VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum- 590014, Karnataka.

LAB RECORD

Artificial Intelligence (23CS5PCAIN)

Submitted by

Lakshitha.L (1BM22CS134)

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

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 **Lakshitha.L (1BM22CS134),** 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.

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

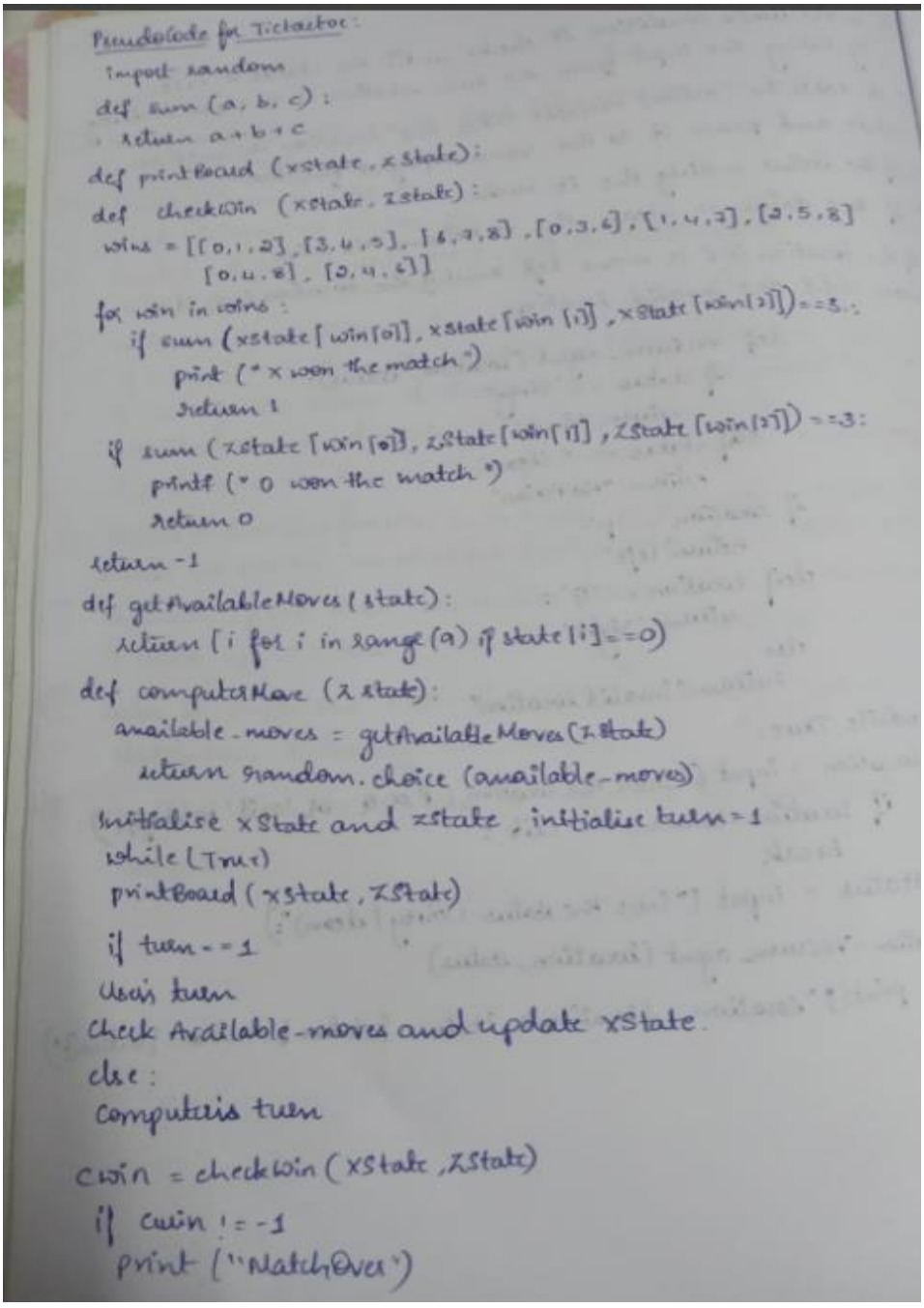
INDEX

Github Link: https://github.com/Lakshitha1818/AI-LAB

LABORATORY PROGRAM – 1

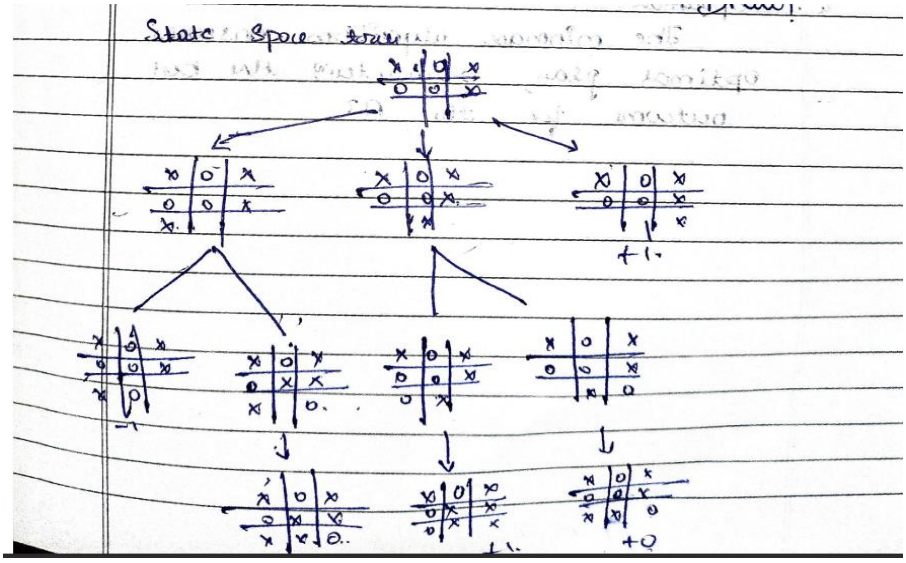
Implement Tic –Tac –Toe Game

PSEUDOCODE OR ALGORITHM



```
Pseudocode for Tictactoe:
import random
def sum(a, b, c):
    return a+b+c
def printBoard(xstate, zstate):
def checkWin(xstate, zstate):
wins = [[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8]
        [0,4,8],[2,4,6]]
for win in wins:
    if sum(xstate[win[0]], xstate[win[1]], xstate[win[2]])==3:
        print("x won the match")
        return 1
    if sum(xstate[win[0]], zstate[win[1]], zstate[win[2]])==3:
        printf("0 won the match")
        return 0
return -1
def getAvailableMoves(state):
    return [i for i in range(9) if state[i]==0)
def computerMove(xstate):
    available_moves = getAvailableMoves(zstate)
    return random.choice(available_moves)
initialise xState and zstate, initialise turn=1
while(True)
printBoard(xstate, zstate)
if turn==1
User turn
Check Available-moves and update xState.
else:
Computers turn
cwin = checkWin(xState, zState)
if cwin!=-1
    print("MatchOver")
```

STATE SPACE TREE



CODE

```
import random

board = ["-", "-", "-",
         "-", "-", "-",
         "-", "-", "-"]

def p_board(player):
    print(board[0] + "|" + board[1] + "|" + board[2])
    print(board[3] + "|" + board[4] + "|" + board[5])
    print(board[6] + "|" + board[7] + "|" + board[8])

def check_win():

  for i in range(0, 7, 3):
    if board[i] == board[i + 1] == board[i + 2] and board[i] != '-':
      return board[i]

  for i in range(3):
    if board[i] == board[i + 3] == board[i + 6] and board[i] != '-':
      return board[i]

  if board[0] == board[4] == board[8] and board[0] != '-':
    return board[0]
  if board[2] == board[4] == board[6] and board[2] != '-':
    return board[2]

  return None

def check_tie():
  if '-' not in board:
    return True
  return False

def play_game():
  current_player = "X"
  game_over = False

  while not game_over:
    p_board(current_player)

    if current_player == "X":
      print("It's your turn (X).")
      try:
        position = int(input("Choose a position from 1-9: ")) - 1
        if position < 0 or position > 8 or board[position] != '-':
          print("Invalid move. Try again.")
          continue
      except ValueError:
        print("Invalid input. Please enter a number between 1 and 9.")
        continue
    else:
```

```
            print("Computer's turn (O).")
            available_moves = [i for i, spot in enumerate(board) if spot == '-']
            position = random.choice(available_moves)

        board[position] = current_player
        winner = check_win()

        if winner:
            p_board(current_player)
            print(winner + " won!")
            game_over = True
        elif check_tie():
            p_board(current_player)
            print("It's a tie!")
            game_over = True
        else:
            current_player = "O" if current_player == "X" else "X"

play_game()
```

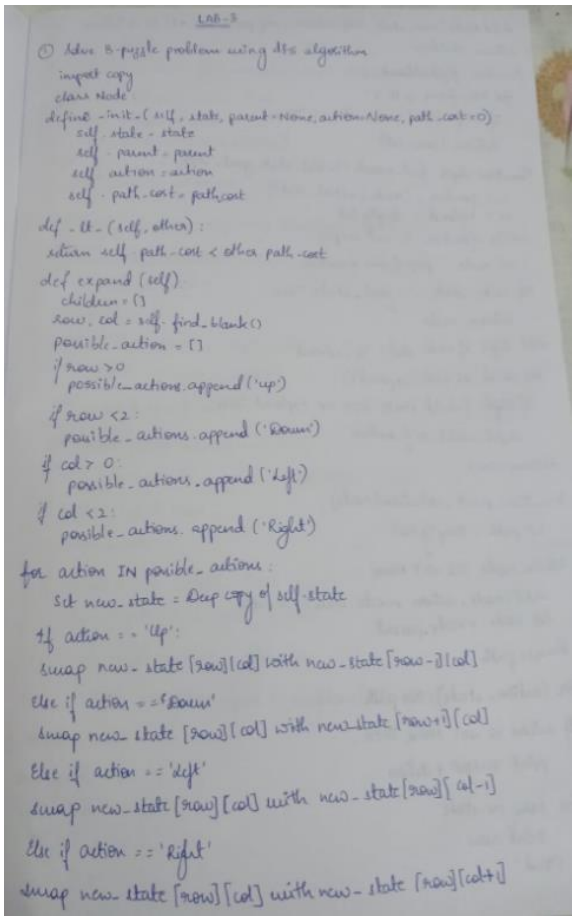
OUTPUT

```
Welcome to Tic Tac Toe
0 | 1 | 2
--|---|---
3 | 4 | 5
--|---|---
6 | 7 | 8
X's Chance. Please enter a value (0-8): 2
0 | 1 | X
--|---|---
3 | 4 | 5
--|---|---
6 | 7 | 8
O's Chance (Computer's Move)
Computer placed O in position 5
0 | 1 | X
--|---|---
3 | 4 | O
--|---|---
6 | 7 | 8
X's Chance. Please enter a value (0-8): 4
0 | 1 | X
--|---|---
3 | X | O
--|---|---
6 | 7 | 8
O's Chance (Computer's Move)
Computer placed O in position 7
0 | 1 | X
--|---|---
3 | X | O
--|---|---
6 | O | 8
X's Chance. Please enter a value (0-8): 6
X Won the match
0 | 1 | X
--|---|---
3 | X | O
--|---|---
X | O | 8
Match over
```

LABORATORY PROGRAM – 2

Solve 8 puzzle problems

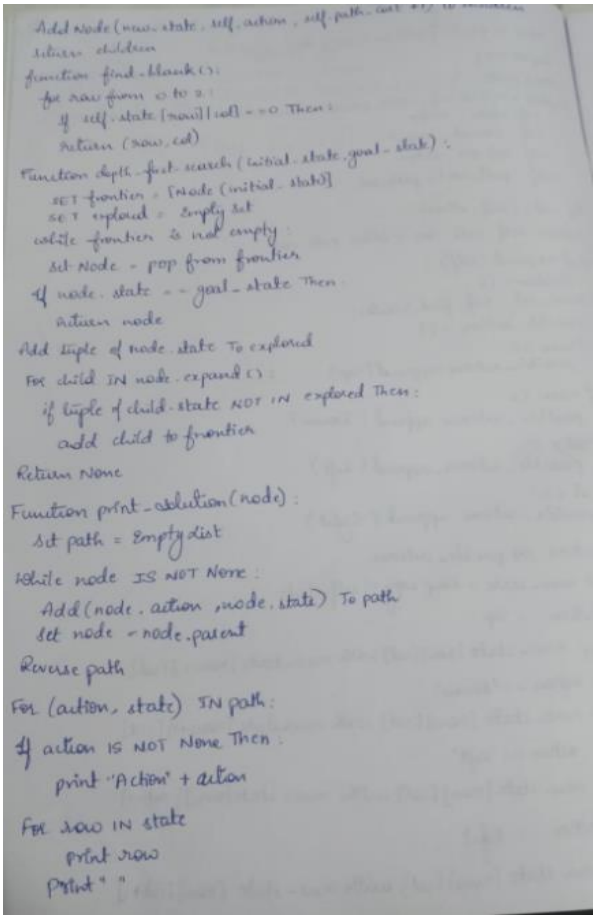
PSEUDOCODE OR ALGORITHM



LAB-3

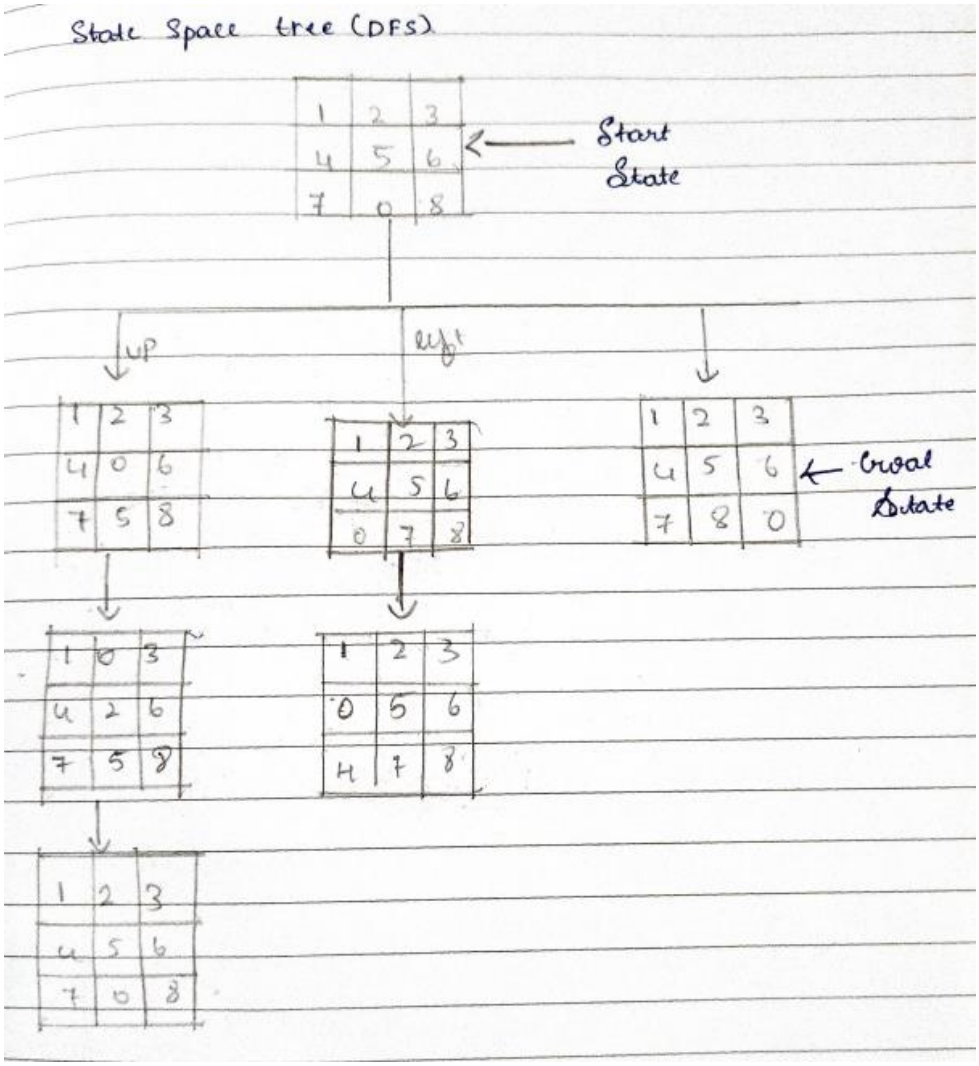
① Solve 8-puzzle problem using dfs algorithm

```
import copy
class state
define __init__(self, state, parent=None, action=None, path_cost=0)
    self.state = state
    self.parent = parent
    self.action = action
    self.path_cost = path_cost
def __lt__(self, other):
return self.path_cost < other.path_cost
def expand(self):
    children = []
    row, col = self.find_blank()
    possible_action = []
    if row > 0
    possible_actions.append('up')
    if row < 2:
    possible_actions.append('Down')
    if col > 0:
    possible_actions.append('Left')
    if col < 2:
    possible_actions.append('Right')
for action IN posible_actions:
    Set new_state = Deep copy of self.state
    if action == 'Up':
    swap new_state[row][col] with new_state[row-1][col]
    Else if action == 'Down'
    swap new_state[row][col] with new_state[row+1][col]
    Else if action == 'Left'
    swap new_state[row][col] with new_state[row][col-1]
    Else if action == 'Right'
    swap new_state[row][col] with new_state[row][col+1]
```



```
Add node (new_state, self, action, self.path_cost+1) to children
return children
function find_blank():
  for row from 0 to 2:
    if self.state[row][col] == 0 Then:
    return (row, col)
Function depth_first_search (initial_state, goal_state):
  SET frontier = [Node (initial_state)]
  SET explored = Empty Set
  while frontier is not empty:
    Set Node = pop from frontier
  if node.state == goal_state Then:
    return node
  Add tuple of node.state To explored
  For child IN node.expand():
    if tuple of child.state NOT IN explored Then:
      add child to frontier
  Return None
Function print_solution(node):
  Set path = Empty list
  While node IS NOT None:
    Add(node.action, node.state) To path
    Set node = node.parent
  Reverse path
  For (action, state) IN path:
  if action IS NOT None Then:
    print "Action" + action
  For row IN state
    print row
    Print " "
```

STATE SPACE TREE



CODE

```
from collections import deque
import time

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

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

def move(state, direction):
    new_state = [row[:] for row in state]
    zero_pos = find_zero(state)
    i, j = zero_pos

    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] = new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] = new_state[i][j+1], new_state[i][j]
    else:
        return None

    return new_state

def is_goal(state):
    return state == goal_state

def print_state(state):
    for row in state:
        print(row)
    print("\n")

def dfs(initial_state):
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        state, path = stack.pop()
        print("Exploring state in DFS:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(str(state))
```

```
        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and str(new_state) not in visited:
                stack.append((new_state, path + [direction]))

    return None

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()

    print("Initial State:")
    print_state(initial_state)

    start_time_dfs = time.time()
    print("Solving using DFS:")
    dfs_solution = dfs(initial_state)
    end_time_dfs = time.time()
    if dfs_solution:
        print("DFS Solution:", dfs_solution)
    else:
        print("No solution found with DFS.")
    print(f"Time taken by DFS: {end_time_dfs - start_time_dfs:.6f} seconds")
```

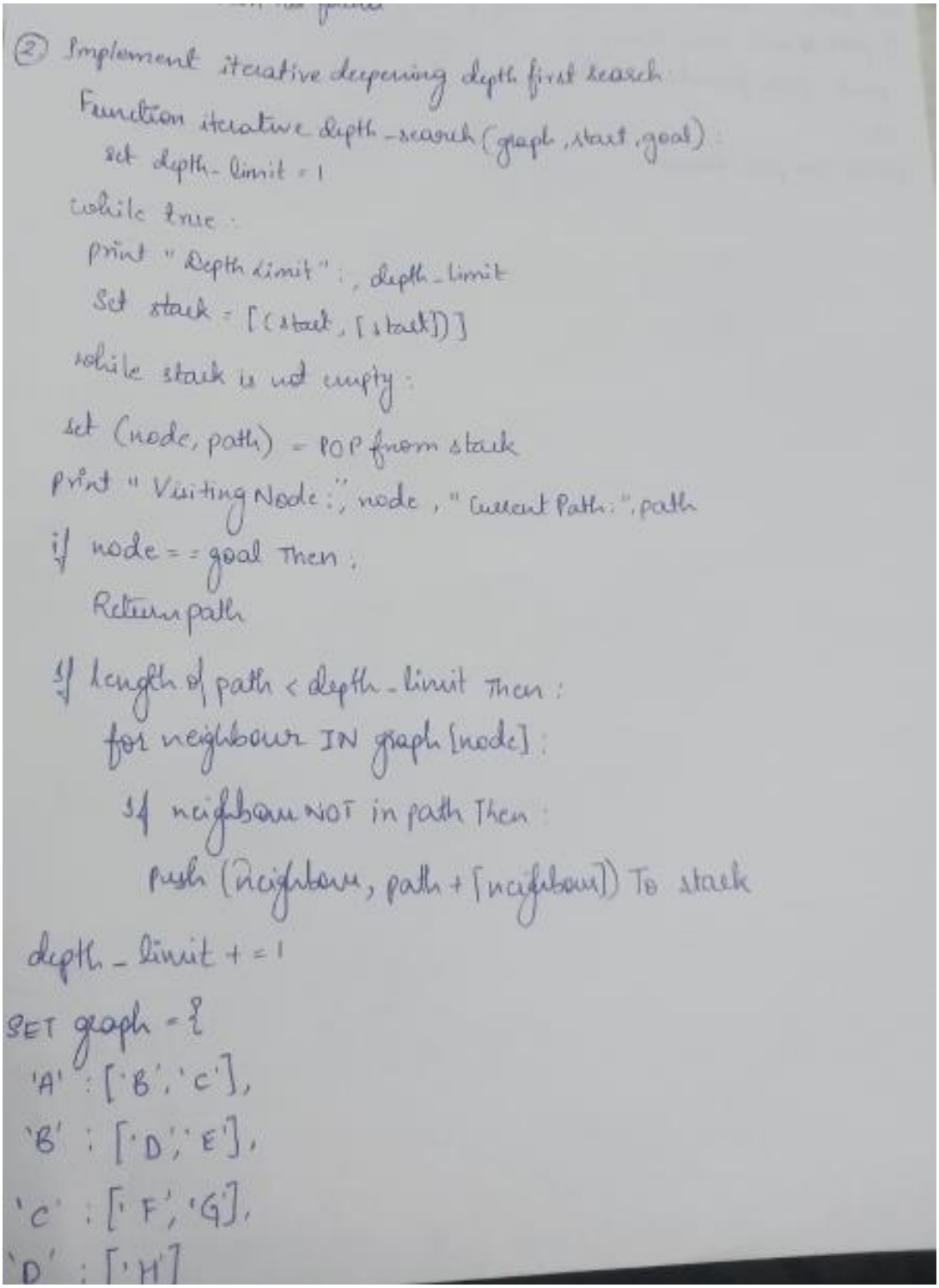
OUTPUT

```
Initial Puzzle:
1 2 3
4 5 6
7   8

Solution found:
1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```

LABORATORY PROGRAM – 3

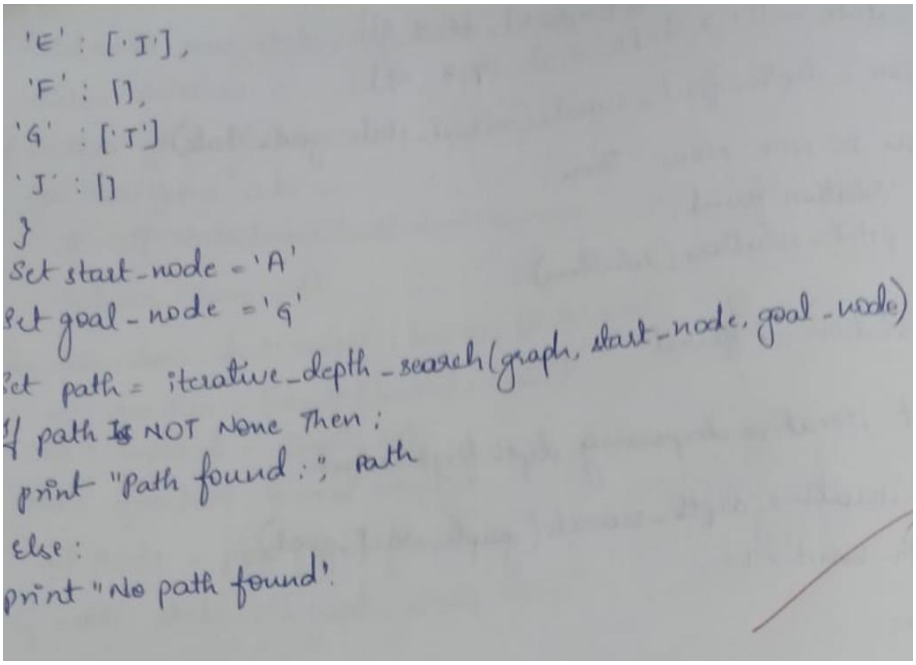
Implement Iterative deepening search algorithm

PSEUDOCODE OR ALGORITHM



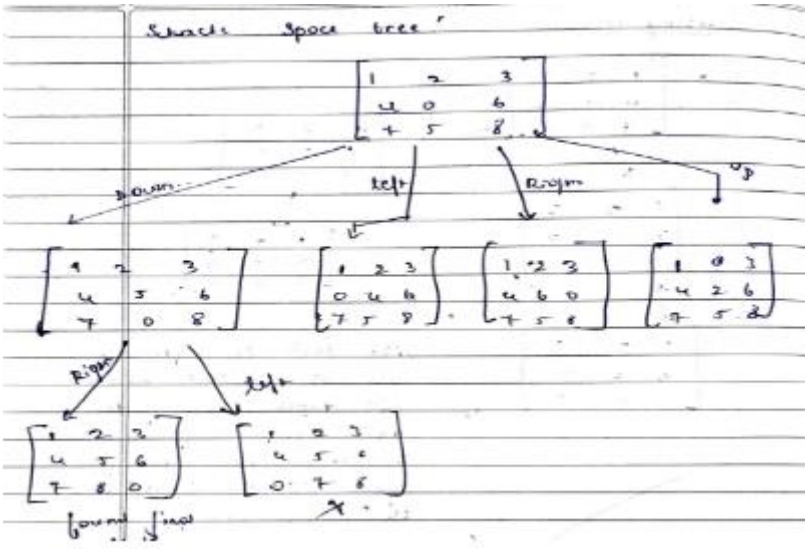
② Implement iterative deepening depth first search

```
Function iterative_depth_search(graph, start, goal):
  set depth_limit = 1
while true:
  print "Depth limit": , depth_limit
  Set stack = [(start, [start])]
  while stack is not empty:
  set (node, path) = POP from stack
  Print "Visiting Node:", node, "Current Path:", path
  if node == goal Then:
    Return path
  if length of path < depth_limit Then:
    for neighbour IN graph[node]:
      if neighbour NOT in path Then:
        push (neighbour, path + [neighbour]) To stack
  depth_limit += 1
SET graph = {
  'A' : ['B', 'C'],
  'B' : ['D', 'E'],
  'C' : ['F', 'G'],
  'D' : ['H']
```



```
'E' : ['I'],
'F' : [],
'G' : ['I']
'I' : []
}
Set start_node = 'A'
Set goal_node = 'G'
Set path = iterative_depth_search(graph, start_node, goal_node)
If path IS NOT None Then:
print "Path found:", path
Else:
print "No path found".
```

STATE SPACE TREE



CODE

```
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent

    def path(self):
        node, result = self, []
        while node:
            result.append(node.state)
            node = node.parent
        return result[::-1]

def iterative_deepening_search(problem):
    depth = 0
    while True:
        print(f"Exploring depth: {depth}")
        result, _ = depth_limited_search(problem, depth)
        if result is not None and result != 'cutoff':
            return result
        depth += 1

def depth_limited_search(problem, limit):
    frontier = [Node(problem.initial_state)]
    explored = set()
    cutoff_occurred = False

    while frontier:
        node = frontier.pop()

        if problem.is_goal(node.state):
            return node.path(), explored

        if node.state not in explored:
            explored.add(node.state)
            if len(node.path()) - 1 < limit:
                for child in problem.expand(node.state):
                    frontier.append(Node(child, node))
            else:
                cutoff_occurred = True

    return 'cutoff' if cutoff_occurred else None, explored

class GraphProblem:
    def __init__(self, initial_state, goal_state, adjacency_list):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.adjacency_list = adjacency_list

    def is_goal(self, state):
        return state == self.goal_state

    def expand(self, state):
        return [neighbor for neighbor in self.adjacency_list.get(state, [])]
```

```
def get_graph_from_input():
    adjacency_list = {}
    initial_state = input("Enter the initial state: ").strip()
    goal_state = input("Enter the goal state: ").strip()

    print("Enter the adjacency list for the graph (neighbors of each node).")
    print("Type 'done' when finished.")

    while True:
        node = input("Enter node (or 'done' to finish): ").strip()
        if node.lower() == 'done':
            break
        neighbors_input = input(f"Enter neighbors of {node} separated by spaces: ").strip()
        neighbors = neighbors_input.split()
        adjacency_list[node] = [neighbor.strip() for neighbor in neighbors]

    return GraphProblem(initial_state, goal_state, adjacency_list)

if __name__ == "__main__":
    problem = get_graph_from_input()
    solution = iterative_deepening_search(problem)

    if solution:
        print("Solution Path:", solution)
    else:
        print("No solution found.")
```

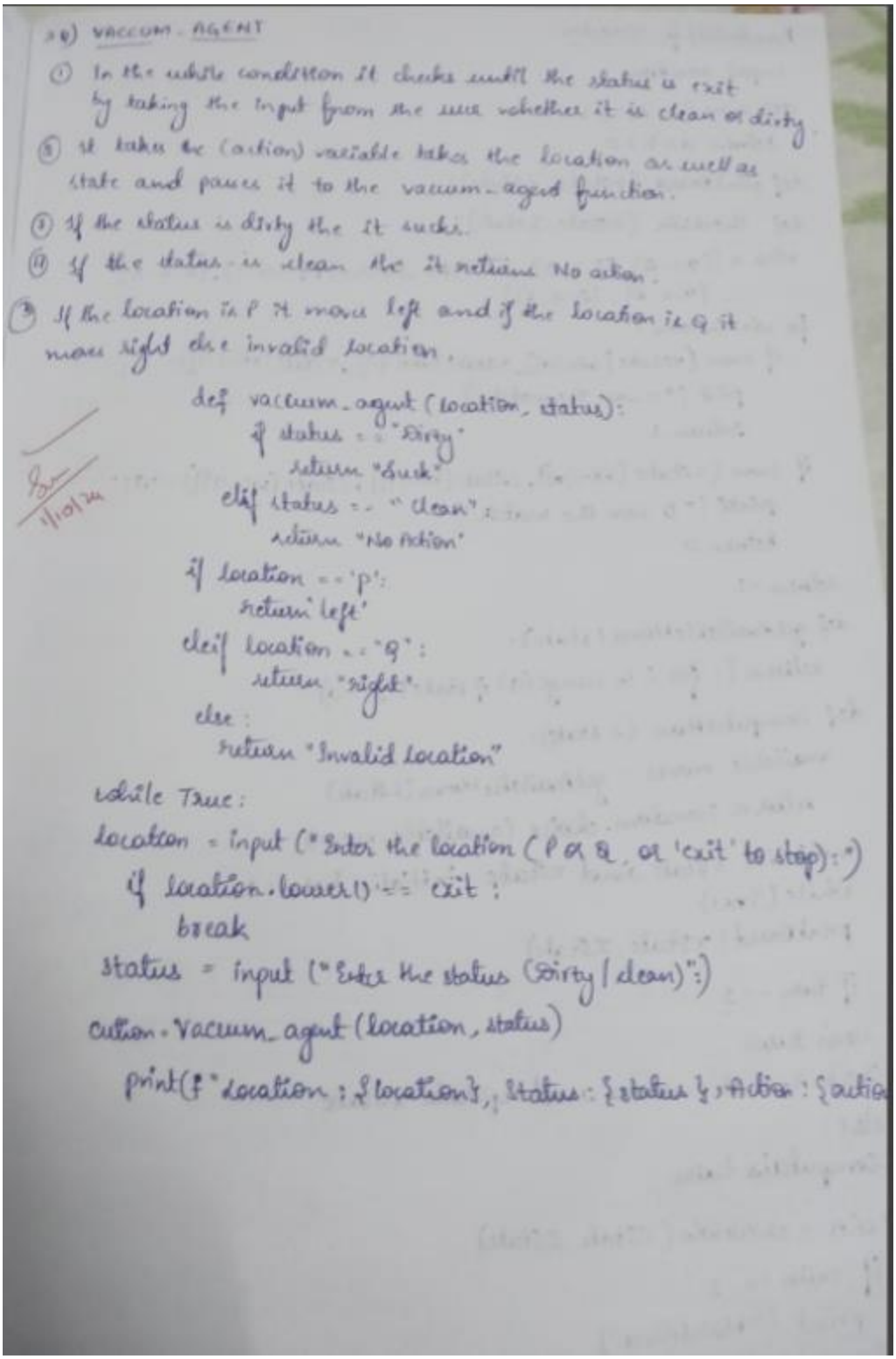
OUTPUT

```
Exploring depth limit: 1
Current node: A, Current path: ['A']
Exploring depth limit: 2
Current node: A, Current path: ['A']
Current node: C, Current path: ['A', 'C']
Current node: B, Current path: ['A', 'B']
Exploring depth limit: 3
Current node: A, Current path: ['A']
Current node: C, Current path: ['A', 'C']
Current node: G, Current path: ['A', 'C', 'G']
Path found: ['A', 'C', 'G']
```

LABORATORY PROGRAM – 4

Implement vacuum cleaner agent

PSEUDOCODE OR ALGORITHM



a) VACCUM - AGENT

① In the while condition it checks until the status is exit by taking the input from the user whether it is clean or dirty.

② It takes the (action) variable takes the location as well as state and passes it to the vacuum_agent function.

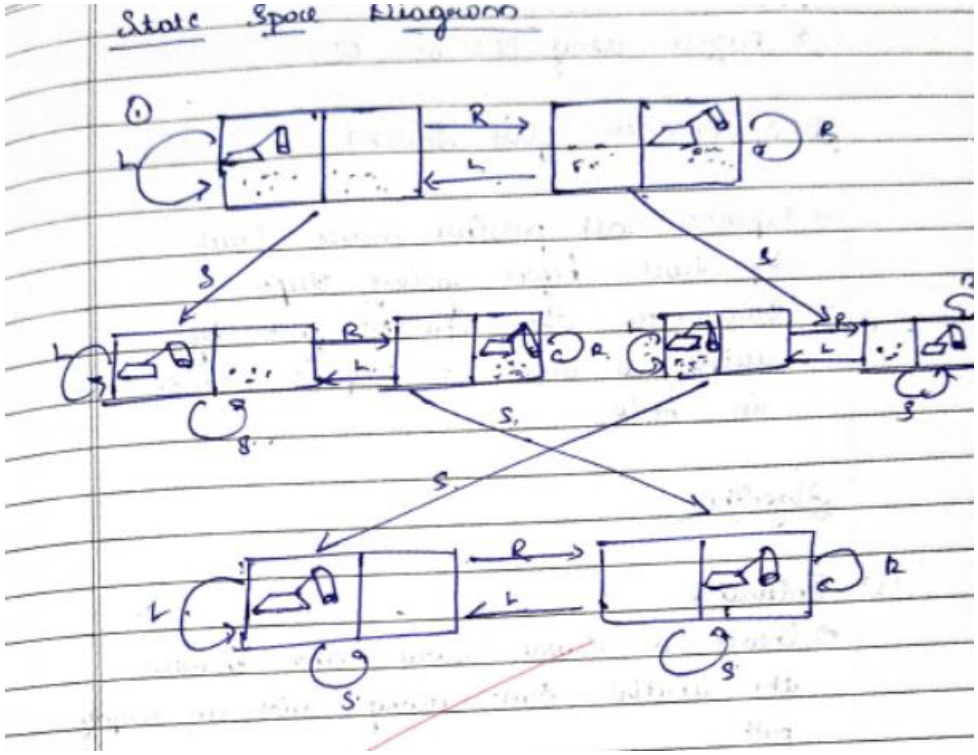
③ If the status is dirty the it sucks.

④ If the status is clean the it returns No action.

⑤ If the location is P it move left and if the location is Q it move right else invalid location.

```
def vaccum_agent (location, status):
    if status == "Dirty"
        return "Suck"
    elif status == "clean":
        return "No Action"
    if location == 'P':
        return 'left'
    elif location == "Q":
        return "right"
    else:
        return "Invalid location"
while True:
location = input ("Enter the location (P or Q, or 'exit' to stop):")
    if location.lower() == 'exit':
        break
status = input ("Enter the status (Dirty / clean)":)
action = vacuum_agent (location, status)
    print(f"Location : {location}, Status : {status}, Action : {actio
```

STATE SPACE TREE



CODE

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter the initial location of the vacuum cleaner (A or B): ")
    status_input = input(f"Enter the status of room {location_input} (0 for Clean, 1 for Dirty): ")
    status_input_complement = input(f"Enter the status of the other room ({'B' if location_input == 'A'
else 'A'}) (0 for Clean, 1 for Dirty): ")

    # Set the initial state based on user input
    initial_state = {
        'A': status_input if location_input == 'A' else status_input_complement,
        'B': status_input_complement if location_input == 'A' else status_input
    }

    print("\nInitial Location Condition:", initial_state)

    if location_input == 'A':
        print("\nVacuum cleaner is placed in room A.")

        if status_input == '1':
            print("Room A is Dirty. Cleaning room A...")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for cleaning room A:", cost)

            if status_input_complement == '1':
                print("\nRoom B is also Dirty. Moving to room B...")
                cost += 1
                print("Cost for moving to room B:", cost)
                print("Cleaning room B...")
                goal_state['B'] = '0'
                cost += 1
                print("Cost for cleaning room B:", cost)
            else:
                print("\nRoom B is already Clean. No further action needed.")

        else:
            print("Room A is already Clean.")

            if status_input_complement == '1':
                print("\nRoom B is Dirty. Moving to room B...")
                cost += 1
                print("Cost for moving to room B:", cost)
                print("Cleaning room B...")
                goal_state['B'] = '0'
                cost += 1
                print("Cost for cleaning room B:", cost)
            else:
                print("\nRoom B is already Clean. No further action needed.")

    else:
        print("\nVacuum cleaner is placed in room B.")
```

```
        if status_input == '1':
            print("Room B is Dirty. Cleaning room B...")
            goal_state['B'] = '0'
            cost += 1
            print("Cost for cleaning room B:", cost)

            if status_input_complement == '1':
                print("\nRoom A is also Dirty. Moving to room A...")
                cost += 1
                print("Cost for moving to room A:", cost)
                print("Cleaning room A...")
                goal_state['A'] = '0'
                cost += 1
                print("Cost for cleaning room A:", cost)
            else:
                print("\nRoom A is already Clean. No further action needed.")

        else:
            print("Room B is already Clean.")

            if status_input_complement == '1':
                print("\nRoom A is Dirty. Moving to room A...")
                cost += 1
                print("Cost for moving to room A:", cost)
                print("Cleaning room A...")
                goal_state['A'] = '0'
                cost += 1
                print("Cost for cleaning room A:", cost)
            else:
                print("\nRoom A is already Clean. No further action needed.")

    print("\nGOAL STATE:", goal_state)
    print("Total cost for cleaning:", cost)

vacuum_world()
```

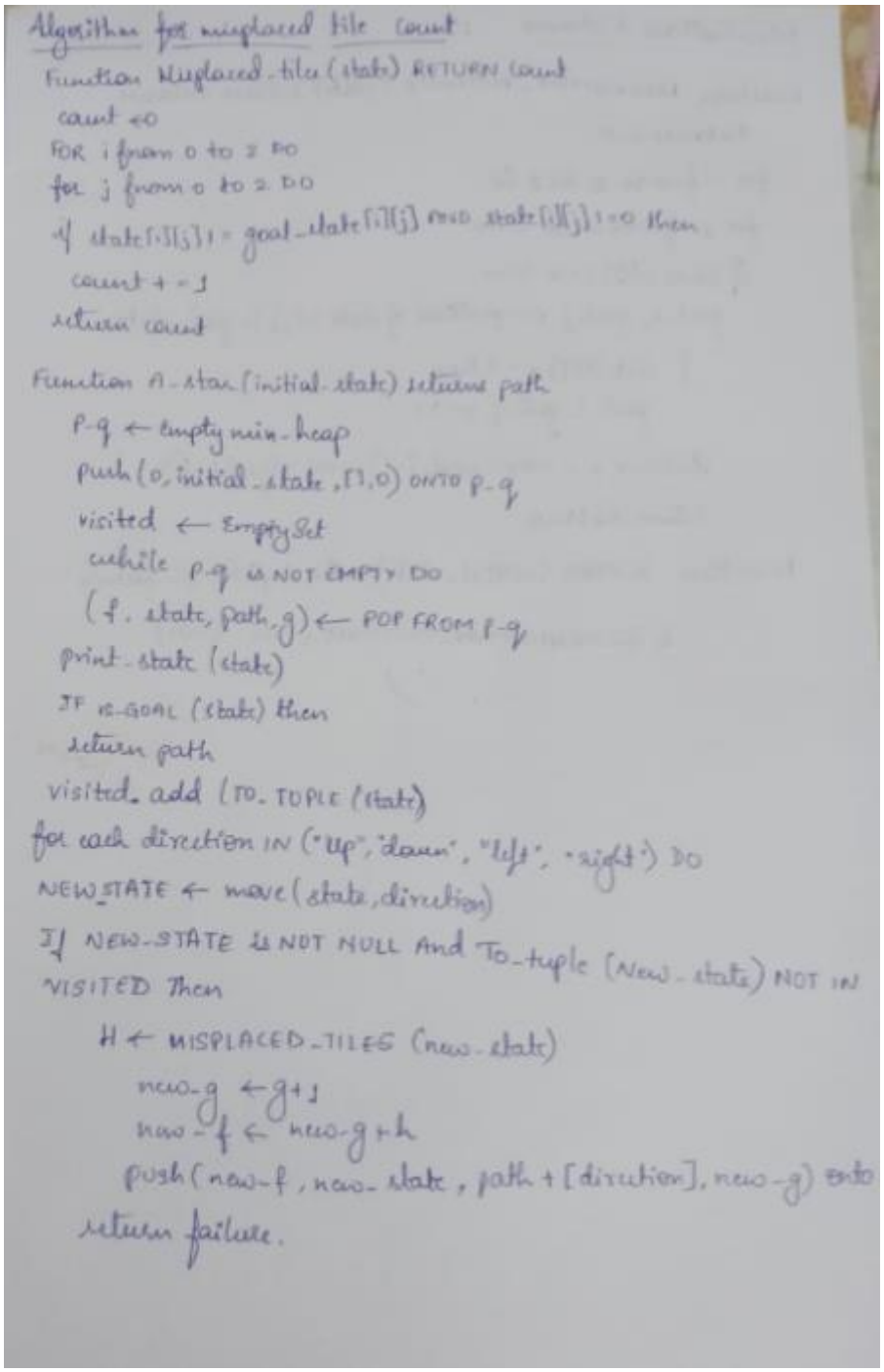
OUTPUT

```
Enter the location (P or Q, or 'exit' to stop): p
Enter the status (Dirty or Clean): dirty
Location: p, Status: dirty, Action: Invalid Location
Enter the location (P or Q, or 'exit' to stop): P
Enter the status (Dirty or Clean): Dirty
Location: P, Status: Dirty, Action: Suck
Enter the location (P or Q, or 'exit' to stop): P
Enter the status (Dirty or Clean): Clean
Location: P, Status: Clean, Action: No Action
Enter the location (P or Q, or 'exit' to stop): exit
```

LABORATORY PROGRAM – 5(A)

Implement A* search algorithm

PSEUDOCODE OR ALGORITHM



Algorithm for misplaced tile count

Function Misplaced_tile(state) RETURN count

count ←0

FOR i from 0 to 2 DO

for j from 0 to 2 DO

if state[i][j] = goal_state[i][j] AND state[i][j] != 0 then

count + = 1

return count

Function A-star(initial_state) returns path

P-q ← empty min-heap

Push(0, initial_state, [],0) ONTO p-q

visited ← EmptySet

while p-q is NOT EMPTY DO

(f, state, path, g) ← POP FROM p-q

print_state(state)

IF is-GOAL(state) then

return path

visited.add(TO_TUPLE(state))

for each direction IN ("Up", "down", "left", "right") DO

NEW_STATE ← move(state, direction)

If NEW_STATE IS NOT NULL And To_tuple(New_state) NOT IN VISITED Then

H ← MISPLACED_TILES(new_state)

new_g ← g+1

new_f ← new_g+h

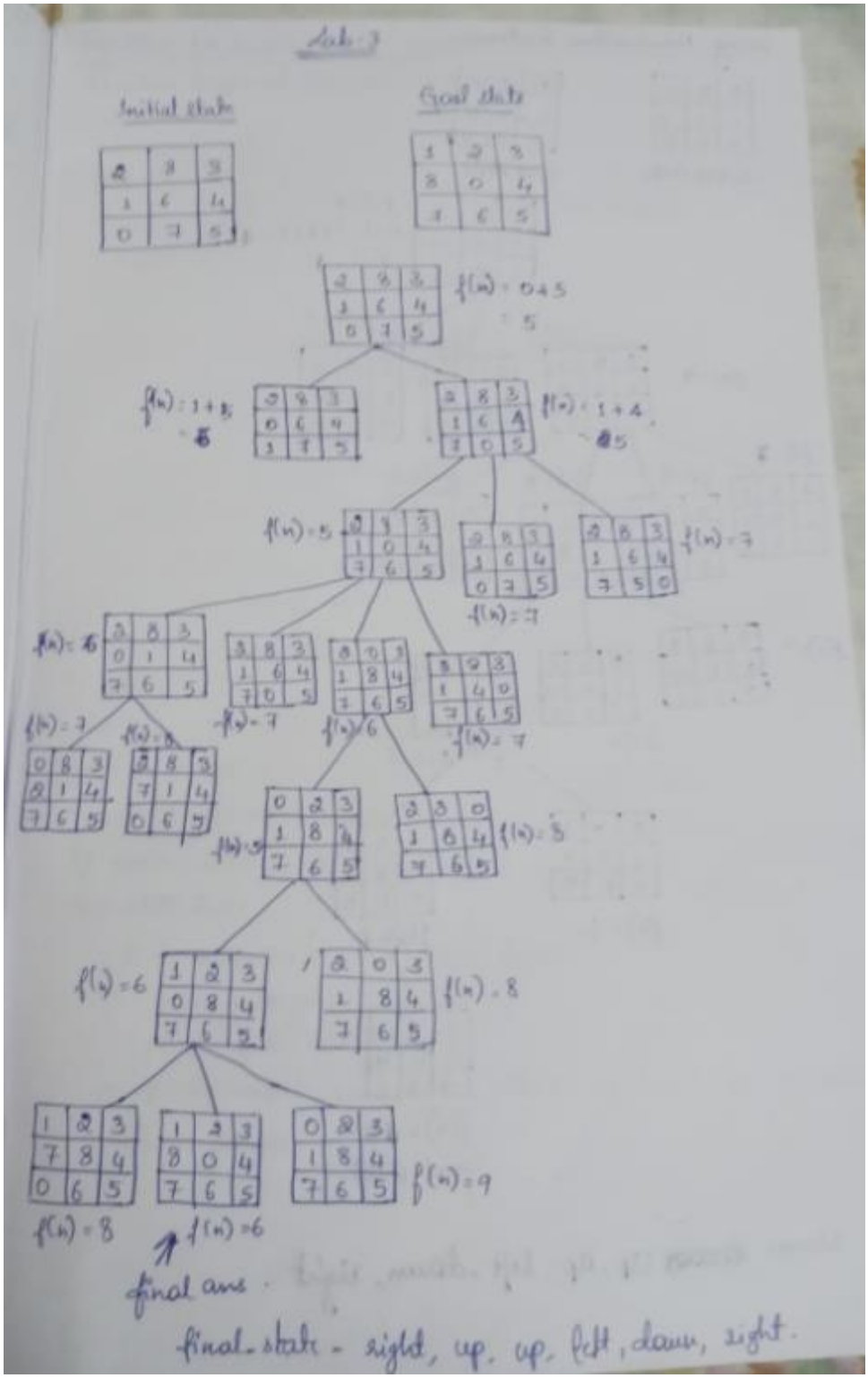
Push(new_f, new_state, path + [direction], new_g) onto

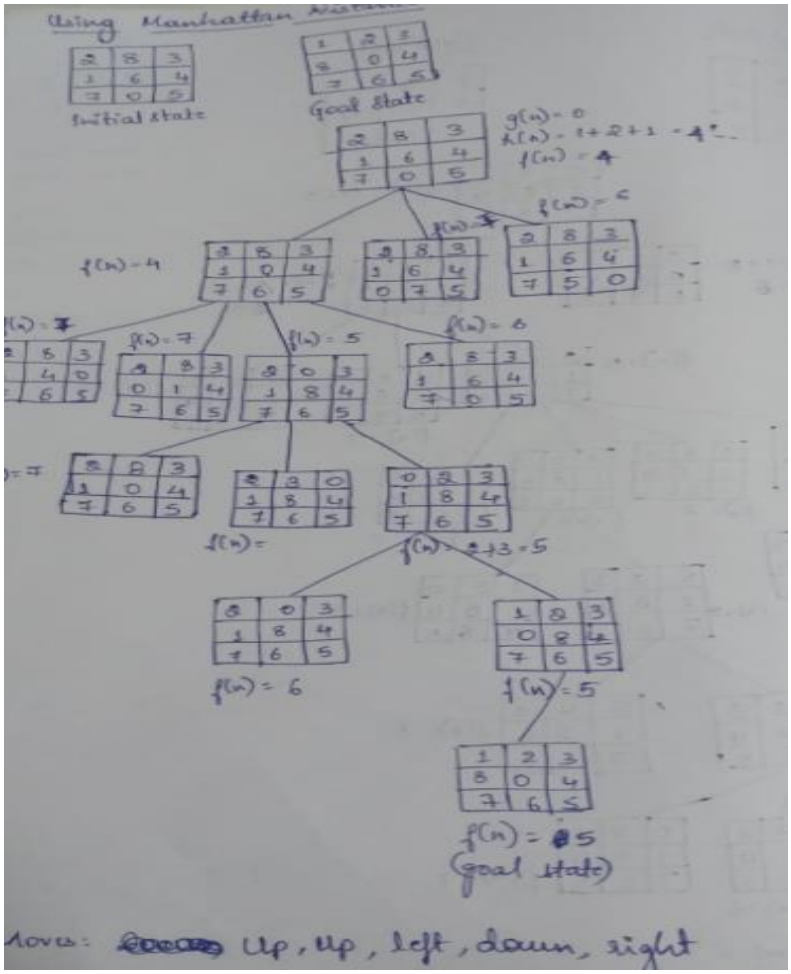
return failure.

Manhattan Distance

```
Function MANHATTAN_DISTANCE (state) return distance
    Distance ← 0
  for i from 0 to 2 do
  for j from 0 to 2 do
  if State[i][j] != 0 then
    goal_i, goal_j ← position of state[i][j] in goal_state
      if state[i][j] == 8 then
        goal_i, goal_j ← 1, 1
    Distance += ABS(goal_i - i) + ABS(goal i - j)
    return distance
```

STATE SPACE TREE




Using Manhattan
Initial state
Goal state
g(n) = 0
h(n) = 1+2+1 = 4
f(n) = 4
f(n) = 4
f(n) = 6
f(n) = 7
f(n) = 5
f(n) = 6
f(n) = 6
f(n) = 5
f(n) = 5
(goal state)
Moves: Up, Up, left, down, right

MISPLACED TILES

```
import heapq

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

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

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

def is_goal(state):
    return state == goal_state

def misplaced_tiles(state):
    return sum(1 for i in range(3) for j in range(3)
               if state[i][j] != goal_state[i][j] and state[i][j] != 0)

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(tuple(map(tuple, state)))

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = misplaced_tiles(new_state)
```

```
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

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

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
```

OUTPUT - MISPLACED TILES

```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Exploring state in A*:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

CODE - MANHATTAN DISTANCE

```
import heapq

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

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

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

def is_goal(state):
    return state == goal_state

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j] - 1, 3)
                if state[i][j] == 8:
                    goal_i, goal_j = 1, 1
                distance += abs(goal_i - i) + abs(goal_j - j)
    return distance

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [], 0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path
```

```
        visited.add(tuple(map(tuple, state)))

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state)) not in visited:
                h = manhattan_distance(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f, new_state, path + [direction], new_g))

    return None

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

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
```

OUTPUT - MANHATTAN DISTANCE

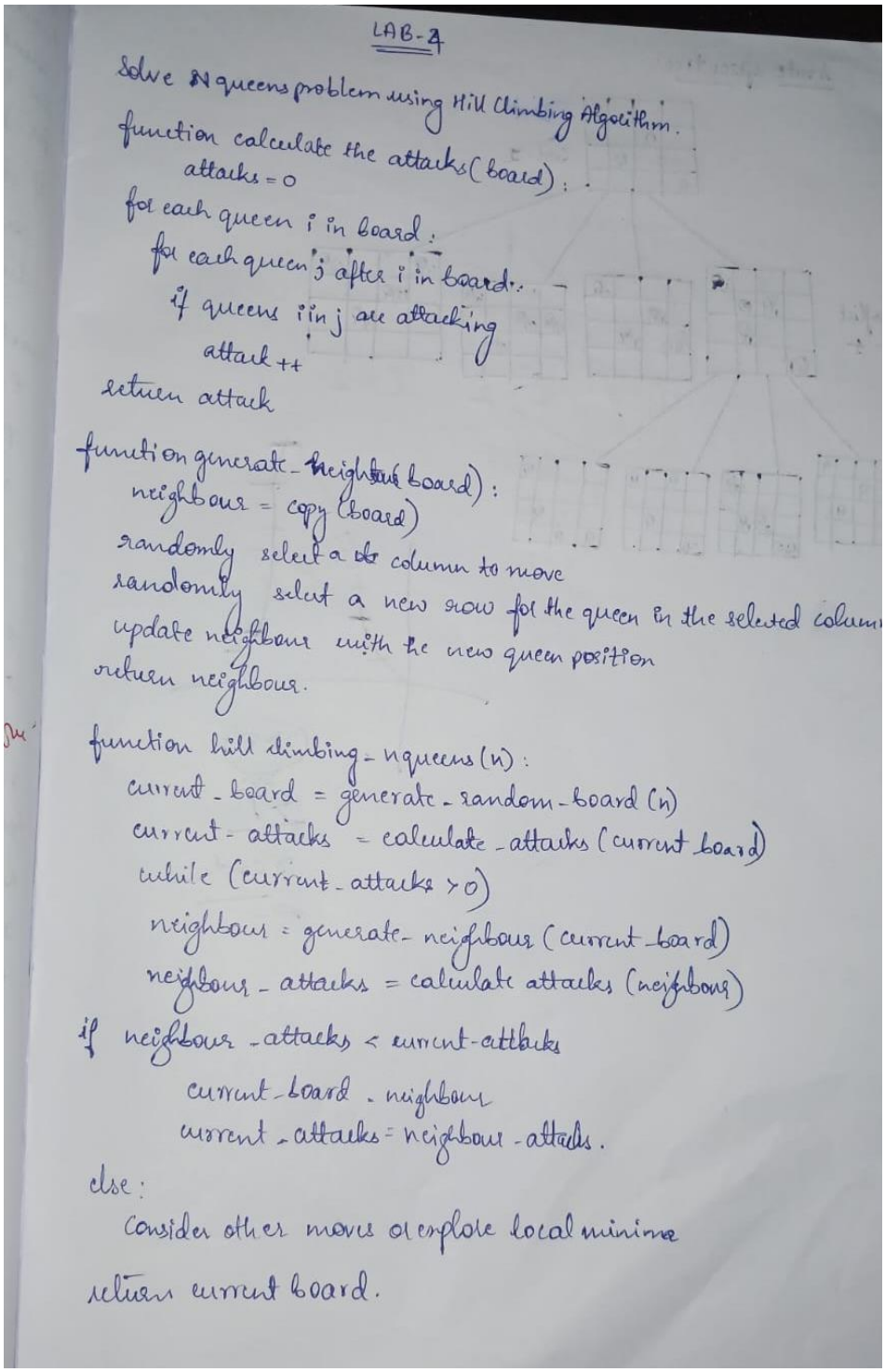
```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 5, 0]

Exploring state in A*:
[2, 8, 3]
[0, 6, 4]
[1, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
```

```
Exploring state in A*:
[2, 8, 3]
[1, 5, 6]
[7, 4, 0]

Exploring state in A*:
[0, 8, 3]
[2, 6, 4]
[1, 7, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 6]
[0, 5, 4]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

LABORATORY PROGRAM – 5(B)

Implement Hill Climbing Algorithm

PSEUDOCODE OR ALGORITHM

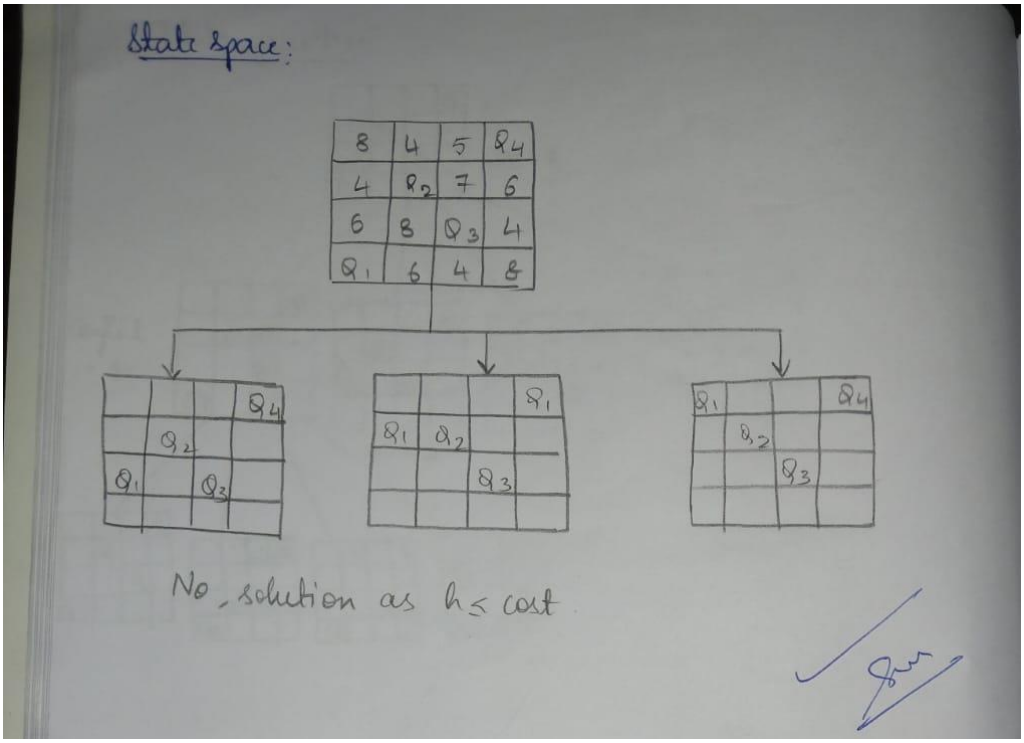


LAB-4

Solve N queens problem using Hill Climbing Algorithm.

```
function calculate the attacks(board):
    attacks = 0
    for each queen i in board:
        for each queen j after i in board:
            if queens i in j are attacking
                attack++
    return attack

function generate_neighbour(board):
    neighbour = copy(board)
    randomly select a column to move
    randomly select a new row for the queen in the selected column
    update neighbour with the new queen position
    return neighbour.

function hill climbing_nqueens(n):
    current_board = generate_random_board(n)
    current_attacks = calculate_attacks(current_board)
    while (current_attacks > 0)
        neighbour = generate_neighbour(current_board)
        neighbour_attacks = calculate attacks(neighbour)
if neighbour_attacks < current_attacks
        current_board = neighbour
        current_attacks = neighbour_attacks.
else:
    Consider other moves or explore local minima
return current board.
```

STATE SPACE TREE



CODE

```
import random

def calculate_cost(state):
    """Calculate the number of conflicts in the current state."""
    cost = 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):
                cost += 1
    return cost

def get_neighbors(state):
    """Generate all possible neighbors by moving each queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:  # Move the queen in column `col` to a different row
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(initial_state, max_iterations=1000):
    """Perform hill climbing search to solve the N-Queens problem."""
    current_state = initial_state
    current_cost = calculate_cost(current_state)

    for iteration in range(max_iterations):
        if current_cost == 0:
            return current_state

        neighbors = get_neighbors(current_state)
        neighbor_costs = [(neighbor, calculate_cost(neighbor)) for neighbor in neighbors]
        next_state, next_cost = min(neighbor_costs, key=lambda x: x[1])

        if next_cost >= current_cost:
            print(f"Local maximum reached at iteration {iteration}. Restarting...")
            return None
        current_state, current_cost = next_state, next_cost
        print(f"Iteration {iteration}: Current state: {current_state}, Cost: {current_cost}")

    print(f"Max iterations reached without finding a solution.")
    return None

try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")

    initial_state = list(map(int, input(f"Enter the initial state as a list of {n} integers (rows for each
column): ").split()))
```

```
    if len(initial_state) != n or any(not (0 <= row < n) for row in initial_state):
        raise ValueError(f"Invalid initial state. Please provide {n} integers between 0 and {n-1}.")
except ValueError as e:
    print(e)
    n = 4
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    print(f"Using random initial state: {initial_state}")

solution = None

while solution is None:
    solution = hill_climbing(initial_state)

print(f"Solution found: {solution}")
```

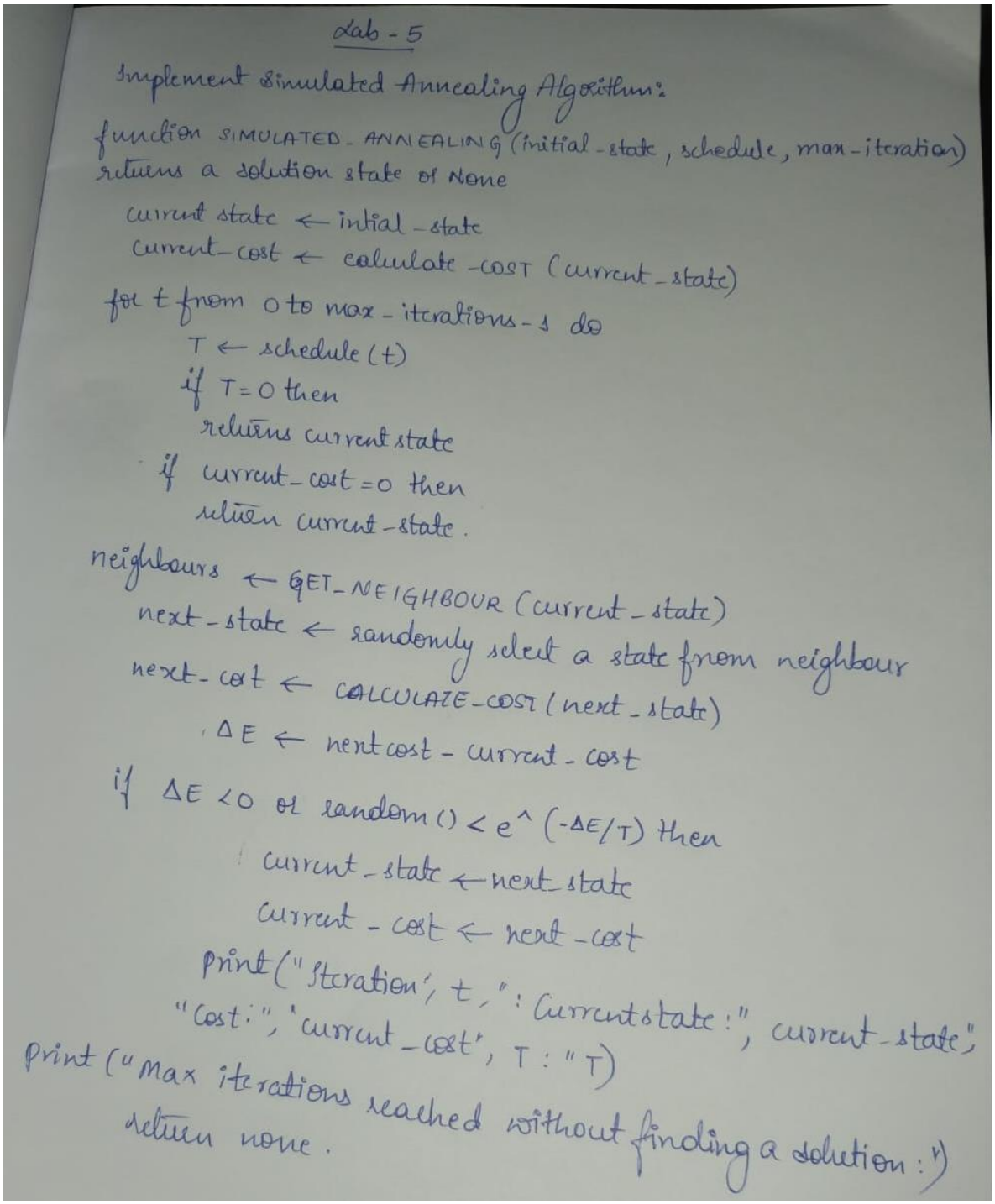
OUTPUT

```
Current State: [3, 1, 2, 0], Heuristic: 2
. . . Q
. Q . .
. . Q .
Q . . .

Current State: [1, 3, 2, 0], Heuristic: 1
. Q . .
. . . Q
. . Q .
Q . . .

Current State: [1, 3, 0, 2], Heuristic: 0
. Q . .
. . . Q
Q . . .
. . Q .

Solution found for 4-Queens problem: [1, 3, 0, 2]
. Q . .
. . . Q
Q . . .
. . Q .
```

LABORATORY PROGRAM – 6

implement Simulated Annealing Algorithm

PSEUDOCODE OR ALGORITHM

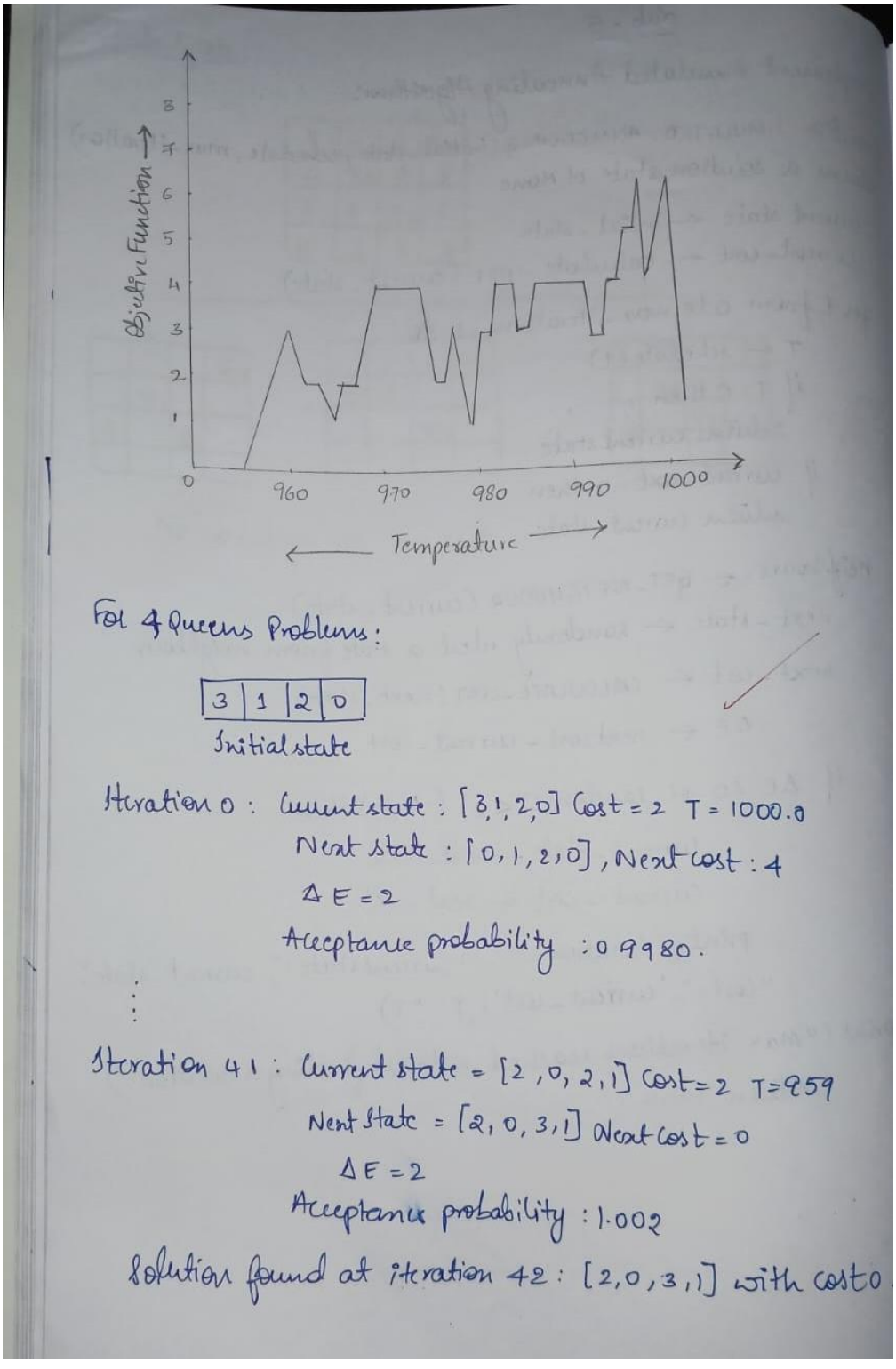


Lab - 5

Implement Simulated Annealing Algorithm:

```
function SIMULATED_ANNEALING (initial_state, schedule, max_iteration)
returns a solution state or None
    current state ← intial_state
    current_cost ← calculate_COST (current_state)
for t from 0 to max_iterations-1 do
        T ← schedule (t)
        if T=0 then
        returns current state
    if current_cost=0 then
        return current_state.
neighbours ← GET_NEIGHBOUR (current_state)
    next_state ← randomly select a state from neighbour
    next_cost ← CALCULATE_COST (next_state)
        ΔE ← next cost - current_cost
if ΔE <0 or random() < e^(-ΔE/T) then
            current_state ← next state
            current_cost ← next_cost
        Print("Iteration", t, ": Current state:", current_state,
    "cost:", "current_cost", T:"T)
Print ("Max iterations reached without finding a solution:")
    return none.
```

STATE SPACE TREE



For 4 Queens Problems:

3	1	2	0

Initial state

Iteration 0 : Current state : [3, 1, 2, 0] Cost = 2 T = 1000.0
Next state : [0, 1, 2, 0], Next cost : 4
$\Delta E = 2$
Acceptance probability : 0.9980.

⋮

Iteration 41 : Current state = [2, 0, 2, 1] Cost = 2 T = 959
Next State = [2, 0, 3, 1] Next cost = 0
$\Delta E = 2$
Acceptance probability : 1.002
Solution found at iteration 42 : [2, 0, 3, 1] with cost 0.

CODE

```
import random
import math
import matplotlib.pyplot as plt

def calculate_cost(state):
    cost = 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):
                cost += 1
    return cost

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def simulated_annealing_with_tracking(initial_state, schedule, max_iterations=1000):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    costs = []
    temperatures = []

    for t in range(max_iterations):
        T = schedule(t)
        if T == 0:
            break

        if current_cost == 0:
            costs.append(current_cost)
            temperatures.append(T)
            print(f"Solution found at iteration {t}: {current_state} with cost {current_cost}")
            break

        neighbors = get_neighbors(current_state)
        next_state = random.choice(neighbors)
        next_cost = calculate_cost(next_state)

        ΔE = next_cost - current_cost
        acceptance_probability = math.exp(-ΔE / T) if T > 0 else 0
        accept = ΔE < 0 or random.random() < acceptance_probability

        print(f"Iteration {t}:")
        print(f"  Current state: {current_state}, Cost: {current_cost}, Temperature (T): {T}")
```

```
            print(f"  Next state: {next_state}, Next cost: {next_cost}")
            print(f"  ΔE = {ΔE}")
            print(f"  Acceptance probability: {acceptance_probability}")
            print(f"  Acceptance condition met: {accept}")

            costs.append(current_cost)
            temperatures.append(T)

            if accept:
                current_state, current_cost = next_state, next_cost

        costs.append(current_cost)
        temperatures.append(T)
        return costs, temperatures

def linear_schedule(t, initial_temp=1000, final_temp=1, max_iter=1000):
    return max(final_temp, initial_temp - (initial_temp - final_temp) * (t / max_iter))

try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")

    initial_state = list(map(int, input(f"Enter the initial state as a list of {n} integers (rows for each
column): ").split()))
    if len(initial_state) != n or any(not (0 <= row < n) for row in initial_state):
        raise ValueError(f"Invalid initial state. Please provide {n} integers between 0 and {n-1}.")
except ValueError as e:
    print(e)
    n = 4
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    print(f"Using random initial state: {initial_state}")

costs, temperatures = simulated_annealing_with_tracking(initial_state, linear_schedule)

plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.plot(costs, label="Objective Function (Cost)")
plt.xlabel("Iterations")
plt.ylabel("Objective Function (Cost)")
plt.title("Objective Function (Cost) over Iterations")
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(temperatures, costs, label="Objective Function (Cost)")
plt.xlabel("Temperature")
plt.ylabel("Objective Function (Cost)")
plt.title("Objective Function (Cost) over Temperature")
plt.legend()

plt.tight_layout()
plt.show()

if costs[-1] == 0:
```

```
    print(f"Solution found: {initial_state}")
else:
    print("Max iterations reached without finding a solution.")
```

OUTPUT

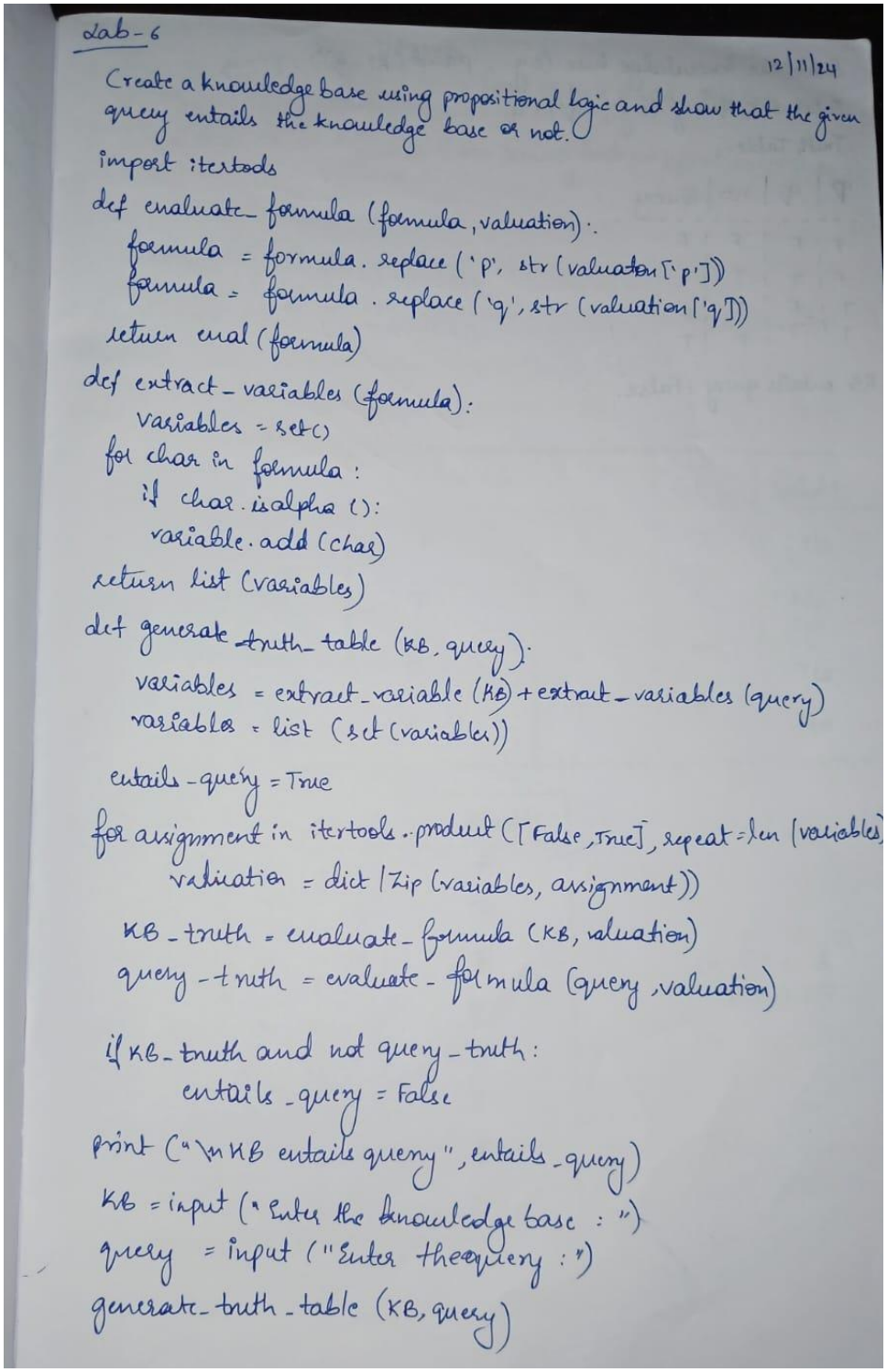
Output

```
Initial configuration:
. . . Q
. Q . .
Q . . .
. . Q .

Initial number of conflicts: 1

Solution found at iteration 3!
Final configuration (solution):
. Q . .
. . . Q
Q . . .
. . Q .

=== Code Execution Successful ===
```

LABORATORY PROGRAM – 7

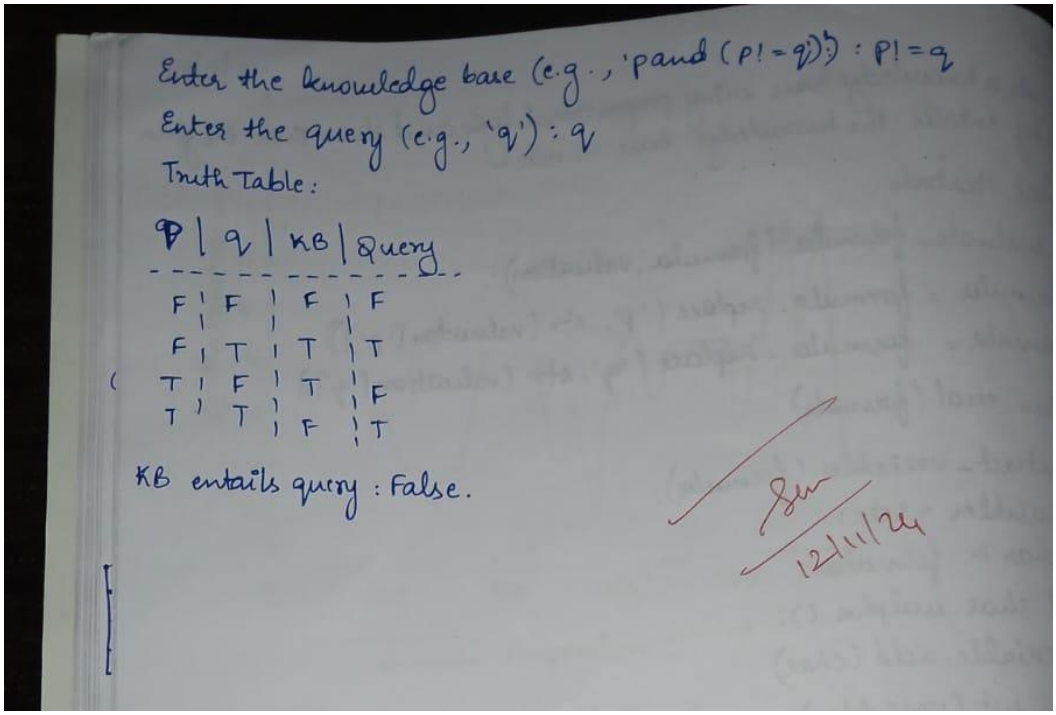
Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

PSEUDOCODE OR ALGORITHM



```
lab-6                                                    12/11/24
Create a knowledge base using propositional logic and show that the given
query entails the knowledge base or not.
import itertools
def evaluate_formula (formula, valuation):
    formula = formula.replace('p', str(valuation['p']))
    formula = formula.replace('q', str(valuation['q']))
    return eval(formula)
def extract_variables (formula):
    variables = set()
    for char in formula:
        if char.isalpha():
            variable.add(char)
    return list(variables)
def generate_truth_table (KB, query):
    variables = extract_variable(KB) + extract_variables(query)
    variables = list(set(variables))
    entails_query = True
    for assignment in itertools.product([False, True], repeat=len(variables)
        valuation = dict(zip(variables, assignment))
        KB_truth = evaluate_formula(KB, valuation)
        query_truth = evaluate_formula(query, valuation)
        if KB_truth and not query_truth:
            entails_query = False
    print("\nKB entails query", entails_query)
    KB = input("Enter the knowledge base : ")
    query = input("Enter the query : ")
    generate_truth_table(KB, query)
```

STATE SPACE TREE



Enter the knowledge base (e.g., 'p and (p!=q)'): p!=q
Enter the query (e.g., 'q'): q
Truth Table:

p	q	KB	Query
F	F	F	F
F	T	T	T
T	F	T	F
T	T	F	T

KB entails query : False.

12/11/24

CODE

```
from itertools import product

# Evaluate a logical formula using the assignment of truth values
def evaluate_formula(formula, assignment):
    return eval(formula, {}, assignment)

# Generate all possible truth assignments for the given variables
def generate_all_assignments(variables):
    return [dict(zip(variables, values)) for values in product([True, False],
repeat=len(variables))]

# Create knowledge base and query from user input
def create_knowledge_base():
    print("Please enter the meanings of the following propositions:")
    p = input("Enter the meaning of proposition p : ")
    q = input("Enter the meaning of proposition q : ")
    r = input("Enter the meaning of proposition r : ")

    print(f"\nYou defined the following propositions:")
    print(f"p: {p}")
    print(f"q: {q}")
    print(f"r: {r}")

    print("\nNow, define the knowledge base (KB) and query (Q) using these propositions.")
    print("You can use 'p', 'q', 'r', 'not', 'or', 'and', and parentheses in your formulas.")

    KB = input("\nEnter the knowledge base (KB) formula : ")
    Q = input("\nEnter the query (Q) formula: ")

    return p, q, r, KB, Q

# Check if the knowledge base (KB) entails the query (Q) by evaluating the truth table
def truth_table_entailment(KB, Q, variables):
    all_assignments = generate_all_assignments(variables)

    print(f"\n{'p':<8}{'q':<8}{'r':<8}{'KB':<8}{'Q':<8}{'Entails'}")

    for assignment in all_assignments:
        KB_value = evaluate_formula(KB, assignment)
        Q_value = evaluate_formula(Q, assignment)
        entails = "Yes" if KB_value == True and Q_value == True else "No"

print(f"{assignment['p']:<8}{assignment['q']:<8}{assignment['r']:<8}{KB_value:<8}{Q_valu
e:<8}{entails}")

        if KB_value == True and Q_value == False:
            return False
    return True
```

```
# Main execution
if __name__ == "__main__":
    p, q, r, KB, Q = create_knowledge_base()
    variables = ['p', 'q', 'r']

    # Check if the KB entails the query Q
    result = truth_table_entailment(KB, Q, variables)

    if result:
        print("\nKB entails Q.")
    else:
        print("\nKB does not entail Q.")
```

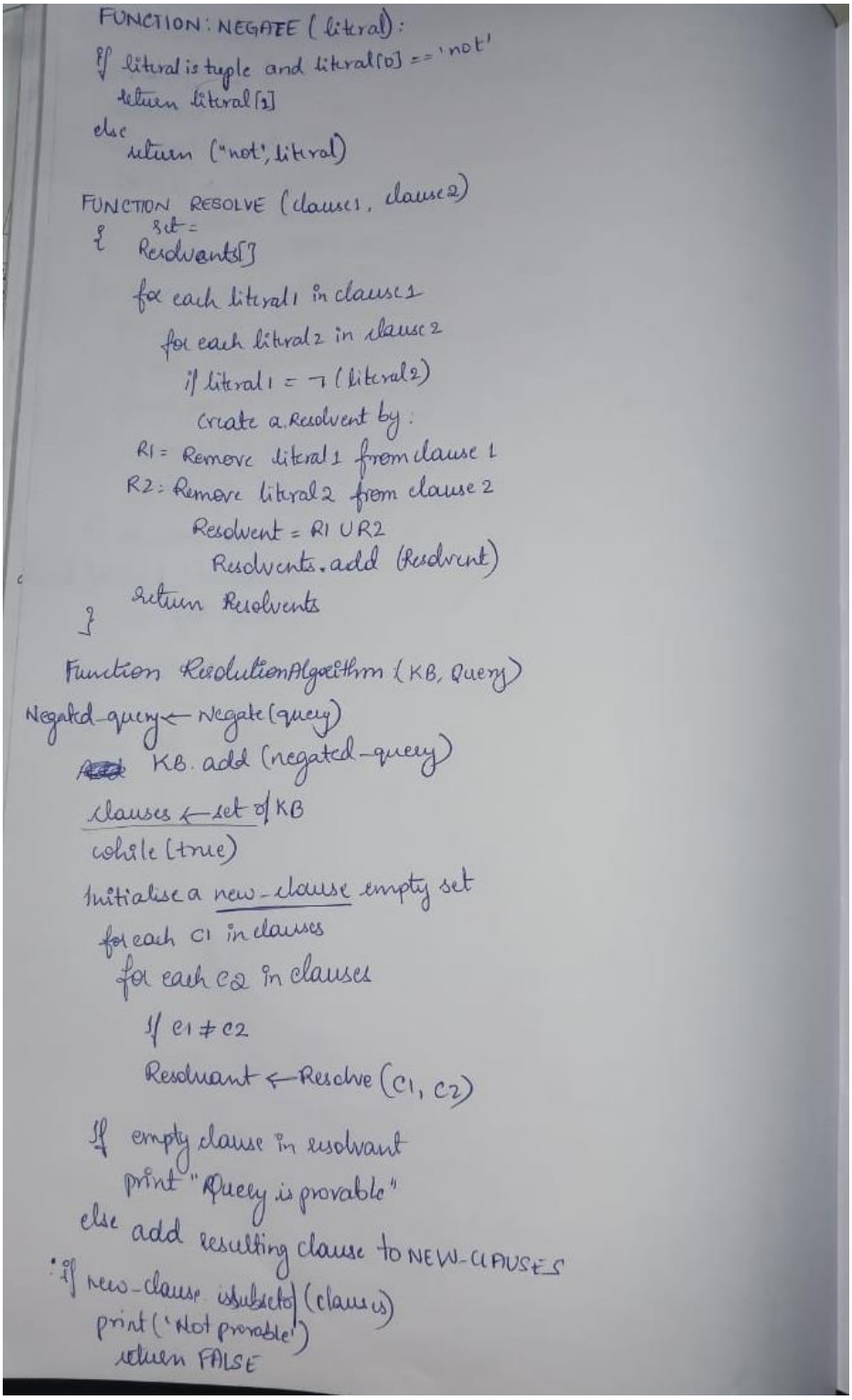
OUTPUT

```
Please enter the meanings of the following propositions:
Enter the meaning of proposition p : "It is raining"
Enter the meaning of proposition q : "The ground is wet"
Enter the meaning of proposition r : "The sun is shining"

You defined the following propositions:
p: "It is raining"
q: "The ground is wet"
r: "The sun is shining"

Now, define the knowledge base (KB) and query (Q) using these propositions.
You can use 'p', 'q', 'r', 'not', 'or', 'and', and parentheses in your formulas.

Enter the knowledge base (KB) formula : p and q

Enter the query (Q) formula: q

p       q       r       KB      Q       Entails
1       1       1       1       1       Yes
1       1       0       1       1       Yes
1       0       1       0       0       No
1       0       0       0       0       No
0       1       1       0       1       No
0       1       0       0       1       No
0       0       1       0       0       No
0       0       0       0       0       No

KB entails Q.
```

LABORATORY PROGRAM – 8

Create a knowledge base using prepositional logic and prove the given query using resolution.

PSEUDOCODE OR ALGORITHM



```
FUNCTION: NEGATEE ( literal):
if literal is tuple and literal[0] == 'not'
  return literal[1]
else
  return ("not', literal)

FUNCTION RESOLVE (clause1, clause2)
{   set =
  Resolvents[]
    for each literal1 in clause1
      for each literal2 in clause2
        if literal1 = ¬ (literal2)
          Create a Resolvent by:
    R1 = Remove literal1 from clause 1
    R2: Remove literal2 from clause 2
        Resolvent = R1 UR2
          Resolvents.add (Resolvent)
    return Resolvents
}

Function ResolutionAlgorithm (KB, Query)
Negated_query ← Negate (query)
  KB. add (negated_query)
  clauses ← set of KB
  while (true)
  initialise a new_clause empty set
   for each c1 in clauses
     for each c2 in clauses
       if c1 ≠ c2
       Resolvant ← Resolve (c1, c2)
   if empty clause in resolvant
     print "Query is provable"
   else add resulting clause to NEW-CLAUSES
   if new_clause. issubset (clauses)
     print ('Not provable')
     return FALSE
```

IMPLEMENTATION OF FORWARD CHAINING

Fact:

America (Robert)
Enemy (A, America)
Owns (A, T1)
Missile (T1)
Sells Weapon (Robert, T1)

Rules:

Missile (x) → Weapon (x)
Enemy (x, America) → Hostile (x)
America (p) and sellsWeapon (p, q) and Enemy (q, America) → Criminal (p

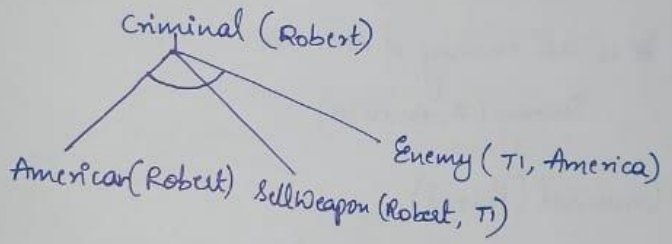
Goal:

Criminal (Robert)

FOL:

∀p ∀q (American(p) ∧ Sells Weapon (p, q) ∧ Enemy (q, America) → Criminal(

PROOF TREE:



PROOF BY RESOLUTION

① It's a crime for an American to sell weapon to hostile nation.

∀p ∀q ∀r (American(p) ∧ Weapon (q) ∧ Sells (p, q, r) ∧ Hostile (r
⇒ Criminal (p))

¬ American (p) ∨ ¬ Weapon (q) ∨ ¬ Sells (p, q, r) ∨ ¬ Hostile(r)
∨ Criminal (p)

2. Country A has some missile

$\exists x \, (Owns(A, x) \wedge Missile(x))$

$Owns(x, T_1) \wedge Missile(T_1)$

3. All Missiles were sold to country A by Robert

$\forall x \, (Missile(x) \wedge Owns(A, x) \Rightarrow Sells(Robert, x, A))$

$\neg Missile(x) \vee \neg Owns(A, x) \vee Sells(Robert, x, A)$

4. Missiles are weapons

$\forall x \, (Missile(x) \Rightarrow Weapon(x))$

$\neg Missile(x) \vee Weapon(x)$

5. Enemy of America is hostile

$\forall x \, (Enemy(x, America) \Rightarrow Hostile(x))$

$\neg Enemy(x, America) \vee Hostile(x)$

6. Robert is an American

$American(Robert)$

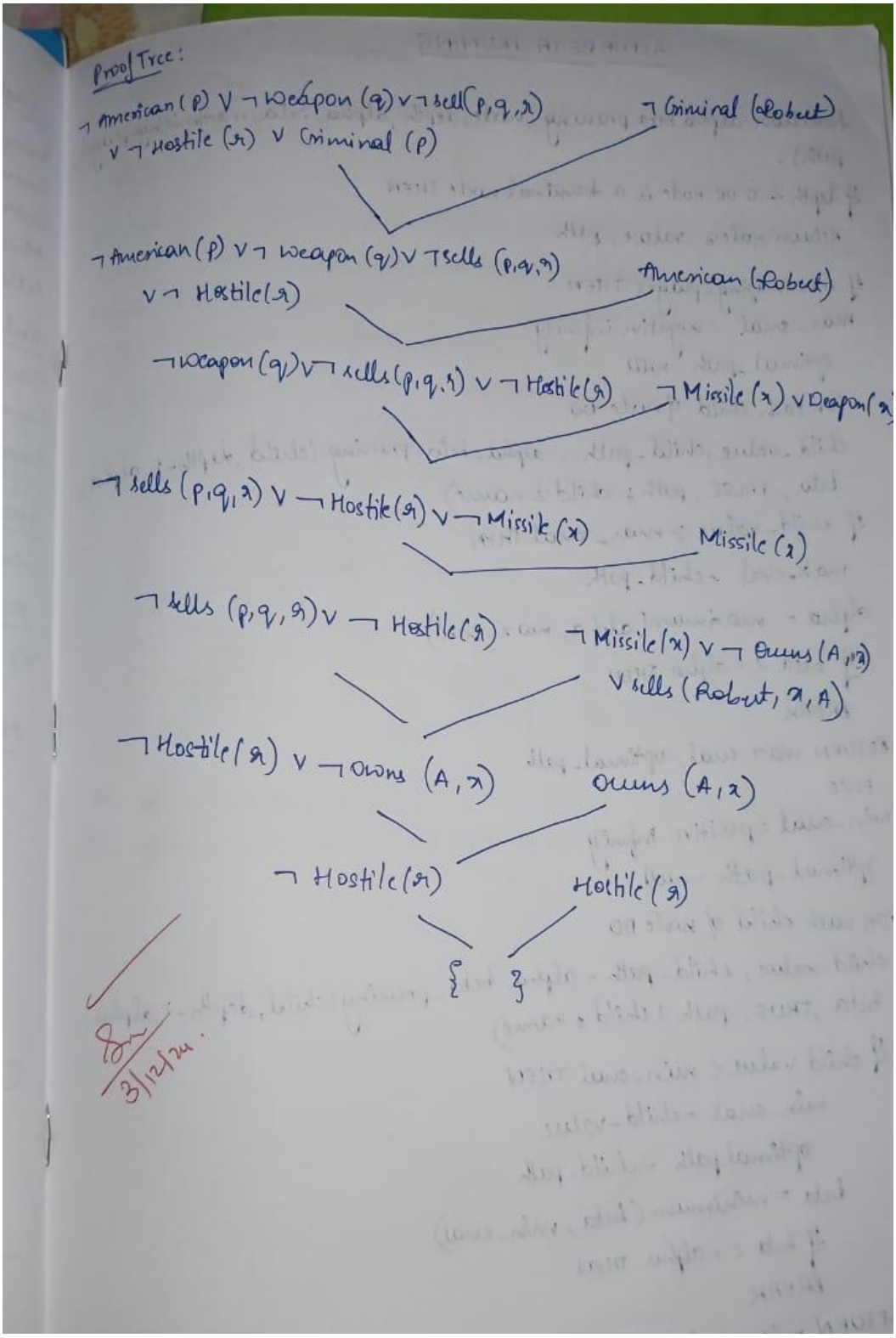
7. Country A is an enemy of America

$Enemy(A, America)$

Goal : $Criminal(Robert)$

Resolution : $\neg Criminal(Robert)$

STATE SPACE TREE



CODE

```
from typing import List, Set, Dict, Union

def unify(literal1: str, literal2: str) -> Union[Dict[str, str], None]:
"""
Unify two literals and return a substitution dictionary, or None if they cannot be unified.
"""
if literal1 == literal2:
return {}
if literal1.startswith("~") and literal2.startswith("~"):
return None
if literal1.startswith("~"):
neg, pos = literal1, literal2
else:
neg, pos = literal2, literal1

if neg[1:] == pos:
return {}
return None

def apply_substitution(clause: Set[str], substitution: Dict[str, str]) -> Set[str]:
"""
Apply a substitution to a clause.
"""
new_clause = set()
for literal in clause:
for var, value in substitution.items():
literal = literal.replace(var, value)
new_clause.add(literal)
return new_clause

def resolve(clause1: Set[str], clause2: Set[str]) -> Union[Set[str], None]:
"""
Resolves two clauses. Returns the resolvent clause or None if resolution is not possible.
"""
for lit1 in clause1:
for lit2 in clause2:
substitution = unify(lit1, lit2)
if substitution is not None:
# Create a new clause with unified literals removed
new_clause = (clause1 - {lit1}) | (clause2 - {lit2})
return apply_substitution(new_clause, substitution)
return None

def resolution(knowledge_base: List[Set[str]], query: Set[str]) -> bool:
"""
Implements the resolution algorithm.
"""
```

```
# Negate the query and add it to the knowledge base
negated_query = {f"~{literal}" if not literal.startswith("~") else literal[1:] for literal in query}
clauses = knowledge_base + [negated_query]

new_clauses = set()

while True:
pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))]
for clause1, clause2 in pairs:
resolvent = resolve(clause1, clause2)
if resolvent is not None:
if not resolvent:  # Empty clause found
return True
new_clauses.add(frozenset(resolvent))

# If no new clauses are generated, resolution has failed
if all(frozenset(c) in new_clauses for c in clauses):
return False

# Add new clauses to the set of clauses
clauses.extend(map(set, new_clauses))

if __name__ == "__main__":
print("Enter knowledge base (clauses) as sets of literals (comma-separated).")
print("Example: Likes(John, Food), Food(Apple). Enter 'done' when finished.")

knowledge_base = []
while True:
clause = input("Clause: ").strip()
if clause.lower() == "done":
break
knowledge_base.append(set(lit.strip() for lit in clause.split(',')))

print("Enter query as a set of literals (comma-separated).")
query = set(lit.strip() for lit in input("Query: ").strip().split(','))

result = resolution(knowledge_base, query)
print("Result:", "Entailed (True)" if result else "Not Entailed (False)")
```

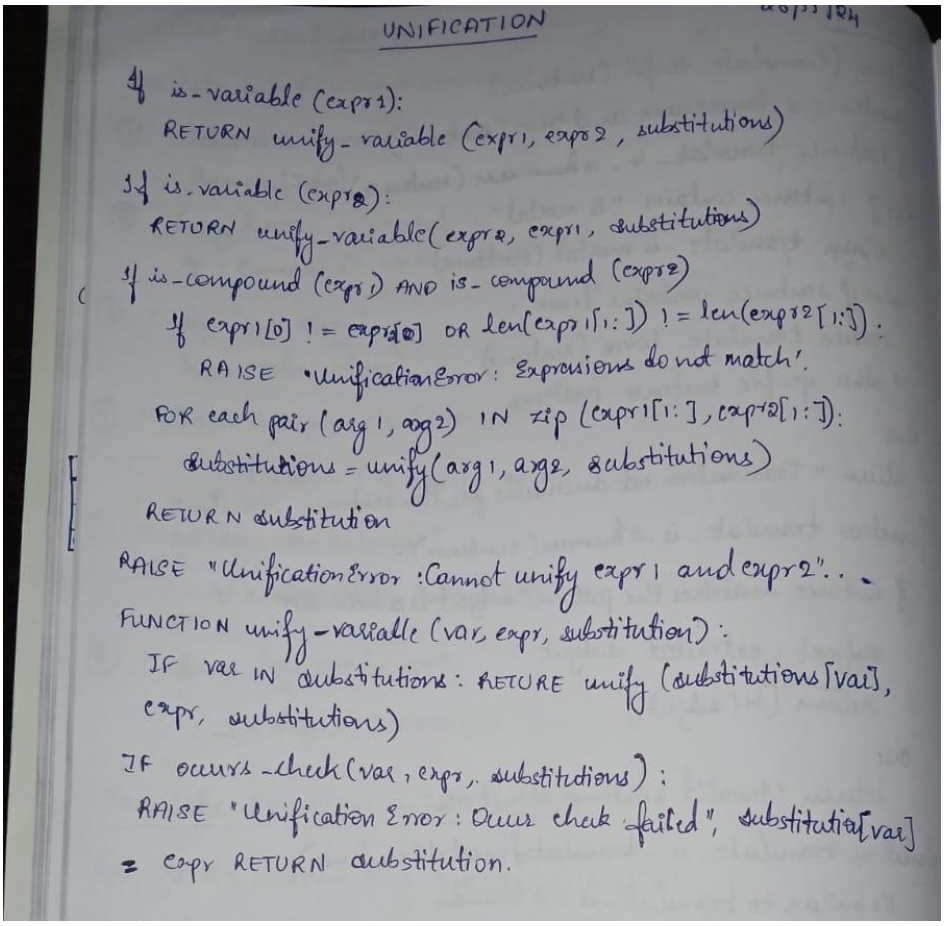
OUTPUT

```
Enter knowledge base (clauses) as sets of literals (comma-separated).
Example: Likes(John, Food), Food(Apple). Enter 'done' when finished.
Clause: Likes(John, Food), Food(Apple)
Clause: ~Likes(John, Apple)
Clause: done
Enter query as a set of literals (comma-separated).
Query: Likes(John, Apple)
Result: Entailed (True)
```

LABORATORY PROGRAM – 9

Implement unification in first order logic.

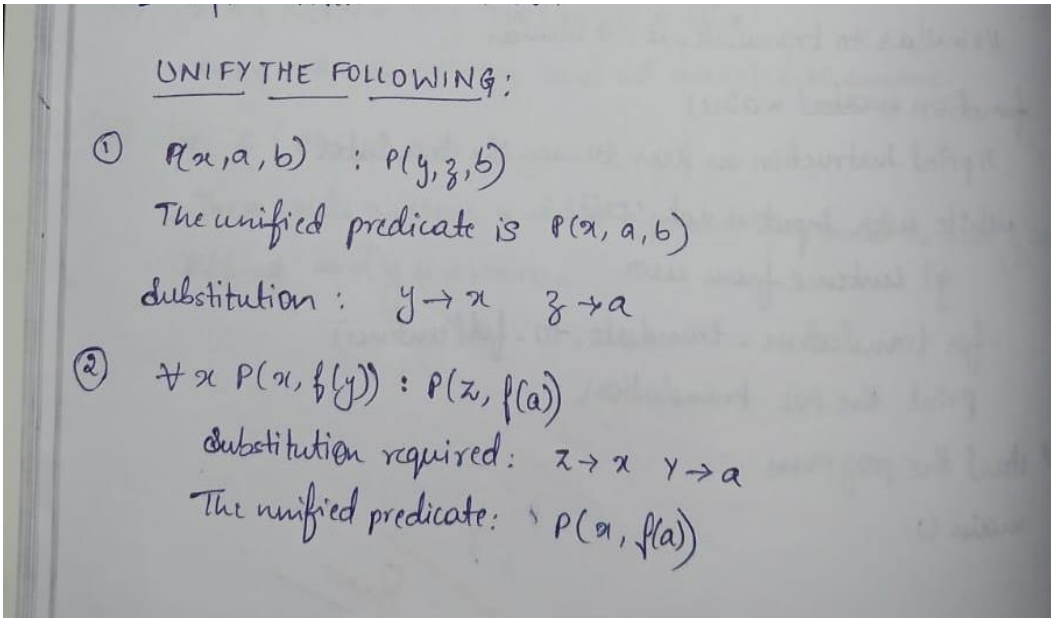
PSEUDOCODE OR ALGORITHM



UNIFICATION

```
If is_variable(expr1):
    RETURN unify_variable(expr1, expr2, substitutions)
If is_variable(expr2):
    RETURN unify_variable(expr2, expr1, substitutions)
If is_compound(expr1) AND is_compound(expr2)
    If expr1[0] != expr2[0] OR len(expr1[1:]) != len(expr2[1:]):
        RAISE "UnificationError: Expressions do not match".
    FOR each pair (arg1, arg2) IN zip(expr1[1:], expr2[1:]):
        substitutions = unify(arg1, arg2, substitutions)
    RETURN substitution
RAISE "UnificationError: Cannot unify expr1 and expr2"..
FUNCTION unify_variable(var, expr, substitution):
    IF var IN substitutions: RETURE unify(substitutions[var],
    expr, substitutions)
IF occurs_check(var, expr, substitutions):
    RAISE "Unification Error: Occurs check failed", substitution[var]
= expr RETURN substitution.
```

STATE SPACE TREE



UNIFY THE FOLLOWING:

① $P(x,a,b)$: $P(y,z,b)$

The unified predicate is $P(x,a,b)$

substitution : $y \to x$ $z \to a$

② $\forall x\ P(x, f(y))$: $P(z, f(a))$

substitution required: $z \to x$ $y \to a$

The unified predicate: $P(a, f(a))$

CODE

```
#Laboratory - 9
#Implement unification in first order logic.

class UnificationError(Exception):
    pass

def unify(expr1, expr2, substitutions=None):
    if substitutions is None:
        substitutions = {}

    # If both expressions are identical, return current substitutions
    if expr1 == expr2:
        return substitutions

    # If the first expression is a variable
    if is_variable(expr1):
        return unify_variable(expr1, expr2, substitutions)

    # If the second expression is a variable
    if is_variable(expr2):
        return unify_variable(expr2, expr1, substitutions)

    # If both expressions are compound expressions
    if is_compound(expr1) and is_compound(expr2):
        if expr1[0] != expr2[0] or len(expr1[1:]) != len(expr2[1:]):
            raise UnificationError("Expressions do not match.")
        return unify_lists(expr1[1:], expr2[1:], unify(expr1[0], expr2[0], substitutions))

    # If expressions are not compatible
    raise UnificationError(f"Cannot unify {expr1} and {expr2}.")

def unify_variable(var, expr, substitutions):
    if var in substitutions:
        return unify(substitutions[var], expr, substitutions)
    elif occurs_check(var, expr, substitutions):
        raise UnificationError(f"Occurs check failed: {var} in {expr}.")
    else:
        substitutions[var] = expr
        return substitutions

def unify_lists(list1, list2, substitutions):
    for expr1, expr2 in zip(list1, list2):
        substitutions = unify(expr1, expr2, substitutions)
    return substitutions

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

```
def is_compound(term):
    return isinstance(term, (list, tuple)) and len(term) > 0

def occurs_check(var, expr, substitutions):
    if var == expr:
        return True
    elif is_compound(expr):
        return any(occurs_check(var, sub, substitutions) for sub in expr)
    elif expr in substitutions:
        return occurs_check(var, substitutions[expr], substitutions)
    return False

# Function to parse input into a usable expression format
def parse_expression(expr_str):
    # Try to evaluate the expression as a tuple or list
    try:
        expr = eval(expr_str)
        if isinstance(expr, (tuple, list)) and len(expr) > 0:
            return expr
        else:
            raise ValueError("Expression must be a non-empty tuple or list.")
    except Exception as e:
        raise ValueError(f"Invalid expression format: {e}")

# Example usage: allow user input for the expressions
try:
    expr1_str = input("Enter the first expression (e.g., ('f', 'x', ('g', 'y'))): ")
    expr2_str = input("Enter the second expression (e.g., ('f', 'a', ('g', 'b'))): ")

    # Parse the user input expressions
    expr1 = parse_expression(expr1_str)
    expr2 = parse_expression(expr2_str)

    # Perform unification
    result = unify(expr1, expr2)
    print("Unified substitutions:", result)
except UnificationError as e:
    print("Unification failed:", e)
except ValueError as e:
    print("Input error:", e)
```

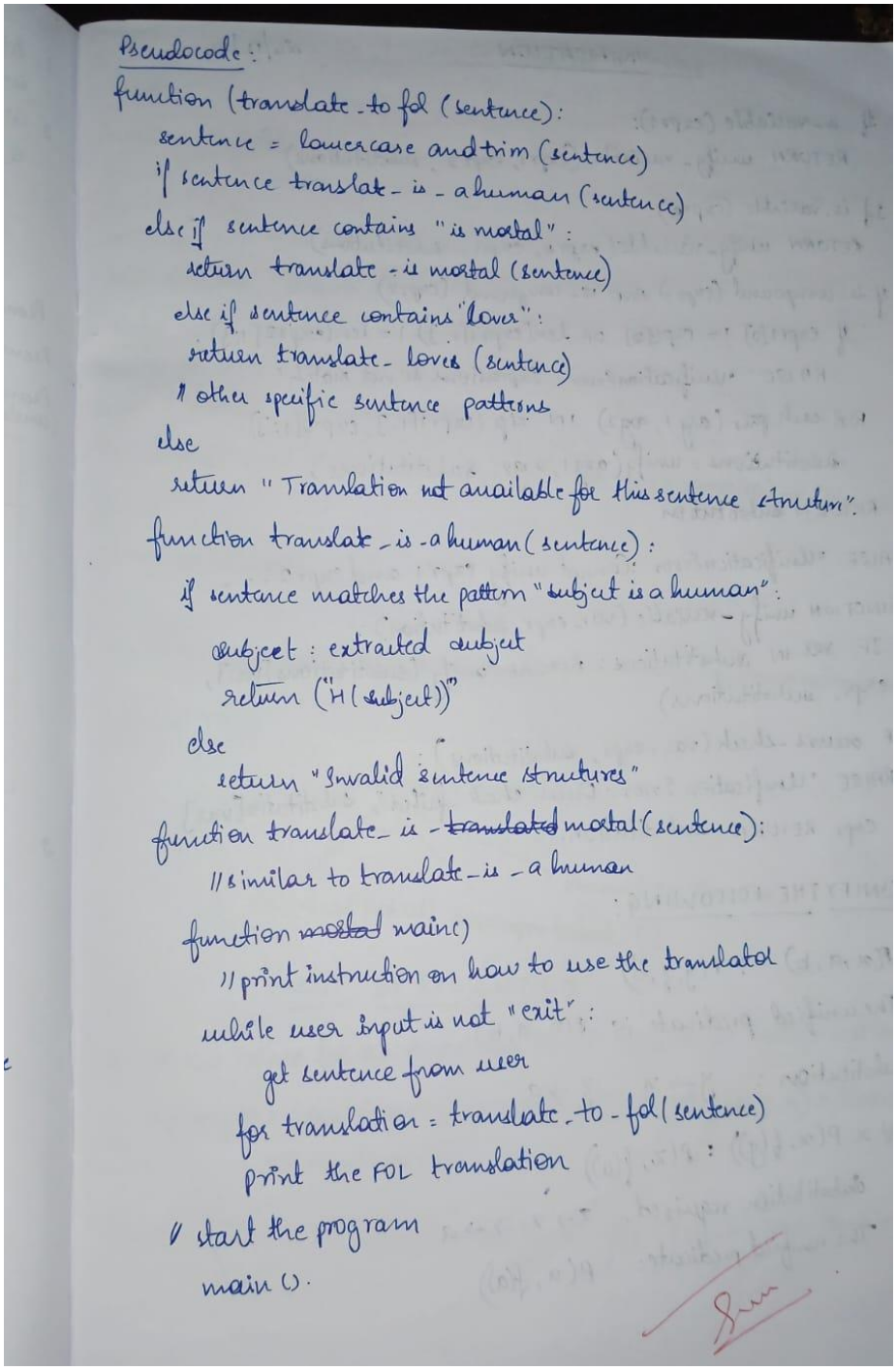
OUTPUT

```
Enter the first expression (e.g., ('f', 'x', ('g', 'y'))): ('P', 'x', 'a', 'b')
Enter the second expression (e.g., ('f', 'a', ('g', 'b'))): ('P', 'y', 'z', 'b')
Unified substitutions: {'x': 'y', 'a': 'z'}
```

LABORATORY PROGRAM – 10

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

PSEUDOCODE OR ALGORITHM



Pseudocode:

```
function (translate_to_fol (sentence):
    sentence = lowercase and trim (sentence)
    if sentence translat_is_a human (sentence)
    else if sentence contains "is mortal":
        return translate_is mortal (sentence)
    else if sentence contains "loves":
        return translate_loves (sentence)
    # other specific sentence patterns
    else
    return "Translation not available for this sentence structure"
function translat_is_a human (sentence):
    if sentence matches the pattern "subject is a human":
        subject : extracted subject
        return ("H(subject))"
    else
        return "Invalid sentence structures"
function translate_is_~~translated~~ mortal (sentence):
    // similar to translate_is_a human
function ~~mortal~~ main()
    // print instruction on how to use the translator
    while user input is not "exit":
        get sentence from user
        for translation = translate_to_fol (sentence)
        print the FOL translation
// start the program
main().
```

STATE SPACE TREE

First Order Logic:

Translate into FOL and vice versa:

① "John is a human".

Human(John)

② "Every human is mortal

$\forall x$ Human(x) $\Rightarrow$ Mortal(x)

③ "John loves Mary"

loves(John, Mary)

④ "There is someone who loves Mary"

$\exists x$ loves(x, Mary)

⑤ "All dogs are animals"

$\forall x$ Dogs(x) $\Rightarrow$ Animals(x)

⑥ "Some dogs are brown"

$\exists x$ dog(x) $\wedge$ brown(x)

Translate FOL expression into English:

(a) $\forall x (H(x) \rightarrow \forall y \neg M(x,y)) \wedge U(x)$

every man is unhappy and not married to anyone.

(b) $\exists z (P(z,x) \wedge S(z,y) \wedge W(y))$

There exists a person x who is a parent of x and z and
siblings and y is a woman.

CODE

```
import re


# Helper functions to apply the transformations step by step.

# 1. Eliminate biconditionals and implications
def eliminate_biconditionals_implications(expr):
    # Eliminate biconditionals (⇔)
    expr = re.sub(r'([A-Za-z0-9()]+) ⇔ ([A-Za-z0-9()]+)', r'(\1 ⇒ \2) ∧ (\2 ⇒ \1)', expr)

    # Eliminate implications (⇒)
    expr = re.sub(r'([A-Za-z0-9()]+) ⇒ ([A-Za-z0-9()]+)', r'¬\1 ∨ \2', expr)

    return expr

# 2. Move negations inward
def move_negations_inward(expr):
    expr = re.sub(r'¬\((∀x [A-Za-z0-9()]+)\)', r'∃x ¬\1', expr)  # Move negation inside ∀
    expr = re.sub(r'¬\((∃x [A-Za-z0-9()]+)\)', r'∀x ¬\1', expr)  # Move negation inside ∃
    expr = re.sub(r'¬\(([^()]+) ∨ ([^()]+)\)', r'¬\1 ∧ ¬\2', expr)  # De Morgan's law: ¬(A ∨ B)
    expr = re.sub(r'¬\(([^()]+) ∧ ([^()]+)\)', r'¬\1 ∨ ¬\2', expr)  # De Morgan's law: ¬(A ∧ B)
    expr = re.sub(r'¬¬([A-Za-z0-9()]+)', r'\1', expr)  # Double negation elimination
    return expr

# 3. Skolemization: Replace existential quantifiers with Skolem constants/functions
def skolemize(expr):
    expr = re.sub(r'∃([A-Za-z0-9]+)', r'G1', expr)  # Replace ∃x with Skolem constant (G1)
    expr = re.sub(r'∃([A-Za-z0-9]+)\(([A-Za-z0-9, ()]+)\)', r'F1(\2)', expr)  # Existential
quantifier -> Skolem function
    return expr

# 4. Drop universal quantifiers
def drop_universal_quantifiers(expr):
    expr = re.sub(r'∀([A-Za-z0-9]+)', '', expr)  # Drop ∀ quantifiers
    return expr

# 5. Distribute AND over OR to get CNF
def distribute_and_over_or(expr):
    expr = re.sub(r'(\([A-Za-z0-9()]+ ∧ [A-Za-z0-9()]+\)) ∨ ([A-Za-z0-9()]+)', r'(\1 ∨ \2)', expr)
    expr = re.sub(r'([A-Za-z0-9()]+ ∧ [A-Za-z0-9()]+) ∨ ([A-Za-z0-9()]+)', r'(\1 ∨ \2)', expr)
    return expr

# Convert to CNF using the above steps
def convert_to_cnf(expr):
    # Step 1: Eliminate biconditionals and implications
    expr = eliminate_biconditionals_implications(expr)

    # Step 2: Move negations inward
```

```
    expr = move_negations_inward(expr)

    # Step 3: Skolemize the expression
    expr = skolemize(expr)

    # Step 4: Drop universal quantifiers
    expr = drop_universal_quantifiers(expr)

    # Step 5: Distribute AND over OR to get CNF
    expr = distribute_and_over_or(expr)

    return expr

# Example FOL expressions (from the problem statement)
fol_expressions = [
    "Mary is the mother of John: Mother(Mary, John)",
    "John and Mary are both students: Student(John) ∧ Student(Mary)",
    "If it is raining, then the ground is wet: Raining ⇒ Wet(Ground)",
    "There is a person who knows every other person: ∃x ∀y (x ≠ y ⇒ Knows(x, y))",
    "Nobody is taller than themselves: ∀x ¬Taller(x, x)",
    "All students in the class passed the exam: ∀x (Student(x) ⇒ Passed(x, Exam))",
    "Mary has a pet dog: ∃x (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))",
    "If Alice is a teacher, then Alice teaches mathematics: Teacher(Alice) ⇒ Teaches(Alice,
Mathematics)",
    "Everyone loves someone: ∀x ∃y Loves(x, y)",
    "No one is both a teacher and a student: ∀x ¬(Teacher(x) ∧ Student(x))",
    "Every man respects his parent: ∀x (Man(x) ⇒ Respects(x, Parent(x)))",
    "Not all students like both Mathematics and Science: ¬∀x (Student(x) ⇒ (Likes(x,
Mathematics) ∧ Likes(x, Science)))"
]

# Convert and print CNF for each expression
for expr in fol_expressions:
    print(f"Original FOL Expression: {expr}")
    expression = expr.split(":")[1].strip()  # Remove the description part and keep the formula
    cnf = convert_to_cnf(expression)
    print(f"CNF: {cnf}\n")
```

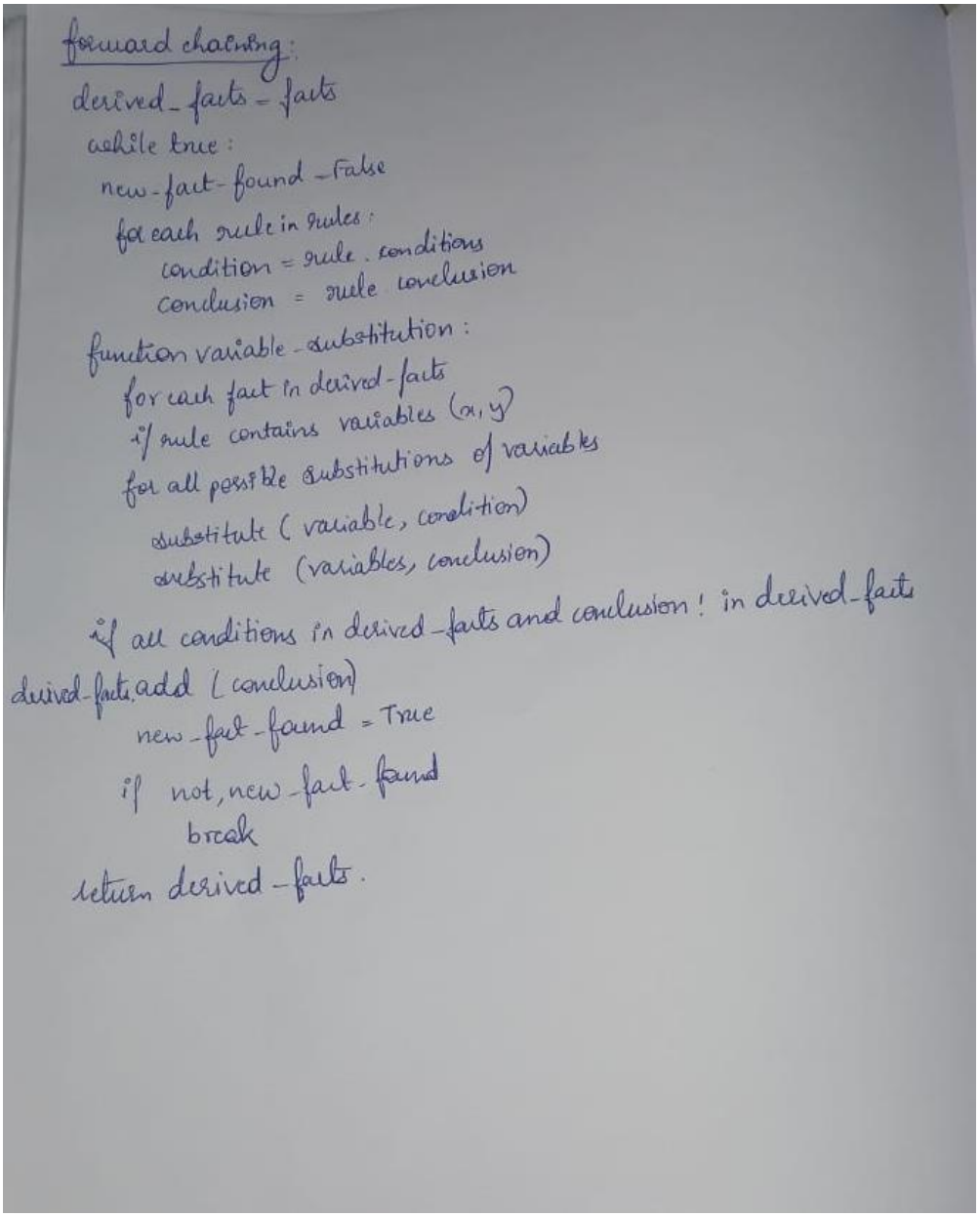
OUTPUT

```
Original FOL Expression: Mary is the mother of John: Mother(Mary, John)
CNF: Mother(Mary, John)

Original FOL Expression: John and Mary are both students: Student(John) ∧ Student(Mary)
CNF: Student(John) ∧ Student(Mary)

Original FOL Expression: If it is raining, then the ground is wet: Raining ⇒ Wet(Ground)
CNF: ¬Raining ∨ Wet(Ground)

Original FOL Expression: There is a person who knows every other person: ∃x ∀y (x ≠ y ⇒ Knows(x, y))
CNF: G1  (x ≠ ¬y ∨ Knows(x, y))

Original FOL Expression: Nobody is taller than themselves: ∀x ¬Taller(x, x)
CNF:  ¬Taller(x, x)

Original FOL Expression: All students in the class passed the exam: ∀x (Student(x) ⇒ Passed(x, Exam))
CNF:  ¬(Student(x) ∨ Passed(x, Exam))

Original FOL Expression: Mary has a pet dog: ∃x (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))
CNF: G1 (Pet(x) ∧ Dog(x) ∧ Has(Mary, x))

Original FOL Expression: If Alice is a teacher, then Alice teaches mathematics: Teacher(Alice) ⇒ Teaches(Alice, Mathematics)
CNF: ¬Teacher(Alice) ∨ Teaches(Alice, Mathematics)

Original FOL Expression: Everyone loves someone: ∀x ∃y Loves(x, y)
CNF:  G1 Loves(x, y)

Original FOL Expression: No one is both a teacher and a student: ∀x ¬(Teacher(x) ∧ Student(x))
CNF:  ¬(Teacher(x) ∧ Student(x))

Original FOL Expression: Every man respects his parent: ∀x (Man(x) ⇒ Respects(x, Parent(x)))
CNF:  ¬(Man(x) ∨ Respects(x, Parent(x)))

Original FOL Expression: Not all students like both Mathematics and Science: ¬∀x (Student(x) ⇒ (Likes(x, Mathematics) ∧ Likes(x, Science)))
CNF: ¬ ¬(Student(x) ∨ (Likes(x, Mathematics) ∧ Likes(x, Science)))
```

LABORATORY PROGRAM – 11

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

PSEUDOCODE OR ALGORITHM



```
forward chaining:
derived_facts = facts
while true:
  new_fact_found = False
  for each rule in rules:
    condition = rule.conditions
    conclusion = rule.conclusion
  function variable_substitution:
    for each fact in derived_facts
    if rule contains variables (x, y)
    for all possible substitutions of variables
      substitute (variable, condition)
      substitute (variables, conclusion)
  if all conditions in derived_facts and conclusion ! in derived_facts
derived_facts.add (conclusion)
    new_fact_found = True
  if not, new_fact_found
      break
return derived_facts.
```

IMPLEMENTATION OF FORWARD CHAINING

Fact:

American (Robert)
Enemy (A, America)
Owns (A, T1)
Missile (T1)
Sells Weapon (Robert, T1)

Rules:

Missile (x) → Weapon (x)
Enemy (x, America) → Hostile (x)
American (p) and sellsWeapon (p, q) and Enemy (q, America) → Criminal (p)

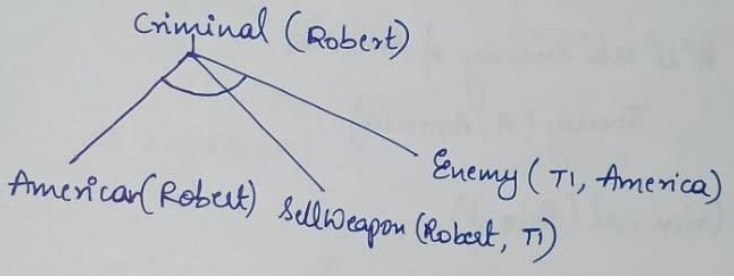
Goal:

Criminal (Robert)

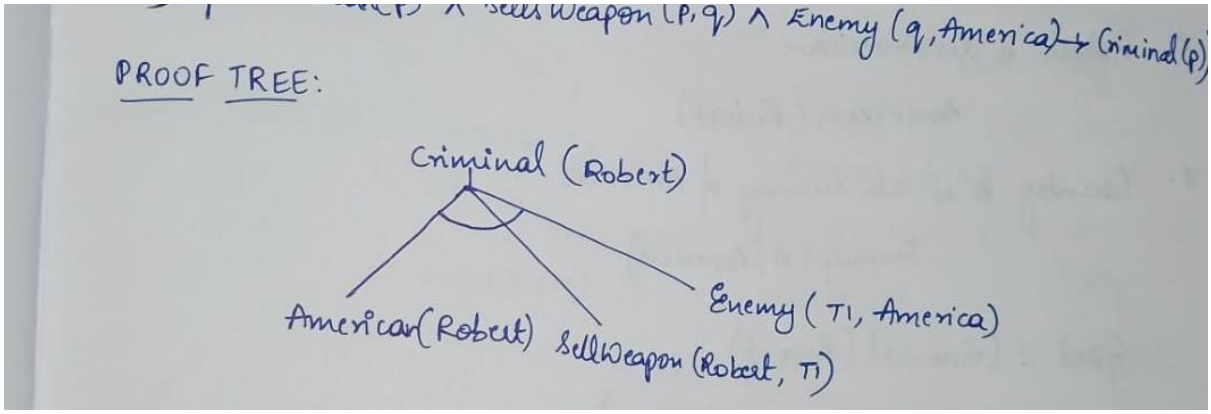
FOL:

$\forall p \; \forall q$ (American(p) ∧ Sells Weapon (p, q) ∧ Enemy (q, America) → Criminal (p)

PROOF TREE:



STATE SPACE TREE

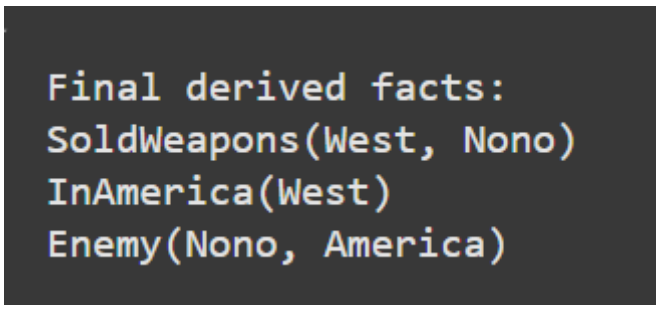


CODE

```
# Define initial facts and rules
facts = {"InAmerica(West)", "SoldWeapons(West, Nono)", "Enemy(Nono, America)"}
rules = [
    {
        "conditions": ["InAmerica(x)", "SoldWeapons(x, y)", "Enemy(y, America)"],
        "conclusion": "Criminal(x)",
    },
    {
        "conditions": ["Enemy(y, America)"],
        "conclusion": "Dangerous(y)",
    },
]

# Forward chaining function
def forward_chaining(facts, rules):
    derived_facts = set(facts)  # Initialize derived facts
    while True:
        new_fact_found = False

        for rule in rules:
            # Substitute variables and check if conditions are met
            for fact in derived_facts:
                if "x" in rule["conditions"][0]:
                    # Substitute variables (x, y) with specific instances
                    for condition in rule["conditions"]:
                        if "x" in condition or "y" in condition:
                            x = "West"  # Hardcoded substitution for simplicity
                            y = "Nono"
                            conditions = [
                                cond.replace("x", x).replace("y", y)
                                for cond in rule["conditions"]
                            ]
                            conclusion = (
                                rule["conclusion"].replace("x", x).replace("y", y)
                            )

                            # Check if all conditions are satisfied
                            if all(cond in derived_facts for cond in conditions) and conclusion not in
derived_facts:
                                derived_facts.add(conclusion)
                                print(f"New fact derived: {conclusion}")
                                new_fact_found = True

        # Exit loop if no new fact is found
        if not new_fact_found:
            break

    return derived_facts
```

```
# Run forward chaining
final_facts = forward_chaining(facts, rules)
print("\nFinal derived facts:")
for fact in final_facts:
    print(fact)
```

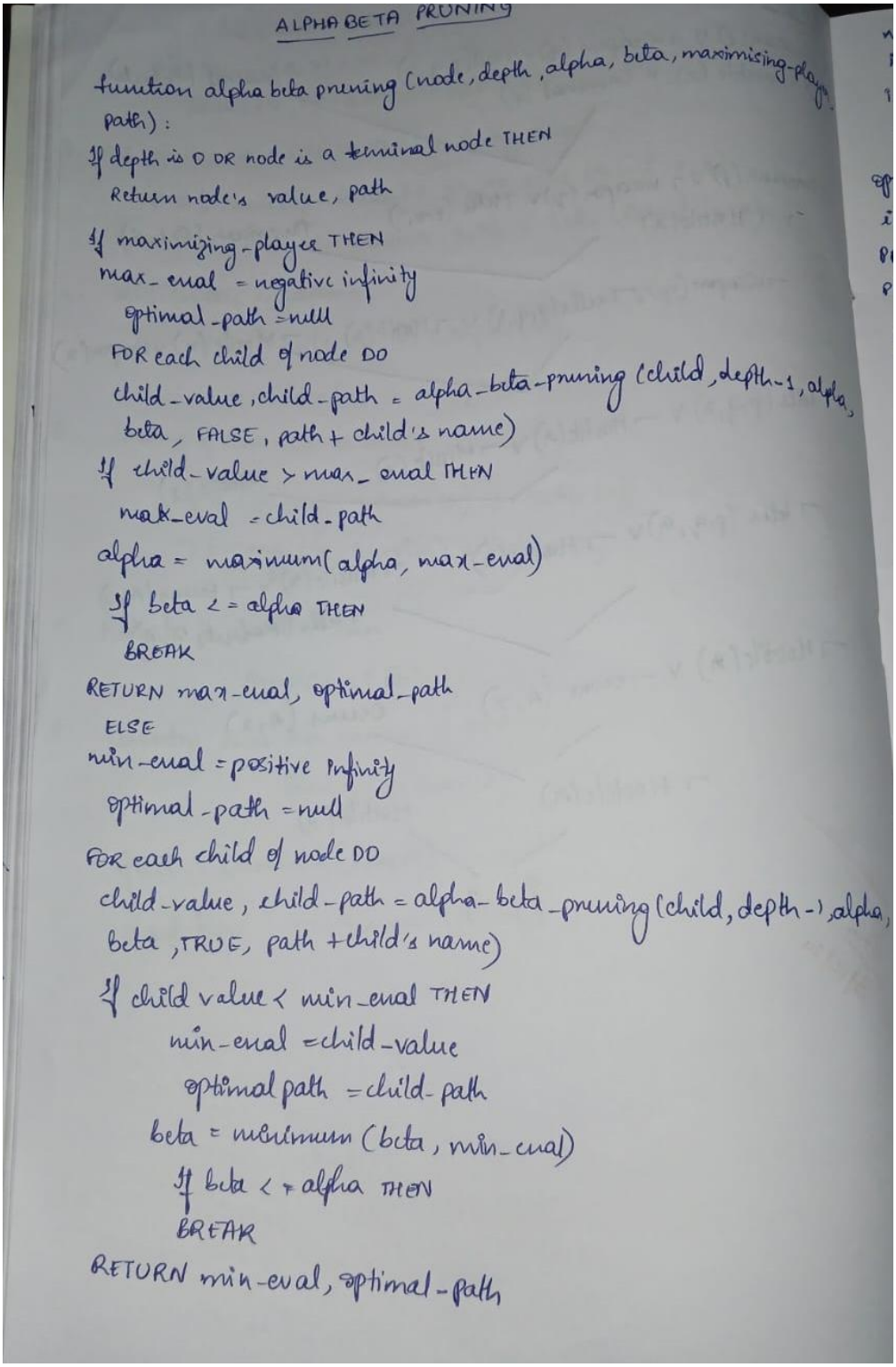
OUTPUT



LABORATORY PROGRAM – 12

Implement Alpha-Beta Pruning

PSEUDOCODE OR ALGORITHM



ALPHA BETA PRUNING

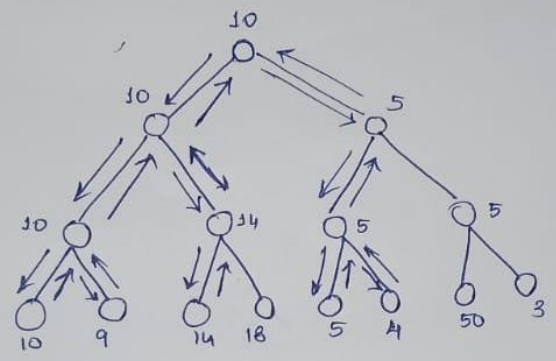
function alpha beta pruning (node, depth, alpha, beta, maximising-player, path):
If depth is 0 OR node is a terminal node THEN
Return node's value, path
If maximizing-player THEN
max_eval = negative infinity
optimal_path = null
FOR each child of node DO
child_value, child_path = alpha_beta_pruning (child, depth-1, alpha, beta, FALSE, path + child's name)
If child_value > max_eval THEN
max_eval = child_path
alpha = maximum(alpha, max_eval)
If beta <= alpha THEN
BREAK
RETURN max_eval, optimal_path
ELSE
min_eval = positive Infinity
optimal_path = null
FOR each child of node DO
child_value, child_path = alpha_beta_pruning (child, depth-1, alpha, beta, TRUE, path + child's name)
If child value < min_eval THEN
min_eval = child_value
optimal path = child_path
beta = minimum (beta, min_eval)
If beta <= alpha THEN
BREAK
RETURN min_eval, optimal_path

STATE SPACE TREE

maximizing_player=TRUE
initial_alpha = negative infinity
initial_beta = positive infinity
depth=3

optimal_value, optimal_path = alpha_beta_pruning(root_node, depth, initial_alpha, initial_beta, maximizing_player)
PRINT "The optimal value is:", optimal_value
PRINT "The optimal path is:", optimal_path

PROOF TREE:



The optimal value is: 10
The optimal path is: A → B → D → H.

17/12/24

CODE

```
class Node:
    def __init__(self, name, value=None, children=None):
        self.name = name
        self.value = value  # The value of the node (used for terminal nodes)
        self.children = children or []  # List of child nodes

def alpha_beta_search(state):
    def max_value(node, alpha, beta, path):
        print(f"MAX: Visiting node {node.name}, alpha={alpha}, beta={beta}")
        if terminal_test(node):
            print(f"MAX: Terminal node {node.name} has utility {utility(node)}")
            return utility(node), path
        v = float('-inf')
        best_path = []
        for child in node.children:
            value, new_path = min_value(child, alpha, beta, path + [child.name])
            print(f"MAX: From node {node.name}, child {child.name} → value={value}")
            if value > v:
                v = value
                best_path = new_path
            if v >= beta:
                print(f"MAX: Pruning at node {node.name} with value={v} ≥ beta={beta}")
                return v, best_path
            alpha = max(alpha, v)
        print(f"MAX: Returning value={v} for node {node.name}")
        return v, best_path

    def min_value(node, alpha, beta, path):
        print(f"MIN: Visiting node {node.name}, alpha={alpha}, beta={beta}")
        if terminal_test(node):
            print(f"MIN: Terminal node {node.name} has utility {utility(node)}")
            return utility(node), path
        v = float('inf')
        best_path = []
        for child in node.children:
            value, new_path = max_value(child, alpha, beta, path + [child.name])
            print(f"MIN: From node {node.name}, child {child.name} → value={value}")
            if value < v:
                v = value
                best_path = new_path
            if v <= alpha:
                print(f"MIN: Pruning at node {node.name} with value={v} ≤ alpha={alpha}")
                return v, best_path
            beta = min(beta, v)
        print(f"MIN: Returning value={v} for node {node.name}")
        return v, best_path

    print("Starting Alpha-Beta Search...\n")
```

```
    final_value, final_path = max_value(state, float('-inf'), float('inf'), [state.name])
    return final_value, final_path

# Helper Functions

# Terminal test function (checks if a node is terminal)
def terminal_test(node):
    return node.value is not None  # If the node has a value, it's a terminal node

# Utility function (returns the utility of a terminal node)
def utility(node):
    return node.value  # Return the value of the terminal node

# Example usage:

# Create the terminal nodes (leaf nodes)
H = Node('H', value=10)
I = Node('I', value=9)
J = Node('J', value=14)
K = Node('K', value=18)
L = Node('L', value=5)
M = Node('M', value=4)
N = Node('N', value=50)
O = Node('O', value=3)

# Create the non-terminal nodes with children
D = Node('D', children=[H, I])
E = Node('E', children=[J, K])
F = Node('F', children=[L, M])
G = Node('G', children=[N, O])

# Create the parent nodes
B = Node('B', children=[D, E])
C = Node('C', children=[F, G])

# Create the root node
A = Node('A', children=[B, C])

# Perform Alpha-Beta Search starting from the root node 'A'
final_value, final_path = alpha_beta_search(A)
print(f"Best path: {final_path}, with final value: {final_value}")
```

OUTPUT

```
Starting Alpha-Beta Search...

MAX: Visiting node A, alpha=-inf, beta=inf
MIN: Visiting node B, alpha=-inf, beta=inf
MAX: Visiting node D, alpha=-inf, beta=inf
MIN: Visiting node H, alpha=-inf, beta=inf
MIN: Terminal node H has utility 10
MAX: From node D, child H → value=10
MIN: Visiting node I, alpha=10, beta=inf
MIN: Terminal node I has utility 9
MAX: From node D, child I → value=9
MAX: Returning value=10 for node D
MIN: From node B, child D → value=10
MAX: Visiting node E, alpha=-inf, beta=10
MIN: Visiting node J, alpha=-inf, beta=10
MIN: Terminal node J has utility 14
MAX: From node E, child J → value=14
MAX: Pruning at node E with value=14 ≥ beta=10
MIN: From node B, child E → value=14
MIN: Returning value=10 for node B
MAX: From node A, child B → value=10
MIN: Visiting node C, alpha=10, beta=inf
MAX: Visiting node F, alpha=10, beta=inf
MIN: Visiting node L, alpha=10, beta=inf
MIN: Terminal node L has utility 5
MAX: From node F, child L → value=5
MIN: Visiting node M, alpha=10, beta=inf
MIN: Terminal node M has utility 4
MAX: From node F, child M → value=4
MAX: Returning value=5 for node F
MIN: From node C, child F → value=5
MIN: Pruning at node C with value=5 ≤ alpha=10
MAX: From node A, child C → value=5
MAX: Returning value=10 for node A
Best path: ['A', 'B', 'D', 'H'], with final value: 10
```