VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Rushil Magazine (1BM22CS225)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Rushil Magazine (1BM22CS225),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

Program 1

Tic-Tac-Toe:

Algorithm:

24/9/24

1) Construct a 3x3 Matrix

2) Every player plays alternatively until one player gets a horizontal, diagonal or vertical filled with the same symbol.

3) Once a player gets 3 in a sequence they win, the computer must find the most optimal place to put the symbol to block the sequence

4) Once a player gets a sequence they win otherwise if all the places are occupied then it results in a tie.

if the A.I wins the toss it starts
row = random.randint (1,3)
col = random.randint (1,3)
placing in the random place.
now subtract [row-1][col-1]. this will show the position occupied

AI moves in priorities:
1) To check if it can win
2) To check if player can win and block.

Code:

```
import random

def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]
    return None

def is_board_full(board):
    return all(cell != " " for row in board for cell in row)

def ai_move(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                if check_winner(board) == "O":
                    return
                board[i][j] = " "

    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "X"
                if check_winner(board) == "X":
                    board[i][j] = "O"
                    return
                board[i][j] = " "

    if board[1][1] == " ":
        board[1][1] = "O"
        return

    corners = [(0, 0), (0, 2), (2, 0), (2, 2)]
```

```
    random.shuffle(corners)
    for corner in corners:
        if board[corner[0]][corner[1]] == " ":
            board[corner[0]][corner[1]] = "O"
            return

    sides = [(0, 1), (1, 0), (1, 2), (2, 1)]
    random.shuffle(sides)
    for side in sides:
        if board[side[0]][side[1]] == " ":
            board[side[0]][side[1]] = "O"
            return

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic Tac Toe!")
    print_board(board)

    while True:
        # Player move
        while True:
            try:
                row = int(input("Enter row (1-3): ")) - 1
                col = int(input("Enter column (1-3): ")) - 1
                if board[row][col] == " ":
                    board[row][col] = "X"
                    break
                else:
                    print("Cell already taken, choose another.")
            except (ValueError, IndexError):
                print("Invalid input. Please enter numbers between 1 and 3.")

        print_board(board)

        if check_winner(board) == "X":
            print("You win!")
            break
        if is_board_full(board):
            print("It's a draw!")
            break

        # AI move
        print("AI's turn...")
        ai_move(board)
        print_board(board)

        if check_winner(board) == "O":
```

```
            print("AI wins!")
            break
        if is_board_full(board):
            print("It's a draw!")
            break

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

Output:

```
---------
Enter row (1-3): 1
Enter column (1-3): 2
X | X |
---------
  | O |
---------
  |   |
---------
AI's turn...
X | X | O
---------
  | O |
---------
  |   |
---------
Enter row (1-3): 1
Enter column (1-3): 2
Cell already taken, choose another.
Enter row (1-3): 3
Enter column (1-3): 1
X | X | O
---------
  | O |
---------
X |   |
---------
AI's turn...
X | X | O
---------
O | O |
---------
X |   |
---------
Enter row (1-3): 3
Enter column (1-3): 2
X | X | O
---------
O | O |
---------
X | X |
---------
AI's turn...
X | X | O
---------
O | O | O
---------
X | X |
---------
AI wins!
```

LAB 2
Vacuum Cleaner Agent:

Algorithm:

1) Automatic Vacuum Cleaner :

1) ~~It~~ The vaccum cleaner visits all the rooms and cleans them.

2) If the room is already clean then it goes to the next room.

3) Each room can either be dirty or clean.

4) It starts with an initial room and inspects the room
- If it is ~~clean~~ not clean it should clean the room otherwise go the other room.

After both the rooms are clean it can exit.

```
if left_room is dirty :
      clean left_room
Move to right_room;
If right_room is dirty:
      Clean right_room.
If both rooms are clean
  TERMINATE.

- a = input ("Room A state")
- b = input ("Room B state)
for i in room_list:
      IF (i.state = "dirty"):
            i.clean().
```

```
def clean (self):
    self.state = "clean".
```

```
room_list = [ ]
```

Proceed

Code:

```
agent_table = {
    ('Clean', 'A'): 'MoveRight',
    ('Clean', 'B'): 'MoveLeft',
    ('Dirty', 'A'): 'Suck',
    ('Dirty', 'B'): 'Suck',
}

class vacuumcleaner:
    def __init__(self, status_a='Clean', status_b='Clean', location='A'):
        self.location = location
        self.status = {'A': status_a, 'B': status_b}

    def percept(self):
        return self.status[self.location]

    def act(self, action):
        if action == 'MoveRight':
            self.location = 'B'
        elif action == 'MoveLeft':
            self.location = 'A'
        elif action == 'Suck':
            self.status[self.location] = 'Clean'

def table_driven_agent(percept):
    return agent_table.get(percept, 'NoOp')

if __name__ == "__main__":
    status_a = input("Is room A Clean or Dirty? ").strip().capitalize()
    status_b = input("Is room B Clean or Dirty? ").strip().capitalize()
    vacuum = vacuumcleaner(status_a=status_a, status_b=status_b)

    for _ in range(3):
        current_percept = vacuum.percept()
        action = table_driven_agent((current_percept, vacuum.location))
        print(f"Percept: {current_percept}, Action: {action}")

        if action != 'NoOp':
            vacuum.act(action)

        print(f"Location: {vacuum.location}, Status: {vacuum.status}\n")
```

Output:

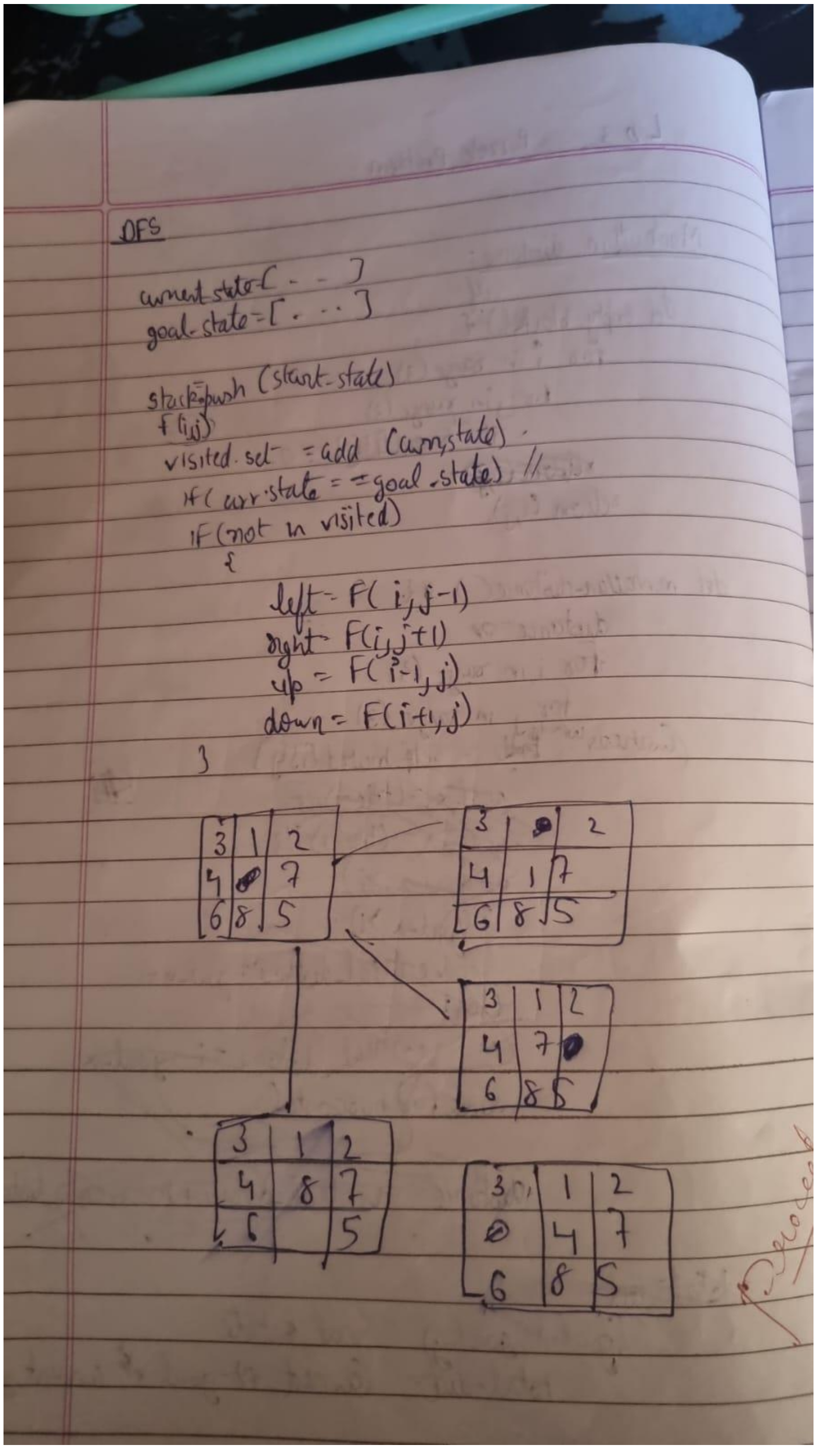
```
Is room A 'Clean' or 'Dirty'? Dirty
Is room B 'Clean' or 'Dirty'? Dirty
Percept: Dirty, Action: Suck
Location: A, Status: {'A': 'Clean', 'B': 'Dirty'}

Percept: Clean, Action: MoveRight
Location: B, Status: {'A': 'Clean', 'B': 'Dirty'}

Percept: Dirty, Action: Suck
Location: B, Status: {'A': 'Clean', 'B': 'Clean'}
```

```
Is room A 'Clean' or 'Dirty'? dirty
Is room B 'Clean' or 'Dirty'? dirty
Is room C 'Clean' or 'Dirty'? dirty
Is room D 'Clean' or 'Dirty'? dirty
Percept: Dirty, Action: Suck
Location: A, Status: {'A': 'Clean', 'B': 'Dirty', 'C': 'Dirty', 'D': 'Dirty'}

Percept: Clean, Action: MoveRight
Location: B, Status: {'A': 'Clean', 'B': 'Dirty', 'C': 'Dirty', 'D': 'Dirty'}

Percept: Dirty, Action: Suck
Location: B, Status: {'A': 'Clean', 'B': 'Clean', 'C': 'Dirty', 'D': 'Dirty'}

Percept: Clean, Action: MoveRight
Location: C, Status: {'A': 'Clean', 'B': 'Clean', 'C': 'Dirty', 'D': 'Dirty'}

Percept: Dirty, Action: Suck
Location: C, Status: {'A': 'Clean', 'B': 'Clean', 'C': 'Clean', 'D': 'Dirty'}

Percept: Clean, Action: MoveRight
Location: D, Status: {'A': 'Clean', 'B': 'Clean', 'C': 'Clean', 'D': 'Dirty'}

Percept: Dirty, Action: Suck
Location: D, Status: {'A': 'Clean', 'B': 'Clean', 'C': 'Clean', 'D': 'Clean'}

All rooms are clean!
```

Program 3

Depth First Search

Algorithm:



Code:

```
import heapq

def manhattan(puzzle, goal):
    dist = 0
    for i in range(9):
        if puzzle[i] != 0:
            goal_idx = goal.index(puzzle[i])
            dist += abs(i // 3 - goal_idx // 3) + abs(i % 3 - goal_idx % 3)
    return dist

def a_star_manhattan(puzzle, goal):
    # Priority queue for A* search, stores tuples of (cost, puzzle_state, path)
    pq = [(manhattan(puzzle, goal), puzzle, [puzzle])]
    visited = set()

    while pq:
        cost, current, path = heapq.heappop(pq)

        if current == goal:
            return path

        visited.add(tuple(current))
        idx = current.index(0)
        # Define possible moves for the blank space
        moves = [(1, 3), (-1, 3), (3, 1), (-3, 1)]

        for move, cond in moves:
            new_idx = idx + move
            if 0 <= new_idx < 9 and (new_idx // 3 == idx // 3 or new_idx % 3 == idx % 3):
                new_puzzle = current[:]
                new_puzzle[idx], new_puzzle[new_idx] = new_puzzle[new_idx], new_puzzle[idx]

                if tuple(new_puzzle) not in visited:
                    heapq.heappush(pq, (cost + manhattan(new_puzzle, goal), new_puzzle, path + 
[new_puzzle]))

    return None

def prettify_step(step, index):
    print(f"Step {index}:")
    for i in range(0, 9, 3):
        print(f"{step[i]} {step[i+1]} {step[i+2]}")
    print("-" * 8)
```

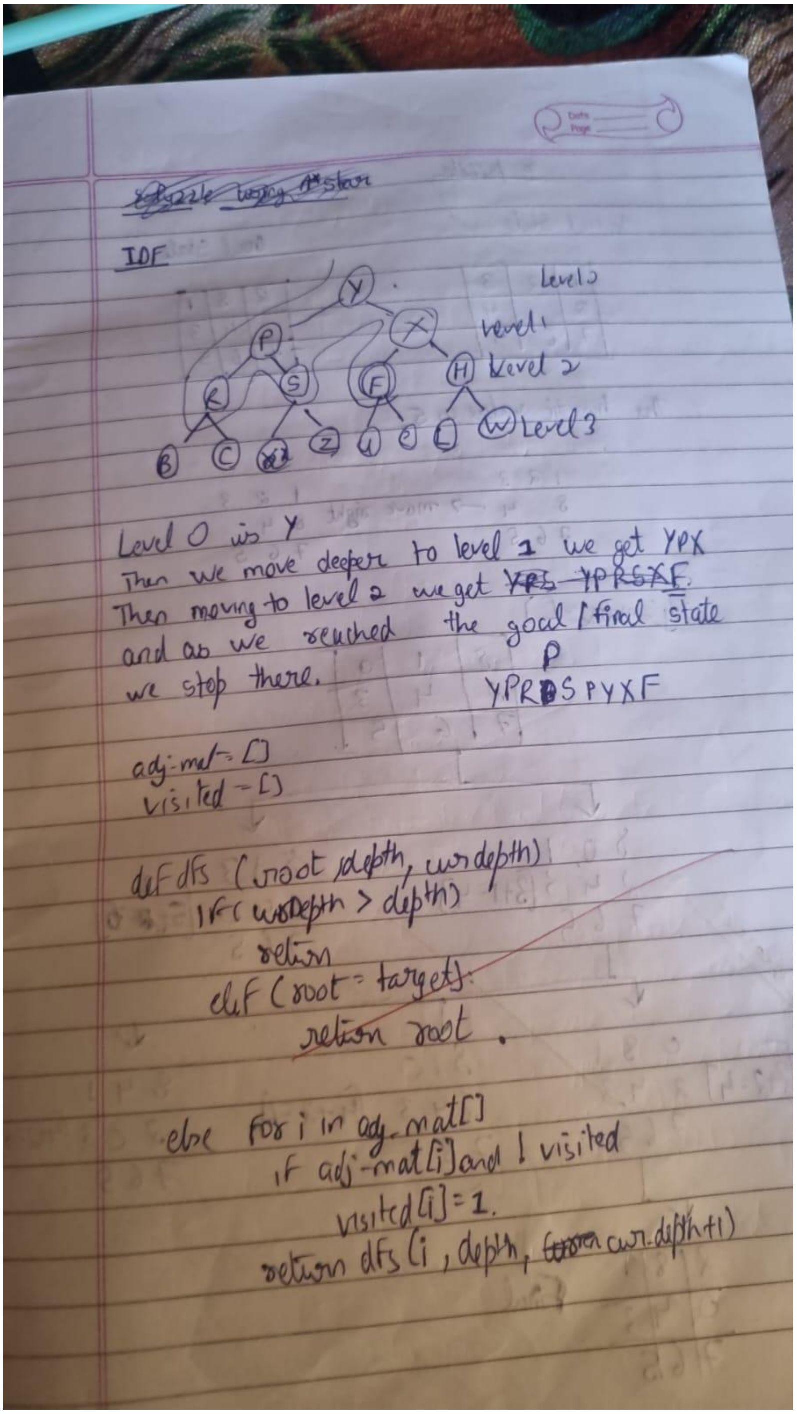
```
start = [1, 2, 3, 4, 0, 5, 6, 7, 8]
goal = [0, 1, 2, 3, 4, 5, 6, 7, 8]

# Run A* search
result = a_star_manhattan(start, goal)

# Print the solution steps, but limit to every 2nd step for brevity
if result:
    for index, step in enumerate(result):
        if index % 2 == 0:  # Print only every 2nd step
            prettify_step(step, index)
    print(f"Total moves: {len(result)}")
else:
    print("No solution found.")
```

Output:

```
Step 0:
1 2 3
4 0 5
6 7 8
--------
Step 2:
0 2 3
1 4 5
6 7 8
--------
Step 4:
2 3 0
1 4 5
6 7 8
--------
Step 6:
2 3 5
1 0 4
6 7 8
--------
Step 8:
0 2 5
1 3 4
6 7 8
--------
Step 10:
1 2 5
3 0 4
6 7 8
--------
Step 12:
1 2 0
3 4 5
6 7 8
--------
Step 14:
0 1 2
3 4 5
6 7 8
--------
Total moves: 15
```

LAB 4
Iterative Deepening Search:

Algorithm:



~~A* star~~

IDF

Level 0
level 1
Level 2
Level 3

Y X P H F S R W B C Z

Level 0 is Y
Then we move deeper to level 1 we get YPX
Then moving to level 2 we get ~~YPS~~ ~~YPRSXF~~.
Then moving to level 2 we get the goal / final state
and as we reached
we stop there.

P
YPRBSPYXF

```
adj-mat = []
visited = []

def dfs (root, depth, curr depth)
    if (currDepth > depth)
        return
    elif (root = target):
        return root.

else for i in adj-mat[]
        if adj-mat[i] and ! visited
            visited[i] = 1.
        return dfs (i, depth, curr.depth+1)
```

Code:

```
from queue import PriorityQueue

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos
        self.moves = moves
        self.previous = previous

    def __lt__(self, other):
        return (self.moves + self.heuristic()) < (other.moves + other.heuristic())

    def heuristic(self):
        # Manhattan distance heuristic
        distance = 0
        goal_positions = {1: (0, 0), 2: (0, 1), 3: (0, 2),
                          4: (1, 0), 5: (1, 1), 6: (1, 2),
                          7: (2, 0), 8: (2, 1), 0: (2, 2)}
        for i in range(3):
            for j in range(3):
                value = self.board[i][j]
                goal_x, goal_y = goal_positions[value]
                distance += abs(goal_x - i) + abs(goal_y - j)
        return distance

    def get_neighbors(self):
        # Possible moves (up, down, left, right)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        neighbors = []
        x, y = self.zero_pos

        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                # Swap the zero with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                neighbors.append(PuzzleState(new_board, (new_x, new_y), self.moves + 1, self))

        return neighbors

def a_star(start, goal):
    start_zero_pos = next((i, j) for i in range(3) for j in range(3) if start[i][j] == 0)
    goal_flat = [value for row in goal for value in row]
```

```
    open_set = PriorityQueue()
    open_set.put(PuzzleState(start, start_zero_pos))
    visited = set()

    while not open_set.empty():
        current_state = open_set.get()
        current_board_tuple = tuple(tuple(row) for row in current_state.board)
        visited.add(current_board_tuple)

        if current_state.board == goal:
            print("Solution found in", current_state.moves, "moves.")
            path = []
            while current_state:
                path.append(current_state.board)
                current_state = current_state.previous
            for step in reversed(path):
                for row in step:
                    print(row)
                print()
            return

        for neighbor in current_state.get_neighbors():
            neighbor_board_tuple = tuple(tuple(row) for row in neighbor.board)
            if neighbor_board_tuple not in visited:
                open_set.put(neighbor)

    print("No solution found.")


def get_input_state(prompt):
    while True:
        state_input = input(prompt)
        try:
            state = list(map(int, state_input.split()))
            if len(state) == 9 and all(x in range(9) for x in state):
                return [state[i:i+3] for i in range(0, 9, 3)]
            else:
                print("Invalid input. Please enter 9 numbers (0-8).")
        except ValueError:
            print("Invalid input. Please enter numbers only.")

def main():
    print("Enter the start state (9 numbers, use 0 for the blank space):")
    start_state = get_input_state("Start state: ")

    print("Enter the goal state (9 numbers, use 0 for the blank space):")
```

```
    goal_state = get_input_state("Goal state: ")

    a_star(start_state, goal_state)

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

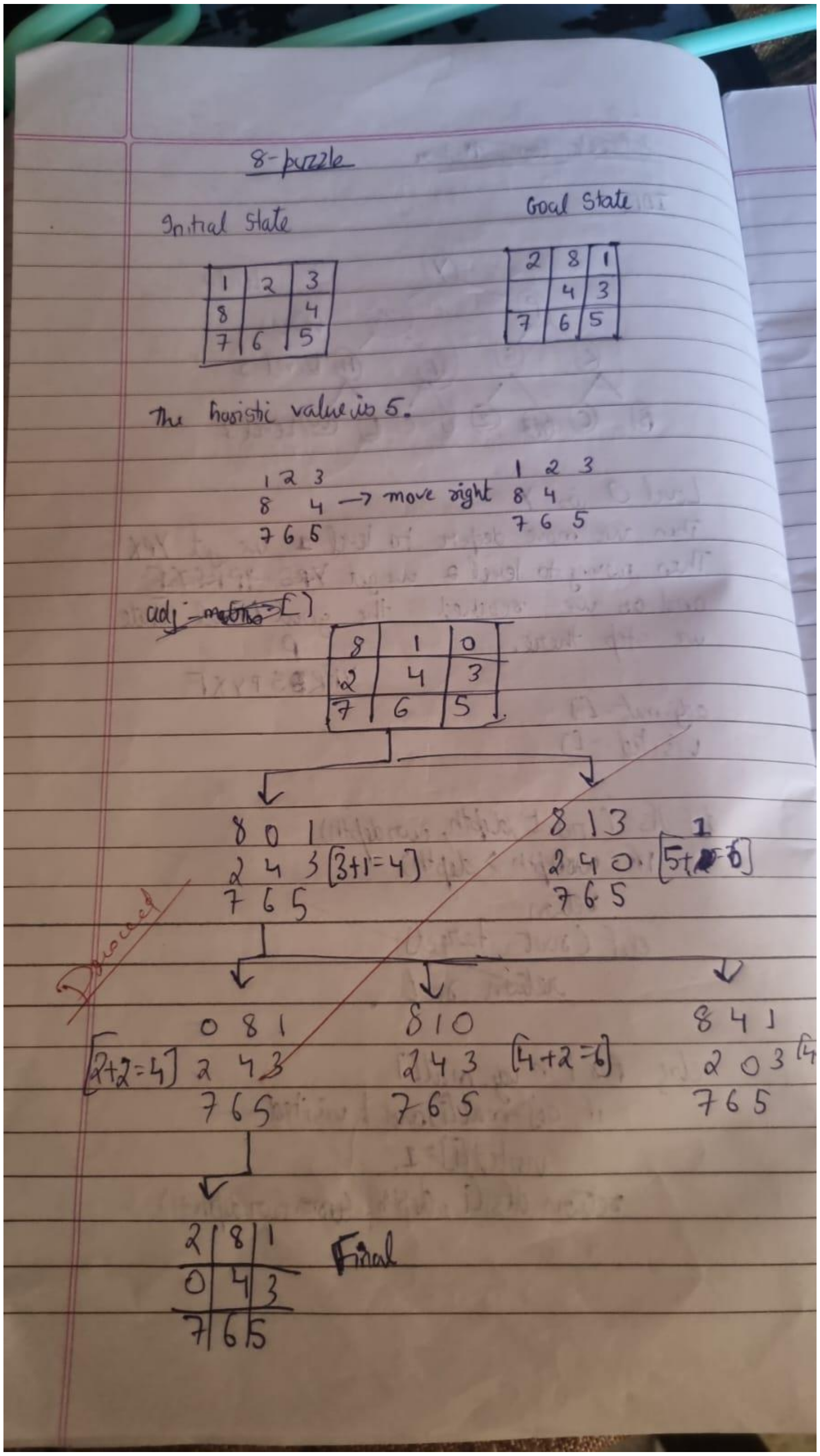
Output:

```
Enter the start state (9 numbers, use 0 for the blank space):
Start state: 1 2 3 8 0 4 7 6 5
Enter the goal state (9 numbers, use 0 for the blank space):
Goal state: 2 8 1 0 4 3 7 6 5
Solution found in 9 moves.
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

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

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

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

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

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

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

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

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

Program 2

A* using Manhattan Distance:

Algorithm:



Code:

```
from queue import PriorityQueue

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos
        self.moves = moves
        self.previous = previous

    def __lt__(self, other):
        return (self.moves + self.heuristic()) < (other.moves + other.heuristic())

    def heuristic(self):
        # Manhattan distance heuristic
        distance = 0
        goal_positions = {1: (0, 0), 2: (0, 1), 3: (0, 2),
                          4: (1, 0), 5: (1, 1), 6: (1, 2),
                          7: (2, 0), 8: (2, 1), 0: (2, 2)}
        for i in range(3):
            for j in range(3):
                value = self.board[i][j]
                goal_x, goal_y = goal_positions[value]
                distance += abs(goal_x - i) + abs(goal_y - j)
        return distance

    def get_neighbors(self):
        # Possible moves (up, down, left, right)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        neighbors = []
        x, y = self.zero_pos

        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                # Swap the zero with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                neighbors.append(PuzzleState(new_board, (new_x, new_y), self.moves + 1, self))

        return neighbors

def a_star(start, goal):
    start_zero_pos = next((i, j) for i in range(3) for j in range(3) if start[i][j] == 0)
    goal_flat = [value for row in goal for value in row]
```

```
    open_set = PriorityQueue()
    open_set.put(PuzzleState(start, start_zero_pos))
    visited = set()

    while not open_set.empty():
        current_state = open_set.get()
        current_board_tuple = tuple(tuple(row) for row in current_state.board)
        visited.add(current_board_tuple)

        if current_state.board == goal:
            print("Solution found in", current_state.moves, "moves.")
            path = []
            while current_state:
                path.append(current_state.board)
                current_state = current_state.previous
            for step in reversed(path):
                for row in step:
                    print(row)
                print()
            return

        for neighbor in current_state.get_neighbors():
            neighbor_board_tuple = tuple(tuple(row) for row in neighbor.board)
            if neighbor_board_tuple not in visited:
                open_set.put(neighbor)

    print("No solution found.")


def get_input_state(prompt):
    while True:
        state_input = input(prompt)
        try:
            state = list(map(int, state_input.split()))
            if len(state) == 9 and all(x in range(9) for x in state):
                return [state[i:i+3] for i in range(0, 9, 3)]
            else:
                print("Invalid input. Please enter 9 numbers (0-8).")
        except ValueError:
            print("Invalid input. Please enter numbers only.")

def main():
    print("Enter the start state (9 numbers, use 0 for the blank space):")
    start_state = get_input_state("Start state: ")

    print("Enter the goal state (9 numbers, use 0 for the blank space):")
```

```
    goal_state = get_input_state("Goal state: ")

    a_star(start_state, goal_state)

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

Output:

```
Enter the start state (9 numbers, use 0 for the blank space):
Start state: 1 2 3 8 0 4 7 6 5
Enter the goal state (9 numbers, use 0 for the blank space):
Goal state: 2 8 1 0 4 3 7 6 5
Solution found in 9 moves.
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

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

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

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

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

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

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

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

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

Program 4

Hill Climbing Search to Solve N-Queens Problem:

Algorithm:

Lab-06

~~A* search algorithm~~ hill climbing algorithm for 8 queens

Step 1: generate board.
Generate a random board

Step 2: Calculate conflicts
Count how many pairs of queens are attacking each other

Step 3: Calculate the best position for each queen to reduce the no: of conflicts.

Step 4: Then use hill_climbing to repeatedly get the best position for the queens ~~or~~ to keep reducing conflicts until no improvements can be made.

```
def hill_climbing(){
    generate board
    board = generate_board()
    current_conflicts = calculate_conflicts(board)
    while current_conflicts >0
        new_board, new_conflicts = get_best_move(board
        if new_conflicts >= current_conflicts :
            return False
        board = new_board
        current_conflicts = new_conflicts.
```

Code:

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

def new(s):
    best=s
    for i in range(len(s)):
        for j in range(1,9):
            if j!=s[i]:
                n=s[:i]+[j]+s[i+1:]
                if h(n)<h(best):
                    best=n
    return best

def hc():
    curr=[random.randint(1,8) for i in range(8)]
    while True:
        ch=h(curr)
        curr=new(curr)
        if h(curr)==0:
            return curr
        if h(curr)>=ch:
            curr=[random.randint(1,8) for i in range(8)]

def print_board(solution):
    print("Solution for 8 Queens Hill climbing is: ",solution)
    if solution is None:
        print("No solution found.")
        return

    board = [['.' for _ in range(8)] for _ in range(8)]

    for row in range(len(solution)):
        col = solution[row] - 1
        board[row][col] = 'Q'

    for row in board:
        print(' '.join(row))
```

```
print(print_board(hc()))
```

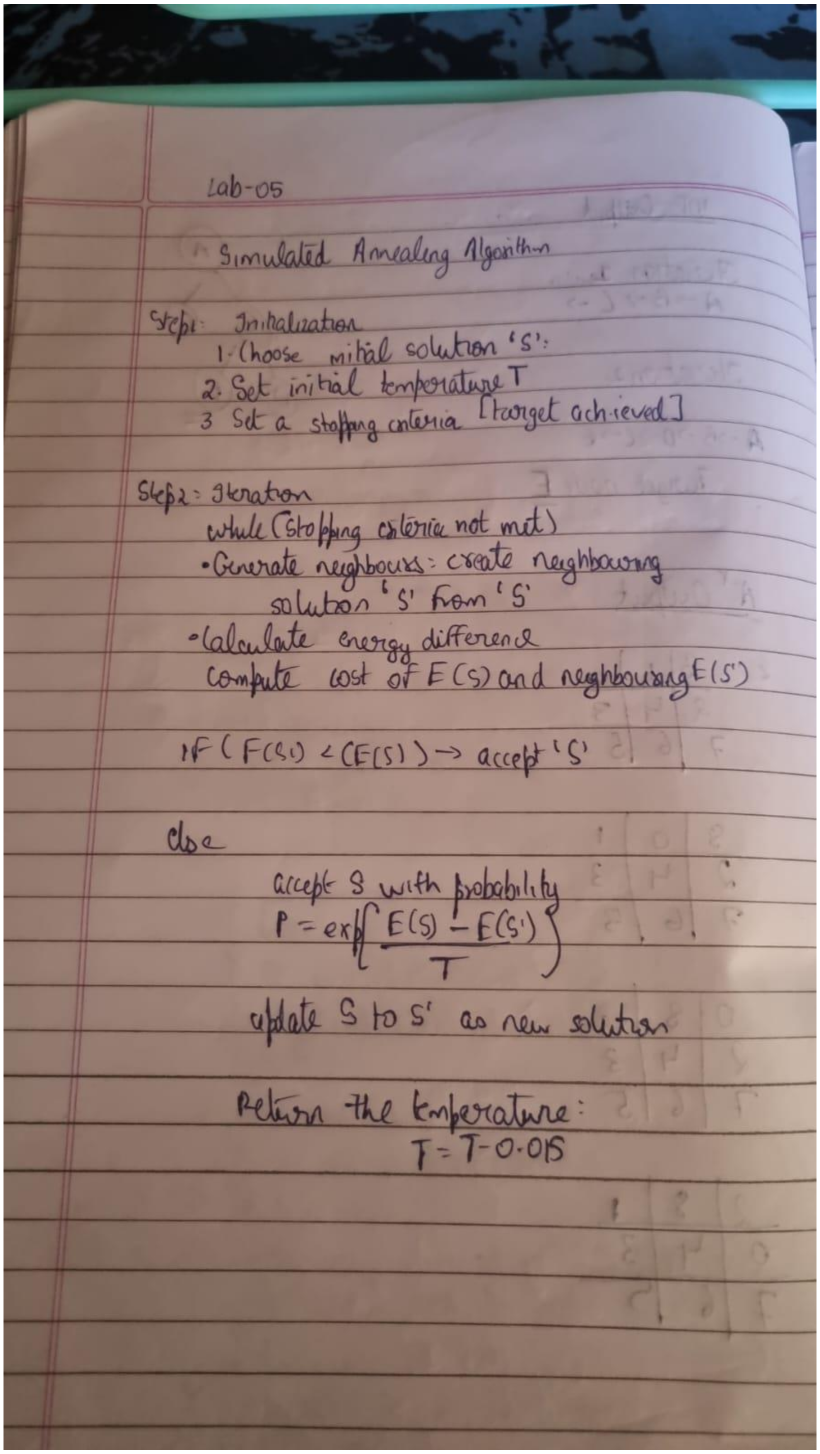
Output:

```
Solution for 8 Queens Hill climbing is:  [5, 7, 1, 4, 2, 8, 6, 3]
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
None
```

Program 5

Simulated Annealing to Solve 8-Queens problem:

Algorithm:



Lab-05

Simulated Annealing Algorithm

Step1: Initialization
1. Choose initial solution 'S':
2. Set initial temperature T
3 Set a stopping criteria [target achieved]

Step2: Iteration
while (stopping criteria not met)
• Generate neighbours: create neighbouring solution 'S' from 'S'
• Calculate energy difference
compute cost of E(S) and neighbouring E(S')

if (F(S') < (E(S))) → accept 'S'

else

accept S with probability
$$P = \exp\left\{\frac{E(S) - E(S')}{T}\right\}$$

update S to S' as new solution

Return the temperature:
T = T-0.015

Code:

```
import random
import math

def simanl(ini, initemp, cr, it):
    # Initialize the current state, best state, and best cost
    curr = ini
    bstate = curr
    bcost = obj(curr)
    temp = initemp

    # Continue until the temperature is above 1
    while temp > 1:
        # Perform iterations at the current temperature
        for i in range(it):
            nst = neighbour(curr)
            currcost = obj(curr)
            ncost = obj(nst)

            # Decide whether to accept the neighbor based on the acceptance probability
            if ap(currcost, ncost, temp) > random.random():
                curr = nst

            # Update the best state and best cost if the new cost is better
            if ncost < bcost:
                bstate = nst
                bcost = ncost

        # Cool down the temperature
        temp *= cr

    return bstate, bcost

def obj(state):
    # Objective function: Calculate the cost (sum of squares)
    cost = 0
    for ele in state:
        cost += ele**2
    return cost

def neighbour(state):
    # Generate a neighbor state by slightly modifying one element
    nstate = state.copy()
    ind = random.randint(0, len(state) - 1)
    nstate[ind] += random.uniform(-1, 1)
    return nstate
```

```
def ap(curr, ncost, temp):
    # Acceptance probability function
    if(ncost < curr):
        return 1
    return math.exp((curr - ncost) / temp)

# main function
print(simanl([1, 2, 3, 4, 5], 1000, 0.99, 100))
```

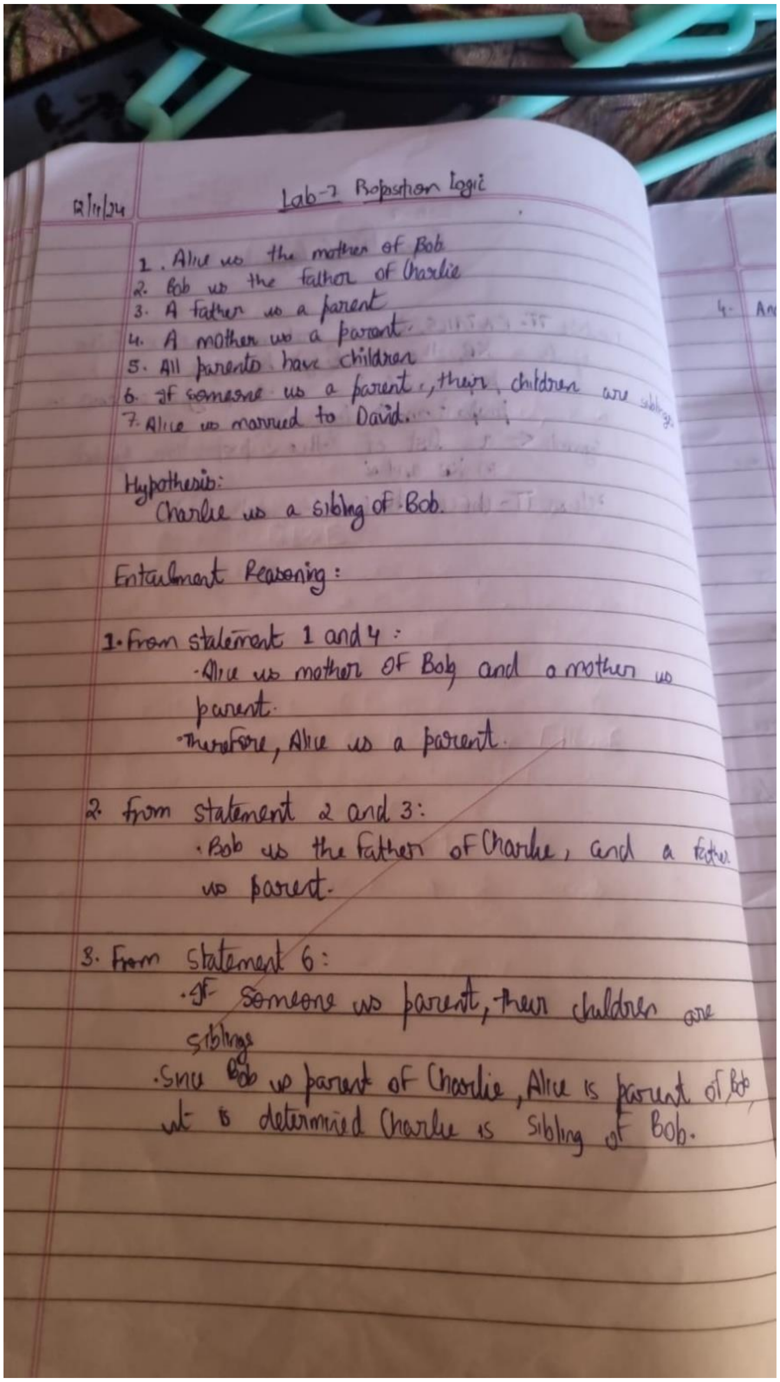
Output:

```
Output:-

it: 1 , temp : 950.000, cursol: 9.44654 , best sol : 9.449654
it: 2 , temp : 902.500, cursol: 8.937 , bestsol: 8.8378
it: 3 , temp :- 857.375 , cursol: 9.487 , bestsol: 8.837
it: 4 , temp: 773.38 , cursol: 8.056 , bestsol: 8.0567
it: 5 , temp: 735.891 , cursol: 7.675 , bestsol: 7.695
it: 6 , temp: 694.756 , cursol: 7.6 , bestsol: 7.6
it: 7 , temp: 605.21 , cursol: 7.7 , bestsol: 7.6
it: 8 , temp: 638.77 , cursol: 8.4 , bestsol: 7.6
it: 9 , temp: 630.57 , cursol: 8.4 , bestsol: 7.656
it: 10 , temp: 598.66 , cursol: 8.7 , bestsol: 7.656
```

Program 6

Knowledge Base using Propositional Logic:

Algorithm:



2/11/24 Lab-7 Proposition logic

1. Alice is the mother of Bob
2. Bob is the father of Charlie
3. A father is a parent
4. A mother is a parent.
5. All parents have children
6. If someone is a parent, their children are siblings.
7. Alice is married to David.

Hypothesis:
Charlie is a sibling of Bob.

Entailment Reasoning:

1. From statement 1 and 4:
   - Alice is mother of Bob and a mother is parent.
   - Therefore, Alice is a parent.
2. From statement 2 and 3:
   - Bob is the father of Charlie, and a father is parent.
3. From statement 6:
   - If someone is parent, their children are siblings
   - Since Bob is parent of Charlie, Alice is parent of Bob, it is determined Charlie is sibling of Bob.

4. Analysis using statement 6:
- Statement 6 implies that children of a parent are siblings
- ~~They cannot be siblings~~

Conclusion:
The hypothesis is entailed.

$(F(X) \rightarrow P(X)$ (If X is a Father the X is a parent)

A mother is a parent.
$\forall x\,(M(X) \rightarrow P(X))$ (If X is a mother then X is a parent)

All parents have children
$\forall x\,(P(x) \rightarrow \exists y\, \text{Child}(x,y))$ (If x is a parent there exists some child y of x)

Output:
1) Alice is mother of Bob: True
2) Bob is Father of Charlie: True
3) A Father is parent: True
4) A mother is a parent: True
5) All parents have children: True
6) If someone is a parent, their children are siblings
7) Alice is married to David: False

Conclusion: Charlie is sibling of Bob: False

Code:

```
import itertools


def evaluate_formula(formula, valuation):

    formula = formula.replace('p', str(valuation['p']))
    formula = formula.replace('q', str(valuation['q']))

    return eval(formula)


def extract_variables(formula):
    variables = set()
    for char in formula:
        if char.isalpha():
            variables.add(char)
    return list(variables)


def generate_truth_table(KB, query):

    variables = extract_variables(KB) + extract_variables(query)
    variables = list(set(variables))

    print("Truth Table:")
    print(" | ".join(variables + ["KB", "Query"]))
    print("-" * (len(variables) * 4 + 12))


    entails_query = True


    for assignment in itertools.product([False, True], repeat=len(variables)):
        valuation = dict(zip(variables, assignment))

        KB_truth = evaluate_formula(KB, valuation)
        query_truth = evaluate_formula(query, valuation)


        row = [str('T' if valuation[var] else 'F') for var in variables]
        row.append(str('T' if KB_truth else 'F'))
        row.append(str('T' if query_truth else 'F'))
        print(" | ".join(row))


        if KB_truth and not query_truth:
```

```
            entails_query = False

    print("\nKB entails query:", entails_query)


KB = input("Enter the knowledge base (e.g., 'p and (p != q)'): ")
query = input("Enter the query (e.g., 'q'): ")


generate_truth_table(KB, query)
```

Output:

```
KB entails R
```

```
KB does not entail R
```

Program 7

Unification in First Order Logic:

Algorithm:

Expression A : Loves (x,y)
" B : Loves (Alice, Bob)

~~Step 1~~ :

Step 1: Both have same predicate (Loves), so we can proceed.

Step 2: First argument : x in Expression A is a variable and Alice in B is a constant so, x = Alice

Step 3: Do the same for the 2nd argument => y = Bob

Step 4: Replace x with Alice and y with Bob in Expression A

Step 5: ~~After substitution~~ After substitution, Expression A = Loves (Alice, Bob)

Step 6: Now both the expressions are identical.

$\forall x \, Male(x) \Leftrightarrow \neg Female(x)$

Output

Substitutions: {'x': 'John', 'y': 'Mary'}

Unified Expression : ('Loves', 'John', 'Mary')

Code:

```
import re

# Predicates for translation
predicates = {
    "is a human": "H",  # e.g., John is a human
    "is mortal": "M",  # e.g., John is mortal
    "loves": "L",       # e.g., John loves Mary
    "is a dog": "D",    # e.g., John is a dog
    "is an animal": "A", # e.g., John is an animal
    "is brown": "B",    # e.g., John is brown
    "is a person": "P",  # e.g., John is a person
    "is a teacher": "T",  # e.g., John is a teacher
    "is a student": "S",  # e.g., John is a student
    "respects": "R",     # e.g., John respects Mary
    "knows": "K",       # e.g., John knows Mary
    "likes mathematics": "Lm",  # John likes mathematics
    "likes science": "Ls",     # John likes science
    "is married to": "Ma",  # John is married to Mary
    "is a bachelor": "Bch", # John is a bachelor
    "is a parent of": "Pnt", # John is a parent of someone
    "is raining": "R",    # It is raining
    "is wet": "G",        # The ground is wet
    "is a man": "R",      # John is a man
    "is a woman": "W",    # Mary is a woman
}

# Constants: John (j) and Mary (m)
constants = {
    "John": "j",
    "Mary": "m",
    "Alice": "a",
}

# Function to handle sentence translation
def translate_to_fol(sentence):
    sentence = sentence.strip().lower()

    # Handle sentence structures
    if "is a human" in sentence:
        return translate_is_a_human(sentence)

    if "is mortal" in sentence:
        return translate_is_mortal(sentence)

    if "loves" in sentence:
```

```
        return translate_loves(sentence)

    if "every" in sentence:
        return translate_every(sentence)

    if "there exists" in sentence or "there is" in sentence:
        return translate_exists(sentence)

    if "not all" in sentence:
        return translate_not_all(sentence)

    if "if" in sentence and "then" in sentence:
        return translate_if_then(sentence)

    if "nobody" in sentence:
        return translate_nobody(sentence)

    if "and" in sentence:
        return translate_conjunction(sentence)

    return "Translation not available for this sentence structure."

# Helper functions for specific cases
def translate_is_a_human(sentence):
    match = re.match(r"([a-zA-Z]+) is a human", sentence)
    if match:
        subject = match.group(1)
        subject_const = constants.get(subject, subject)
        return f"H({subject_const})"
    return "Invalid sentence structure."

def translate_is_mortal(sentence):
    match = re.match(r"([a-zA-Z]+) is mortal", sentence)
    if match:
        subject = match.group(1)
        subject_const = constants.get(subject, subject)
        return f"M({subject_const})"
    return "Invalid sentence structure."

def translate_loves(sentence):
    match = re.match(r"([a-zA-Z]+) loves ([a-zA-Z]+)", sentence)
    if match:
        subject = match.group(1)
        object_ = match.group(2)
        subject_const = constants.get(subject, subject)
        object_const = constants.get(object_, object_)
        return f"L({subject_const}, {object_const})"
```

```
    return "Invalid sentence structure."

def translate_every(sentence):
    match = re.match(r"every ([a-zA-Z]+) is ([a-zA-Z]+)", sentence)
    if match:
        subject = match.group(1)
        predicate = match.group(2)
        return f"∀x ({subject}(x) → {predicate}(x))"
    return "Invalid sentence structure."

def translate_exists(sentence):
    match = re.match(r"there exists ([a-zA-Z]+) who ([a-zA-Z]+) ([a-zA-Z]+)", sentence)
    if match:
        subject = match.group(1)
        predicate = match.group(2)
        object_ = match.group(3)
        subject_const = constants.get(subject, subject)
        object_const = constants.get(object_, object_)
        return f"∃x ({predicate}(x, {object_const}))"
    return "Invalid sentence structure."

def translate_not_all(sentence):
    match = re.match(r"not all ([a-zA-Z]+) like both ([a-zA-Z]+) and ([a-zA-Z]+)", sentence)
    if match:
        subject = match.group(1)
        subject1 = match.group(2)
        subject2 = match.group(3)
        return f"¬∀x ({subject}(x) → ({subject1}(x) ∧ {subject2}(x)))"
    return "Invalid sentence structure."

def translate_if_then(sentence):
    match = re.match(r"if ([a-zA-Z]+) is ([a-zA-Z]+), then ([a-zA-Z]+) teaches mathematics",
sentence)
    if match:
        subject = match.group(1)
        subject_const = constants.get(subject, subject)
        return f"{subject_const}(x) → Teaches(x, Mathematics)"
    return "Invalid sentence structure."

def translate_conjunction(sentence):
    match = re.match(r"([a-zA-Z]+) and ([a-zA-Z]+) are both students", sentence)
    if match:
        subject1 = match.group(1)
        subject2 = match.group(2)
        subject1_const = constants.get(subject1, subject1)
        subject2_const = constants.get(subject2, subject2)
        return f"S({subject1_const}) ∧ S({subject2_const})"
```

```
    return "Invalid sentence structure."

# Function to handle "nobody" sentences
def translate_nobody(sentence):
    match = re.match(r"nobody is ([a-zA-Z]+) than themselves", sentence)
    if match:
        predicate = match.group(1)  # For example: "taller"
        return f"¬∃x ({predicate}(x, x))"  # This means "Nobody is taller than themselves"
    return "Invalid sentence structure."

# Main loop to interact with the user
def main():
    print("Enter a sentence like:")
    print("1. John is a human.")
    print("2. Every human is mortal.")
    print("3. John loves Mary.")
    print("4. There exists someone who loves Mary.")
    print("Type 'exit' to quit.")

    while True:
        sentence = input("\nEnter a sentence: ").strip()

        if sentence.lower() == 'exit':
            print("Goodbye!")
            break

        # Translate the sentence into FOL
        fol_translation = translate_to_fol(sentence)
        print("First-Order Logic Translation:", fol_translation)

# Run the program
if _name___== "_main_":
    main()
```

Output:

```
Enter a sentence like:
1. John is a human.
2. Every human is mortal.
3. John loves Mary.
4. There exists someone who loves Mary.
Type 'exit' to quit.

Enter a sentence: John is a human.
First-Order Logic Translation: H(john)

Enter a sentence: Mary is a human.
First-Order Logic Translation: H(mary)

Enter a sentence: John loves Mary.
First-Order Logic Translation: L(john, mary)
```

Program 9

Knowledge Base consisting of First Order Logic Statements and proof using Forward Reasoning:

Algorithm:

3/12/24

Forward Chaining

As per the law, it is a crime for an American to sell hostile weapons to hostile nations.
Prove that 'Robert is a criminal'.

Lets say p, q and r are variables

American (p) ∧ Weapon(q) ∧ Sells (p,q,r) ∧ Hostile(r) => Criminal (p)

Country A has some missiles.

∃ x Owns (A,x) ∧ Missile(x)

Owns (A,T1)
Missile(T1)

All of the missiles were sold to country A by Robert
∀ x Missile(x) ∧ Owns (A,x) => Sells (Robert, x, A)

Missiles are weapons
Missile(x) => Weapons (x)

Enemy of America is known as hostile
∀ x Enemy (x, America) => Hostile(x)

Robert is an American
American (Robert)

The country A, an enemy of America
Enemy (A, America)

Criminal(Robert)

Code:

```
class ForwardChaining:
    def __init__(self, facts, rules):
        """
        Initialize the Forward Chaining algorithm with facts and rules.
        :param facts: Set of known facts (initial facts).
        :param rules: List of rules where each rule is a tuple (premise, conclusion).
        """
        self.facts = set(facts)
        self.rules = rules
        self.inferred_facts = set(facts)  # Set of facts derived during the process

    def apply_rule(self, rule):
        """
        Applies a rule to derive new facts from existing facts.
        :param rule: A rule represented as (premise, conclusion).
        :return: True if a new fact is derived, False otherwise.
        """
        premise, conclusion = rule
        premise_facts = set(premise.split(','))  # Split the premise into individual facts
        # Check if the premise of the rule is fully satisfied by current facts
        if premise_facts.issubset(self.facts):  # Ensure all premises are in facts
            if conclusion not in self.facts:
                print(f"Inferred new fact: {conclusion}")
                self.facts.add(conclusion)  # Add the conclusion to the set of facts
                return True
        return False

    def forward_chaining(self):
        """
        Applies forward chaining to derive new facts until no more facts can be derived.
        """
        new_inference = True
        while new_inference:
            new_inference = False
            # Go through all the rules and try to apply them
            for rule in self.rules:
                if self.apply_rule(rule):
                    new_inference = True
            if new_inference:
```

```
            print(f"Current facts: {self.facts}")
        print("Forward Chaining completed.")

    def is_goal_reached(self, goal):
        """
        Checks if the goal has been reached (i.e., if the goal is in the facts).
        :param goal: The goal fact to check for.
        :return: True if the goal is in the facts, otherwise False.
        """
        return goal in self.facts


def main():
    print("Forward Chaining System")

    # Define the initial facts
    facts = {
        "american(p)",
        "weapon(q)",
        "sells(p,q,r)",
        "hostile(r)",
        "american(robert)",
        "enemy(a, america)"
    }

    # Define the rules (premise -> conclusion)
    rules = [
        ("american(p),weapon(q),sells(p,q,r),hostile(r)", "criminal(p)"),  # Rule 1
        ("owns(a,x),missile(x)", "sells(robert,x,a)"),  # Rule 2
        ("missile(x)", "weapon(x)"),  # Rule 3
        ("enemy(a,america)", "hostile(a)")  # Rule 4
    ]

    # Create an instance of ForwardChaining with the facts and rules
    fc = ForwardChaining(facts, rules)

    # Perform forward chaining to infer new facts
    fc.forward_chaining()

    # Define the goal (fact you want to check)
    goal = "criminal(robert)"
```

```
    # Check if the goal is reached
    if fc.is_goal_reached(goal):
        print(f"The goal '{goal}' is reached!")
    else:
        print(f"The goal '{goal}' is reached.")

# Run the main function
if _name___== "_main_":
    main()
```

Output:

```
Forward Chaining System
Forward Chaining completed.
The goal 'criminal(robert)' is reached.
```

Alpha-Beta Pruning

Algorithm:

3/12/24

Alpha-Beta Pruning

1) Initalize the Board

2) Set values of alpha and beta,
initialize the value of alpha as $-\infty$ (maximize for placing queens).

Beta = ∞, best value of minimizer (avoiding conflicts).

3) Start from first row.
If the current state leads to a value worse than Alpha, prune the branch.

If the current leads to a value better than beta, prune the branch

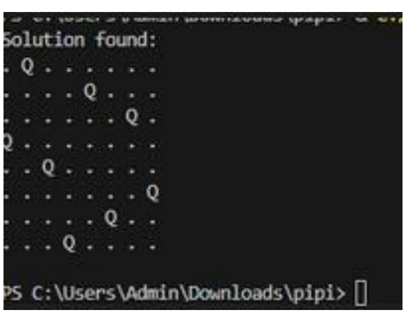
```
function alphabeta (state, depth, alpha, beta):
    if depth == 8:
        return evaluate(state)
    best = -∞
    for each valid move:
        newState = applyMove(state, move)
        value = alphaBeta(newState, depth+1, alpha, beta)
        best = max(best, value)
        alpha = max(alpha, value)
        if beta <= alpha
            break.
```

Code:

```
class EightQueens:
    def __init__(self, size=8):
        self.size = size

    def is_safe(self, board, row, col):
        """Check if placing a queen at board[row][col] is safe."""
        for i in range(col):
            if board[row][i] == 1:  # Check this row on the left
                return False

        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):  # Check upper diagonal
            if board[i][j] == 1:
                return False

        for i, j in zip(range(row, self.size), range(col, -1, -1)):  # Check lower diagonal
            if board[i][j] == 1:
                return False

        return True

    def alpha_beta_search(self, board, col, alpha, beta, maximizing_player):
        """Alpha-Beta Pruning Search."""
        if col >= self.size:  # If all queens are placed
            return 0, [row[:] for row in board]  # Return 0 as heuristic since it's a valid solution

        if maximizing_player:
            max_eval = float('-inf')
            best_board = None
            for row in range(self.size):
                if self.is_safe(board, row, col):
                    board[row][col] = 1
                    eval_score, potential_board = self.alpha_beta_search(board, col + 1, alpha, beta, False)
                    board[row][col] = 0
                    if eval_score > max_eval:
                        max_eval = eval_score
                        best_board = potential_board
                    alpha = max(alpha, eval_score)
                    if beta <= alpha:  # Beta cutoff
                        break
            return max_eval, best_board
        else:
            min_eval = float('inf')
            best_board = None
            for row in range(self.size):
                if self.is_safe(board, row, col):
```

```
                board[row][col] = 1
                eval_score, potential_board = self.alpha_beta_search(board, col + 1, alpha, beta, True)
                board[row][col] = 0
                if eval_score < min_eval:
                    min_eval = eval_score
                    best_board = potential_board
                beta = min(beta, eval_score)
                if beta <= alpha:  # Alpha cutoff
                    break
            return min_eval, best_board

    def solve(self):
        """Solve the 8-Queens problem."""
        board = [[0] * self.size for _ in range(self.size)]
        _, solution = self.alpha_beta_search(board, 0, float('-inf'), float('inf'), True)
        return solution

    def print_board(self, board):
        """Print the chessboard."""
        for row in board:
            print(" ".join("Q" if col else "." for col in row))
        print()


if _name___== "_main_":
    game = EightQueens()
    solution = game.solve()
    if solution:
        print("Solution found:")
        game.print_board(solution)
    else:
        print("No solution exists.")
```

Output:



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

PS C:\Users\Admin\Downloads\pipi>
```

MinMax Tic-Tac-Toe

Algorithm:

3/12/24

Alpha-Beta Pruning

1) Initialize the Board

2) Set values of alpha and beta,
initialize the value of alpha as $-\infty$ (maximize for placing queens).

Beta = ∞, best value of minimizer (avoiding conflicts).

3) Start from first row.
If the current state leads to a value worse than alpha, prune the branch.

If the current leads to a value better than beta, prune the branch

```
Function alphabeta (state, depth, alpha, beta):
    if depth == 8:
        return evaluate(state)
    best = -∞
    for each valid move:
        newState = applyMove(state, move)
        value = alphaBeta(newState, depth +1, alpha, beta)
        best = max(best, value)
        alpha = max(alpha, value)
        if beta <= alpha
            break.
```

Code:

```
def evaluate(state):
    """
    Function to heuristic evaluation of state.
    :param state: the state of the current board
    :return: +1 if the computer wins; -1 if the human wins; 0 draw
    """
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0

    return score


def wins(state, player):
    """
    This function tests if a specific player wins. Possibilities:
    * Three rows    [X X X] or [O O O]
    * Three cols    [X X X] or [O O O]
    * Two diagonals [X X X] or [O O O]
    :param state: the state of the current board
    :param player: a human or a computer
    :return: True if the player wins
    """
    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False


def game_over(state):
    """
    This function test if the human or computer wins
```

```
    :param state: the state of the current board
    :return: True if the human or computer wins
    """
    return wins(state, HUMAN) or wins(state, COMP)


def empty_cells(state):
    """
    Each empty cell will be added into cells' list
    :param state: the state of the current board
    :return: a list of empty cells
    """
    cells = []

    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell == 0:
                cells.append([x, y])

    return cells


def valid_move(x, y):
    """
    A move is valid if the chosen cell is empty
    :param x: X coordinate
    :param y: Y coordinate
    :return: True if the board[x][y] is empty
    """
    if [x, y] in empty_cells(board):
        return True
    else:
        return False


def set_move(x, y, player):
    """
    Set the move on board, if the coordinates are valid
    :param x: X coordinate
    :param y: Y coordinate
    :param player: the current player
    """
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False
```

```
def minimax(state, depth, player):
    """
    AI function that choice the best move
    :param state: current state of the board
    :param depth: node index in the tree (0 <= depth <= 9),
    but never nine in this case (see iaturn() function)
    :param player: an human or a computer
    :return: a list with [the best row, best col, best score]
    """
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
            if score[2] > best[2]:
                best = score  # max value
        else:
            if score[2] < best[2]:
                best = score  # min value

    return best


def clean():
    """
    Clears the console
    """
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')
```

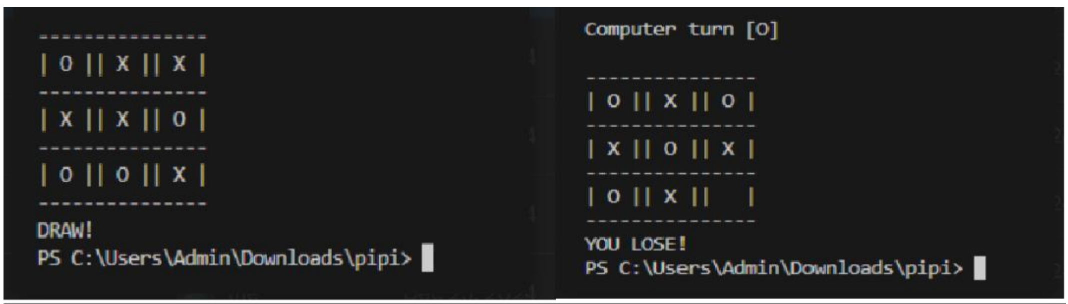
```
def render(state, c_choice, h_choice):
    """
    Print the board on console
    :param state: current state of the board
    """

    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)


def ai_turn(c_choice, h_choice):
    """
    It calls the minimax function if the depth < 9,
    else it choices a random coordinate.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]

    set_move(x, y, COMP)
    time.sleep(1)
```

```
def human_turn(c_choice, h_choice):
    """
    The Human plays choosing a valid move.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
            can_move = set_move(coord[0], coord[1], HUMAN)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')


def main():
    """
    Main function that calls all functions
    """
    clean()
    h_choice = ''  # X or O
```

```
c_choice = ''  # X or O
first = ''  # if human is the first

# Human chooses X or O to play
while h_choice != 'O' and h_choice != 'X':
    try:
        print('')
        h_choice = input('Choose X or O\nChosen: ').upper()
    except (EOFError, KeyboardInterrupt):
        print('Bye')
        exit()
    except (KeyError, ValueError):
        print('Bad choice')

# Setting computer's choice
if h_choice == 'X':
    c_choice = 'O'
else:
    c_choice = 'X'

# Human may starts first
clean()
while first != 'Y' and first != 'N':
    try:
        first = input('First to start?[y/n]: ').upper()
    except (EOFError, KeyboardInterrupt):
        print('Bye')
        exit()
    except (KeyError, ValueError):
        print('Bad choice')

# Main loop of this game
while len(empty_cells(board)) > 0 and not game_over(board):
    if first == 'N':
        ai_turn(c_choice, h_choice)
        first = ''

    human_turn(c_choice, h_choice)
    ai_turn(c_choice, h_choice)

# Game over message
if wins(board, HUMAN):
    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)
    print('YOU WIN!')
elif wins(board, COMP):
```

```
        clean()
        print(f'Computer turn [{c_choice}]')
        render(board, c_choice, h_choice)
        print('YOU LOSE!')
    else:
        clean()
        render(board, c_choice, h_choice)
        print('DRAW!')

    exit()


if _name___== '_main_':
    main()
```

OUTPUT:



LAB 10

CIE QUESTIONS IMPLEMENTATIONS

Lab 10 17/12/24

CIE 10 Marks Questions

Q.1) Prove that Mary Sneezes

→ Mary Sneezes : True

Q.2) FOL representation i) $\forall x\,(\exists y\,((Real(x)\,\&\,Real(y)\,\&\,(y_1 - (x)))$

FOL representation ii) $\forall x\,[\exists y\,((Person(y)\,\&\,Loves\,(x,y)))]$

FOL representation iii) $\exists x\,(\forall y\,(Person\,(y) \rightarrow \sim(Loves\,(x,y))$

FOL representation iv): $\forall x\,((Bought(Ronald, x) \rightarrow Brought($

FOL representation v: $\{Green\,(Parrot)\,\&. \sim(Green(Rabbit$

Q.3 Proving 'John Likes Peanuts' using forward chaining

Result (Forward Chaining) True

Proving ('John Likes Peanuts') using Backward Chaining

Result (Backward chaining): True.

Q.4) The optimal value for the root node is: 9.