VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum- 590014, Karnataka.

LAB RECORD

Artificial Intelligence (23CS5PCAIN)

Submitted by

Mohith Jain

(1BM22CS162)

in partial fulfilment 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

Academic Year 2024 -25 (odd)

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 **Mohith Jain (1BM22CS162),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Laboratory report has been approved as it satisfies the academic requirements of the above-mentioned subject and the work prescribed for the said degree.

Signature of Faculty In Charge

Dr. Asha G R
Associate Professor
Department of CSE, BMSCE

Signature of HOD

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

INDEX

Sl No.	Date	Experiment Title	Page No.
1	01.10.24	Implement Tic –Tac –Toe Game.	4
2	01.10.24	Implement a vacuum cleaner agent.	10
3	08.10.24	Solve 8 puzzle problems.	12
4	08.10.24	Implement Iterative Deepening Search Algorithm	16
5	15.10.24 22.10.24	a. Implement A* search algorithm. b. Implement Hill Climbing Algorithm.	19 25
6	29.10.24	Write a program to implement Simulated Annealing Algorithm	28
7	12.11.24	Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.	31
8	19.11.24	Convert a given first order logic statement into Conjunctive Normal Form (CNF).	33
9	26.11.24	Implement unification in first order logic.	36
10	03.12.24	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	38
11	03.12.24	Create a knowledge base using prepositional logic and prove the given query using resolution.	40
12	17.12.24	Implement Alpha-Beta Pruning.	42

Github Link: https://github.com/mohithjain/AILab-1BM22CS162

DATE - 1/10/2024

TIC-TAC -TOE GAME

Program:

```
import random
def draw_board(board):
    """Prints the Tic Tac Toe board."""
    print('   |   |')
    print(' ' + board[0] + ' | ' + board[1] + ' | ' + board[2])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[3] + ' | ' + board[4] + ' | ' + board[5])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[6] + ' | ' + board[7] + ' | ' + board[8])
    print('   |   |')

def input_player_letter():
    """Lets the player choose X or O."""
    letter = ''
    while letter not in ['X', 'O']:
        letter = input("Do you want to be X or O? ").upper()
    return letter

def who_goes_first():
    """Randomly decides who goes first."""
    return 'computer' if random.randint(0, 1) == 0 else 'player'

def make_move(board, letter, move):
    """Places the player's letter on the board."""
    board[move] = letter

def is_winner(board, letter):
    """Checks if the specified letter has won."""
    winning_combinations = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]              # Diagonals
    ]
    return any(all(board[i] == letter for i in combo) for combo in winning_combinations)

def is_board_full(board):
```

```
    """Checks if the board is full."""
    return all(space != ' ' for space in board)

def get_player_move(board):
    """Gets the player's move."""
    move = -1
    while move not in range(9) or board[move] != ' ':
        try:
            move = int(input("What is your next move? (0-8): "))
        except ValueError:
            print("Invalid input. Please enter a number between 0 and 8.")
            continue
    return move

def get_computer_move(board, computer_letter, player_letter):
    """Determines the computer's move using a basic strategy."""

    # Check for winning move
    for i in range(9):
        if board[i] == ' ':
            board[i] = computer_letter
            if is_winner(board, computer_letter):
                return i
            board[i] = ' '  # Undo move

    # Block player's winning move
    for i in range(9):
        if board[i] == ' ':
            board[i] = player_letter
            if is_winner(board, player_letter):
                return i
            board[i] = ' '  # Undo move

    # If no immediate win or block, choose randomly
    available_moves = [i for i in range(9) if board[i] == ' ']
    return random.choice(available_moves) if available_moves else None

# Main Game Loop
print("Welcome to Tic Tac Toe!")

# Initialize scores
player_score = 0
computer_score = 0
tie_score = 0

while True:
```

```
# Initialize the game
board = [' '] * 9
player_letter = input_player_letter()
computer_letter = 'O' if player_letter == 'X' else 'X'

turn = who_goes_first()

while True:
    if turn == 'player':
        draw_board(board)
        move = get_player_move(board)
        make_move(board, player_letter, move)

        if is_winner(board, player_letter):
            draw_board(board)
            print("Congratulations! You have won!")
            player_score += 1
            break

        if is_board_full(board):
            draw_board(board)
            print("It's a tie!")
            tie_score += 1
            break

        turn = 'computer'
    else:
        move = get_computer_move(board, computer_letter, player_letter)
        make_move(board, computer_letter, move)

        if is_winner(board, computer_letter):
            draw_board(board)
            print("The computer has won! You lose.")
            computer_score += 1
            break

        if is_board_full(board):
            draw_board(board)
            print("It's a tie!")
            tie_score += 1
            break

        turn = 'player'

# Print current scores after each game round
print("\nCurrent Scores:")
```

```
    print(f"Player: {player_score}, Computer: {computer_score}, Ties: {tie_score}")

    # Ask to play again; if yes, continue to next round without resetting scores.
    play_again = input("Do you want to play again? (yes/no): ").lower()

    if play_again != 'yes':
        break

print("Thank you for playing!")
```

OUTPUT

```
Welcome to Tic Tac Toe!
Do you want to be X or O?  X
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
What is your next move? (0-8):  0
   |   |
 X |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   | O
   |   |
What is your next move? (0-8):  4
   |   |
 X |   |
   |   |
-----------
  What is your next move? (0-8):  3
    | X |
    |   |
-----------
    |   |
 O  |   | O
    |   |
What is your next move? (0-8):  7
    |   |
 X  | O |
    |   |
-----------
    |   |
    | X |
    |   |
-----------
    |   |
 O  | X | O
    |   |
What is your next move? (0-8):  2
    |   |
 X  | O | X
    |   |
-----------
    |   |
    | X | O
    |   |
-----------
    |   |
 O  | X | O

   What is your next move? (0-8):  4
```

```
  |   |
X | O | X
  |   |
-----------
  |   |
X | X | O
  |   |
-----------
  |   |
O | X | O
  |   |
```

It's a tie!

Current Scores:
Player: 0, Computer: 0, Ties: 1
Do you want to play again? (yes/no): yes
Do you want to be X or O? x

```
  |   |
  |   |
  |   |
-----------
  |   |
  |   |
  |   |
-----------
  |   |
  |   |
  |   |
```

What is your next move? (0-8): 0

```
  |   |
X | O |
  |   |
-----------
  |   |
  |   |
  |   |
-----------
  |   |
  |   |
  |   |
```

```
  |   |
X | O |
  |   |
-----------
  |   |
  | X |
  |   |
-----------
  |   |
  |   | O
  |   |
```

What is your next move? (0-8): 6

```
  |   |
X | O | O
  |   |
-----------
  |   |
  | X |
  |   |
-----------
  |   |
X |   | O
  |   |
```

What is your next move? (0-8): 3

```
  |   |
X | O | O
  |   |
-----------
  |   |
X | X |
  |   |
-----------
  |   |
X |   | O
  |   |
```

Congratulations! You have won!

Current Scores:
Player: 1, Computer: 0, Ties: 1
Do you want to play again? (yes/no): no

OBSERVATION

Lab-01 01

① ⇒ Tic-Tac-Toe Game

Algorithm:

Step 1:
BEGIN Tic-Tac-Toe game
// Initialize game state
board = [' ' for i in range(9)]
scoreboard = { }
player1 = prompt "Enter player 1 name:"
player2 = prompt "Enter player 2 name"
scoreboard [player1] = 0
scoreboard [player2] = 0

Step 2:
win-combination [[0,1,2], [3,4,5], [6,7,8],
[0,3,6], [1,4,7], [2,5,8],
[0,4,8], [2,4,6]]

Step 3:

while true DO
current-player = player1
while DO
Display board

Step 4: Display the board
MOVE = prompt "Player" + current player + ",
enter your move (1-9):"

Step 5:
if move < 1 or move > 9 or board [move-1] ! = ' '
Display "Invalid move."
continue

01/10/24

board [move-1] = current-player

Step 6:

if checkWin(board, current-player, win-combination):
Display board
Display current-player + "wins!"
scoreboard [current-player] += 1
Break

if checkTie(board) then
Display board
Display "It is tie"
Break

Step 7: // Switch player
if current-player == player1 Then
current-player = player2
else
current-player = player1

Step 8:
Replay = prompt "Do you want to play again? (y/n)"

if replay == "no" then
Display scoreboard
Break
board = [" " for i in range(9)] // Reset

End Tic Tac Toe

Bafna Gold 02

Output:
The first player name
Please mention your name: Nidhith

The second player name
Please mention your name:

SCOREBOARD for TIC TAC TOE

Nidhith 0
Nidhithjain 0

Nidhith, it's the chance to make the choice
Please press 1 for X
Please press 2 for O
Please press 3 for exit

The player X turn to choose Block: 1

The player O turn to choose Block: 2

DATE - 1/10/24

VACUUM CLEANER AGENT

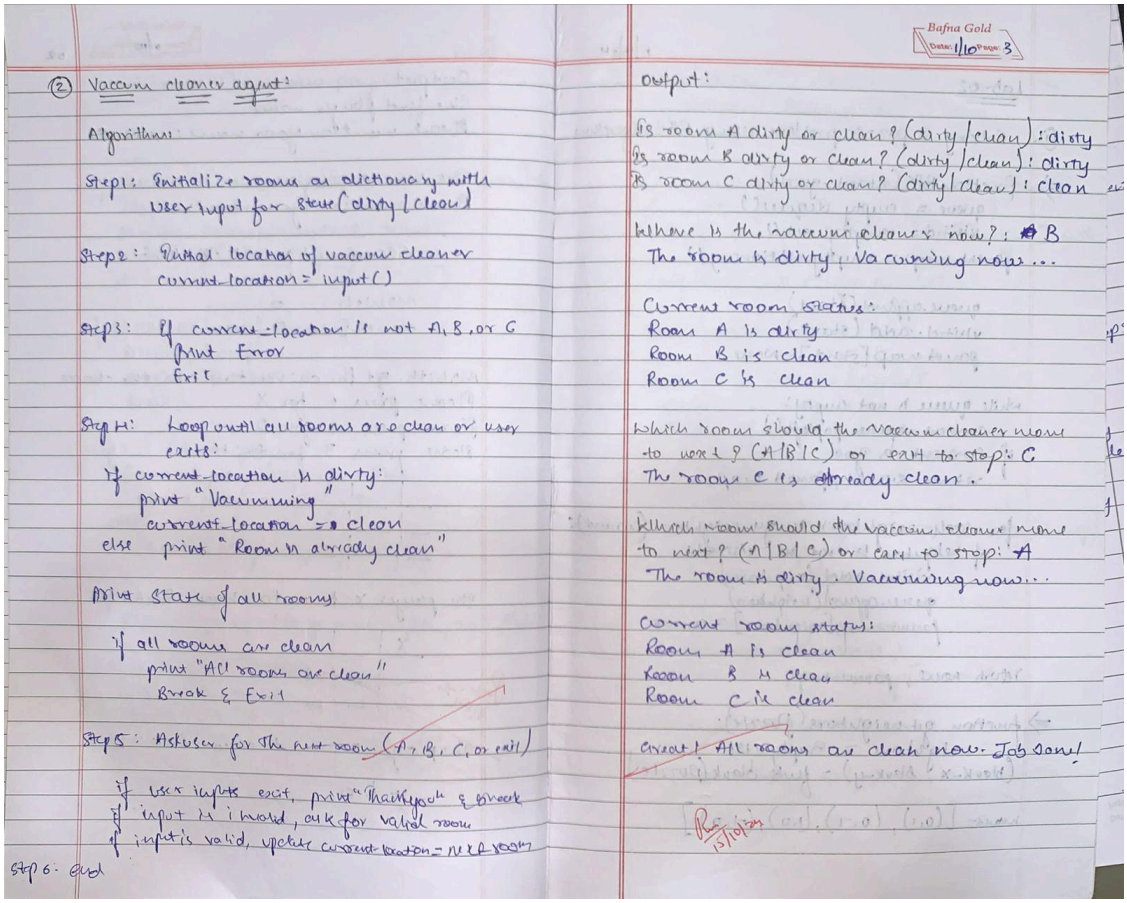
PROGRAM:

```
def vacuum_cleaner():
    # Initial states of the rooms, asking the user for input
    rooms = {
        'A': input("Is room A dirty or clean? (dirty/clean): ").strip().lower(),
        'B': input("Is room B dirty or clean? (dirty/clean): ").strip().lower(),
        'C': input("Is room C dirty or clean? (dirty/clean): ").strip().lower()
    }

    # Get the initial location of the vacuum cleaner
    current_location = input("Where is the vacuum cleaner now? (A/B/C): ").strip().upper()

    # Make sure the input is valid
    if current_location not in rooms:
        print("Invalid location! Please choose A, B, or C.")
        return

    # Keep cleaning until all rooms are clean or the user decides to stop
    while True:
        # Check if the current room is dirty or clean
        if rooms[current_location] == 'dirty':
            print(f"The room {current_location} is dirty. Vacuuming now...")
            rooms[current_location] = 'clean'  # Mark the room as clean after vacuuming
        else:
            print(f"The room {current_location} is already clean. No need to vacuum here.")

        # Show the status of all the rooms
        print("\nCurrent room status:")
        for room, status in rooms.items():
            print(f"Room {room} is {status}.")

        # Check if all rooms are clean
        if all(status == 'clean' for status in rooms.values()):
            print("\nGreat! All rooms are clean now. Job done!")
            break

        # Ask where to go next, or let the user stop the program
        next_room = input("\nWhich room should the vacuum cleaner move to next? (A/B/C) or
type 'exit' to stop: ").strip().upper()

        # If the user wants to exit
        if next_room == 'EXIT':
```

```
        print("Okay, stopping the vacuum cleaner. Goodbye!")
        break

    # Validate the next room input
    if next_room not in rooms:
        print("Invalid room choice! Please enter A, B, or C.")
    else:
        current_location = next_room  # Update the current location to the next room

# Run the vacuum cleaner program
vacuum_cleaner()
```

OUTPUT:

```
Is room A dirty or clean? (dirty/clean): dirty
Is room B dirty or clean? (dirty/clean): dirty
Is room C dirty or clean? (dirty/clean): clean
Where is the vacuum cleaner now? (A/B/C): A
The room A is dirty. Vacuuming now...

Current room status:
Room A is clean.
Room B is dirty.
Room C is clean.
```



DATE-8/10/2024

8 PUZZLE PROBLEM

PROGRAM:

```
from collections import deque
# Function to print the puzzle
def print_puzzle(puzzle):
    for row in puzzle:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()

# Function to find the position of the blank tile (0)
def find_blank(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i, j

# Function to generate all possible states by moving the blank tile
def get_neighbors(puzzle):
    neighbors = []
    blank_x, blank_y = find_blank(puzzle)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up

    for dx, dy in moves:
        new_x = blank_x + dx
        new_y = blank_y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:  # Check if within bounds
            # Create a new state by swapping the blank tile
            new_puzzle = [row[:] for row in puzzle]  # Make a copy of the current puzzle
                                        new_puzzle[blank_x][blank_y],    new_puzzle[new_x][new_y]    =
new_puzzle[new_x][new_y], new_puzzle[blank_x][blank_y]
            neighbors.append(new_puzzle)
    return neighbors

# BFS function to solve the puzzle
def bfs(start, goal):
    queue = deque([start])  # Queue for BFS
    visited = set()  # Set to track visited states
    visited.add(tuple(map(tuple, start)))  # Add initial state to visited

    parent_map = {tuple(map(tuple, start)): None}  # To track the path

    while queue:
        current = queue.popleft()  # Get the current state
```

```
        if current == goal:  # Check if we reached the goal
            return current, parent_map

        for neighbor in get_neighbors(current):  # Explore neighbors
            neighbor_tuple = tuple(map(tuple, neighbor))  # Convert to tuple for set
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                queue.append(neighbor)
                parent_map[neighbor_tuple] = tuple(map(tuple, current))  # Track parent state

    return None, parent_map  # Return None if no solution is found

# Function to backtrack and get the path from start to goal
def get_solution_path(start, goal, parent_map):
    path = []
    current = tuple(map(tuple, goal))
    while current is not None:
        path.append(current)
        current = parent_map[current]
    return path[::-1]  # Reverse the path to get it from start to goal

# Main function to run the puzzle solver
def solve_8_puzzle():
    print("Enter the starting state of the 8-puzzle (use 0 for the blank space):")
    start_state = []
    for i in range(3):
        row = input(f"Enter row {i + 1} (3 numbers separated by space): ")
        start_state.append([int(x) for x in row.split()])

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

    print("Starting puzzle:")
    print_puzzle(start_state)

    solution, parent_map = bfs(start_state, goal_state)

    if solution:
        print("Solution found:")
        print_puzzle(solution)

        # Ask the user if they want to see the transition states
```

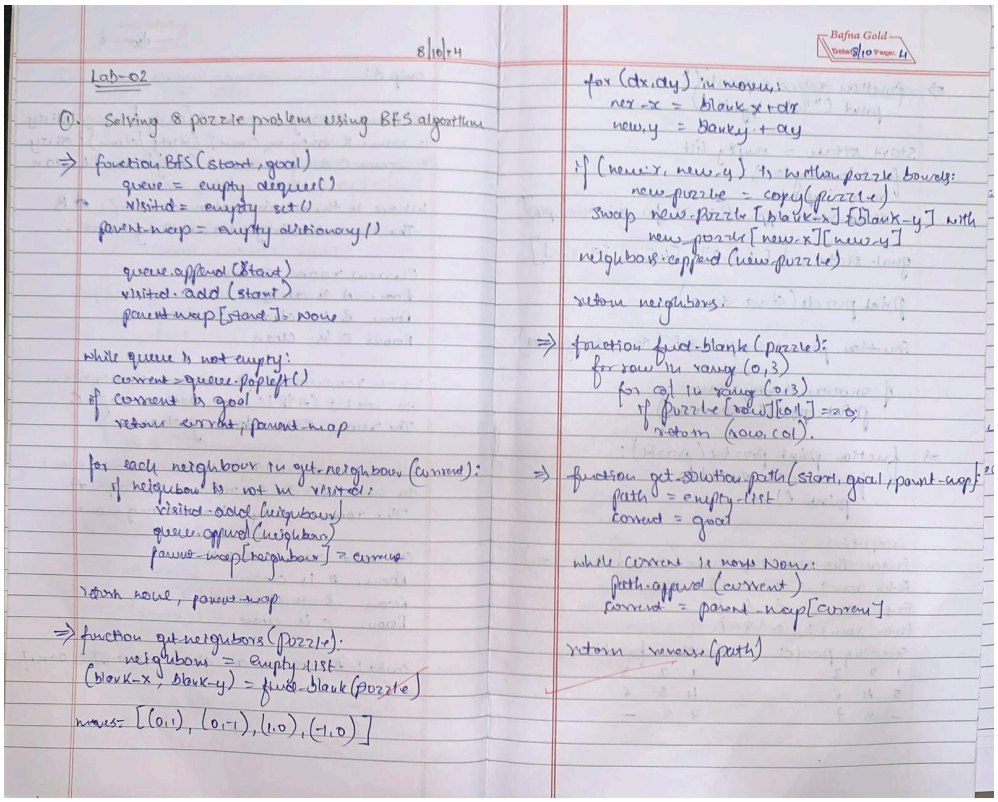
```
        show_transition = input("Do you want to see the transition states from start to goal? (yes/no): ").strip().lower()
        if show_transition == 'yes':
            path = get_solution_path(start_state, solution, parent_map)
            print("Transition states:")
            for state in path:
                print_puzzle(state)
    else:
        print("No solution found.")

# Run the solver
solve_8_puzzle()
```

OUTPUT:

```
Enter the starting state of the 8-puzzle (use 0 for the blank space):
Enter row 1 (3 numbers separated by space): 0 1 2
Enter row 2 (3 numbers separated by space): 3 4 5
Enter row 3 (3 numbers separated by space): 6 7 8
Starting puzzle:
_ 1 2
3 4 5
6 7 8

Solution found:
1 2 3
4 5 6
7 8 _

Do you want to see the transition states from start to goal? (yes/no): yes
Transition states:
```

_ 1 2 3 4 5 6 7 8	1 4 2 6 _ 3 7 8 5	1 4 2 7 _ 3 8 6 5
1 _ 2 3 4 5 6 7 8	1 4 2 _ 6 3 7 8 5	1 _ 2 7 4 3 8 6 5
1 4 2 3 _ 5 6 7 8	1 4 2 7 6 3 _ 8 5	1 2 _ 7 4 3 8 6 5
1 4 2 _ 3 5 6 7 8	1 4 2 7 6 3 8 _ 5	1 4 2 6 3 5 _ 7 8

1 4 2 6 3 5 7 _ 8 1 4 2 6 3 5 7 8 _ 1 4 2 6 3 _ 7 8 5 1 2 3 7 4 5 8 6 _ 1 2 3 7 4 5 8 _ 6	1 2 3 7 4 5 _ 8 6 1 2 3 _ 4 5 7 8 6 1 2 3 4 _ 5 7 8 6 1 2 3 4 5 _ 7 8 6 1 2 3 4 5 6 7 8 _	



8/10/24

Lab-02

1. Solving 8 puzzle problem using BFS algorithm

⇒ function BFS(start, goal)
queue = empty deque()
visited = empty set()
parent_map = empty dictionary()

queue.append(start)
visited.add(start)
parent_map[start] = None

while queue is not empty:
current = queue.popleft()
if current is goal
return current, parent_map

for each neighbour in get_neighbours(current):
if neighbour is not in visited:
visited.add(neighbour)
queue.append(neighbour)
parent_map[neighbour] = current

return None, parent_map

⇒ function get_neighbours(puzzle):
neighbours = empty list
(blank_x, blank_y) = find_blank(puzzle)

moves = [(0,1), (0,-1), (1,0), (-1,0)]

Bafna Gold

for (dx,dy) in moves:
new_x = blank_x + dx
new_y = blank_y + dy

if (new_x, new_y) is within puzzle bounds:
new_puzzle = copy(puzzle)
swap new_puzzle[blank_x][blank_y] with new_puzzle[new_x][new_y]
neighbours.append(new_puzzle)

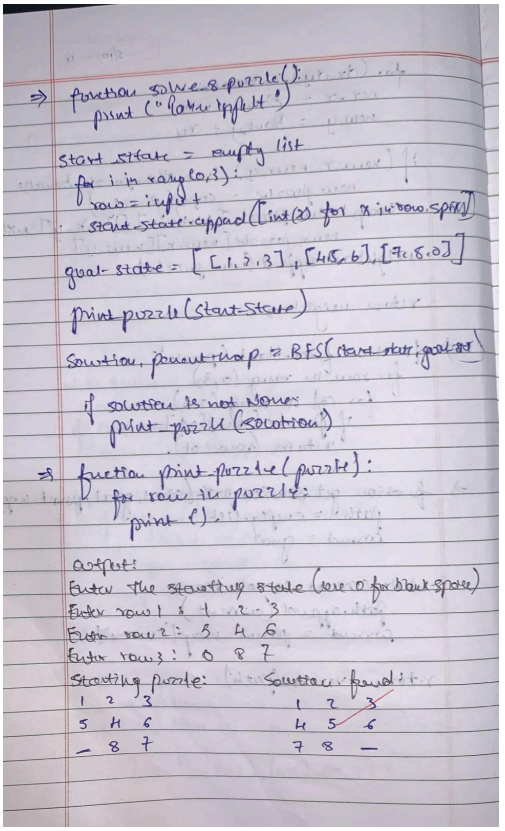
return neighbours

⇒ function find_blank(puzzle):
for row in range(0,3)
for col in range(0,3)
if puzzle[row][col] == 0
return (row, col)

⇒ function get_solution_path(start, goal, parent_map):
path = empty list
current = goal

while current is not None:
path.append(current)
current = parent_map[current]

return reverse(path)



⇒ function solve_8_puzzle():
print("Solve 8 puzzle")

start_state = empty list
for i in range(0,3):
row = input()
start_state.append([int(x) for x in row.split()])

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

print_puzzle(start_state)

solution, parent_map = BFS(start_state, goal_state)

if solution is not None:
print_puzzle(solution)

⇒ function print_puzzle(puzzle):
for row in puzzle:
print()

output:
Enter the starting state (use 0 for blank space)
Enter row 1: 1 2 3
Enter row 2: 5 4 6
Enter row 3: 0 8 7
Starting puzzle:
1 2 3
5 4 6
_ 8 7

Solution found:
1 2 3
4 5 6
7 8 _

DATE-8/10/2024

Iterative Deepening SEARCH

PROGRAM:

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        """Add an edge from node u to node v."""
        if u not in self.graph:
            self.graph[u] = []
        self.graph[u].append(v)

    def DLS(self, src, target, limit):
        """
        Depth-Limited Search (DLS): Search for the target starting from the src node,
        but stop the recursion once the depth limit is reached.
        """
        if src == target:
            return True  # Target found

        if limit <= 0:
            return False  # Limit reached, stop recursion

        # Recur for all adjacent vertices of the src node
        if src in self.graph:
            for neighbor in self.graph[src]:
                if self.DLS(neighbor, target, limit - 1):
                    return True

        return False  # Target not found within this depth limit

    def IDDFS(self, src, target, max_depth):
        """
        Iterative Deepening Depth-First Search (IDDFS):
        Perform DLS for increasing limits (from 0 to max_depth).
        """
        for depth in range(max_depth + 1):
            print(f"Checking with depth limit: {depth}")
            if self.DLS(src, target, depth):
                print(f"Target {target} found at depth level: {depth}")
                return True  # Target found within depth limit
        return False  # Target not reachable within max_depth

# Main function to get user input and execute IDDFS
def main():
```

```
    # Create an instance of the graph
    g = Graph()

    # Take user input for the graph structure
    num_edges = int(input("Enter the number of edges in the graph: "))
    print("Enter each edge as a pair of space-separated integers (source, destination):")
    for _ in range(num_edges):
        u, v = map(int, input().split())  # User inputs edges as two space-separated integers
        g.add_edge(u, v)

    # Take user input for the source, target, and maximum depth
    src = int(input("Enter the source node: "))
    target = int(input("Enter the target node: "))
    max_depth = int(input("Enter the maximum depth limit: "))

    # Perform the Iterative Deepening Depth-First Search (IDDFS)
    if g.IDDFS(src, target, max_depth):
        print(f"\nTarget {target} is reachable from source {src} within depth {max_depth}")
    else:
        print(f"\nTarget {target} is NOT reachable from source {src} within depth {max_depth}")

# Run the main function
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Enter the number of edges in the graph: 6
Enter each edge as a pair of space-separated integers
(source, destination):
0 1
0 2
1 3
1 4
2 5
2 6
Enter the source node: 0
Enter the target node: 5
Enter the maximum depth limit: 3
Checking with depth limit: 0
Checking with depth limit: 1
Checking with depth limit: 2
Target 5 found at depth level: 2

Target 5 is reachable from source 0 within depth 3
```

Bafna Gold
Date 3/10 Page: 5

(7) Implementing Iterative deepening search algorithm.

Algorithm

⇒ class Graph:
function __init__():
graph = empty dictionary

function add_edge(u,v):
if u is not in graph:
graph[u] = empty list
graph[u].append(v)

⇒ function depthSearch(source, target, limit):
if source == target:
return True
if limit <= 0:
return false
if source in graph:
for each neighbor in graph[source]:
if depthSearch(neighbor, target, limit-1)
return True // Found
return False // NOT found

⇒ function IterativeDeep(src, target, max_depth):
for depth in range(0, max_depth+1):
print("depth limit")
if depthSearch(src, target, depth):
print "Target" ++target+ "found at depth level" + depth.
return True.
return False

⇒ function main():
graph = new Graph()
num_edges = input
print to enter edge pair (source, destination)
for i in range(0, num_edges):
u,v = input("Enter edge:").split()
graph.add_edge(int(u), int(v)).
src = input("Enter the source node")
target = input("Enter the target node:")
max_depth = input("Enter max depth limit")
if IterativeDeep(int(src), int(target), int(maxdepth)):
print "reachable target"
else
print "not reachable target".

Bafna Gold
Date 3/10 Page: 6

Output:
Enter the no. of edges in graph: 5
Enter each edge as a pair (source, destination):
0 1
0 2
1 3
1 4
2 5
Enter the source node: 0
Enter the target node: 5
Enter the max depth limit: 3

Checking with depth limit: 0
Checking with depth limit: 1
Checking with depth limit: 2
Target 5 found at depth level: 2
Target 5 is reachable from source 0 with depth 3.

15/10/24

DATE-8/10/2024

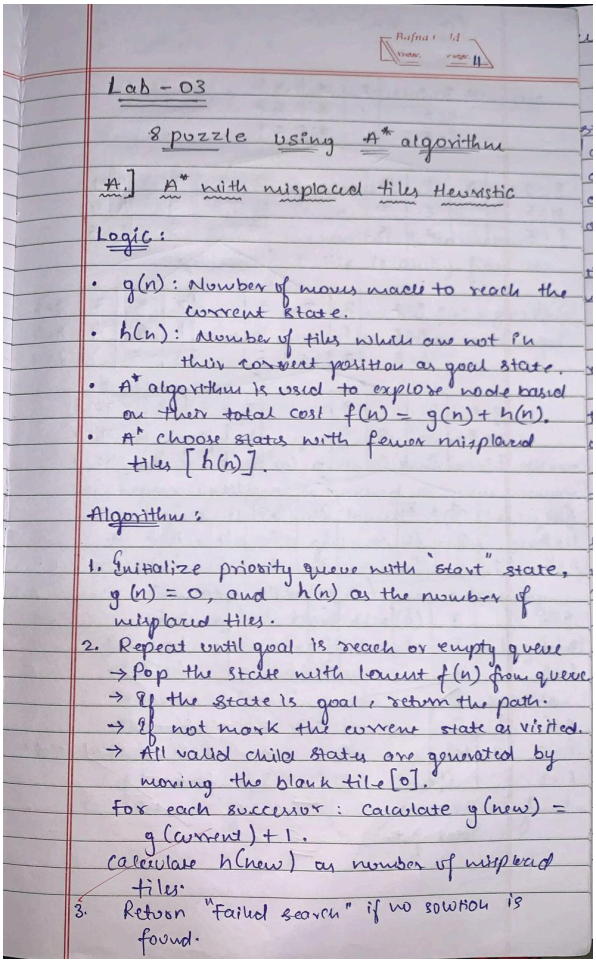
8 PUZZLE PROBLEM A* SEARCH

a) Using Misplaced Tiles

Program:

```
import heapq
# Function to find the index of a tile in a 3x3 board
def find_index(state, tile):
    for i in range(3):
        for j in range(3):
            if state[i][j] == tile:
                return (i, j)
# Heuristic: Number of misplaced tiles
def misplaced_tiles(current, goal):
    count = 0
    for i in range(3):
        for j in range(3):
            if current[i][j] != goal[i][j] and current[i][j] != 0:
                count += 1
    return count
# A* algorithm with misplaced tiles heuristic
def a_star_misplaced_tiles(start, goal):
    pq = []
    heapq.heappush(pq, (0, 0, start, []))  # f(n), g(n), state, path
    visited = set()

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

        if current == goal:
            return path + [current], g  # Return final path and cost

        visited.add(str(current))

        zero_pos = find_index(current, 0)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Move directions

        for d in directions:
            new_pos = (zero_pos[0] + d[0], zero_pos[1] + d[1])
            if 0 <= new_pos[0] < 3 and 0 <= new_pos[1] < 3:
                new_state = [list(row) for row in current]
                new_state[zero_pos[0]][zero_pos[1]] = new_state[new_pos[0]][new_pos[1]]
                new_state[new_pos[0]][new_pos[1]] = 0

                if str(new_state) not in visited:
```

```
                h = misplaced_tiles(new_state, goal)
                f_new = g + 1 + h
                heapq.heappush(pq, (f_new, g + 1, new_state, path + [current]))

    return None, None  # No solution found

# Function to take user input for the start and goal states
def get_user_input():
    print("Enter the start state (row by row, use 0 for the blank):")
    start_state = [list(map(int, input().split())) for _ in range(3)]

    print("Enter the goal state (row by row, use 0 for the blank):")
    goal_state = [list(map(int, input().split())) for _ in range(3)]

    return start_state, goal_state

# Main function to run the A* algorithm and display results
def main():
    start_state, goal_state = get_user_input()

    solution, cost = a_star_misplaced_tiles(start_state, goal_state)

    if solution:
        print("\nStart State:")
        for row in start_state:
            print(row)

        print("\nGoal State:")
        for row in goal_state:
            print(row)

        print("\nSolution Path:")
        for step in solution:
            for row in step:
                print(row)
            print()

        print(f"Total Path Cost: {cost}")
    else:
        print("No solution found."
# Run the main function
if __name__ == "__main__":
    main()
```



Lab - 03

8 puzzle using A* algorithm

A.] A* with misplaced tiles Heuristic

Logic:

- g(n): Number of moves made to reach the current state.
- h(n): Number of tiles which are not in their correct position as goal state.
- A* algorithm is used to explore node based on their total cost f(n) = g(n) + h(n).
- A* choose states with fewer misplaced tiles [h(n)].

Algorithm:

1. Initialize priority queue with "start" state, g(n) = 0, and h(n) as the number of misplaced tiles.
2. Repeat until goal is reach or empty queue
   → Pop the state with lowest f(n) from queue
   → If the state is goal, return the path.
   → If not mark the current state as visited.
   → All valid child states are generated by moving the blank tile[0].
   For each successor: Calculate g(new) = g(current) + 1.
   Calculate h(new) as number of misplaced tiles.
3. Return "Failed search" if no solution is found.

Output

Enter the start state (row by row, use 0 for the blank):
2 8 3
1 6 4
0 7 5
Enter the goal state (row by row, use 0 for the blank):
1 2 3
8 0 4
7 6 5
Start State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]
Goal State:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Solution Path:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

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

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

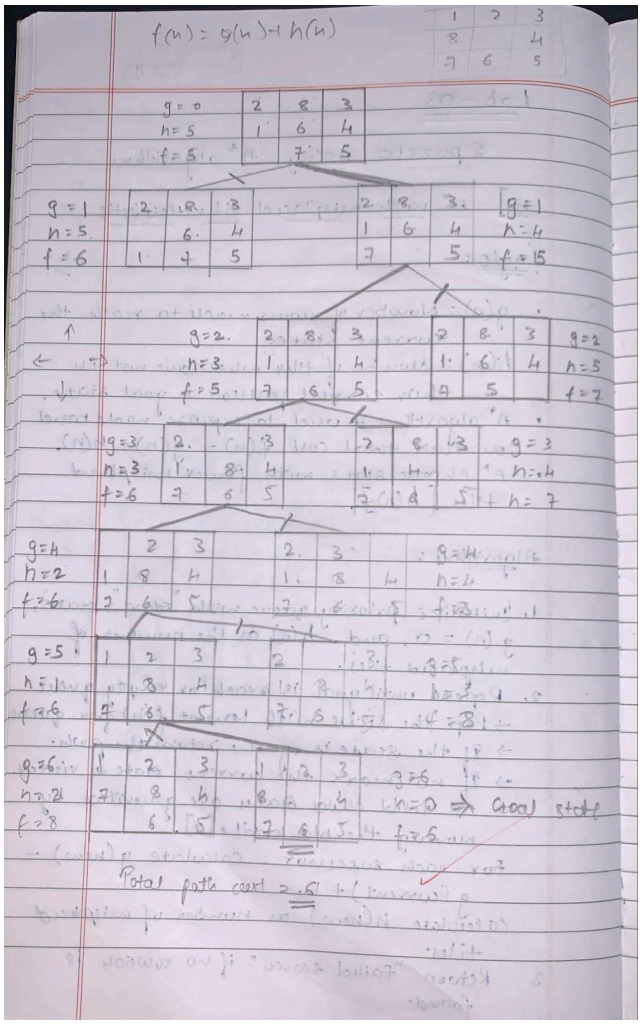
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

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

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

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

Total Path Cost: 6

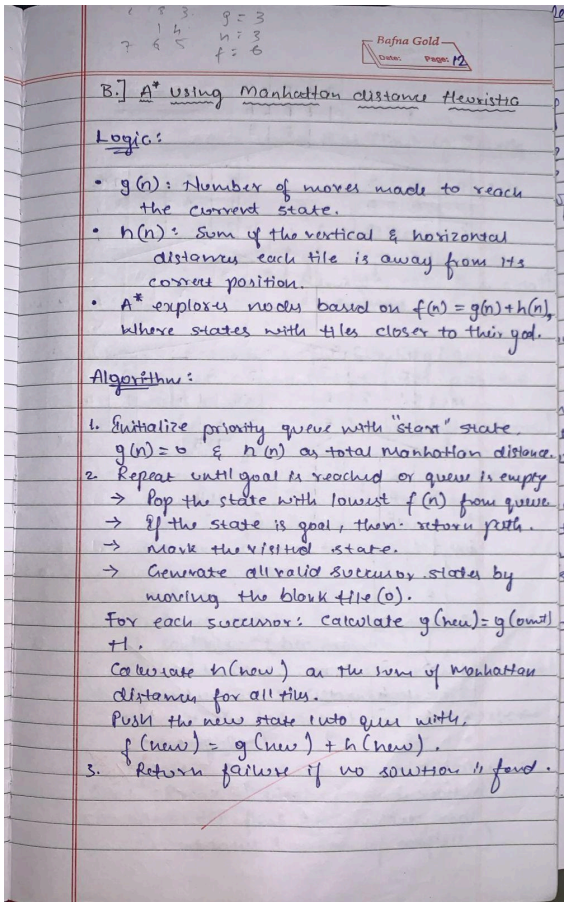


b)Using Manhattan Distance

Program:

```
import heapq

# Function to find the index of a tile in a 3x3 board
def find_index(state, tile):
    for i in range(3):
        for j in range(3):
            if state[i][j] == tile:
                return (i, j)

# Heuristic: Manhattan distance
def manhattan_distance(current, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = current[i][j]
            if tile != 0:
                goal_i, goal_j = find_index(goal, tile)
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

# A* algorithm with Manhattan distance heuristic
def a_star_manhattan_distance(start, goal):
    pq = []
    heapq.heappush(pq, (0, 0, start, []))  # f(n), g(n), state, path
    visited = set()

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

        if current == goal:
            return path + [current], g  # Return final path and path cost

        visited.add(str(current))

        zero_pos = find_index(current, 0)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Move directions

        for d in directions:
            new_pos = (zero_pos[0] + d[0], zero_pos[1] + d[1])
            if 0 <= new_pos[0] < 3 and 0 <= new_pos[1] < 3:
                new_state = [list(row) for row in current]
                new_state[zero_pos[0]][zero_pos[1]] = new_state[new_pos[0]][new_pos[1]]
                new_state[new_pos[0]][new_pos[1]] = 0
```

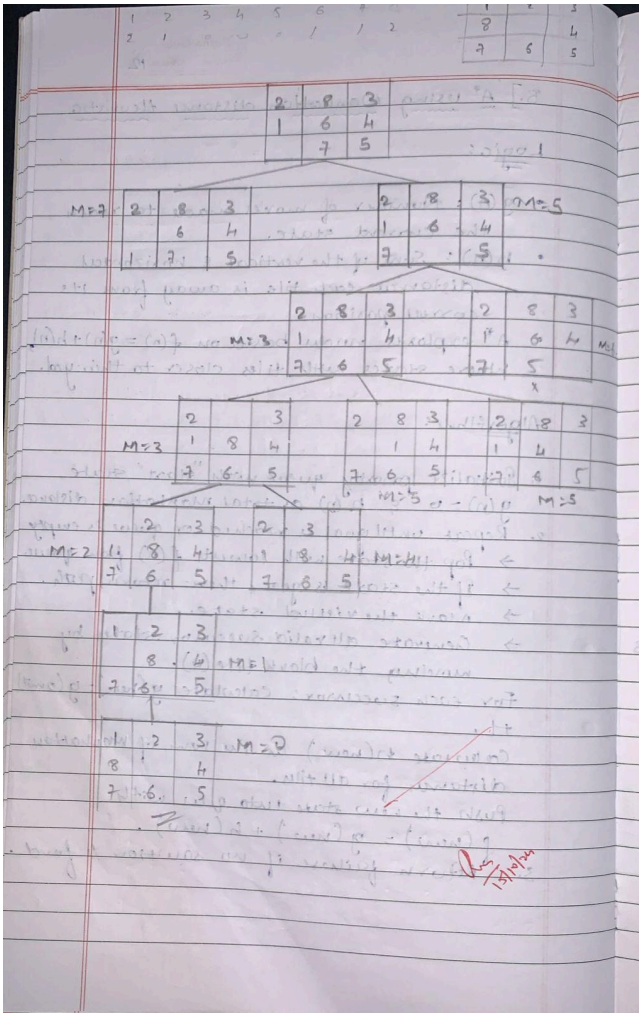
```
            if str(new_state) not in visited:
                h = manhattan_distance(new_state, goal)
                f_new = g + 1 + h
                heapq.heappush(pq, (f_new, g + 1, new_state, path + [current]))

    return None, None  # No solution found

# Function to take user input for the start and goal states
def get_user_input():
    print("Enter the start state (row by row, use 0 for the blank):")
    start_state = [list(map(int, input().split())) for _ in range(3)]

    print("Enter the goal state (row by row, use 0 for the blank):")
    goal_state = [list(map(int, input().split())) for _ in range(3)]

    return start_state, goal_state

# Main function to run the A* algorithm and display results
def main():
    start_state, goal_state = get_user_input()

    solution, cost = a_star_manhattan_distance(start_state, goal_state)

    if solution:
        print("\nStart State:")
        for row in start_state:
            print(row)

        print("\nGoal State:")
        for row in goal_state:
            print(row)

        print("\nSolution Path:")
        for step in solution:
            for row in step:
                print(row)
            print()

        print(f"Total Path Cost: {cost}")
    else:
        print("No solution found.")

# Run the main function
if __name__ == "__main__":
    main()
```



OUTPUT

Enter the start state (row by row, use 0 for the blank):
2 8 3
1 6 4
0 7 5
Enter the goal state (row by row, use 0 for the blank):
1 2 3
8 0 4
7 6 5

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



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

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

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

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

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

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

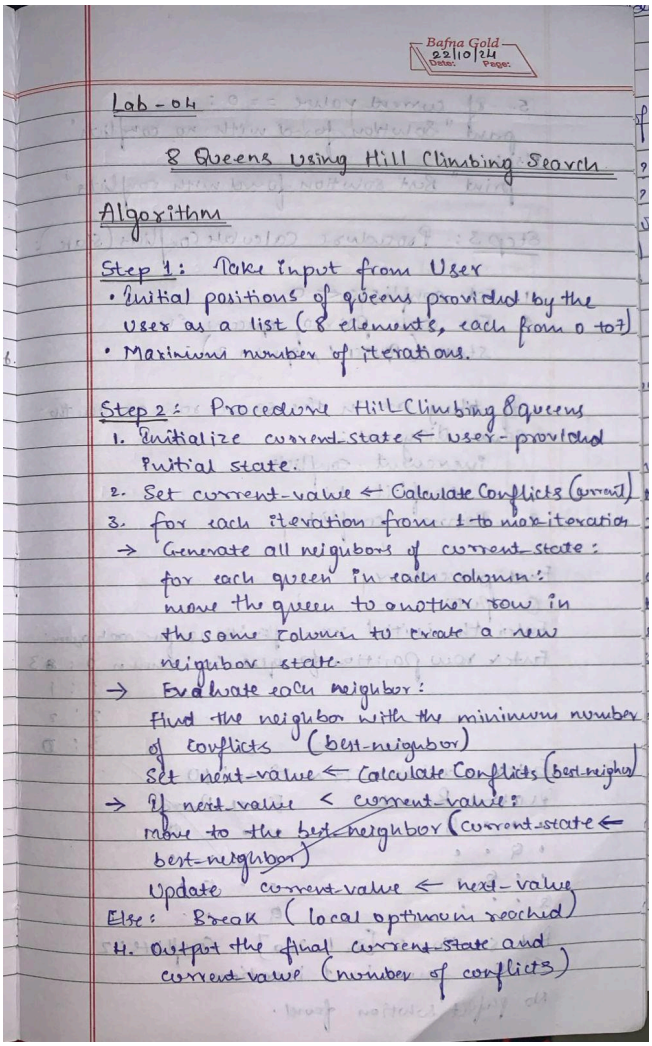
1, 2, 3] [1, 2, 3]
[0, 8, 4] [8, 0, 4]
[7, 6, 5] [7, 6, 5]
Total Path Cost: 6

DATE - 22/10/2024

HILL CLIMBING - N QUEEN PROBLEM

Program

```
def calculate_conflicts(state):
    conflicts = 0
    n = len(state)

    for i in range(n):
        for j in range(i + 1, n):

            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts


def generate_neighbors(state):
    neighbors = []
    n = len(state)

    for col in range(n):
        for row in range(n):
            if row != state[col]:

                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)

    return neighbors


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



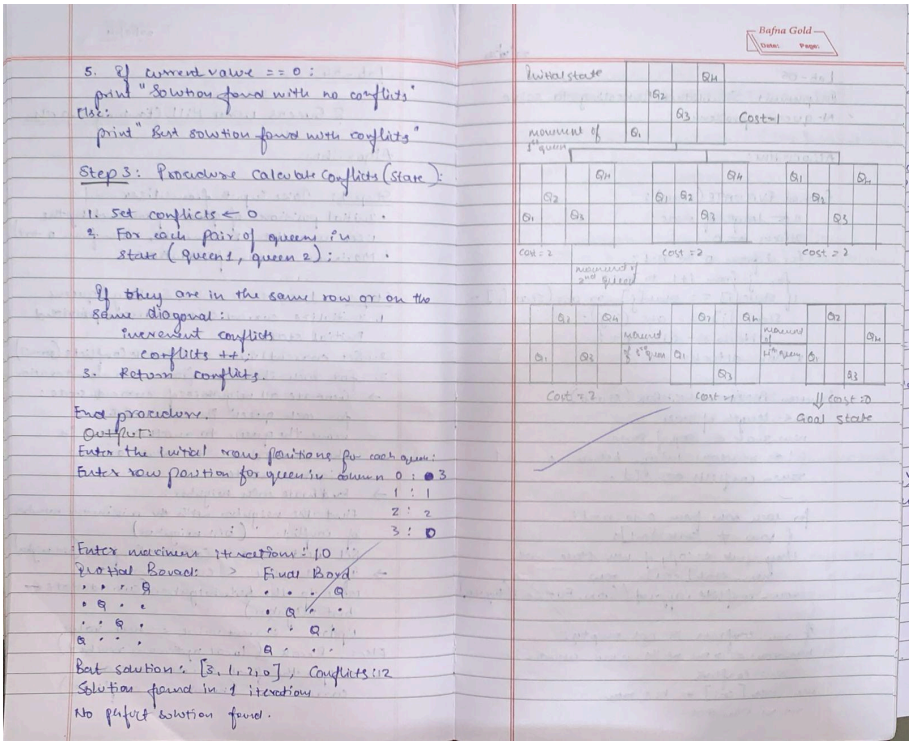

```
def hill_climbing_4_queens(initial_state, max_iterations):
```

```
    current_state = initial_state
    current_value = calculate_conflicts(current_state)

    print("Initial Board:")
    print_board(current_state)

    iteration = 0
    for iteration in range(max_iterations):

        neighbors = generate_neighbors(current_state)
        next_state = min(neighbors, key=calculate_conflicts)
        next_value = calculate_conflicts(next_state)


        if next_value < current_value:
            current_state = next_state
            current_value = next_value
        else:

            break

    print("Final Board:")
    print_board(current_state)

    return current_state, current_value, iteration + 1

try:
    initial_state = []
        print("Enter the initial row positions for each queen (0 to 3) for each column (total 4 queens):")

    for col in range(4):
        row_position = int(input(f"Enter the row position for queen in column {col} (0-3): "))
        if 0 <= row_position <= 3:
            initial_state.append(row_position)
        else:
            raise ValueError("Row position must be between 0 and 3.")

    max_iterations = int(input("Enter the maximum number of iterations: "))


            best_state, best_value, iterations_taken = hill_climbing_4_queens(initial_state, max_iterations)

    print(f"Best Solution: {best_state}, Conflicts: {best_value}")
    print(f"Cost (Number of Conflicts): {best_value}")
```

```
    print(f"Solution found in {iterations_taken} iterations")


    if best_value == 0:
        print("\nSolution found with no conflicts!")
    else:
        print("\nNo perfect solution found. Try increasing the number of iterations.")

except ValueError as ve:
    print(f"Error: {ve}")
```

OUTPUT

```
Enter the initial row positions for each queen (0 to 3) for each column (total 4 queens):
Enter the row position for queen in column 0 (0-3): 1
Enter the row position for queen in column 1 (0-3): 3
Enter the row position for queen in column 2 (0-3): 0
Enter the row position for queen in column 3 (0-3): 2
Enter the maximum number of iterations: 5
Initial Board:
. . Q .
Q . . .
. . . Q
. Q . .

Final Board:
. . Q .
Q . . .
. . . Q
. Q . .

Best Solution: [1, 3, 0, 2], Conflicts: 0
Cost (Number of Conflicts): 0
Solution found in 1 iterations

Solution found with no conflicts!
```



DATE - 29/10/2024

SIMULATED ANNEALING ALGORITHM

PROGRAM

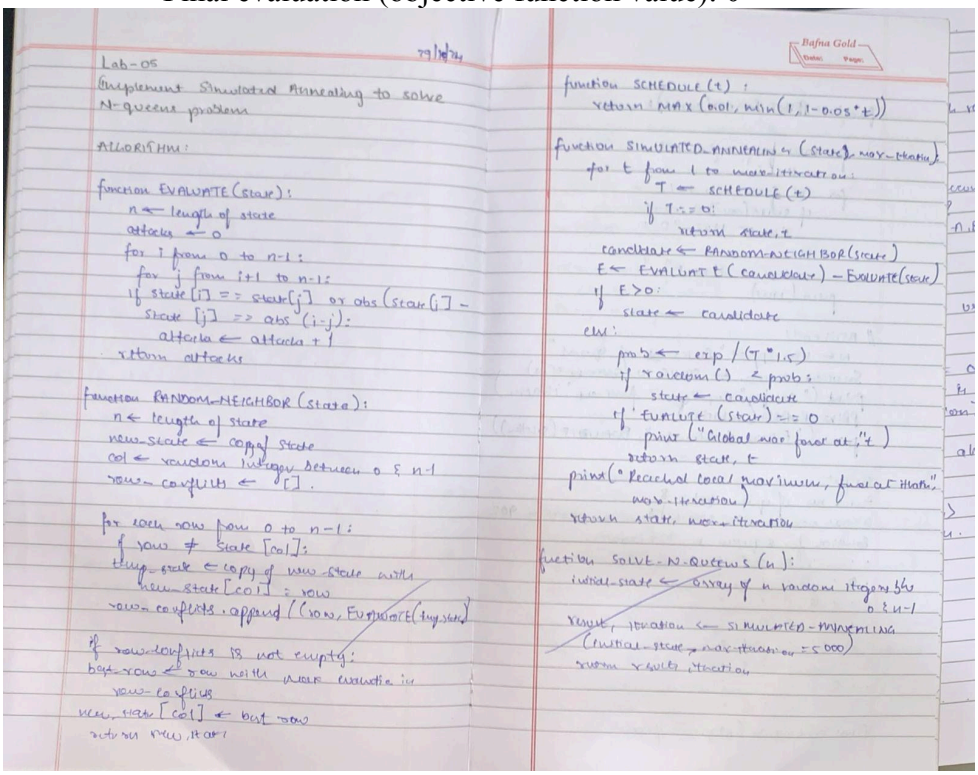
```
import random
import math
def eval_function(state):
    n = len(state)
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):

            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return -attacks


def random_neighbor(state):
    n = len(state)
    new_state = state[:]
    col = random.randint(0, n - 1)
    new_row = random.choice([i for i in range(n) if i != state[col]])
    return new_state


def schedule(t):
    #return max(0.01, min(1, 1 - math.log(t + 1) * 0.001))  # Adjust normal decay factor as needed
(gives large iterations)
    return max(0.01, min(1, 1 - 0.005 * t))  # Faster decay


# Simulated Annealing function
def simulated_annealing(state, schedule, eval_function, random_neighbor,
max_iterations=10000):
    for t in range(1, max_iterations + 1):
        T = schedule(t)
        if T == 0:
            return state, t

        candidate = random_neighbor(state)
        E = eval_function(candidate) - eval_function(state)

        if E > 0:
            state = candidate
        else:
            #prob = math.exp(E / T) normal prob for large iterations
            prob = math.exp(E / (T * 2))  # Decrease the probability of accepting worse solutions
```

```
            if random.random() < prob:
                state = candidate


        if eval_function(state) == 0:
            print(f"Global maximum (no attacks) found at iteration {t}")
            return state, t

    print(f"Reached local maximum/minimum at iteration {max_iterations}")
    return state, max_iterations


def solve_n_queens(n):

    initial_state = [random.randint(0, n - 1) for _ in range(n)]
            result, iterations = simulated_annealing(initial_state, schedule, eval_function,
random_neighbor)
    return result, iterations


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

N = int(input("Enter the number of queens (N): "))
solution, iterations = solve_n_queens(N)
print(f"Solution for {N} queens found in {iterations} iterations:")
print_board(solution)
print("Final evaluation (objective function value):", eval_function(solution))
```

OUTPUT:

Enter the number of queens (N): 8
Global maximum (no attacks) found at iteration 902
Solution for 8 queens found in 902 iterations:
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . Q
. . Q . . . . .
Q . . . .
. . . . . Q .

Final evaluation (objective function value): 0


Lab-05
Implement Simulated Annealing to solve N-queens problem

ALGORITHM:

function EVALUATE(state):
n ← length of state
attacks ← 0
for i from 0 to n-1:
for j from i+1 to n-1:
if state[i] == state[j] or abs(state[i] - state[j]) == abs(i-j):
attacks ← attacks + 1
return attacks

function RANDOM_NEIGHBOR(state):
n ← length of state
new_state ← copy of state
col ← random integer between 0 & n-1
row_conflicts ← [].

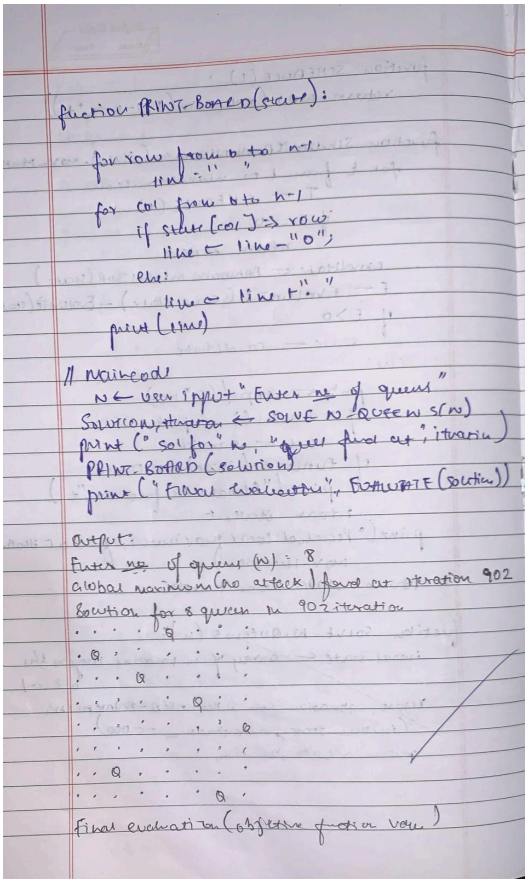
for each row from 0 to n-1:
if row ≠ state[col]:
temp_state ← copy of new_state with new_state[col] = row
row_conflicts.append((row, EVALUATE(temp_state)))

if row_conflicts is not empty:
best_row ← row with least evaluation in row_conflicts
new_state[col] ← best_row
return new_state

function SCHEDULE(t):
return MAX(0.01, min(1, 1 - 0.05*t))

function SIMULATED_ANNEALING(state, max_iterations):
for t from 1 to max_iterations:
T ← SCHEDULE(t)
if T == 0:
return state, t
candidate ← RANDOM_NEIGHBOR(state)
E ← EVALUATE(candidate) - EVALUATE(state)
if E > 0:
state ← candidate
else:
prob ← exp(E / (T * 1.5))
if random() < prob:
state ← candidate
if EVALUATE(state) == 0:
print("Global max found at", t)
return state, t
print("Reached local maximum, not at max", max_iteration)
return state, max_iteration

function SOLVE_N_QUEENS(n):
initial_state ← array of n random integers between 0 & n-1
result, iteration ← SIMULATED_ANNEALING(initial_state, max_iterations=5000)
return result, iteration


function PRINT_BOARD(state):
for row from 0 to n-1:
line = ". "
for col from 0 to n-1:
if state[col] == row:
line ← line + "Q "
else:
line ← line + ". "
print(line)

// Main code
N ← user input "Enter no. of queens"
solution, iterations ← SOLVE_N_QUEENS(N)
print("Sol for", N, "queens found at", iteration)
PRINT_BOARD(solution)
print("Final evaluation", EVALUATE(solution))

Output:
Enter no. of queens (N): 8
Global maximum (no attack) found at iteration 902
Solution for 8 queens in 902 iteration
. . . . Q . . .
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . . . Q
. . Q . . . . .
Q . . . .
. . . . . Q .
Final evaluation (objective function value)

DATE - 12/11/2024

KB ENTAILMENT ALGORITHM

PROGRAM:

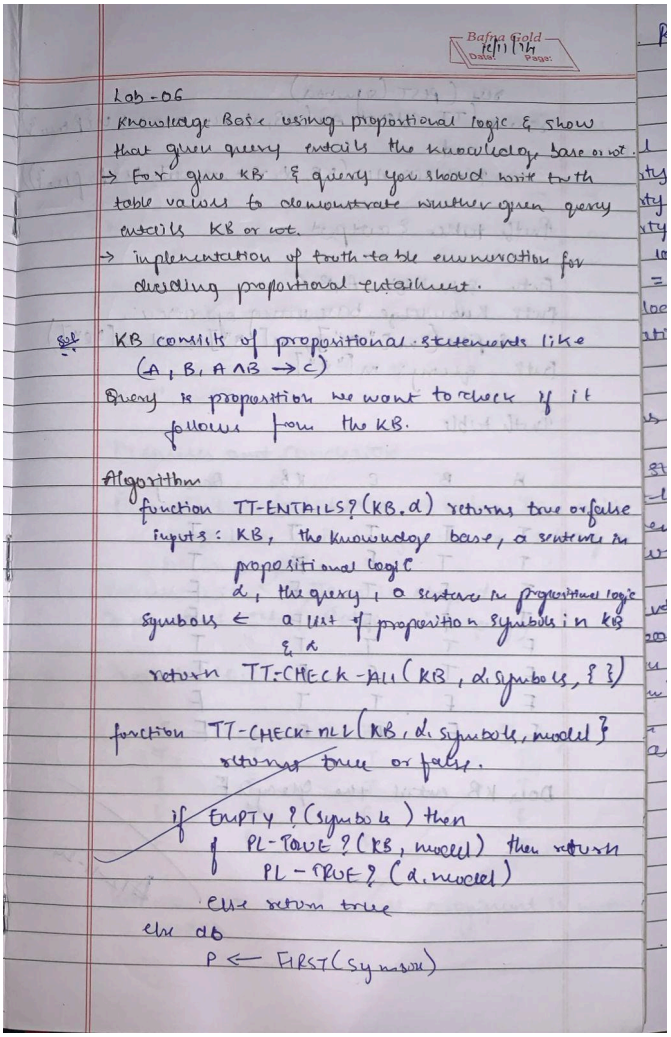
```
from itertools import product
def implies(p, q):
    return not p or q

def iff(p, q):
    return p == q

def tt_entails(kb, query, symbols):
    return all(pl_true(kb, model) <= pl_true(query, model) for model in all_models(symbols))

def all_models(symbols):
    return [dict(zip(symbols, values)) for values in product([True, False], repeat=len(symbols))]

def pl_true(statement, model):
    return statement(model)

symbols = input("Enter the symbols (separated by commas): ").split(',')
symbols = [symbol.strip() for symbol in symbols]

kb_input = input("Enter the knowledge base using operators (e.g., 'implies(m[\"A\"], m[\"B\"]) and m[\"C\"]'): ")
kb = eval(f"lambda m: {kb_input}")

query_input = input("Enter the query (e.g., 'm[\"B\"]'): ")
query = eval(f"lambda m: {query_input}")

print("\nTruth Table:")
print(" | ".join(symbols + ["KB", "Query"]))
print("-" * (len(symbols) * 5 + 15))

for model in all_models(symbols):
    kb_value = pl_true(kb, model)
    query_value = pl_true(query, model)
    values = [model[symbol] for symbol in symbols] + [kb_value, query_value]
    print(" | ".join(str(v) for v in values))

result = tt_entails(kb, query, symbols)
print("\nDoes KB entail the query?", result)
```

Enter the symbols (separated by commas): A,B,C
Enter the knowledge base using operators (e.g., 'implies(m["A"], m["B"]) and m["C"]'): implies(m["A"],m["B"]) and m["C"]
Enter the query (e.g., 'm["B"]'): m["B"]

Truth Table:

A	B	C	KB	Query
True	True	True	True	True
True	True	False	False	True
True	False	True	False	False
True	False	False	False	False
False	True	True	True	True
False	True	False	False	True
False	False	True	True	False
False	False	False	False	False

Does KB entail the query? False



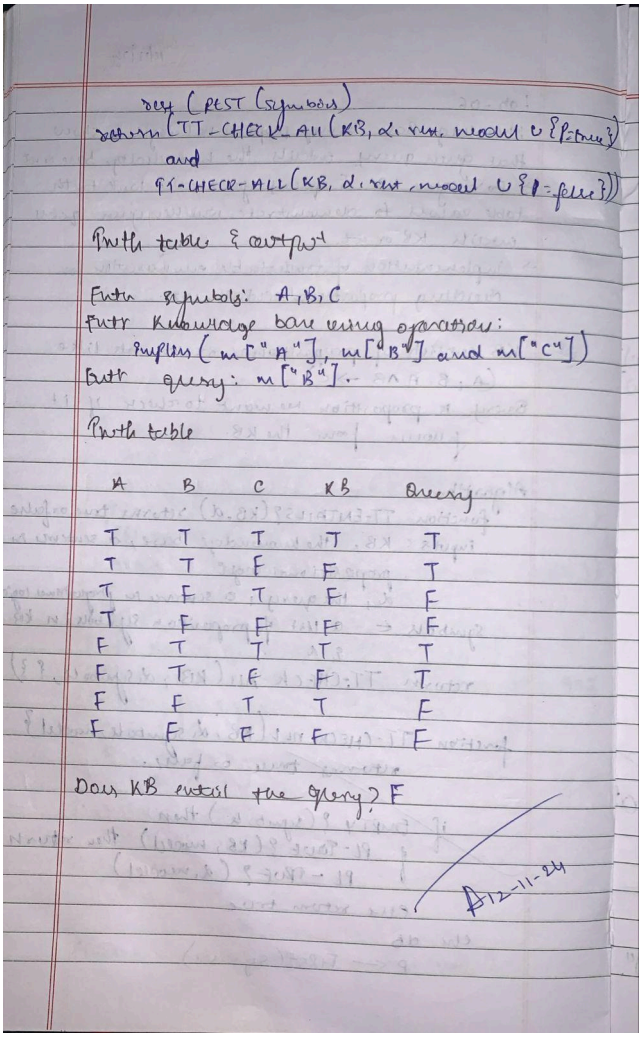
Lab-06
Knowledge Base using propositional logic & show that given query entails the knowledge base or not.
→ For given KB & query you should write truth table values to demonstrate whether given query entails KB or not.
→ implementation of truth-table enumeration for deciding propositional entailment.

KB consists of propositional statements like (A, B, A ∧ B → C)
Query is proposition we want to check if it follows from the KB.

Algorithm
function TT-ENTAILS?(KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic
symbols ← a list of proposition symbols in KB & α
return TT-CHECK-ALL(KB, α, symbols, {})

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false.

if EMPTY?(symbols) then
if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
else return true
else do
P ← FIRST(symbols)



rest ← REST(symbols)
return (TT-CHECK-ALL(KB, α, rest, model ∪ {P=true}) and TT-CHECK-ALL(KB, α, rest, model ∪ {P=false}))

Truth table & output

Enter symbols: A,B,C
Enter Knowledge base using operators: implies(m["A"], m["B"] and m["C"])
Enter query: m["B"].

Truth table

A	B	C	KB	Query
T	T	T	T	T
T	T	F	F	T
T	F	T	F	F
T	F	F	F	F
F	T	T	T	T
F	T	F	F	T
F	F	T	T	F
F	F	F	F	F

Does KB entail the query? F

12-11-24

DATE - 19/11/24

FIRST ORDER LOGIC ALGORITHM

PROGRAM:

```
def validate_and_transform(sentence):
    # Normalize the sentence for processing
    sentence = sentence.strip().lower()

    # Initialize variables
    valid = True
    fol_representation = None

    # Logical consistency checks and transformations
    if "and" in sentence and "are both students" in sentence:
        # Handle sentences like "John and Mary are both students"
        parts = sentence.split(" and ")
        if len(parts) == 2 and "are both students" in parts[1]:
            names = parts[0].split(" ")
            if len(names) == 1:  # Case like "John and Mary"
                name1 = names[0].capitalize()
                name2 = parts[1].replace("are both students", "").strip().capitalize()
            else:
                name1 = names[0].capitalize()
                name2 = names[1].capitalize()
            fol_representation = f"Student({name1}) & Student({name2})"

    # Handle "is human" type sentences
    elif "is human" in sentence:
        name = sentence.split(" ")[0]
        fol_representation = f"Human({name.capitalize()})"

    # Handle "loves" type sentences
    elif "loves" in sentence and "someone" not in sentence:
        parts = sentence.split(" loves ")
        if len(parts) == 2:
            fol_representation = f"Loves({parts[0].capitalize()}, {parts[1].capitalize()})"
        else:
            valid = False

    # Handle "is mortal" type sentences
    elif "is mortal" in sentence:
        name = sentence.split(" ")[0]
        fol_representation = f"Human({name.capitalize()}) -> Mortal({name.capitalize()})"

    # Handle "are animals" type sentences
    elif "are animals" in sentence:
```

```
        subject = sentence.split(" ")[0]
        fol_representation = f"forall x ({subject.capitalize()}(x) -> Animal(x))"

    # Handle "are brown" type sentences
    elif "are brown" in sentence:
        subject = sentence.split(" ")[0]
        fol_representation = f"exists x ({subject.capitalize()}(x) & Brown(x))"

    # Handle "is the mother of" type sentences
    elif "is the mother of" in sentence:
        parts = sentence.split(" is the mother of ")
        if len(parts) == 2:
            fol_representation = f"Mother({parts[0].capitalize()}, {parts[1].capitalize()})"
        else:
            valid = False

    # Handle "is raining" type sentences
    elif "is raining" in sentence:
        fol_representation = "Raining -> Wet(Ground)"

    # Handle "loves someone" type sentences
    elif "loves someone" in sentence:
        name = sentence.split(" ")[0]
        fol_representation = f"exists x (Loves({name.capitalize()}, x))"

    # Handle "there is no person who is both" type sentences
    elif "there is no person who is both" in sentence:
        parts = sentence.replace("there is no person who is both ", "").split(" and ")
        if len(parts) == 2:
            fol_representation = f"¬exists x ({parts[0].capitalize()}(x) & {parts[1].capitalize()}(x))"
        else:
            valid = False

    # Invalid sentence if no patterns match
    else:
        valid = False

    # Return validity and FOL representation
    if valid and fol_representation:
        return f"Sentence is valid.\nFOL Representation: {fol_representation}"
    else:
        return "Sentence is invalid or not recognized!"

# Main program to take input
sentence = input("Enter a sentence: ")
print(validate_and_transform(sentence))
```

OUTPUT:

sentence 1:
Enter a sentence: john is human
FOL Representation: Human(John)

sentence 2:
Enter a sentence: no person is bachelor and married
Sentence is valid.
FOL Representation: ¬exists x (Bachelor(x) & Married(x))

sentence 3:

Enter a sentence: Mary is the mother of john
Sentence is valid.
FOL Representation: Mother(Mary, John)

sentence 4:

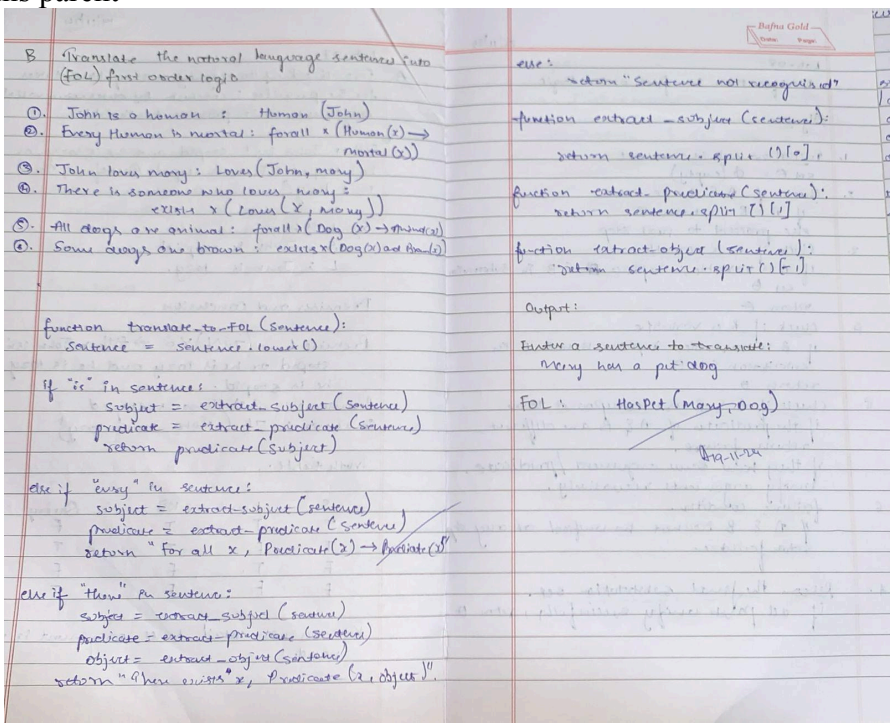
Enter a sentence: john and mary are both students
Sentence is valid.
FOL Representation: Student(John) & Student(Mary)

sentence 5:

Enter a sentence: if it is raining, then the ground is wet
Sentence is valid.
FOL Representation: Raining -> Wet(Ground)

sentence 6:

Enter a sentence: every man respects his parent
Sentence is invalid or not recognized!



DATE - 26/11/24

UNIFICATION IN FOL

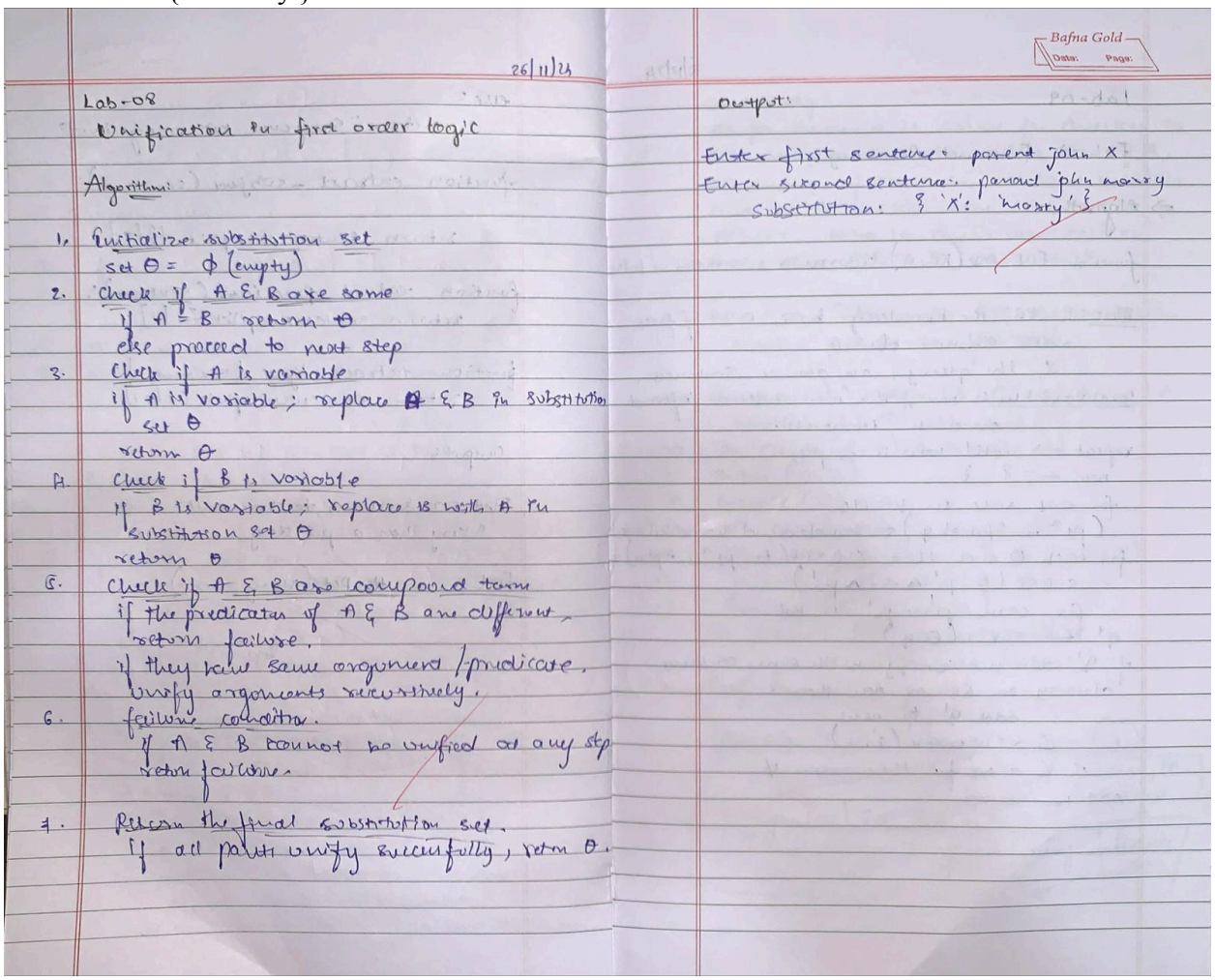
PROGRAM

```
def is_variable(term):
    return isinstance(term, str) and term.islower()

def unify_sentences(s1, s2, theta=None):
    if theta is None:
        theta = {}

    if s1 == s2:
        return theta
    elif is_variable(s1):
        return unify_variable(s1, s2, theta)
    elif is_variable(s2):
        return unify_variable(s2, s1, theta)
    elif isinstance(s1, list) and isinstance(s2, list) and len(s1) == len(s2):
        for a, b in zip(s1, s2):
            theta = unify_sentences(a, b, theta)
            if theta is None:
                return None
        return theta
    else:
        return None

def unify_variable(var, value, theta):
    if var in theta:
        return unify_sentences(theta[var], value, theta)
    elif value in theta:
        return unify_sentences(var, theta[value], theta)
    else:
        theta[var] = value
        return theta

# Get input from the user
def get_input(prompt):
    return input(prompt)

# Parse input into a list representation
def parse_sentence(sentence):
    return sentence.strip().split()

# Main execution
sentence1 = parse_sentence(get_input("Enter the first sentence (e.g., Parent john x): "))
sentence2 = parse_sentence(get_input("Enter the second sentence (e.g., Parent john mary): "))
```

```
result = unify_sentences(sentence1, sentence2)
if result:
    print("Substitution:", result)
else:
    print("Unification failed.")
```

OUTPUT:

```
Enter the first sentence (e.g., Parent john x): parent john x
Enter the second sentence (e.g., Parent john mary): parent john mary
Substitution: {'x': 'mary'}
```



26/11/24

Lab-08

Unification in first order logic

Algorithm:

1. Initialize substitution set
   set θ = ϕ (empty)
2. Check if A & B are same
   if A = B return θ
   else proceed to next step
3. Check if A is variable
   if A is variable; replace A & B in substitution set θ
   return θ
4. Check if B is variable
   if B is variable; replace it with A in substitution set θ
   return θ
5. Check if A & B are compound term
   if the predicates of A & B are different, return failure.
   if they have same argument / predicate, unify arguments recursively.
6. failure condition.
   if A & B cannot be unified at any step return failure.
7. Return the final substitution set.
   if all pairs unify successfully, return θ.

Bafna Gold
Date: Page:

Output:

Enter first sentence: parent john X
Enter second sentence: parent john mary
Substitution: {'X': 'mary'}

DATE - 3/12/2024

FORWARD CHAINING IN FOL

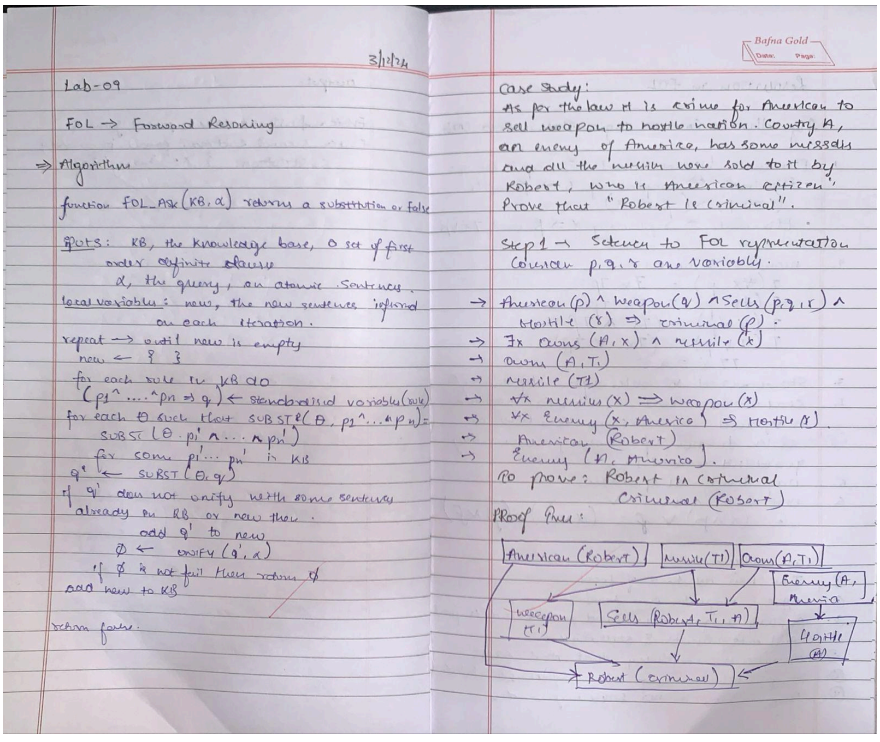
PROGRAM:

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def forward_chain(self):
        new_facts = set(self.facts)
        while True:
            inferred = set()
            for rule in self.rules:
                conclusion, premises = rule
                if all(premise in self.facts for premise in premises):
                    if conclusion not in self.facts:
                        inferred.add(conclusion)
                        self.facts.add(conclusion)
            if not inferred:
                break
            new_facts.update(inferred)
        return new_facts


def main():
    kb = KnowledgeBase()

    kb.add_fact("American(Robert)")
    kb.add_fact("Enemy(A, America)")
    kb.add_fact("Missile(T1)")
    kb.add_fact("Owns(A, T1)")

    kb.add_rule(("Weapon(T1)", ["Missile(T1)"]))
    kb.add_rule(("Hostile(A)", ["Enemy(A, America)"]))
    kb.add_rule(("Sells(Robert, T1, A)", ["Missile(T1)", "Owns(A, T1)"]))
        kb.add_rule(("Criminal(Robert)", ["American(Robert)", "Weapon(T1)", "Sells(Robert, T1,
A)", "Hostile(A)"]))

    print("Starting forward chaining...")
```

```
    inferred_facts = kb.forward_chain()

    print("\nInferred Facts:")
    for fact in inferred_facts:
        print(fact)

    if "Criminal(Robert)" in inferred_facts:
        print("\nConclusion: Robert is a criminal.")
    else:
        print("\nConclusion: Robert is NOT a criminal.")


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

OUTPUT:

```
Starting forward chaining...

Inferred Facts:
Sells(Robert, T1, A)
Missile(T1)
Criminal(Robert)
Weapon(T1)
Owns(A, T1)
Hostile(A)
American(Robert)
Enemy(A, America)
Conclusion: Robert is a criminal.
=== Code Execution Successful ===
```



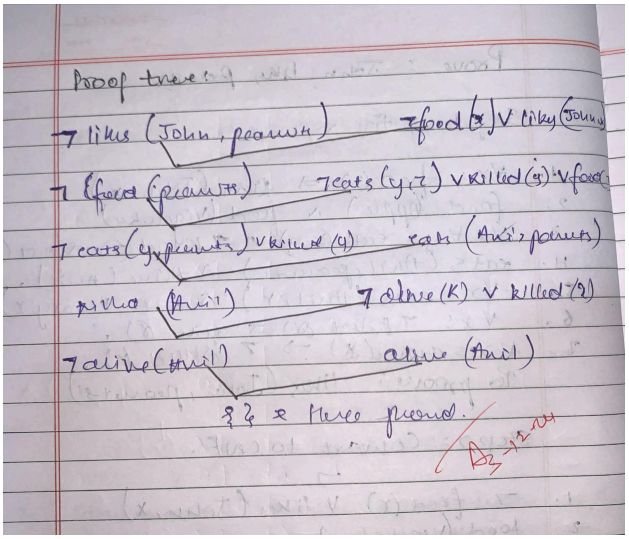
DATE - 3/12/24

RESOLUTION IN FOL

PROGRAM:

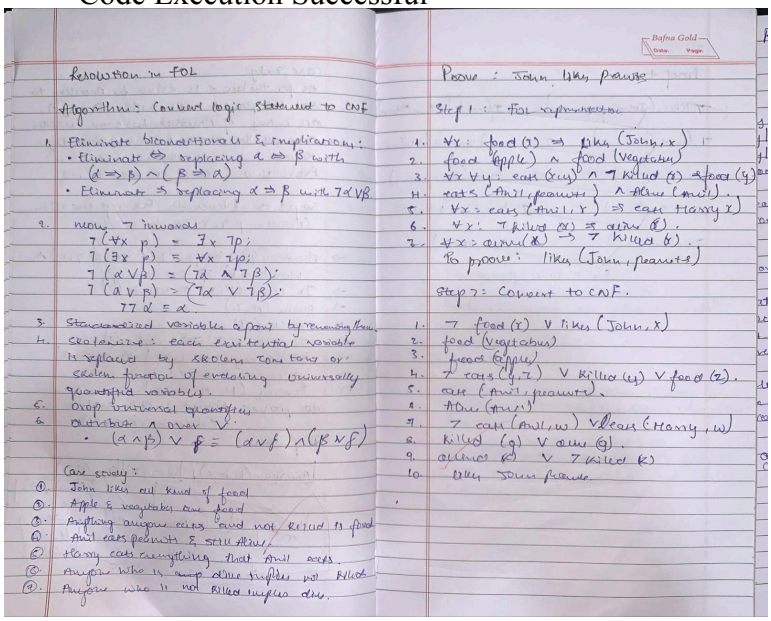
```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def forward_chain(self):
        new_facts = set(self.facts)
        while True:
            inferred = set()
            for rule in self.rules:
                conclusion, premises = rule
                if all(premise in self.facts for premise in premises):
                    if conclusion not in self.facts:
                        inferred.add(conclusion)
                        self.facts.add(conclusion)
            if not inferred:
                break
            new_facts.update(inferred)
        return new_facts


def main():
    kb = KnowledgeBase()

    # Adding initial facts based on the case study
    kb.add_fact("Likes(John, Food)")  # John likes all kinds of food
    kb.add_fact("Food(Apple)")  # Apple is food
    kb.add_fact("Food(Vegetables)")  # Vegetables are food
    kb.add_fact("Eats(Anil, Peanuts)")  # Anil eats peanuts
    kb.add_fact("Alive(Anil)")  # Anil is alive
    kb.add_fact("Eats(Harry, Peanuts)")  # Harry eats peanuts (same as Anil)
    kb.add_fact("Food(Peanut)")  # Explicitly stating that peanuts are food

    # Adding rules (similar to those in the case study)
     kb.add_rule(("Food(FoodItem)", ["Eats(Anil, FoodItem)", "Alive(Anil)"]))  # If Anil eats a
food item and is alive, the food item is food
```

```
    kb.add_rule(("Likes(John, Peanuts)", ["Food(Peanut)"]))  # If peanuts are food, John likes peanuts

    # Forward chaining to infer new facts
    inferred_facts = kb.forward_chain()

    print("Inferred Facts:")
    for fact in inferred_facts:
        print(fact)

    # Check for the conclusion: Does John like Peanuts?
    if "Likes(John, Peanuts)" in inferred_facts:
        print("\nConclusion: John likes Peanuts.")
    else:
        print("\nConclusion: John does not like Peanuts.")


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



OUTPUT:

```
Inferred Facts:
Alive(Anil)
Food(Apple)
Eats(Harry, Peanuts)
Likes(John, Peanuts)
Food(Peanut)
Likes(John, Food)
Eats(Anil, Peanuts)
Food(Vegetables)
Conclusion: John likes Peanuts.
=== Code Execution Successful ===
```



DATE - 17/12/2024

ALPHA-BETA PRUNING ALGORITHM

PROGRAM:

```
import math
import random

# Function to print the Tic-Tac-Toe board
def print_board(board):
    print("Current Board:")
    for row in board:
        print(" | ".join(row))
    print()

# Check for a terminal state (win or draw)
def is_terminal(board):
    for row in board:
        if row.count(row[0]) == 3 and row[0] != " ":
            return True
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != " ":
            return True
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":
        return True
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":
        return True
    return all(board[r][c] != " " for r in range(3) for c in range(3))  # Draw

# Evaluate the utility of terminal states
def evaluate(board):
    for row in board:
        if row.count("X") == 3:
            return 10  # AI wins
        elif row.count("O") == 3:
            return -10  # User wins
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col]:
            if board[0][col] == "X":
                return 10
            elif board[0][col] == "O":
                return -10
    if board[0][0] == board[1][1] == board[2][2]:
        return 10 if board[0][0] == "X" else -10 if board[0][0] == "O" else 0
    if board[0][2] == board[1][1] == board[2][0]:
        return 10 if board[0][2] == "X" else -10 if board[0][2] == "O" else 0
    return 0  # Draw
```

```
# Alpha-Beta Pruning Algorithm
def alpha_beta(board, depth, alpha, beta, is_maximizing, pruned_nodes):
    if is_terminal(board):
        return evaluate(board)

    if is_maximizing:  # AI
        best_value = -math.inf
        for r in range(3):
            for c in range(3):
                if board[r][c] == " ":
                    board[r][c] = "X"
                    value = alpha_beta(board, depth + 1, alpha, beta, False, pruned_nodes)
                    board[r][c] = " "  # Undo
                    best_value = max(best_value, value)
                    alpha = max(alpha, value)
                    if beta <= alpha:
                        pruned_nodes.append((r, c))
                        return best_value
        return best_value
    else:  # User
        best_value = math.inf
        for r in range(3):
            for c in range(3):
                if board[r][c] == " ":
                    board[r][c] = "O"
                    value = alpha_beta(board, depth + 1, alpha, beta, True, pruned_nodes)
                    board[r][c] = " "  # Undo
                    best_value = min(best_value, value)
                    beta = min(beta, value)
                    if beta <= alpha:
                        pruned_nodes.append((r, c))
                        return best_value
        return best_value

# Find the best move for AI
def find_best_move(board):
    best_value = -math.inf
    best_move = (-1, -1)
    pruned_nodes = []

    preferred_moves = [(1, 1), (0, 0), (0, 2), (2, 0), (2, 2), (0, 1), (1, 0), (1, 2), (2, 1)]
    for r, c in preferred_moves:
        if board[r][c] == " ":
            board[r][c] = "X"
            move_value = alpha_beta(board, 0, -math.inf, math.inf, False, pruned_nodes)
```

```
            board[r][c] = " "
            if move_value > best_value:
                best_value = move_value
                best_move = (r, c)

    # Limit the output of pruned nodes to 5 or 6
    displayed_pruned_nodes = pruned_nodes[:6]
     print(f"Pruned Nodes: {displayed_pruned_nodes}{'...and more' if len(pruned_nodes) > 6 else
"}")
    return best_move

# Map single user input to board coordinates
def input_to_coordinates(cell):
    mapping = {
        0: (0, 0), 1: (0, 1), 2: (0, 2),
        3: (1, 0), 4: (1, 1), 5: (1, 2),
        6: (2, 0), 7: (2, 1), 8: (2, 2)
    }
    return mapping.get(cell)

# Function for AI to choose random position for the first move
def ai_first_move(board):
    available_positions = [(r, c) for r in range(3) for c in range(3) if board[r][c] == " "]
    return random.choice(available_positions)  # Randomly select from available positions

# Main function
def play_tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic-Tac-Toe with Alpha-Beta Pruning!")
    print_board(board)

    # AI's first move (randomly selected)
    print("AI's First Move (X):")
    r, c = ai_first_move(board)
    board[r][c] = "X"
    print_board(board)

    while not is_terminal(board):
        # User's move
        while True:
            try:
                user_input = int(input("Your Turn (O): Enter a number (0-8): "))
                if 0 <= user_input <= 8:
                    r, c = input_to_coordinates(user_input)
                    if board[r][c] == " ":
                        board[r][c] = "O"
```

```
                    break
                print("Invalid move! Cell is occupied or out of range. Try again.")
            except ValueError:
                print("Invalid input! Enter a number between 0 and 8.")
        print_board(board)
        if is_terminal(board):
            break

        # AI's move (after first random move)
        print("AI's Turn (X):")
        r, c = find_best_move(board)
        board[r][c] = "X"
        print_board(board)

    # Game over
    if evaluate(board) == 10:
        print("AI Wins!")
    elif evaluate(board) == -10:
        print("You Win!")
    else:
        print("It's a Draw!")

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

OUTPUT:

```
Welcome to Tic-Tac-Toe with Alpha-Beta Pruning!
Current Board:
  |   |
  |   |
  |   |

AI's First Move (X):
Current Board:
  |   |
  |   |
X |   |

Your Turn (O): Enter a number (0-8): 0
Current Board:
O |   |
  |   |
X |   |

AI's Turn (X):
```

Pruned Nodes: [(0, 2), (0, 2), (0, 2), (0, 2), (2, 1), (2, 1)]...and more
Current Board:
O | | X
| |
X | |

Your Turn (O): Enter a number (0-8): 4
Current Board:
O | | X
| O |
X | |

AI's Turn (X):
Pruned Nodes: [(1, 2), (1, 2), (1, 0), (2, 1), (2, 1), (1, 2)]...and more
Current Board:
O | | X
| O |
X | | X

Your Turn (O): Enter a number (0-8): 7
Current Board:
O | | X
| O |
X | O | X

AI's Turn (X):
Pruned Nodes: [(0, 1)]
Current Board:
O | | X
| O | X
X | O | X

AI Wins!