VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Siddharth HG (1BM22CS276)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Siddharth H G (1BM22CS276),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

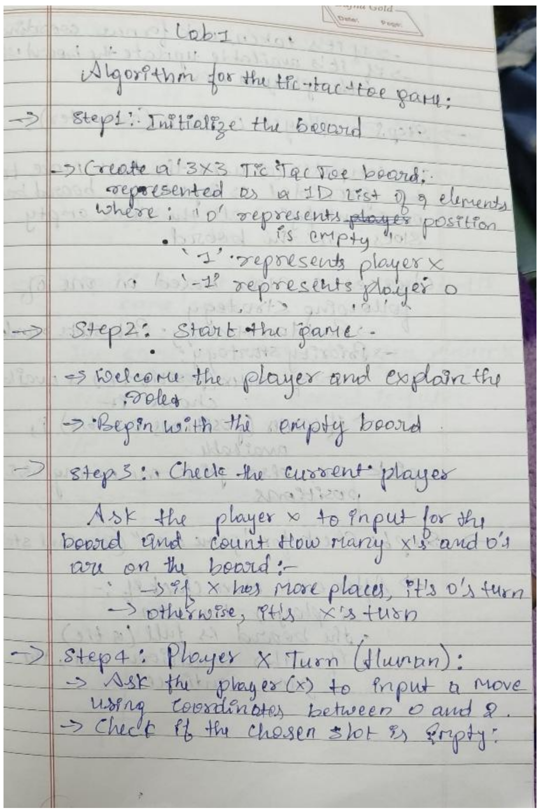
Saritha A N Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/siddharthhg01/artificial_intelligence_lab.git

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



Lab 1

Algorithm for the tic-tac-toe game:

→ Step 1: Initialize the board

→ Create a 3×3 Tic Tac Toe board: represented as a 1D list of 9 elements
where: '0' represents ~~player~~ position is empty
'1' represents player x
'-1' represents player o

→ Step 2: Start the game.

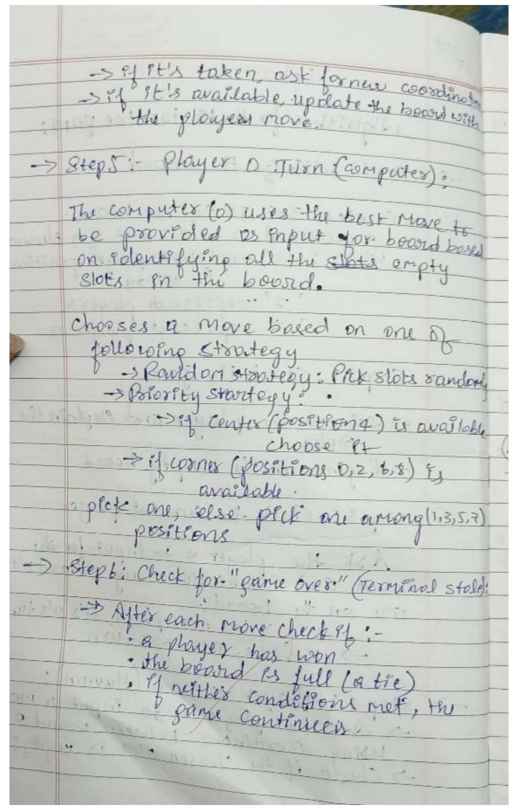
→ Welcome the player and explain the rules

→ Begin with the empty board

→ Step 3: Check the current player

Ask the player x to input for the board and count How many x's and o's are on the board:-
→ if x has more places, it's o's turn
→ otherwise, it's x's turn

→ Step 4: Player X Turn (Human):
→ Ask the player (x) to input a move using coordinates between 0 and 8.
→ Check if the chosen slot is empty:



→ if it's taken, ask for new coordinates
→ if it's available, update the board with the players move.

→ Step 5:- Player O Turn (Computer):

The computer (o) uses the best move to be provided as input for board based on identifying all the ~~slots~~ empty slots in the board.

Chooses a move based on one of following strategy
→ Random strategy: Pick slots randomly
→ Priority strategy:
→ if center (position 4) is available choose it
→ if corner (positions 0, 2, 6, 8) is available.
pick one, else pick one among (1,3,5,7) positions

→ Step 6: Check for "game over" (Terminal state):
→ After each move check if:-
• a player has won
• the board is full (a tie)
• if neither conditions met, the game continues.

Code:

1. Tic Tac Toe

```
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 get_available_moves(board):
    return [(r, c) for r in range(3) for c in range(3) if board[r][c] == ' ']

def play_game():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    player = 'X'

    while True:
        print_board(board)
        winner = check_winner(board)
        if winner:
            print(f"{winner} wins!")
            break
        if not get_available_moves(board):
            print("It's a draw!")
            break

        row, col = map(int, input(f"Player {player}, enter row and column (0-2): ").split())
        if board[row][col] == ' ':
            board[row][col] = player
            player = 'O' if player == 'X' else 'X'
        else:
            print("Invalid move! Try again.")

play_game()
```

output:

```
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter row and column (0-2): 0 1
  | X |
---------
  |   |
---------
  |   |
---------
Player O, enter row and column (0-2): 1 1
  | X |
---------
  | O |
---------
  |   |
---------
Player X, enter row and column (0-2): 2 1
  | X |
---------
  | O |
---------
  | X |
---------
Player O, enter row and column (0-2): 0 0
O | X |
---------
  | O |
---------
  | X |
---------
Player X, enter row and column (0-2): 1 2
O | X |
---------
  | O | X
---------
  | X |
---------
Player O, enter row and column (0-2): 2 2
O | X |
---------
  | O | X
---------
  | X | O
---------
O wins!
```

2. vacuum cleaner agent
Alggorithm:

Lab 1

Algorithm for the tic-tac-toe game:

→ Step1: Initialize the board

→ Create a 3X3 Tic Tac Toe board:
represented as a 1D list of 9 elements
where: '0' represents ~~player~~ position is empty
'1' represents player x
'-1' represents player o

→ Step2: Start the game.

→ Welcome the player and explain the rules
→ Begin with the empty board.

→ step3: Check the current player

Ask the player x to input for the board and count how many x's and o's are on the board:-
→ if x has more places, it's o's turn
→ otherwise, it's x's turn

→ Step4: Player X Turn (Human):
→ Ask the player (x) to input a move using coordinates between 0 and 8.
→ Check if the chosen slot is empty.

Code:
#2 Quandrants

```
# Initial state setup
current_state = ['A', 1, 'B', 1]  # Initial state with both rooms dirty
goal_state = ['A', 0, 'B', 0]      # Goal state (both rooms clean)
total_cost = 0                      # Initialize total cost

def print_status():
    print(f"Current state: {current_state}")
    print(f"Vacuum is placed in Location {position}")
    print(f"Total cleaning cost so far: {total_cost}")

def check_and_clean(start_room):
    process_room(start_room, current_state[current_state.index(start_room) + 1])
    other_room = 'B' if start_room == 'A' else 'A'
    process_room(other_room, current_state[current_state.index(other_room) + 1])
    check_goal_state()

def process_room(room, status):
    if status == 1:
        clean_room(room)
    else:
        print(f"Location {room} is already clean.")

def clean_room(room):
    global total_cost
    print(f"Location {room} is Dirty.")
    current_state[current_state.index(room) + 1] = 0  # Set status to 0 (cleaned)
    print(f"Location {room} has been Cleaned.")

    # Increase total cost for cleaning
    cost_of_cleaning = 1  # Define the cost for each cleaning operation
    total_cost += cost_of_cleaning
    print(f"COST for SUCK at Location {room}: {cost_of_cleaning}")

    # Move the vacuum to the next room after cleaning
    move_to_next_room(room)
    print_status()

def move_to_next_room(room):
    global position
    if room == 'A':
        position = 'B'
        print(f"Moving right to Location B.")
    elif room == 'B':
        position = 'A'
        print(f"Moving left to Location A.")
```

```
def check_goal_state():
    if current_state == goal_state:
        print("Final goal state reached:", goal_state)
    else:
        print("Goal state not yet reached.")

def get_room_status():
    for room in ['A', 'B']:
        status = input(f"Is location {room} dirty? (yes/no): ").strip().lower()
        if status == 'yes':
            current_state[current_state.index(room) + 1] = 1  # Set status to 1 (dirty)
        elif status == 'no':
            current_state[current_state.index(room) + 1] = 0  # Set status to 0 (clean)
        else:
            print("Invalid input. Assuming the room is clean.")
            current_state[current_state.index(room) + 1] = 0  # Default to clean

if __name__ == "__main__":
    position = input("Which room do you want to start cleaning? (A/B): ").strip().upper()

    if position in ['A', 'B']:
        get_room_status()  # Ask user for the status of each room
        print_status()
        check_and_clean(position)
    else:
        print("Invalid choice. Please restart and choose either A or B.")

#4 Quandrants
# Initial state setup
current_state = ['A', 1, 'B', 1, 'C', 1, 'D', 1]  # Initial state with all rooms dirty
goal_state = ['A', 0, 'B', 0, 'C', 0, 'D', 0]      # Goal state (all rooms clean)
total_cost = 0                                     # Initialize total cost
position = None                                    # Initial position is not set

def print_status():
    print(f"Current state: {current_state}")
    print(f"Vacuum is placed in Location {position}")
    print(f"Total cleaning cost so far: {total_cost}")

def check_and_clean(start_room):
    process_room(start_room, current_state[current_state.index(start_room) + 1])

    # Check and clean remaining rooms
    for i in range(0, len(current_state), 2):
        room = current_state[i]
        if room != start_room:
```

```
        process_room(room, current_state[i + 1])

    check_goal_state()

def process_room(room, status):
    if status == 1:
        clean_room(room)
    else:
        print(f"Location {room} is already clean.")

def clean_room(room):
    global total_cost
    print(f"Location {room} is Dirty.")
    current_state[current_state.index(room) + 1] = 0  # Set status to 0 (cleaned)
    print(f"Location {room} has been Cleaned.")

    # Increase total cost for cleaning
    cost_of_cleaning = 1  # Define the cost for each cleaning operation
    total_cost += cost_of_cleaning
    print(f"COST for SUCK at Location {room}: {cost_of_cleaning}")

    print_status()

def check_goal_state():
    if current_state == goal_state:
        print("Final goal state reached:", goal_state)
    else:
        print("Goal state not yet reached.")

def get_room_status():
    for room in ['A', 'B', 'C', 'D']:
        status = input(f"Is location {room} dirty? (yes/no): ").strip().lower()
        if status == 'yes':
            current_state[current_state.index(room) + 1] = 1  # Set status to 1 (dirty)
        elif status == 'no':
            current_state[current_state.index(room) + 1] = 0  # Set status to 0 (clean)
        else:
            print("Invalid input. Assuming the room is clean.")
            current_state[current_state.index(room) + 1] = 0  # Default to clean

if __name__ == "__main__":
    position = input("Which room do you want to start cleaning? (A/B/C/D): ").strip().upper()

    if position in ['A', 'B', 'C', 'D']:
        get_room_stantus()  # Ask user for the status of each room
        print_status()
        check_and_clean(position)
```

```
    else:
        print("Invalid choice. Please restart and choose either A, B, C, or D.")
```

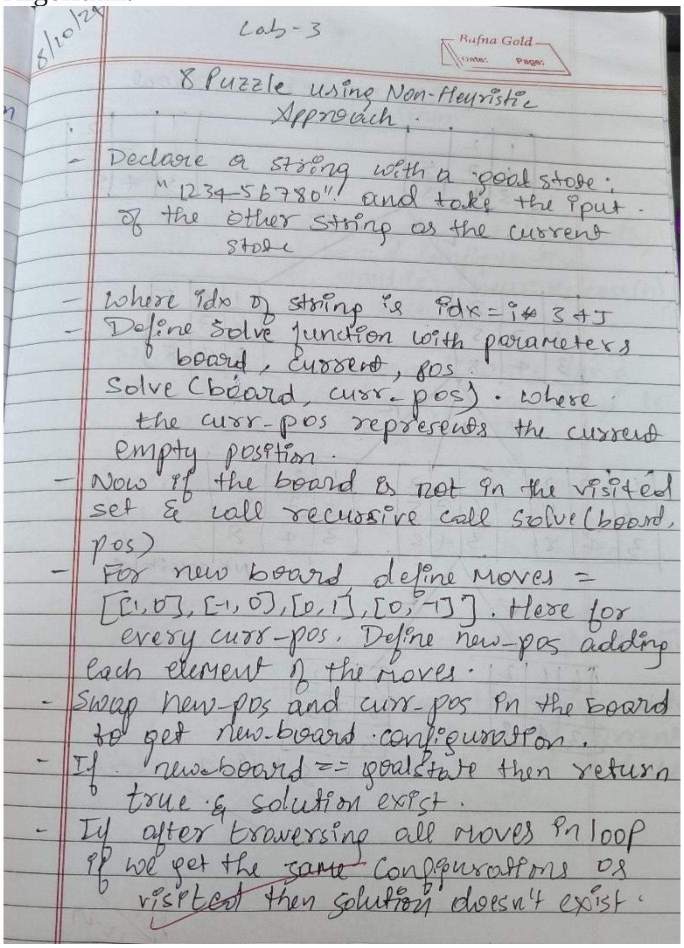
output:
2 Quadrants:

```
Which room do you want to start cleaning? (A/B): A
Is location A dirty? (yes/no): No
Is location B dirty? (yes/no): yes
Current state: ['A', 0, 'B', 1]
Vacuum is placed in Location A
Total cleaning cost so far: 0
Location A is already clean.
Location B is Dirty.
Location B has been Cleaned.
COST for SUCK at Location B: 1
Moving left to Location A.
Current state: ['A', 0, 'B', 0]
Vacuum is placed in Location A
Total cleaning cost so far: 1
Final goal state reached: ['A', 0, 'B', 0]
```

4 Quadrants:

```
Which room do you want to start cleaning? (A/B/C/D): a
Is location A dirty? (yes/no): no
Is location B dirty? (yes/no): no
Is location C dirty? (yes/no): yes
Is location D dirty? (yes/no): yes
Current state: ['A', 0, 'B', 0, 'C', 1, 'D', 1]
Vacuum is placed in Location A
Total cleaning cost so far: 0
Location A is already clean.
Location B is already clean.
Location C is Dirty.
Location C has been Cleaned.
COST for SUCK at Location C: 1
Current state: ['A', 0, 'B', 0, 'C', 0, 'D', 1]
Vacuum is placed in Location A
Total cleaning cost so far: 1
Location D is Dirty.
Location D has been Cleaned.
COST for SUCK at Location D: 1
Current state: ['A', 0, 'B', 0, 'C', 0, 'D', 0]
Vacuum is placed in Location A
Total cleaning cost so far: 2
Final goal state reached: ['A', 0, 'B', 0, 'C', 0, 'D', 0]
```

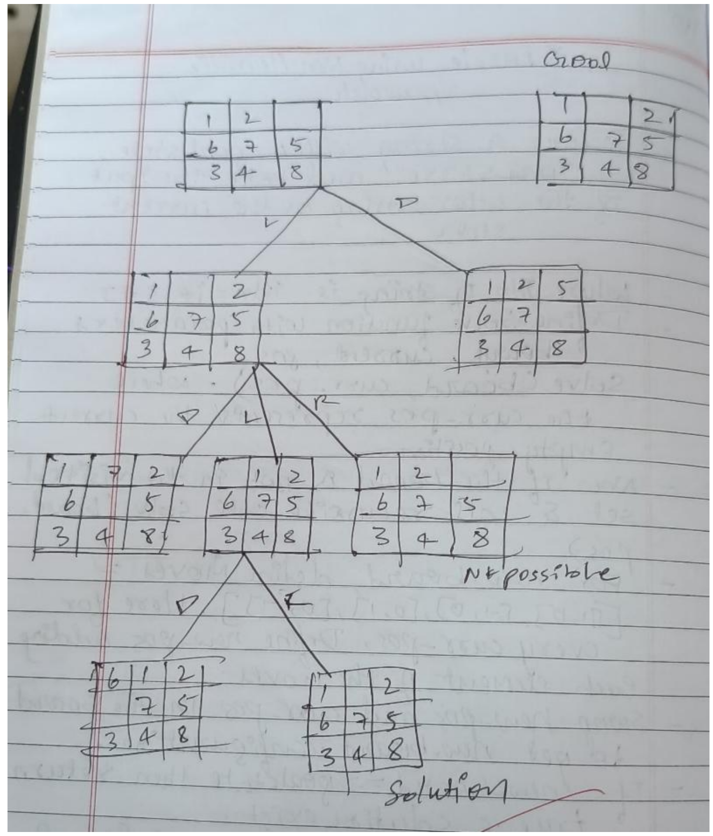
Program 2:
Implement 8 puzzle problems using Depth First Search (DFS)
Algorithm:


Lab-3

8/10/24

8 Puzzle using Non-Heuristic Approach.

- Declare a string with a goal state: "1234-56780" and take the input of the other string as the current state
- where idx of string is idx = i * 3 + J
- Define solve function with parameters board, current, pos
  Solve (board, curr_pos). where the curr_pos represents the current empty position.
- Now if the board is not in the visited set & call recursive call solve(board, pos)
- For new board define Moves = [[1,0],[-1,0],[0,1],[0,-1]]. Here for every curr_pos. Define new_pos adding each element of the moves.
- Swap new_pos and curr_pos in the board to get new-board configuration.
- If new-board == goalstate then return true & solution exist.
- If after traversing all moves in loop if we get the same configurations or visited then solution doesn't exist.



Code:
```
import numpy as np
from collections import deque

class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action

class Puzzle:
    def __init__(self, start, goal):
        self.start = (tuple(map(tuple, start)), self.find_empty(start))
# Dynamically find empty space
        self.goal = (tuple(map(tuple, goal)), self.find_empty(goal))
        self.solution = None
        self.num_explored = 0
        self.total_states_generated = 0  # Counter for total states
generated

    def find_empty(self, state):
```

```
        # Find the position of the empty space (0)
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    return (i, j)

    def neighbors(self, state):
        mat, (row, col) = state
        results = []
        directions = [(1, 0, 'down'), (-1, 0, 'up'), (0, 1, 'right'), (0,
-1, 'left')]

        for dr, dc, action in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:  # Corrected
condition
                mat1 = np.copy(mat)
                # Swap the empty space (0) with the adjacent number
                mat1[row][col], mat1[new_row][new_col] =
mat1[new_row][new_col], mat1[row][col]
                results.append((action, (tuple(map(tuple, mat1)),
(new_row, new_col))))
        return results

    def solve(self):
        start_node = Node(state=self.start, parent=None, action=None)
        queue = deque([start_node])  # Use deque for efficient pops from
the left
        explored = set()  # Use a set for explored states

        while queue:
            node = queue.popleft()  # Dequeue the first node
            self.num_explored += 1

            # Check if the current node's state is the goal
            if node.state[0] == self.goal[0]:
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return  # Found a solution

            # Mark the state as explored
            explored.add(node.state[0])

            # Explore neighbors
            for action, state in self.neighbors(node.state):
                if state[0] not in explored and all(node.state[0] !=
state[0] for node in queue):
                    child = Node(state=state, parent=node, action=action)
                    queue.append(child)  # Enqueue the child node
                    self.total_states_generated += 1  # Increment the
total states generated counter
```

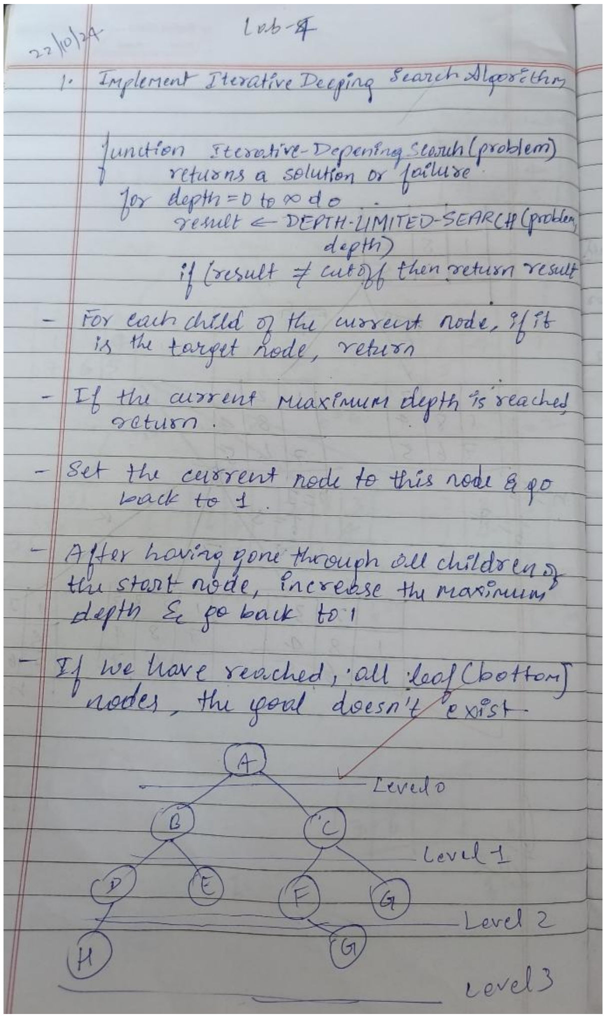
```
    def print_solution(self):
        if self.solution is None:
            print("No solution found.")
            return

        print("Start State:\n", np.array(self.start[0]), "\n")
        print("Goal State:\n", np.array(self.goal[0]), "\n")
        print("\nStates Explored: ", self.num_explored)
        print("Total States Generated: ", self.total_states_generated,
"\n")

        print("Actions Taken to Reach the Goal:\n")
        for action, cell in zip(self.solution[0], self.solution[1]):
            print("Action: ", action)
            print(np.array(cell[0]), "\n")
        print("Goal Reached!!")

# Example usage
start = np.array([[1, 2, 3], [0, 4, 6], [7, 5, 8]])
goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])

p = Puzzle(start, goal)
p.solve()
p.print_solution()
```

output:

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

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


States Explored:  14
Total States Generated:  26

Actions Taken to Reach the Goal:

Action:  right
[[1 2 3]
 [4 0 6]
 [7 5 8]]

Action:  down
[[1 2 3]
 [4 5 6]
 [7 0 8]]

Action:  right
[[1 2 3]
 [4 5 6]
 [7 8 0]]

Goal Reached!!
```

Implement Iterative deepening search algorithm

Algorithm:

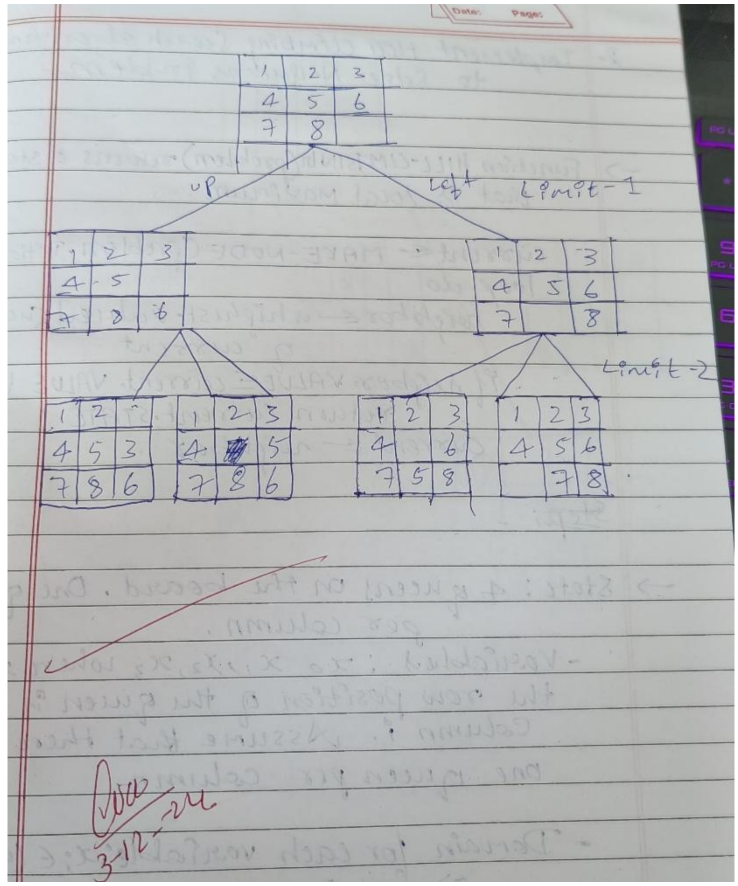


22/10/24 Lab-4

1. Implement Iterative Deeping Search Algorithm

function Iterative-Depening Search(problem) returns a solution or failure
for depth = 0 to ∞ do
result ← DEPTH-LIMITED-SEARCH(problem, depth)
if (result ≠ cutoff then return result

- For each child of the current node, if it is the target node, return
- If the current maximum depth is reached, return.
- Set the current node to this node & go back to 1.
- After having gone through all children of the start node, increase the maximum depth & go back to 1
- If we have reached all leaf (bottom) nodes, the goal doesn't exist.



Code:

```
#DFS
class Puzzle:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.rows = 3
        self.cols = 3

    def get_neighbors(self, state):
        # Find the position of the blank (0)
        zero_pos = [(i, j) for i in range(self.rows) for j in
range(self.cols) if state[i][j] == 0][0]
        x, y = zero_pos

        # Possible directions to move the blank space: up, down, left,
right
        directions = [(-1, 0, 'up'), (1, 0, 'down'), (0, -1, 'left'), (0,
1, 'right')]
        neighbors = []

        for dx, dy, action in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < self.rows and 0 <= new_y < self.cols:
                new_state = [list(row) for row in state]  # Create a copy
of the state
                # Swap blank with the neighboring tile
                new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
                neighbors.append((new_state, action))

        return neighbors

    def dfs(self):
        # Stack stores the current state, the path to the state, and the
actions taken
        stack = [(self.initial_state, [], [])]  # (state, path, actions)
        visited = set()

        while stack:
            current_state, path, actions = stack.pop()

            # If we reached the goal, return the solution
            if current_state == self.goal_state:
                return path + [current_state], actions

            # Mark the current state as visited
            state_tuple = tuple(tuple(row) for row in current_state)
            if state_tuple not in visited:
                visited.add(state_tuple)

                # Explore all neighboring states
                for neighbor, action in
self.get_neighbors(current_state):
                    stack.append((neighbor, path + [current_state],
actions + [action]))
```

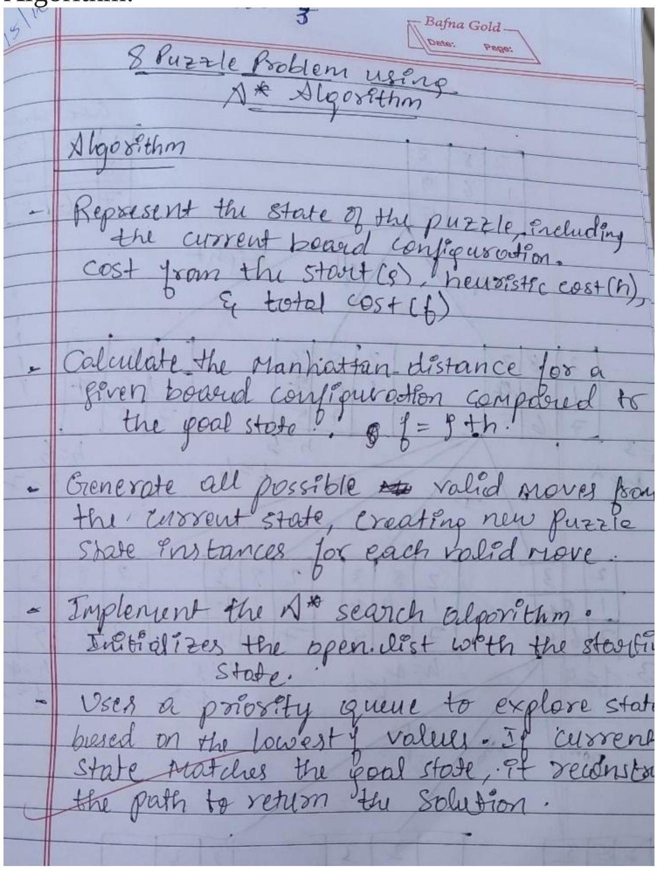
```
        return None, None  # If no solution found

    def print_solution(self, solution, actions):
        if solution:
            print("Solution found!")
            for step, action in zip(solution, actions + ['Goal
Reached!!']):
                for row in step:
                    print(row)
                print(f"Action: {action}\n")
        else:
            print("No solution exists.")

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

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

puzzle = Puzzle(initial_state, goal_state)
solution, actions = puzzle.dfs()
puzzle.print_solution(solution, actions)
```

output:

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

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

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

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

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

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

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Action: Goal Reached!!
```

```
Program 3:
```

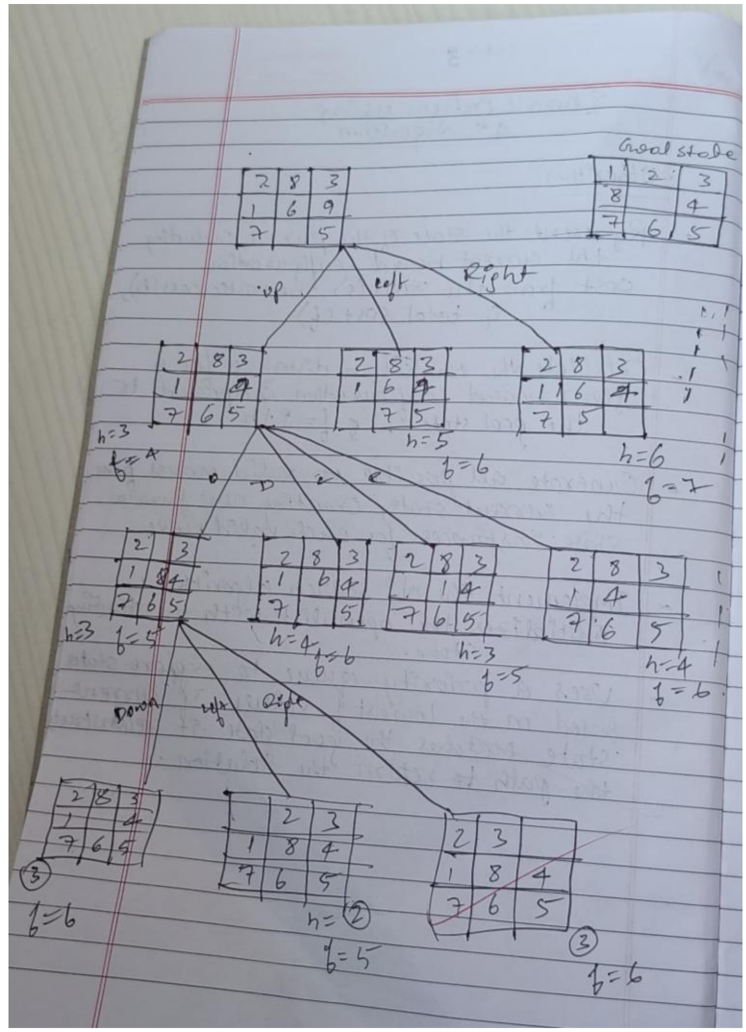
Implement A* search algorithm:
Algorithm:



8 Puzzle Problem using A* Algorithm

Algorithm

- Represent the state of the puzzle, including the current board configuration, cost from the start (g), heuristic cost (h), & total cost (f)
- Calculate the Manhattan distance for a given board configuration compared to the goal state. f = g + h.
- Generate all possible valid moves from the current state, creating new puzzle state instances for each valid move.
- Implement the A* search algorithm. Initializes the open list with the starting state.
- Uses a priority queue to explore states based on the lowest f values. If current state matches the goal state, it reconstructs the path to return the solution.



Code:

```
#BFS
import numpy as np
from collections import deque

class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action

class Puzzle:
    def __init__(self, start, goal):
        self.start = (tuple(map(tuple, start)), self.find_empty(start))
```

```
# Dynamically find empty space
        self.goal = (tuple(map(tuple, goal)), self.find_empty(goal))
        self.solution = None
        self.num_explored = 0
        self.total_states_generated = 0  # Counter for total states
generated

    def find_empty(self, state):
        # Find the position of the empty space (0)
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    return (i, j)

    def neighbors(self, state):
        mat, (row, col) = state
        results = []
        directions = [(1, 0, 'down'), (-1, 0, 'up'), (0, 1, 'right'), (0,
-1, 'left')]

        for dr, dc, action in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:  # Corrected
condition
                mat1 = np.copy(mat)
                # Swap the empty space (0) with the adjacent number
                mat1[row][col], mat1[new_row][new_col] =
mat1[new_row][new_col], mat1[row][col]
                results.append((action, (tuple(map(tuple, mat1)),
(new_row, new_col))))
        return results

    def solve(self):
        start_node = Node(state=self.start, parent=None, action=None)
        queue = deque([start_node])  # Use deque for efficient pops from
the left
        explored = set()  # Use a set for explored states

        while queue:
            node = queue.popleft()  # Dequeue the first node
            self.num_explored += 1

            # Check if the current node's state is the goal
            if node.state[0] == self.goal[0]:
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return  # Found a solution

            # Mark the state as explored
            explored.add(node.state[0])
```

```
            # Explore neighbors
            for action, state in self.neighbors(node.state):
                if state[0] not in explored and all(node.state[0] !=
state[0] for node in queue):
                    child = Node(state=state, parent=node, action=action)
                    queue.append(child)  # Enqueue the child node
                    self.total_states_generated += 1  # Increment the
total states generated counter

    def print_solution(self):
        if self.solution is None:
            print("No solution found.")
            return

        print("Start State:\n", np.array(self.start[0]), "\n")
        print("Goal State:\n", np.array(self.goal[0]), "\n")
        print("\nStates Explored: ", self.num_explored)
        print("Total States Generated: ", self.total_states_generated,
"\n")

        print("Actions Taken to Reach the Goal:\n")
        for action, cell in zip(self.solution[0], self.solution[1]):
            print("Action: ", action)
            print(np.array(cell[0]), "\n")
        print("Goal Reached!!")

# Example usage
start = np.array([[1, 2, 3], [0, 4, 6], [7, 5, 8]])
goal = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 0]])

p = Puzzle(start, goal)
p.solve()
p.print_solution()
```

output:

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

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


States Explored:  14
Total States Generated:  26

Actions Taken to Reach the Goal:

Action:  right
[[1 2 3]
 [4 0 6]
 [7 5 8]]

Action:  down
[[1 2 3]
 [4 5 6]
 [7 0 8]]

Action:  right
[[1 2 3]
 [4 5 6]
 [7 8 0]]

Goal Reached!!
```

```
#DFS
class Puzzle:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.rows = 3
        self.cols = 3

    def get_neighbors(self, state):
        # Find the position of the blank (0)
        zero_pos = [(i, j) for i in range(self.rows) for j in
range(self.cols) if state[i][j] == 0][0]
        x, y = zero_pos

        # Possible directions to move the blank space: up, down, left,
right
        directions = [(-1, 0, 'up'), (1, 0, 'down'), (0, -1, 'left'), (0,
1, 'right')]
        neighbors = []

        for dx, dy, action in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < self.rows and 0 <= new_y < self.cols:
                new_state = [list(row) for row in state]  # Create a copy
of the state
                # Swap blank with the neighboring tile
                new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]
                neighbors.append((new_state, action))

        return neighbors

    def dfs(self):
        # Stack stores the current state, the path to the state, and the
actions taken
        stack = [(self.initial_state, [], [])]  # (state, path, actions)
        visited = set()

        while stack:
            current_state, path, actions = stack.pop()

            # If we reached the goal, return the solution
            if current_state == self.goal_state:
                return path + [current_state], actions

            # Mark the current state as visited
            state_tuple = tuple(tuple(row) for row in current_state)
            if state_tuple not in visited:
                visited.add(state_tuple)

                # Explore all neighboring states
                for neighbor, action in
self.get_neighbors(current_state):
                    stack.append((neighbor, path + [current_state],
actions + [action]))

        return None, None  # If no solution found
```

```
    def print_solution(self, solution, actions):
        if solution:
            print("Solution found!")
            for step, action in zip(solution, actions + ['Goal
Reached!!']):
                for row in step:
                    print(row)
                print(f"Action: {action}\n")
        else:
            print("No solution exists.")
```

output:

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

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

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

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

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

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

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
Action: Goal Reached!!
```

Program 4:

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorihtm:

2. Implement Hill Climbing Search algorithm to solve N-Queens problem

→ Function HILL-CLIMBING(problem) returns a state that is local maximum.

current ← MAKE-NODE(problem.Initial state)
loop do
neighbor ← a highest-valued successor of current
if neighbor.VALUE ≤ current.VALUE then
return current.STATE.
current ← neighbor.

~~Step~~: -

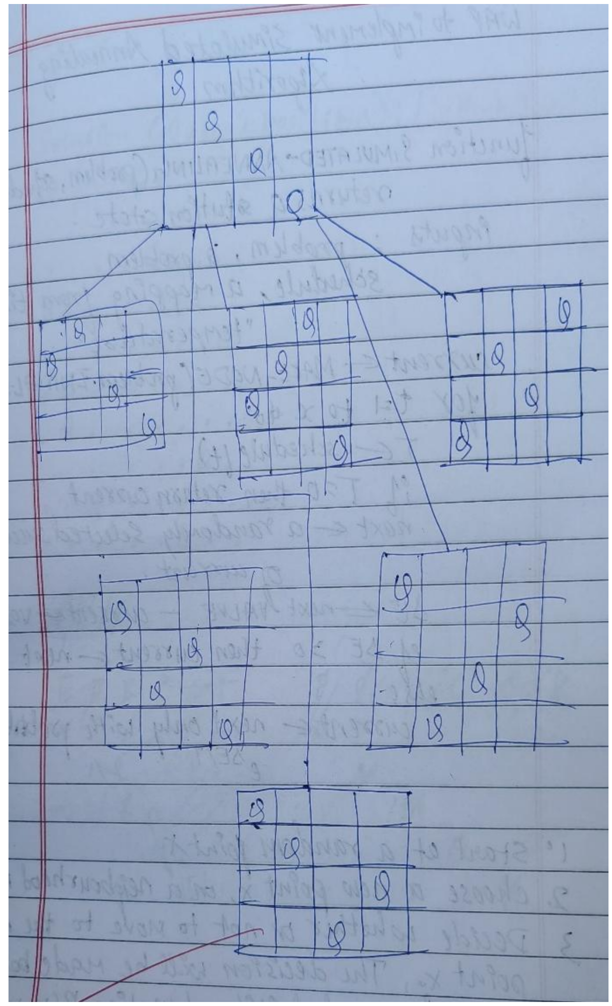
→ State: 4 queens on the board. One queen per column.

- Variables: x_0, x_1, x_2, x_3 where x_i is the row position of the queen in column i. Assume that there is one queen per column.

- Domain for each variable: $x_i \in \{0,1,2,3\}, \forall i$

→ Initial state: a random state

→ Goal state: 4 queens on the board. No pair of queens are attacking each other



Code:

```
# N QUEENS USING HILL CLIMBING SEARCH ALGORITHM
import random

# Function to print the 4x4 board
def print_board(board):
    for row in range(len(board)):
        line = ""
        for col in range(len(board)):
            if board[row] == col:
                line += " Q "
            else:
                line += " . "
        print(line)
    print()

# Function to calculate the number of conflicts (heuristic)
def calculate_conflicts(board):
```

```
    conflicts = 0
    n = len(board)

    for i in range(n):
        for j in range(i + 1, n):
            # Check if queens are on the same column or on the same 
diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == 
abs(i - j):
                conflicts += 1
    return conflicts

# Function to perform the Hill Climbing algorithm
def hill_climbing(n=4):
    # Start with a random configuration
    board = [random.randint(0, n - 1) for _ in range(n)]

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

    while True:
        current_conflicts = calculate_conflicts(board)

        # If no conflicts, we have found the solution
        if current_conflicts == 0:
            print("Solution found!")
            return board

        # Generate all possible neighbor configurations by moving one 
queen
        neighbors = []
        for row in range(n):
            for col in range(n):
                if board[row] != col:  # Only consider moves that 
change the column of the queen
                    neighbor = board[:]
                    neighbor[row] = col
                    neighbors.append(neighbor)

        # Evaluate neighbors and select the best one (with fewer 
conflicts)
        best_neighbor = None
        best_conflicts = current_conflicts

        for neighbor in neighbors:
            conflicts = calculate_conflicts(neighbor)
            if conflicts < best_conflicts:
                best_neighbor = neighbor
                best_conflicts = conflicts

        # If no better neighbors, the algorithm is stuck at a local 
minimum
        if best_neighbor is None or best_conflicts >= 
current_conflicts:
            print("Stuck at local minimum. Restarting...")
            board = [random.randint(0, n - 1) for _ in range(n)]
            print("Restarting with new board:")
            print_board(board)
```

```
        else:
            # Move to the better neighbor
            board = best_neighbor
            print(f"Current Conflicts: {best_conflicts}")
            print_board(board)

# Run the Hill Climbing algorithm for the 4-Queens problem
hill_climbing()
```

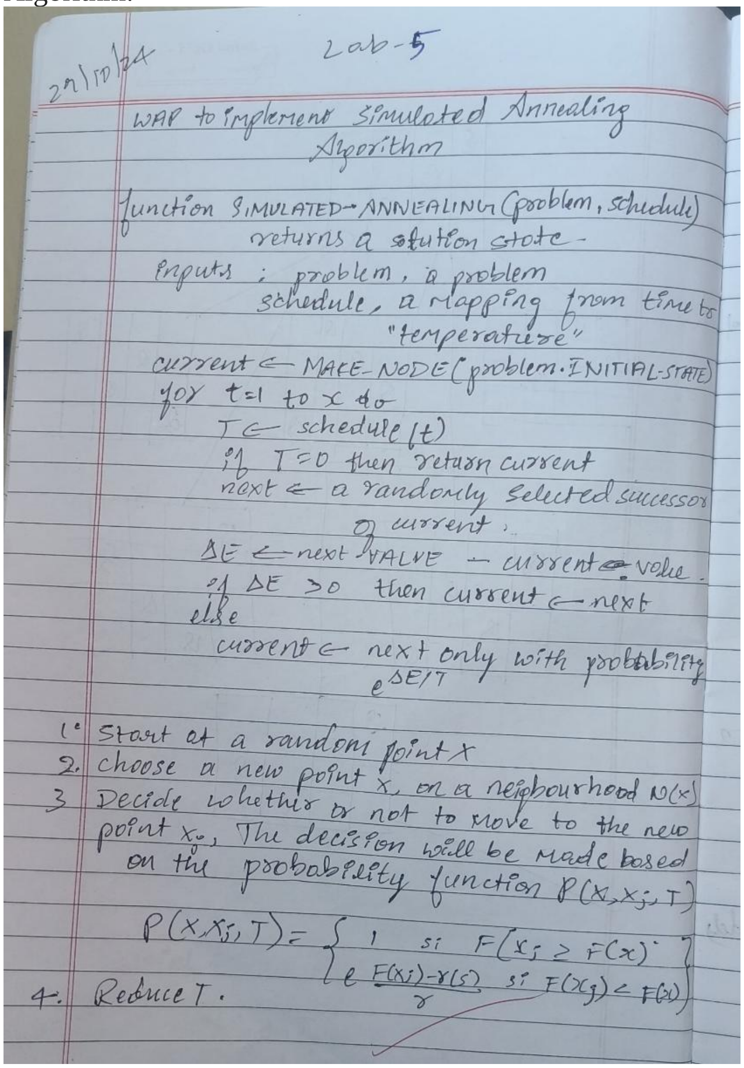
output:

```
Initial Board:
 .  .  .  Q
 Q  .  .  .
 .  .  Q  .
 .  Q  .  .

Stuck at local minimum. Restarting...
Restarting with new board:
 .  .  Q  .
 .  .  Q  .
 .  .  .  Q
 Q  .  .  .

Current Conflicts: 1
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 Q  .  .  .

Current Conflicts: 0
 .  .  Q  .
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .

Solution found!
]:  [2, 0, 3, 1]
```

Program 5:

Simulated Annealing to Solve 8-Queens problem

Algorithm:



27/10/24 Lab-5

WAP to implement Simulated Annealing Algorithm

function SIMULATED-ANNEALING(problem, schedule) returns a solution state.

inputs: problem, a problem

schedule, a mapping from time to "temperature"

current ← MAKE-NODE(problem.INITIAL-STATE)

for t=1 to x do

T ← schedule(t)

if T=0 then return current

next ← a randomly selected successor of current.

ΔE ← next.VALUE − current.value.

if ΔE > 0 then current ← next

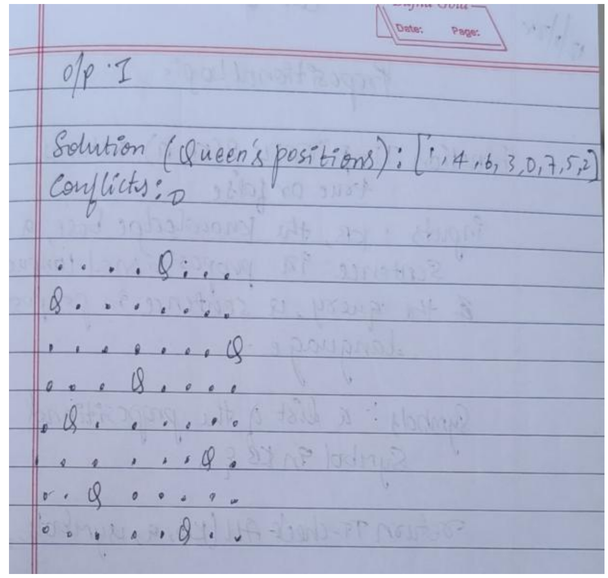
else

current ← next only with probability $e^{\Delta E/T}$

1. Start at a random point x
2. Choose a new point x_j on a neighbourhood N(x)
3. Decide whether or not to move to the new point x_j. The decision will be made based on the probability function $P(x,x_j,T)$

$$P(x,x_j,T) = \begin{cases} 1 & si\ F(x_j) \geq F(x) \\ e^{\frac{F(x_j)-F(x)}{T}} & si\ F(x_j) < F(x) \end{cases}$$

4. Reduce T.



o/p: I

Solution (Queen's positions): [1,4,6,3,0,7,5,2]

Conflicts: 0

. . . . Q . . .
Q . . . .
. . . . Q
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .

code:

```
# Simulated Annealing Algorithm
import numpy as np

def calculate_attacks(position):
    attacks = 0
    n = len(position)

    for i in range(n):
        for j in range(i + 1, n):
            if position[i] == position[j] or abs(position[i] -
position[j]) == j - i:
                attacks += 1
    return attacks

def simulated_annealing(n, max_iters=1000, initial_temp=100,
cooling_rate=0.99):
```

```
    current_position = np.random.permutation(n)
    current_attacks = calculate_attacks(current_position)
    temperature = initial_temp

    for i in range(max_iters):
        if current_attacks == 0:
            break

        new_position = current_position.copy()
        i, j = np.random.choice(n, 2, replace=False)
        new_position[i], new_position[j] = new_position[j],
new_position[i]


        new_attacks = calculate_attacks(new_position)


        if new_attacks < current_attacks or np.random.rand() <
np.exp((current_attacks - new_attacks) / temperature):
            current_position = new_position
            current_attacks = new_attacks

        temperature *= cooling_rate

    return current_position, current_attacks


n = 8
solution, attacks = simulated_annealing(n)

print("Best position found:", solution)
print("Number of attacks:", attacks)
```

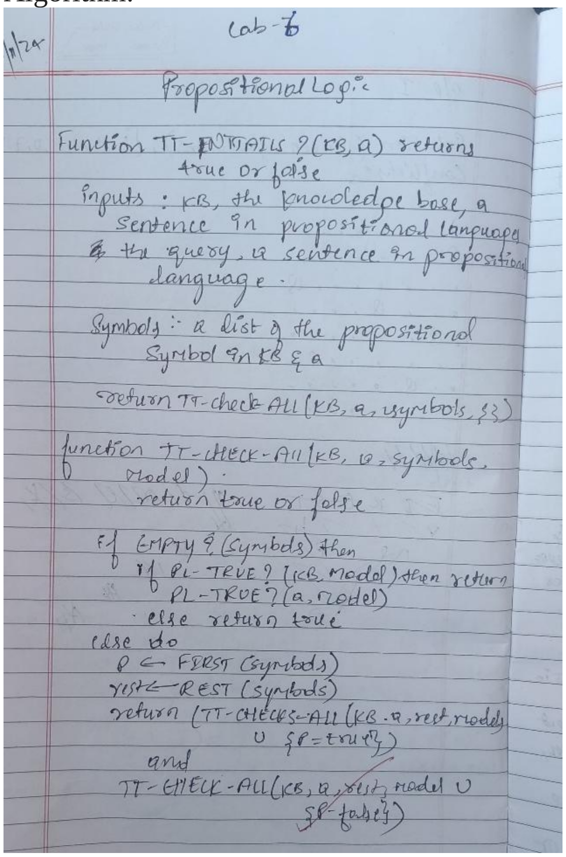
output:

```
Best position found: [0 5 7 2 6 3 1 4]
Number of attacks: 0
```

Program 6:

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

Algorithm:



Lab-6

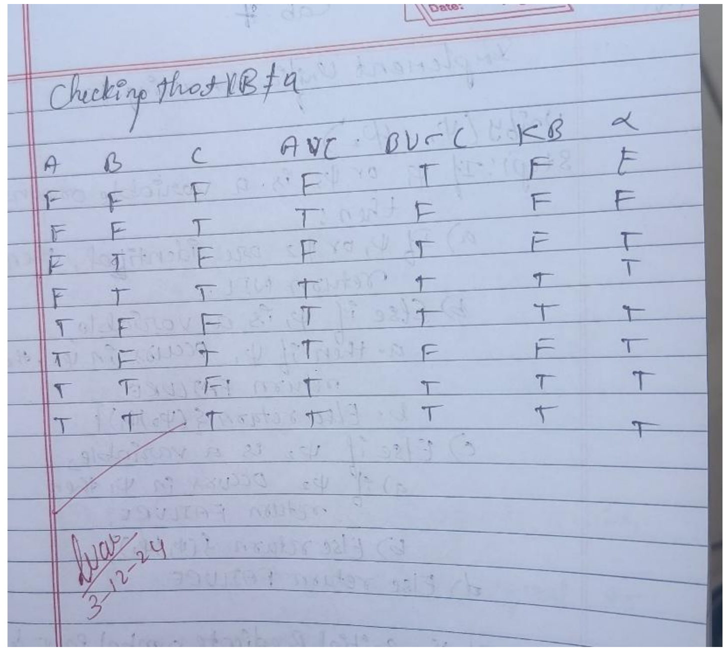
Propositional Logic

Function TT-ENTAILS?(KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional language
α, the query, a sentence in propositional language.

Symbols: a list of the propositional symbol in KB & α

return TT-CHECK-ALL(KB, α, symbols, {})

function TT-CHECK-ALL(KB, α, symbols, model) return 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}))



Checking that KB ⊨ α

A	B	C	A∨C	B∨¬C	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

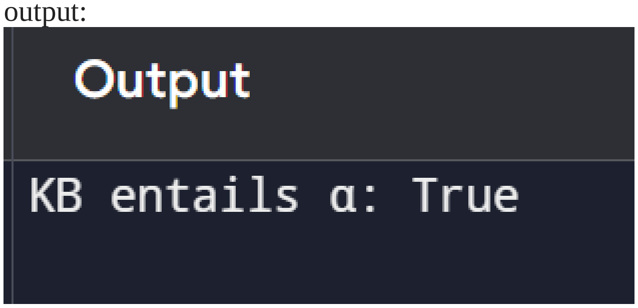
3/12/24

Code:

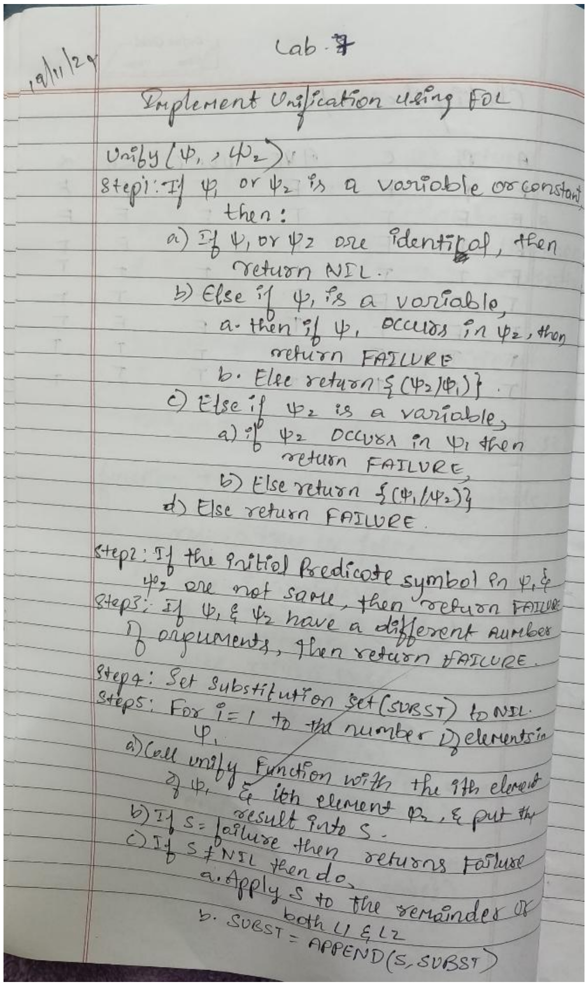
```
from itertools import product

def extract_variables(expression):
    """Extract unique variables from an expression."""
    variables = set()
    for char in expression:
        if char.isalpha() and char.isupper():  # Assuming variables are single uppercase letters
            variables.add(char)
    return sorted(variables)

def truth_table(variables):
    """Generate all possible truth assignments for given variables."""
    return list(product([True, False], repeat=len(variables)))

def evaluate_expression(expression, assignment):
    """Evaluate the expression with the given truth assignment."""
    local_dict = dict(zip(variables, assignment))
    return eval(expression, {}, local_dict)

def check_entailment(KB, alpha):
```

```
    """Check if KB entails alpha using a truth table approach."""
    global variables
    variables = extract_variables(KB + alpha)  # Identify unique variables
    assignments = truth_table(variables)  # Generate all possible truth assignments

    for assignment in assignments:
        KB_value = evaluate_expression(KB, assignment)
        alpha_value = evaluate_expression(alpha, assignment)

        # If KB is True and alpha is False for any assignment, entailment fails
        if KB_value and not alpha_value:
            return False

    # If no assignment contradicts entailment, return True
    return True

# Example usage
KB = "(A and B) or (C and D)"
alpha = "A or C"

result = check_entailment(KB, alpha)
print("KB entails α:", result)
```

output:



Program 7:
Implement unification in first order logic
Algorithm:



19/11/24 Lab 7

Implement Unification using FOL

Unify(Ψ_1, Ψ_2)

Step1: If Ψ_1 or Ψ_2 is a variable or constant then:

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

b) Else if Ψ_1 is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

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

c) Else if Ψ_2 is a variable,

a) if Ψ_2 occurs in Ψ_1 then return FAILURE,

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

d) Else return FAILURE.

Step2: If the initial Predicate symbol in Ψ_1 & Ψ_2 are not same, then return FAILURE

Step3: If Ψ_1 & Ψ_2 have a different number of arguments, then return FAILURE.

Step4: Set Substitution set(SUBST) to NIL.

Step5: For i=1 to the number of elements in Ψ_1

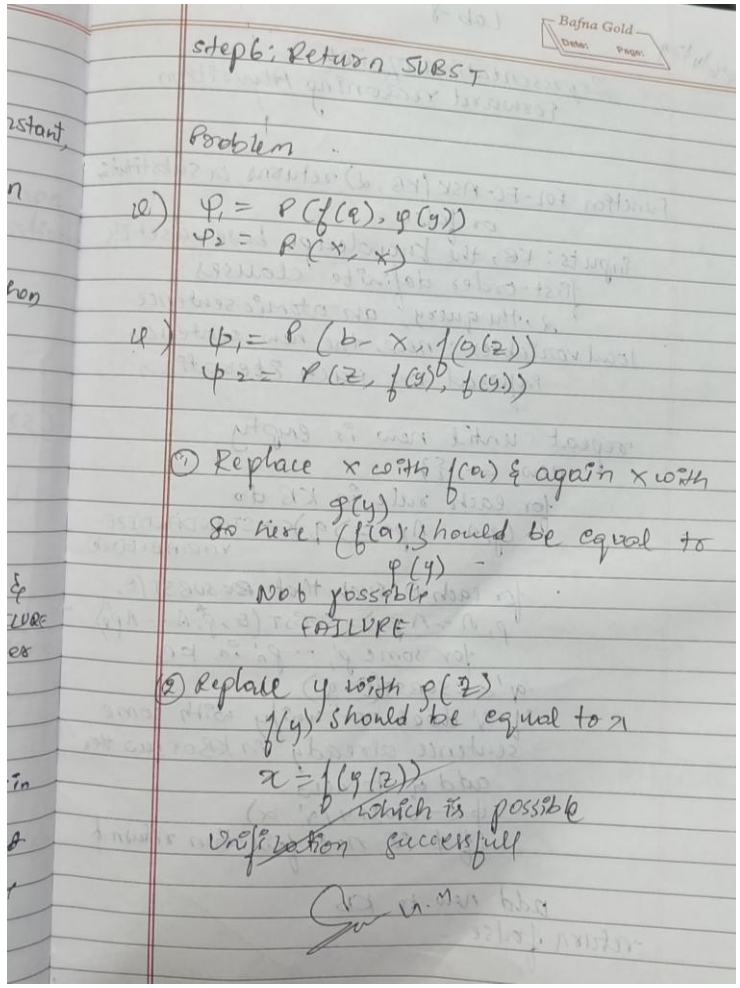
a) Call unify function with the ith element of Ψ_1 & ith element Ψ_2, & put the result into S.

b) If S = failure then returns Failure

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 & L2

b. SUBST = APPEND(S, SUBST)



Step6: Return SUBST

Problem

1) $\Psi_1 = P(f(a), g(y))$
$\Psi_2 = P(x, x)$

2) $\Psi_1 = P(b, x, f(g(z)))$
$\Psi_2 = P(z, f(y), f(y))$

(1) Replace x with f(a) & again x with g(y)
So here f(a) should be equal to g(y)
Not possible
FAILURE

(2) Replace y with g(z)
f(y) should be equal to x
x = f(g(z))
which is possible
Unification successfull

Code:

```
# Define a function to apply substitutions to a list of terms
def apply_substitution(terms, substitution):
    return [substitution.get(term, term) for term in terms]

# Function to check if a term is a variable (it assumes variables
are single letters)
def is_variable(term):
    return term.isalpha()

# Function to unify terms recursively
def unify_terms(term1, term2, substitutions):
```

```
    # Case 1: If both terms are the same, no substitution needed
    if term1 == term2:
        return term1, term2, substitutions

    # Case 2: If one term is a variable and the other is not,
substitute the variable with the term
    elif is_variable(term1):
        if term1 in substitutions:
            return unify_terms(substitutions[term1], term2,
substitutions)
        substitutions[term1] = term2
        return term2, term2, substitutions

    elif is_variable(term2):
        if term2 in substitutions:
            return unify_terms(term1, substitutions[term2],
substitutions)
        substitutions[term2] = term1
        return term1, term1, substitutions

    # Case 3: If both terms are functions (e.g., f(g(Z)) and
f(Y)), unify the inner terms
    elif isinstance(term1, str) and term1.startswith('f(') and
isinstance(term2, str) and term2.startswith('f('):
        inner1 = term1[2:-1]  # Extract the argument inside
f(...)
        inner2 = term2[2:-1]  # Extract the argument inside
f(...)
        inner1, inner2, substitutions = unify_terms(inner1,
inner2, substitutions)
        return f"f({inner1})", f"f({inner2})", substitutions

    else:
        raise ValueError(f"Cannot unify terms: {term1} and
{term2}")

# Function to perform unification
def unify(Ψ1, Ψ2):
    substitutions = {}

    # Ensure both terms have the same number of arguments
    if len(Ψ1) != len(Ψ2):
        raise ValueError("The terms have different numbers of
arguments and cannot be unified.")

    # Unify corresponding arguments
    for i in range(len(Ψ1)):
        Ψ1[i], Ψ2[i], substitutions = unify_terms(Ψ1[i], Ψ2[i],
substitutions)

    return Ψ1, Ψ2, substitutions

# Function to take user input and parse it
def get_input():
    print("Enter the first term (e.g., p(b, X, f(g(Z))):")
    term1 = input("Enter Ψ1: ")
    print("Enter the second term (e.g., p(Z, f(Y), f(Y))):")
    term2 = input("Enter Ψ2: ")
```

```
    # Convert the input strings into lists (representing the
terms' arguments)
    Ψ1 = term1[2:-1].split(', ')  # Extract arguments from the
p(...) form
    Ψ2 = term2[2:-1].split(', ')  # Extract arguments from the
p(...) form

    return Ψ1, Ψ2

# Get input from the user
Ψ1, Ψ2 = get_input()

# Perform unification
try:
    unified_Ψ1, unified_Ψ2, final_substitution = unify(Ψ1, Ψ2)

    print("\nUnified Ψ1:", unified_Ψ1)
    print("Unified Ψ2:", unified_Ψ2)
    print("Final Substitution:", final_substitution)
except ValueError as e:
    print(f"Unification failed: {e}")
```

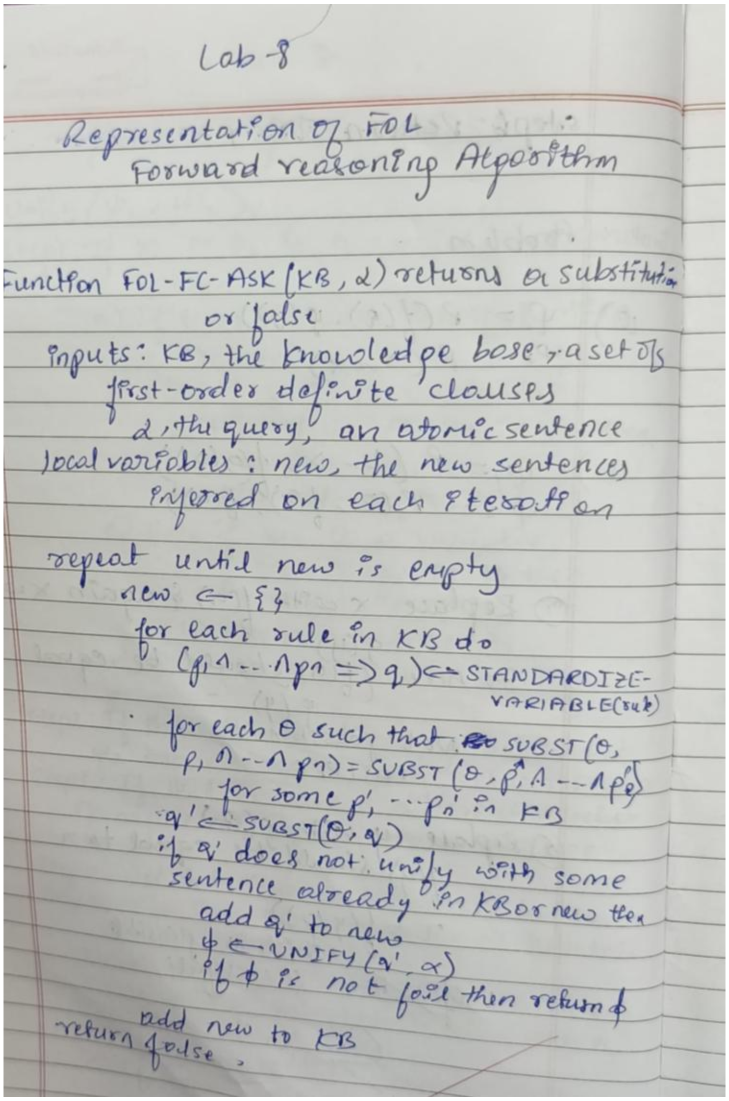
output:

```
Enter the first term (e.g., p(b, X, f(g(Z))):
Enter Ψ1: p(b, X, f(g(Z)))
Enter the second term (e.g., p(Z, f(Y), f(Y))):
Enter Ψ2: p(Z, f(Y), f(Y))

Unified Ψ1: ['Z', 'f(Y)', 'f(g(Z))']
Unified Ψ2: ['Z', 'f(Y)', 'f(g(Z))']
Final Substitution: {'b': 'Z', 'X': 'f(Y)', 'Y': 'g(Z)'}
```

Program 8:

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

Algorithm:

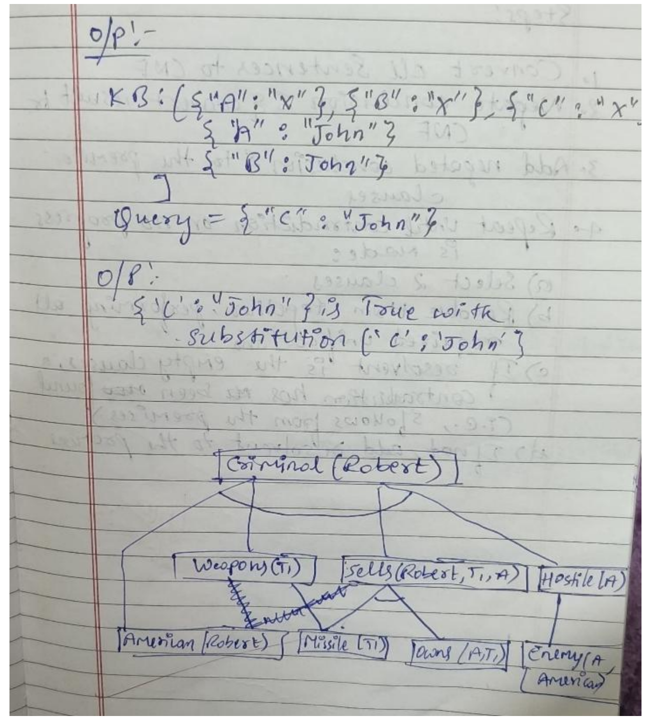


Lab-8

Representation of FOL
Forward reasoning Algorithm

function FOL-FC-ASK(KB, α) returns a substitution or false
inputs: KB, the knowledge base, a set of first-order definite clauses
α, the query, an atomic sentence
local variables: new, the new sentences inferred on each iteration

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



o/p:-
KB: [{"A": "x"}, {"B": "x"}, {"C": "x"}
{"A": "John"}
{"B": "John"}
]
Query = {"C": "John"}
O/P:
{'C': 'John'} is True with the substitution {'C': 'John'}

Code:

```
# Define facts and rules based on the diagram
facts = {
    "Criminal(Robert)": True,
    "American(Robert)": True,
    "Missile(T1)": True,
```

```python
    "Hostile(A)": True,
}

# Define the inference rules
rules = [
    # Rule 1: If Criminal(X) and Weapons(T1), then Sells(X, T1, A)
    ("Criminal(X) and Weapons(T1)", "Sells(X, T1, A)"),

    # Rule 2: If Criminal(X) and Sells(X, T1, A), then Hostile(A)
    ("Criminal(X) and Sells(X, T1, A)", "Hostile(A)"),

    # Rule 3: If Hostile(A) and American(X), then Enemy(A, America)
    ("Hostile(A) and American(X)", "Enemy(A, America)"),

    # Rule 4: If American(X) and owns(A, T1), then Weapons(T1)
    ("American(X) and owns(A, T1)", "Weapons(T1)")
]

# Function to check if a statement is true
def check_fact(statement):
    return facts.get(statement, False)

# Forward reasoning function
def forward_reasoning():
    inferred_facts = set(facts.keys())
    new_inferences = True

    while new_inferences:
        new_inferences = False
        for condition, conclusion in rules:
            # Parse condition into individual facts
            condition_facts = condition.split(" and ")
            # Check if all facts in the condition are known
            if all(check_fact(fact) for fact in condition_facts):
                # If all facts are true, infer the conclusion
                if conclusion not in inferred_facts:
                    inferred_facts.add(conclusion)
                    new_inferences = True
                    print(f"New inference: {conclusion}")

    return inferred_facts

# Run forward reasoning
inferred_facts = forward_reasoning()

# Print the final set of inferred facts
print("\nFinal Inferred Facts:")
for fact in inferred_facts:
    print(fact)

# Define the facts (initial knowledge)
facts = {
    "American(Robert)": True,  # Robert is American
    "Missile(T1)": True,       # T1 is a missile
    "Enemy(A, America)": True, # Country A is an enemy of America
    "Owns(A, T1)": True,       # Country A owns T1
    "Hostile(A)": False,       # Initially, A is not hostile
    "Weapon(T1)": False,       # Initially, T1 is not considered a weapon
```

```
    "Sells(Robert, T1, A)": False,  # Initially, Robert doesn't sell T1 to A
    "Criminal(Robert)": False,    # Initially, Robert is not considered a
criminal
}

# Function to check if a fact is true
def check_fact(fact):
    return facts.get(fact, False)

# Function to infer facts based on the rules
def forward_reasoning():
    new_inferences = True
    while new_inferences:
        new_inferences = False

        # Rule 1: If American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r), then
Criminal(p)
        if check_fact("American(Robert)") and check_fact("Weapon(T1)") and
check_fact("Sells(Robert, T1, A)") and check_fact("Hostile(A)"):
            if not check_fact("Criminal(Robert)"):
                facts["Criminal(Robert)"] = True
                new_inferences = True
                print("Inferred: Robert is a criminal (Criminal(Robert))")

        # Rule 2: If Owns(A, p) ∧ Missile(p), then Weapon(p)
        if check_fact("Owns(A, T1)") and check_fact("Missile(T1)"):
            if not check_fact("Weapon(T1)"):
                facts["Weapon(T1)"] = True
                new_inferences = True
                print("Inferred: T1 is a weapon (Weapon(T1))")

        # Rule 3: If Missile(p) ∧ Owns(A, p), then Sells(Robert, p, A)
        if check_fact("Missile(T1)") and check_fact("Owns(A, T1)"):
            if not check_fact("Sells(Robert, T1, A)"):
                facts["Sells(Robert, T1, A)"] = True
                new_inferences = True
                print("Inferred: Robert sells T1 to A (Sells(Robert, T1, A))")

        # Rule 4: If Enemy(p, America), then Hostile(p)
        if check_fact("Enemy(A, America)"):
            if not check_fact("Hostile(A)"):
                facts["Hostile(A)"] = True
                new_inferences = True
                print("Inferred: A is hostile (Hostile(A))")

    return facts

# Function to start the reasoning and print inferred facts
def print_inferred_facts():
    # Perform forward reasoning to infer facts
    forward_reasoning()

    # Print the final set of inferred facts
    print("\nFinal Inferred Facts:")
    for fact, value in facts.items():
        if value:
            print(f"{fact} is TRUE")
```

```python
        else:
            print(f"{fact} is FALSE")

# Start the reasoning and print the results
print_inferred_facts()
```

Output:

```
Final Inferred Facts:
Hostile(A)
American(Robert)
Missile(T1)
Criminal(Robert)
Inferred: T1 is a weapon (Weapon(T1))
Inferred: Robert sells T1 to A (Sells(Robert, T1, A))
Inferred: A is hostile (Hostile(A))
Inferred: Robert is a criminal (Criminal(Robert))

Final Inferred Facts:
American(Robert) is TRUE
Missile(T1) is TRUE
Enemy(A, America) is TRUE
Owns(A, T1) is TRUE
Hostile(A) is TRUE
Weapon(T1) is TRUE
Sells(Robert, T1, A) is TRUE
Criminal(Robert) is TRUE
```

Program 9:

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

Algorithm:

Lab-9

Convert a given FOL Statement into ~~CNF~~ Resolution

Steps:

- Convert all Sentences to CNF
- Negate conclusion S & convert result to CNF
- Add negated conclusion S to the premise clauses
- Repeat until contradiction or no progress is made:
  a) Select 2 clauses
  b) Resolve them together, performing all required unifications
  c) If resolvent is the empty clauses, a contradiction has been found (i.e., S follows from the premises)
  d) If not, add resolvent to the premises

28-11-24

o/p

KB

① ¬food (x) → likes (John, x)
② food (Apple)
③ ¬eats (Y, Z) ∨ Killed (Y) ∨ food (z)
④ eats (Anil, peanuts)
⑤ alive (Anil
⑥ ¬eats (Anil, w) ∨ eats (Harry, w)
⑦ Killed (g) alive (g)
⑧ ¬alive (x) ∨ ¬ killed (x)
⑨ likes (John, Peanuts)

¬likes (John, Peanuts)　　¬food (x) ∨ likes (John, x)

¬food (peanuts)　　¬eats (Y, z) ∨ killed ∨ food Z (Peanuts/z)

¬eats (y, Peanuts) ∨ Killed (x)　　eats (Anil, Peanuts)

Killed (Anil)　　¬alive (x) ∨ ¬killed (anil)

¬alive (Anil)　　alive

o/p

Proved by Contradiction John likes Peanuts.

Code:

```
from sympy import symbols, And, Or, Not, Implies, to_cnf

# Define constants (entities in the problem)
John, Anil, Harry, Apple, Vegetables, Peanuts, x, y = symbols('John Anil Harry
Apple Vegetables Peanuts x y')

# Define predicates as symbols (this works as a workaround)
Food = symbols('Food')
Eats = symbols('Eats')
Likes = symbols('Likes')
Alive = symbols('Alive')
Killed = symbols('Killed')

# Knowledge Base (Premises) in First-Order Logic
premises = [
    # 1. John likes all kinds of food: Food(x) → Likes(John, x)
    Implies(Food, Likes),
```

```
    # 2. Apples and vegetables are food: Food(Apple) ∧ Food(Vegetables)
    And(Food, Food),

    # 3. Anything anyone eats and is not killed is food: (Eats(y, x) ∧ ¬Killed(y)) →
Food(x)
    Implies(And(Eats, Not(Killed)), Food),

    # 4. Anil eats peanuts and is still alive: Eats(Anil, Peanuts) ∧ Alive(Anil)
    And(Eats, Alive),

    # 5. Harry eats everything that Anil eats: Eats(Anil, x) → Eats(Harry, x)
    Implies(Eats, Eats),

    # 6. Anyone who is alive implies not killed: Alive(x) → ¬Killed(x)
    Implies(Alive, Not(Killed)),

    # 7. Anyone who is not killed implies alive: ¬Killed(x) → Alive(x)
    Implies(Not(Killed), Alive),
]

# Negated conclusion to prove: ¬Likes(John, Peanuts)
negated_conclusion = Not(Likes)

# Convert all premises and the negated conclusion to Conjunctive Normal Form
(CNF)
cnf_clauses = [to_cnf(premise, simplify=True) for premise in premises]
cnf_clauses.append(to_cnf(negated_conclusion, simplify=True))

# Function to resolve two clauses
def resolve(clause1, clause2):
    """
    Resolve two CNF clauses to produce resolvents.
    """
    clause1_literals = clause1.args if isinstance(clause1, Or) else [clause1]
    clause2_literals = clause2.args if isinstance(clause2, Or) else [clause2]
    resolvents = []

    for literal in clause1_literals:
        if Not(literal) in clause2_literals:
            # Remove the literal and its negation and combine the rest
            new_clause = Or(
                *[l for l in clause1_literals if l != literal],
                *[l for l in clause2_literals if l != Not(literal)]
            ).simplify()
            resolvents.append(new_clause)

    return resolvents

# Function to perform resolution on the set of CNF clauses
def resolution(cnf_clauses):
    """
    Perform resolution on CNF clauses to check for a contradiction.
    """
    clauses = set(cnf_clauses)
    new_clauses = set()
```

```
    while True:
        clause_list = list(clauses)
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                if False in resolvents:  # Empty clause found
                    return True  # Contradiction found; proof succeeded
                new_clauses.update(resolvents)

        if new_clauses.issubset(clauses):  # No new information
            return False  # No contradiction; proof failed

        clauses.update(new_clauses)

# Perform resolution to check if the conclusion follows
result = resolution(cnf_clauses)
print("Does John like peanuts? ", "Yes, proven by resolution." if result else
"No, cannot be proven.")
```

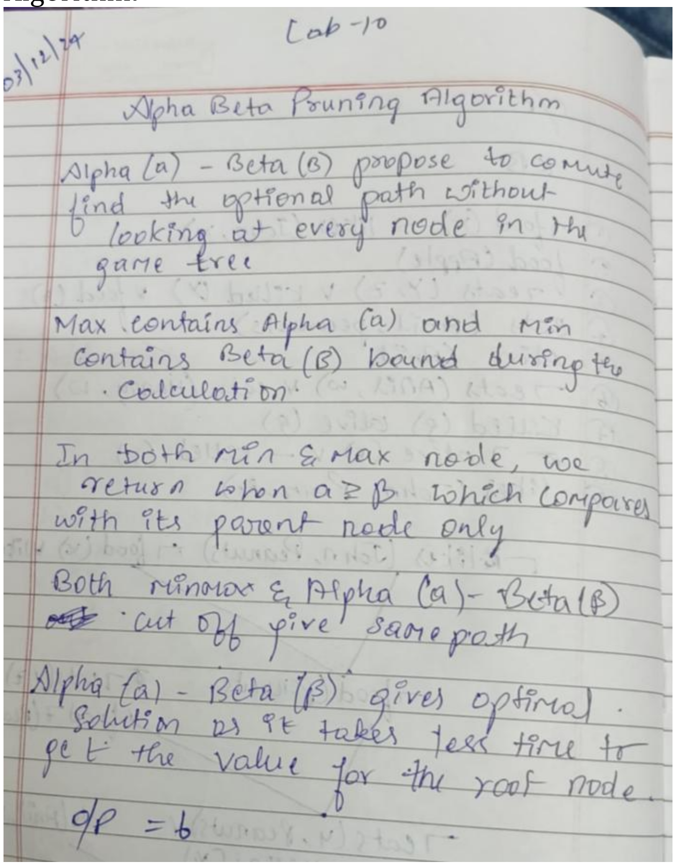
Output:

```
Does John like peanuts?  Yes, proven by resolution.
```

Program 10:

Implement Alpha-Beta Pruning.

Algorithm:



03/12/24

Lab-10

Alpha Beta Pruning Algorithm

Alpha (α) - Beta (β) propose to compute find the optional path without looking at every node in the game tree

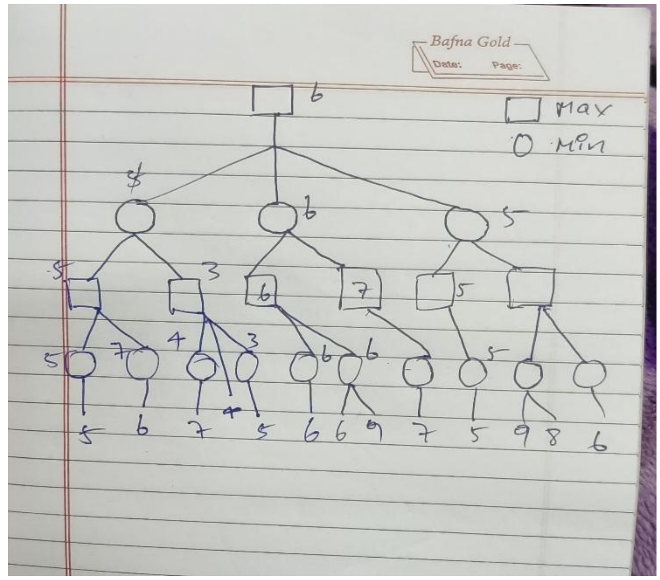
Max contains Alpha (α) and Min contains Beta (β) bound during the calculation

In both min & max node, we return when α ≥ β which compares with its parent node only

Both minmax & Alpha (α) - Beta (β) cut off give same path

Alpha (α) - Beta (β) gives optimal solution as it takes less time to get the value for the root node.

o/p = 6



Code:

```
class State:
    def __init__(self, value=None, actions=None):
        self.value = value  # Utility value if terminal
        self.actions = actions or []  # List of child states

def terminal_test(state):
    """Return True if the state is terminal."""
    return state.value is not None

def utility(state):
    """Return the utility value of a terminal state."""
    if state.value is None:
        raise ValueError("State is not terminal, utility called incorrectly.")
    return state.value
    """Return the list of actions (child states)."""
    return state.actions

def result(state, action):
    """Return the resulting state after taking an action."""
    return action  # The action is already the child state

def alpha_beta_search(state):
```

```python
    """Perform Alpha-Beta Search to find the best action."""
    v, best_action = max_value(state, float('-inf'), float('inf'))
    print("Best utility value:", v)  # Debug: display the best utility value
    return best_action

def max_value(state, alpha, beta):
    """Max-Value function for Alpha-Beta Pruning."""
    if terminal_test(state):
        return utility(state), None

    v = float('-inf')
    best_action = None
    for action in actions(state):
        min_val, _ = min_value(result(state, action), alpha, beta)
        if min_val > v:
            v = min_val
            best_action = action
        if v >= beta:
            return v, best_action
        alpha = max(alpha, v)
    return v, best_action

def min_value(state, alpha, beta):
    """Min-Value function for Alpha-Beta Pruning."""
    if terminal_test(state):
        return utility(state), None

    v = float('inf')
    best_action = None
    for action in actions(state):
        max_val, _ = max_value(result(state, action), alpha, beta)
        if max_val < v:
            v = max_val
            best_action = action
        if v <= alpha:
            return v, best_action
        beta = min(beta, v)
    return v, best_action

# Construct a minimax tree
leaf1 = State(value=3)
leaf2 = State(value=5)
leaf3 = State(value=6)
leaf4 = State(value=9)
leaf5 = State(value=1)
leaf6 = State(value=2)
leaf7 = State(value=0)
leaf8 = State(value=8)

node1 = State(actions=[leaf1, leaf2])  # MIN layer
node2 = State(actions=[leaf3, leaf4])  # MIN layer
node3 = State(actions=[leaf5, leaf6])  # MIN layer
node4 = State(actions=[leaf7, leaf8])  # MIN layer

root1 = State(actions=[node1, node2])  # MAX layer
root2 = State(actions=[node3, node4])  # MAX layer

root = State(actions=[root1, root2])  # Root (MAX layer)
```

```python
# Perform Alpha-Beta search
best_action = alpha_beta_search(root)

# Safely check and print the utility of the best action
if terminal_test(best_action):
    print("Best action leads to utility value:", utility(best_action))
else:
    print("Best action leads to a non-terminal state with further actions.")
```

Output:

```
Best utility value: 5
Best action leads to a non-terminal state with further actions.
```