VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Mandaar Adarsh(1BM22CS358)

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
Sep-2024 to Jan-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 **Mandaar Adarsh(1BM22CS358) ,**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.

Saritha A.N Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor& HOD Department of CSE, BMSCE

Index

GITHUB LINK: https://github.com/mandaar017/ailab

Program 1
Implement Tic –Tac –Toe
Game

Algorithm:

LAB-1

Date 4/10/24
Page 1

(1) TIC TAC TOE

I/P

```
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 check move for win(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)
```

Date __/__/__
Page 3

```
        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

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 = -1000
    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 (checkmove for win(bot)):
        return 1
    elif (check move for win (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()
```

O/P

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

Enter position for O:

04.10.24

Code:

```
def print_board(board):
    print("\n")
    for row in board:
        print("|".join(row))
        print("-" * 5)
    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 board[0][0] == player and board[1][1] == player and board[2][2] == player:
        return True
    if board[0][2] == player and board[1][1] == player and board[2][0] == player:
        return True
    return False


def is_board_full(board):
    return all([cell != ' ' for row in board for cell in row])
```

```
def player_move(board, player):
    while True:
        try:
            move = int(input(f"Player {player}, enter your move (1-9): ")) - 1
            if move < 0 or move >= 9:
                raise ValueError
            row, col = divmod(move, 3)
            if board[row][col] == ' ':
                board[row][col] = player
                break
            else:
                print("This spot is already taken. Try again.")
        except ValueError:
            print("Invalid input. Enter a number between 1 and 9.")


def play_game():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    current_player = 'X'
    game_over = False
    print("Welcome to Tic Tac Toe!")
    print("Player X goes first.")
    print("Enter a number between 1-9 to make your move (1 is top-left and 9 is
bottom-right).")
```

```
    print_board(board)

    while not game_over:

        player_move(board, current_player)

        print_board(board)

        if check_winner(board, current_player):

            print(f"Player {current_player} wins!")

            game_over = True

        elif is_board_full(board):

            print("It's a tie!")

            game_over = True

        else

  current_player = 'O' if current_player == 'X' else 'X'

if _name___== "_main_":

    play_game(
```

Implement Vaccum Cleaner Agent

→ Vaccum - Cleaner agent :-

Algorithm :

1.) Initialize the agent starting (x, y)

2.) loop untill all cells are clean :
- Perciève the current cell
- If the cell is directly
clean the current cell
- else
- check if surrounding cell is dirty
- Move to dirty cell using DFS or BFS or randomly .
- If no dirty cells are percived, stop.

3.) End .

Sem
18.10

Code:

```
if state['A'] == 0 and state['B'] == 0:

print("Turning vacuum off") return


     if state[loc] == 1:

        state[loc] = 0

        count += 1

        print(f"Cleaned {loc}.")

        next_loc = 'B' if loc == 'A' else 'A'

        state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): "))

        if(state[next_loc]!=1):

          state[next_loc]=int(input(f"Is {next_loc} dirty? (0 if clean, 1 if dirty): "))

     if(state[loc]==1):

       rec(state,loc)

     else:

       next_loc = 'B' if loc == 'A' else 'A'

       dire="left" if loc=="B" else "right"

       print(loc,"is clean")

       print(f"Moving vacuum {dire}")

       if state[next_loc] == 1:

          rec(state, next_loc)


   state = {}
```

```
state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): "))

state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): "))

loc = input("Enter location (A or B): ")

rec(state, loc)

print("Cost:",count)

print(state)
```

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

Program 2

Implement 8 puzzle problems using (DFS) and (BFS)

LAB-2

Date
Page

8-Puzzle Problem :-

→ Breadth-First Search :

Algorithm

Let fringe be a list containing the initial state.

loop

if fringe is empty return failure

Node ← remove-first (fringe)

if Node is a goal

then return the path from initial state to Node.

else generate all successors of node, and add generated nodes to the back of fringe in queue

end loop.

→ Depth-First Search :

Algorithm

Let fringe be a list containing the initial state

loop

if fringe is empty return failure

Node ← remove-first(fringe)

if Node is a goal

return the path from initial state to Node

else generate all successors of Node, and add generated nodes to the front ~~back~~ of fringe ~~in queue~~.

Endloop.

CODE:for dfs

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

    def is_goal(state):
        return state == goal_state

    def find_blank(state):
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    return i, j

    def swap(state, i1, j1, i2, j2):
        new_state = [row[:] for row in state]
        new_state[i1][j1], new_state[i2][j2] = new_state[i2][j2], new_state[i1][j1]
        return new_state

    def get_neighbors(state):
        neighbors = []
        i, j = find_blank(state)
        if i > 0:
            neighbors.append(swap(state, i, j, i - 1, j))
        if i < 2:
            neighbors.append(swap(state, i, j, i + 1, j))
        if j > 0:
            neighbors.append(swap(state, i, j, i, j - 1))
        if j < 2:
```

```
        neighbors.append(swap(state, i, j, i, j + 1))
    return neighbors
def dfs(state, visited, path):
    state_tuple = tuple(tuple(row) for row in state)
    if state_tuple in visited:
        return None
    visited.add(state_tuple)
    if is_goal(state):
        return path
    for neighbor in get_neighbors(state):
        result = dfs(neighbor, visited, path + [neighbor])
        if result is not None:
            return result

    return None

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

visited = set()
solution = dfs(initial_state, visited, [])
```

```
if solution:
    print("Solution found in", len(solution), "steps:")
    for step in solution:
        for row in step:
            print(row)
        print()
else:
    print("No solution found.")
```

```
Solution found in 2 steps:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

CODE: for bfs

```
class PuzzleState:
    def __init__(self, board, moves=0):

        self.board = board
        self.blank_index = board.index(0)  # Find the index of the blank space (0)
        self.moves = moves

    def get_possible_moves(self):
        possible_moves = []
        row, col = divmod(self.blank_index, 3)

        # Define possible movements: up, down, left, right
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # (row_change, col_change)

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_blank_index = new_row * 3 + new_col
                new_board = self.board[:]
                # Swap the blank with the adjacent tile
                new_board[self.blank_index], new_board[new_blank_index] =
new_board[new_blank_index], new_board[self.blank_index]
                possible_moves.append(PuzzleState(new_board, self.moves + 1))
        return possible_moves

    def is_goal(self, goal_state):
```

```
        return self.board == goal_state

def depth_limited_search(state, depth, goal_state):
    if state.is_goal(goal_state):
        return state
    if depth == 0:
        return None

    for next_state in state.get_possible_moves():
        result = depth_limited_search(next_state, depth - 1, goal_state)
        if result is not None:
            return result

    return None

def iterative_deepening_search(initial_state, goal_state):
    depth = 0
    while True:
        result = depth_limited_search(initial_state, depth, goal_state)
        if result is not None:
            return result
        depth += 1
```

```python
# Example Usage
if __name___== "__main__":
    initial_board = [2, 8, 3, 1, 6, 4, 7, 0, 5]  # Initial state
    goal_state = [2, 0, 3, 1, 8, 4, 7, 6, 5]  # Final state
    initial_state = PuzzleState(initial_board)

    solution = iterative_deepening_search(initial_state, goal_state)

    if solution:
        print("Solution found!")
        print("Moves:", solution.moves)
        print("Final Board State:", solution.board)
    else:
        print("No solution found.")
```

```
Solution found!
Moves: 2
Final Board State: [2, 0, 3, 1, 8, 4, 7, 6, 5]
```

Program 3

Implement A* Search Algorithm

Misplaced Tiles:

LAB-03

→ A* Algorithms :

function A* search(problem) returns a solution or failure
node ← a node n with n.state problem initial state, n.g=0
frontier ← a priority queue ordered by ascending g+h, only element n

loop do
  if empty?(frontier) then return failure
  n ← pop(frontier)
  if problem.goalTest(n.state) then return sol(n)
  for each action a in problem.actions(n.state) do
    n' ← childnode(problem, n, a)
    insert(n', g(n') + h(n'), frontier).

```
import heapq
def manhattan_distance(state, goal):
distance = 0

for i in range(3):

    for j in range(3):

        tile = state[i][j]

        if tile != 0:

            for r in range(3):

                for c in range(3):

                    if goal[r][c] == tile:

                        target_row, target_col = r, c

                        break

            distance += abs(target_row - i) + abs(target_col - j)

return distance


def findmin(open_list, goal):

    minv = float('inf')

    best_state = None

    for state in open_list:

        h = manhattan_distance(state['state'], goal)

        f = state['g'] + h

        if f < minv:

            minv = f

            best_state = state

    open_list.remove(best_state)
```

```
    return best_state

def operation(state):
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None
```

```
def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state


def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))
initial_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]
goal_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]


open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []
```

```
while open_list:
    best_state = findmin(open_list, goal_state)

    h = manhattan_distance(best_state['state'], goal_state)
    f = best_state['g'] + h

    print(f"g(n) = {best_state['g']}, h(n) = {h}, f(n) = {f}")
    print_state(best_state['state'])
    print()

    if h == 0:
        print("Goal state reached!")
        break

    visited_states.append(best_state['state'])
    next_states = operation(best_state)

    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

if h == 0:
```

```
        moves = []
        goal_state_reached = best_state
        while goal_state_reached['move'] is not None:
            moves.append(goal_state_reached['move'])
            goal_state_reached = goal_state_reached['parent']
        moves.reverse()
        print("\nMoves to reach the goal state:", moves)
    else:
        print("No solution found.")
```

```
g(n) = 0, h(n) = 5, f(n) = 5
2 8 3
1 6 4
7 0 5

g(n) = 1, h(n) = 4, f(n) = 5
2 8 3
1 0 4
7 6 5

g(n) = 2, h(n) = 3, f(n) = 5
2 0 3
1 8 4
7 6 5

g(n) = 3, h(n) = 2, f(n) = 5
0 2 3
1 8 4
7 6 5

g(n) = 4, h(n) = 1, f(n) = 5
1 2 3
0 8 4
7 6 5

g(n) = 5, h(n) = 0, f(n) = 5
1 2 3
8 0 4
7 6 5

Goal state reached!

Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']
```

Misplaced Tiles:

```
import heapq

defind_blank_tile(st
ate):
for i in range(3):
    for j in range(3):
if state[i][j] == 0:
    return i, j
return None

def count_misplaced_tiles(state, goal):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal[i][j]:
                misplaced += 1
    return misplaced
def generate_moves(state):
    moves = []
    x, y = find_blank_tile(state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
```

```
            moves.append(new_state)
    return moves

def print_state(state):
    for row in state:
        print(row)
    print()

def a_star_8_puzzle(start, goal):

    open_list = []

    heapq.heappush(open_list, (count_misplaced_tiles(start, goal), 0, start, None))

    visited = set()

    while open_list:

        f_n, g_n, current_state, previous_state = heapq.heappop(open_list)

        print(f"g(n) = {g_n}, h(n) = {f_n - g_n}, f(n) = {f_n}")
        print_state(current_state)
```

```
        if current_state == goal:
            print("Goal state reached!")
            return
        visited.add(tuple(map(tuple, current_state)))
        for move in generate_moves(current_state):
            move_tuple = tuple(map(tuple, move))
            if move_tuple not in visited:

                g_move = g_n + 1
                h_move = count_misplaced_tiles(move, goal)
                f_move = g_move + h_move
                heapq.heappush(open_list, (f_move, g_move, move, current_state))


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


a_star_8_puzzle(start_state, goal_state)
```

```
g(n) = 0, h(n) = 4, f(n) = 4
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

g(n) = 1, h(n) = 3, f(n) = 4
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

g(n) = 2, h(n) = 3, f(n) = 5
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 2, h(n) = 3, f(n) = 5
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]

g(n) = 3, h(n) = 2, f(n) = 5
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 4, h(n) = 1, f(n) = 5
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

g(n) = 5, h(n) = 0, f(n) = 5
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Goal state reached!
```

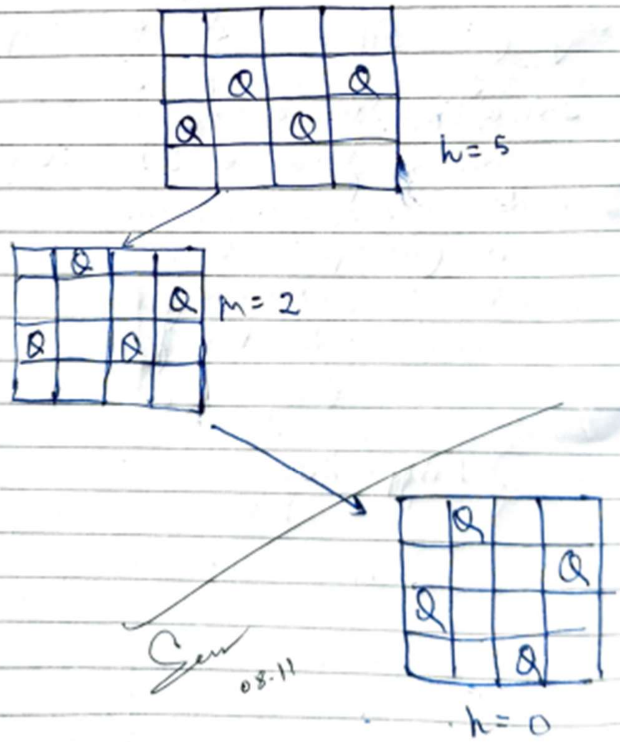
Program 4

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

Hill Climbing Algorithm :-

→ fuction HillClimbing(problem)
returns a solution (or) failure
current ← node write state = problem. initial
loop do
neighbour ← a highest valued successor of current
if neighbour. value ← current value then
return current
current ← neighbour

Solution



08.11

```
import random

class NQueens:

    def _init_(self, n):

        self.n = n

        self.board = self.init_board()

    def init_board(self):

        # Randomly place one queen in each column

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



    def fitness(self, board):

        # Count the number of pairs of queens attacking each other

        conflicts = 0

        for col in range(self.n):

            for other_col in range(col + 1, self.n):

                if board[col] == board[other_col] or abs(board[col] - board[other_col]) == abs(col -
other_col):

                    conflicts += 1

        return conflicts

    def get_neighbors(self, board):

        neighbors = []

        for col in range(self.n):

            for row in range(self.n):

                if row != board[col]:  # Move queen to a different row in the same column

                    new_board = board[:]
```

```
                new_board[col] = row
                neighbors.append(new_board)
        return neighbors
    def hill_climbing(self):
        current_board = self.board
        current_fitness = self.fitness(current_board)

        while current_fitness > 0:
            neighbors = self.get_neighbors(current_board)
            next_board = None
            next_fitness = current_fitness
            for neighbor in neighbors:
                neighbor_fitness = self.fitness(neighbor)
                if neighbor_fitness < next_fitness:
                    next_fitness = neighbor_fitness
                    next_board = neighbor
            if next_board is None:
                # Stuck at local maximum, can either return or restart
                print("Stuck at local maximum. Restarting...")
                self.board = self.init_board()
                current_board = self.board
                current_fitness = self.fitness(current_board)
            else:
```

```
            current_board = next_board
            current_fitness = next_fitness
        return current_board

# Example usage
if __name__ == "__main__":
    n = 4  # Size of the board (N)
    n_queens_solver = NQueens(n)
    solution = n_queens_solver.hill_climbing()

    print("Solution:")
    for row in solution:
        line = ['Q' if i == row else '.' for i in range(n)]
        print(' '.join(line))
```

```
Solution:
. Q . .
. . . Q
Q . . .
. . Q .
```

Program 5

Simulated Annealing to Solve 8-Queens problem.

LAB-5

Date 15/11/24
Page 10

→ Simulated Annealing

```
current ← initial state
T ← a large positive value
while T>0 do
    next ← a random neighbour of current
    ΔE ← current.cost − next.cost
    If ΔE > 0 then
        current ← next
    else
        current ← next with probability P=e^(ΔE/T)
    end if
    decrease T
end while
return current
```

→ for n-Queen's problem

```
initial state(N):
    return [random.randint(0,N-1) for _ in
            range(N)]
// here n-queens are generated
cost():
conf = 0
N = len (state)
    for i in range(N):
        for j in range(i+1, N):
            if conflict == true
                conflict +1; return conflict
```

```python
import random
import math

def print_board(state):
    size = len(state)
    for i in range(size):
        row = ['.'] * size
        row[state[i]] = 'Q'
        print(' '.join(row))
    print()

def calculate_conflicts(state):
    conflicts = 0
    size = len(state)
    for i in range(size):
        for j in range(i + 1, size):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def random_state(size):
    return [random.randint(0, size - 1) for _ in range(size)]
```

```
def neighbor(state):
    new_state = state[:]
    idx = random.randint(0, len(state) - 1)
    new_state[idx] = random.randint(0, len(state) - 1)
    return new_state


def simulated_annealing(size, initial_temp, cooling_rate):
    current_state = random_state(size)
    current_conflicts = calculate_conflicts(current_state)
    temperature = initial_temp

    while temperature > 1:
        new_state = neighbor(current_state)
        new_conflicts = calculate_conflicts(new_state)

        # If new state is better, accept it
        if new_conflicts < current_conflicts:
            current_state, current_conflicts = new_state, new_conflicts
        else:
            # Accept with a probability based on temperature
            acceptance_probability = math.exp((current_conflicts - new_conflicts) / temperature)
            if random.random() < acceptance_probability:
                current_state, current_conflicts = new_state, new_conflicts
```

```
        temperature *= cooling_rate

    return current_state

def main():
    size = 8
    initial_temp = 1000
    cooling_rate = 0.995

    solution = simulated_annealing(size, initial_temp, cooling_rate)
    print("Solution found:")
    print_board(solution)
    print("Conflicts:", calculate_conflicts(solution))

if __name__ == "__main__":
    main()
```

```
Solution found:
. . . . . . Q .
. . Q . . . . .
. . . . . . . Q
Q . . . . . . .
. . . . Q . . .
. . . Q . . . .
. . . . Q . . .
. . . . . Q . .

Conflicts: 6
```

Program 6:

```
def truth_table_entailment():
    print(f"{'A':<7}{'B':<7}{'C':<7}{'A or C':<12}{'B or not C':<15}{'KB':<8}{'alpha':<10}")
    print("-" * 65)
    all_entail = True
    for A in [False, True]:
        for B in [False, True]:
            for C in [False, True]:
                # Calculate individual components
                A_or_C = A or C                    # A or C
                B_or_not_C = B or (not C)          # B or not C
                KB = A_or_C and B_or_not_C         # KB = (A or C) and (B or not C)
                alpha = A or B                     # alpha = A or B

                # Determine if KB entails alpha for this row
                kb_entails_alpha = (not KB) or alpha  # True if KB implies alpha

                # If in any row KB does not entail alpha, set flag to False
                if not kb_entails_alpha:
                    all_entail = False

                # Print the results for this row

print(f"{str(A):<7}{str(B):<7}{str(C):<7}{str(A_or_C):<12}{str(B_or_not_C):<15}{str(KB):<8
}{str(alpha):<10}")
```

```
    # Final result based on all rows
    if all_entail:
        print("\nKB entails alpha for all cases.")
    else:
        print("\nKB does not entail alpha for all cases.")


# Run the function to display the truth table and final result
truth_table_entailment()
```

```
A      B      C      A or C      B or not C      KB      alpha
-----------------------------------------------------------------
False  False  False  False       True            False   False
False  False  True   True        False           False   False
False  True   False  False       True            False   True
False  True   True   True        True            True    True
True   False  False  True        True            True    True
True   False  True   True        False           False   True
True   True   False  True        True            True    True
True   True   True   True        True            True    True

KB entails alpha for all cases.
```

Program 7

Implement unification in first order logic.

Unification Algorithm:

Step 1:- If Ψ_1 or Ψ_2 is a variable or constant, then:

a.) If Ψ_1 or Ψ_2 are identical, return NIL

b.) Else if Ψ_1 is a variable,

a.) then if Ψ_1 occurs in Ψ_2, return F

b.) Else return $\{(\Psi_2 / \Psi_1)\}$.

c.) Else if Ψ_2 is a variable,

a.) If Ψ_2 occurs in Ψ_1 then return F

b.) Else return $\{(\Psi_1 / \Psi_2)\}$.

d.) Else return FAILURE.

Step 2: If initial predicate symbol in Ψ_1 & Ψ_2 are not same, return FAILURE

Step 3: If Ψ_1 & Ψ_2 have diff. no. of arguments, return failure

Step 4: Set Substitution set (SUBST) to NIL

Step 5: for i=1 to no. of elements in Ψ_1.

a) call Unify function with ith element of Ψ_1 and corresponding Ψ_2 and put in S

b.) if S = Failure, return FAILURE

c.) If S ≠ NIL then do,

a) Apply S to remainder of both L1 L2

b) SUBST = APPEND (S, SUBST).

Step 6: Return SUBST

12/11/24

```
Perform unification on two expressions in first-order logic.

Args:
    expr1: The first expression (can be a variable, constant, or list representing a function).
    expr2: The second expression.
    substitution: The current substitution (dictionary).

Returns:
    A dictionary representing the most general unifier (MGU), or None if unification fails.
"""
if substitution is None:
    substitution = {}

# Debug: Print inputs and current substitution
print(f"Unifying {expr1} and {expr2} with substitution {substitution}")

# Apply existing substitutions to both expressions
expr1 = apply_substitution(expr1, substitution)
expr2 = apply_substitution(expr2, substitution)

# Debug: Print expressions after applying substitution
print(f"After substitution: {expr1} and {expr2}")
```

```
# Case 1: If expressions are identical, no substitution is needed
if expr1 == expr2:
    return substitution

# Case 2: If expr1 is a variable
if is_variable(expr1):
    return unify_variable(expr1, expr2, substitution)

# Case 3: If expr2 is a variable
if is_variable(expr2):
    return unify_variable(expr2, expr1, substitution)

# Case 4: If both are compound expressions (e.g., functions or predicates)
if is_compound(expr1) and is_compound(expr2):
    if expr1[0] != expr2[0] or len(expr1) != len(expr2):
        print(f"Failure: Predicate names or arity mismatch {expr1[0]} != {expr2[0]}")
        return None  # Function names or arity mismatch
    for arg1, arg2 in zip(expr1[1:], expr2[1:]):
        substitution = unify(arg1, arg2, substitution)
        if substitution is None:
            print(f"Failure: Could not unify arguments {arg1} and {arg2}")
            return None
```

```
        return substitution

    # Case 5: Otherwise, unification fails
    print(f"Failure: Could not unify {expr1} and {expr2}")
    return None

def unify_variable(var, expr, substitution):
    """
    Handles the unification of a variable with an expression.

    Args:
        var: The variable.
        expr: The expression to unify with.
        substitution: The current substitution.

    Returns:
        The updated substitution, or None if unification fails.
    """
    if var in substitution:
        # Apply substitution recursively
        return unify(substitution[var], expr, substitution)
    elif occurs_check(var, expr):
        # Occurs check fails if the variable appears in the term it's being unified with
```

```
        print(f"Occurs check failed: {var} in {expr}")
        return None
    else:
        substitution[var] = expr
        print(f"Substitution added: {var} -> {expr}")
        return substitution


def occurs_check(var, expr):
    """
    Checks if a variable occurs in an expression (to prevent cyclic substitutions).


    Args:
        var: The variable to check.
        expr: The expression to check against.


    Returns:
        True if the variable occurs in the expression, otherwise False.
    """
    if var == expr:
        return True
    elif is_compound(expr):
        return any(occurs_check(var, arg) for arg in expr[1:])
    return False
```

```
def is_variable(expr):
    """Checks if the expression is a variable."""
    return isinstance(expr, str) and expr[0].islower()

def is_compound(expr):
    """Checks if the expression is compound (e.g., function or predicate)."""
    return isinstance(expr, list) and len(expr) > 0

def apply_substitution(expr, substitution):
    """
    Applies a substitution to an expression.

    Args:
        expr: The expression to apply the substitution to.
        substitution: The current substitution.

    Returns:
        The updated expression with substitutions applied.
    """
    if is_variable(expr) and expr in substitution:
        return apply_substitution(substitution[expr], substitution)
    elif is_compound(expr):
```

```
        return [apply_substitution(arg, substitution) for arg in expr]
    return expr

# Example Usage:
expr1 = ['P', 'X', 'Y']
expr2 = ['P', 'a', 'Z']
result = unify(expr1, expr2)
print("Unification Result:", result)
```

```
Unifying ['P', 'X', 'Y'] and ['P', 'a', 'Z'] with substitution {}
After substitution: ['P', 'X', 'Y'] and ['P', 'a', 'Z']
Unifying X and a with substitution {}
After substitution: X and a
Substitution added: a -> X
Unifying Y and Z with substitution {'a': 'X'}
After substitution: Y and Z
Failure: Could not unify Y and Z
Failure: Could not unify arguments Y and Z
Unification Result: None
```

Program 8

FOL

FORWARD CHAINING :

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of first-order definite clauses α, the query, an atomic sentence

repeat until until new is empty
new ← { }
for each rule in KB do
(p1 ∧ ... ∧ pn => q) ← STANDARDIZE VARIABLES(rule)
foreach θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p1' ∧ ... ∧ pn') for some p1', ..., pn' in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentences already in KB or new then
add q' to new
φ unify (q', α)
if φ is not fail return
add new to KB
return false

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

```
Class Forward_reasoninig:
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts
```

```
    def infer(self):
        applied_rules = True

        while applied_rules:
            applied_rules = False
            for rule in self.rules:
                condition, result = rule
                if condition.issubset(self.facts) and result not in self.facts:
                    self.facts.add(result)
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")
        return self.facts


# Define rules as (condition, result) where condition is a set
rules = [
    ({"A"}, "B"),
    ({"B"}, "C"),
    ({"C", "D"}, "E"),
    ({"E"}, "F")
]

# Define initial facts
facts = {"A", "D"}

# Initialize and run forward reasoning
reasoner = ForwardReasoning(rules, facts)
final_facts = reasoner.infer()

print("\nFinal facts:")
print(final_facts)
```

```
Applied rule: {'A'} -> B
Applied rule: {'B'} -> C
Applied rule: {'C', 'D'} -> E
Applied rule: {'E'} -> F

Final facts:
{'C', 'E', 'B', 'F', 'A', 'D'}
```

Program 9

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

KB using PL and proving query using Resolution

Procedure Res (KB, Query)
Negate (Query)
convert KB → CNF
Add (¬Q) to set of clause

Initialise set of clauses:
clauses = KB ∪ {¬Q}

while true:
select 2 clauses from set
Not found , move to next pair

If ¬Literals found :
resolve two clauses

if clause are empty
return True:

If clauses ≠ { }
return False

```
# Define the knowledge base (KB) as a set of facts KB =
set()
    # Premises based on the provided FOL problem
    KB.add('American(Robert)')
    KB.add('Enemy(America, A)')
    KB.add('Missile(T1)')
    KB.add('Owns(A, T1)')
    # Define inference rules
    def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion
    """
    if fact1 in KB and fact2 in KB:
    KB.add(conclusion)
    print(f"Inferred: {conclusion}")
    def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
    KB.add('Weapon(T1)')
    print(f"Inferred: Weapon(T1)")
    1
    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
    KB.add('Sells(Robert, T1, A)')
    print(f"Inferred: Sells(Robert, T1, A)")
    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
    KB.add('Hostile(A)')
    print(f"Inferred: Hostile(A)")
    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and
    'Hostile(A)' in KB:
```

```
KB.add('Criminal(Robert)')
print("Inferred: Criminal(Robert)")
# Check if we've reached our goal
if 'Criminal(Robert)' in KB:
print("Robert is a criminal!")
else:
print("No more inferences can be made.")
# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

Program 10

Implement Alpha-Beta Pruning.

Alpha-beta pruning

Alphabeta (node, depth, α, β, max):

if depth = 0 or terminal node:
    return h(n)

If max:
    initialize maxval to $-\infty$
    for each child in node:
        Evaluate Alpha(child, depth-1, α, β, false)
        maxEval = max(EvalChild, maxEval)
        if $\alpha \geq \beta$; break (prune)
    return maxEval

else (min player)
    eval $\rightarrow +\infty$
    for each child in tree
        evalute AlphaBeta (child, depth, α, β, true)
        update β to min of β & minEval
        if $\beta \leq \alpha$, break
    return minEval

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(node, alpha, beta, maximizing_player):
# Base case: If it's a leaf node, return its value (simulating evaluation of the node)
if type(node) is int:
return node
# If not a leaf node, explore the children
if maximizing_player:
max_eval = -float('inf')
for child in node: # Iterate over children of the maximizer node
eval = alpha_beta_pruning(child, alpha, beta, False)
max_eval = max(max_eval, eval)
alpha = max(alpha, eval) # Maximize alpha
if beta <= alpha: # Prune the branch
break
return max_eval
else:
min_eval = float('inf')
for child in node: # Iterate over children of the minimizer node
eval = alpha_beta_pruning(child, alpha, beta, True)
min_eval = min(min_eval, eval)
beta = min(beta, eval) # Minimize beta
if beta <= alpha: # Prune the branch
1
break
return min_eval
# Function to build the tree from a list of numbers
def build_tree(numbers):
# We need to build a tree with alternating levels of maximizers and minimizers
# Start from the leaf nodes and work up
current_level = [[n] for n in numbers]
while len(current_level) > 1:
next_level = []
for i in range(0, len(current_level), 2):
if i + 1 < len(current_level):
next_level.append(current_level[i] + current_level[i + 1]) # Combine two nodes
else:
```

```
next_level.append(current_level[i]) # Odd number of elements, just carry forward
current_level = next_level
return current_level[0] # Return the root node, which is a maximizer
# Main function to run alpha-beta pruning
def main():
# Input: User provides a list of numbers
numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))
2
# Build the tree with the given numbers
tree = build_tree(numbers)
# Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
alpha = -float('inf')
beta = float('inf')
maximizing_player = True # The root node is a maximizing player
# Perform alpha-beta pruning and get the final result
result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)
print("Final Result of Alpha-Beta Pruning:", result)
if _name___== "_main_":
main()
```

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```