,
expr2) if result:
    for k, v in result.items():
        print(f"{k} / {v}")
else:
    print("Unification failed.")
```

Program 8:

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

Code :

```
from copy import deepcopy

def occurs_check(var, expr):
    if var == expr:
        return True
```

```
elif isinstance(expr, list):
        return any(occurs_check(var, subexpr) for subexpr in expr)
    return False


def substitute(expr, subst):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    elif isinstance(expr, list):
        return [substitute(e, subst) for e in expr]
    return expr
def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    if subst is
        None: return
        None
    if x == y:
        return
        subst
    elif isinstance(x, str) and
        x.islower(): if x in subst:
            return unify(subst[x], y,
        subst) elif occurs_check(x, y):
            return None
        else:
            subst[x] = y
```

```
        return subst
    elif isinstance(y, str) and y.islower():
```

```
        return unify(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for a, b in zip(x, y):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
    else:
        return None


def parse_sentence(sentence):
    """Parse sentence like 'Parent(John, x)' → ['Parent', 'John', 'x']"""
    sentence = sentence.strip()
    if '(' in sentence and ')' in sentence:
        pred = sentence[:sentence.index('(')]
        args = sentence[sentence.index('(') + 1:sentence.index(')')].split(',')
        args = [a.strip() for a in args]
        return [pred] + args
    else:
        return [sentence]


def to_string(expr):
    if len(expr) ==
    1:
        return expr[0]
    else:
```

```
return f"{expr[0]}({', '.join(expr[1:])})"
```

```
def fol_fc_ask(KB, query):
    print("FORWARD CHAINING START ")
    print("Initial Knowledge Base:")
    for fact in KB:
        print(" ", fact)
    print("Query:", query)


    iteration = 0
    new = set()

    while True:
        iteration += 1
        print(f"\n--- Iteration {iteration} ---")
        n_new = set()

        for rule in KB.copy():
            if "=>" in rule:
                premise, conclusion = rule.split("=>")
                premise = premise.strip()
                conclusion = conclusion.strip()
                premises = [p.strip() for p in premise.split("^")]

                print(f"\nChecking rule: {rule}")

                substitutions = []
```

```
        for fact in KB:
            if "=>" not in fact:
                for p in premises:
                    s = unify(parse_sentence(p), parse_sentence(fact))
                    if s is not None:
                        print(f" Premise '{p}' unified with fact '{fact}' using {s}")
                        substitutions.append(s)



        for s in substitutions:
            new_fact = to_string(substitute(parse_sentence(conclusion),
            s)) if new_fact not in KB and new_fact not in n_new:
                print(f" => New fact inferred: {new_fact}")
                n_new.add(new_fact)
                phi = unify(parse_sentence(new_fact), parse_sentence(query))
                if phi is not None:
                    print("\n Query proved!")
                    print(f"Substitution set: {phi}")
                    return phi

    if not n_new:
        print("\nNo new inferences. Forward chaining ends.")
        print("Query cannot be proved.")
        return False
```

```
        print("\nNewly inferred facts this
        iteration:") for fact in n_new:
            print("  ", fact)

        KB |= n_new
        print("\nUpdated Knowledge Base:")
        for fact in KB:
            print("  ", fact)

KB = {
    "Parent(John, Mary)",
    "Parent(Mary, Alice)",
    "Parent(x, y) ^ Parent(y, z) => Grandparent(x, z)"
}

query = "Grandparent(John, Alice)"

result = fol_fc_ask(deepcopy(KB), query)
```

Program 9:

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

Algorithm :

12/11/25

First Order Logic.

Resolution is a theorem proving technique that proceeds by building refutation proofs i.e, by contradiction

1. Eliminate biconditional and implication:
   i.e, replacing $(\alpha \to \beta) \wedge (\beta \to \alpha) \equiv \alpha \Leftrightarrow \beta$
   Eliminate $\to$, replacing $\alpha \to \beta$ with $\neg\alpha \vee \beta$.

2. Move $\neg$ inwards:
   i.e, $\neg(\forall x\ p) \equiv \exists x\ \neg p$,
   $\neg(\exists x\ p) \equiv \forall x\ \neg p$;
   $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$,
   $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$,
   $\neg\neg\alpha \equiv \alpha$.

3. Standardize variables apart by renaming them i.e, each quantifier should use a different variable.

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

5. Drop Universal quantifiers.

6. Distribute $\wedge$ over $\vee$:
   $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Code :

```
import copy

def is_variable(x):
    return isinstance(x, str) and x[0].islower()
```

```
def unify(x, y, theta=None):
    if theta is None:
        theta = {}
    if theta == "FAIL":
        return "FAIL"
    elif x == y:
        return theta
    elif is_variable(x):
        return unify_var(x, y, theta)
    elif is_variable(y):
        return unify_var(y, x, theta)
    elif isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        return unify(x[1:], y[1:], unify(x[0], y[0], theta))
    else:
        return "FAIL"


def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x,
    theta) elif x in theta:
        return unify(var, theta[x],
    theta) else:
        if occurs_check(var, x, theta):
            return "FAIL"
        theta_copy = theta.copy()
```

```
        theta_copy[var] = x
        return theta_copy

def occurs_check(var, x, theta):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_check(var, arg, theta) for arg in x)
    elif isinstance(x, str) and x in theta:
        return occurs_check(var, theta[x], theta)
    return False


def substitute(theta, clause):
    new_clause = []
    for pred in clause:
        name = pred[0]
        args = pred[1]
        new_args = [(theta[arg] if arg in theta else arg) for arg in args]
        new_clause.append([name, new_args])
    return new_clause


def resolve(ci, cj):
    resolvents = []
```

```
    for pi in ci:
        for pj in
        cj:

            if pi[0] == "~" + pj[0] or pj[0] == "~" + pi[0]:
                theta = unify(pi[1], pj[1],
                {}) if theta != "FAIL":

                    ci_new = substitute(theta, [x for x in ci if x !=
                    pi]) cj_new = substitute(theta, [x for x in cj if x
                    != pj])

                    resolvent = []
                    for term in ci_new +
                        cj_new: if term not in
                        resolvent:
                            resolvent.append(term)

                    resolvents.append(resolvent)

    return resolvents


def clause_to_hashable(clause):
    """
    clause = [["Pred", ["a","b"]], ["~Q", ["x"]]]
```

```
→ (("Pred", ("a","b")), ("~Q", ("x",)))
"""
return tuple((pred[0], tuple(pred[1])) for pred in clause)
```

```python
def hashable_to_clause(tup):
    """ reverse conversion """
    return [[pred, list(args)] for pred, args in tup]


def resolution_algorithm(KB, query):

    KB = copy.deepcopy(KB)

    neg_query = []
    for q in query:
        if q[0].startswith("~"):
            neg_query.append([q[0][1:], q[1]])
        else:
            neg_query.append(["~" + q[0], q[1]])
    KB.append(neg_query)

    print("\nInitial KB + neg(query):")
    for c in KB:
        print(c)

    new = set()
```

```
while True:

    pairs = [(KB[i], KB[j]) for i in range(len(KB)) for j in range(i+1, len(KB))]

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

        for r in resolvents:

            if r == []:
                print("\n❗ Contradiction found → QUERY PROVED.\n")
                return True

            r_hash = clause_to_hashable(r)

            if r_hash not in new:
                new.add(r_hash)

    if all(hashable_to_clause(r) in KB for r in new):
        print("\nNo new clauses → QUERY NOT
        PROVED.\n") return False
```

```
    for r in new:
        clause =
        hashable_to_clause(r) if
        clause not in KB:
            KB.append(clause)
KB = [
  [["Parent", ["x", "y"]], ["~Mother", ["x",
  "y"]]], [["Mother", ["Mary", "John"]]]
]
query = [["Parent", ["Mary", "John"]]]
print("Trying to prove:", query)
resolution_algorithm(KB, query)
```

Program 10:

Implement Alpha-Beta Pruning.

Algorithm :

Adversial Search.

function Alpha-beta-(state).
return an action.

u ← max-value (state, -∞, +∞)
return the action in actions (state) with value v.

function max-value (state; α, β) return a utility value.
If terminal-test (state) then return utility (state).
v ← -∞.

for each a in Actions (state) do
v ← max (v, min-val (result (s, a), α, β))
If v ≥ β then return v.
α ← MAX (α, v).
return v.

function Min-Value (state, α, β) returns a utility val
if Terminal-test (state).
return Utility (state).
v ← -∞.
for each α in Action (state) do.
v ← MAX (v, Min-Value (Result (s, a), α, β))

if v ≤ α then return v.
β ← Min (β, v).
return v.

Code :

```
import math


def alphabeta(node, depth, alpha, beta, maximizingPlayer):
```

```
if depth == 0 or isinstance(node, int):
    return node

if maximizingPlayer:
    value = -math.inf
    for child in node:
        value = max(value, alphabeta(child, depth - 1, alpha, beta, False))
        alpha = max(alpha, value)
        if beta <= alpha:
            print(f"Pruned in MAX node: alpha={alpha}, beta={beta}")
            break
    return value

else:
    value = math.inf
    for child in node:
        value = min(value, alphabeta(child, depth - 1, alpha, beta, True))
        beta = min(beta, value)
        if beta <= alpha:
            print(f"Pruned in MIN node: alpha={alpha}, beta={beta}")
            break
    return value
```

```
game_tree = [
    [3, 5, 6],
    [1, 2, 4],
    [7, 9, 8]
]


result = alphabeta(game_tree, 2, -math.inf, math.inf, True)
print("\nFinal Result (Best value for Max):", result)
```