

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  
“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT**  
**on**  
**Artificial Intelligence (23CS5PCAIN)**

*Submitted by*

**Sanjana Suresh (1BM22CS239)**

*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 **Sanjana Suresh(1BM22CS239)**, who is a 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.

Dr. Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE
--	---

# Index

<b>Sl. No.</b>	<b>Date</b>	<b>Experiment Title</b>	<b>Page No.</b>
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	
3	14-10-2024	Implement A* search algorithm	
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	
7	2-12-2024	Implement unification in first order logic	
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	
10	16-12-2024	Implement Alpha-Beta Pruning.	

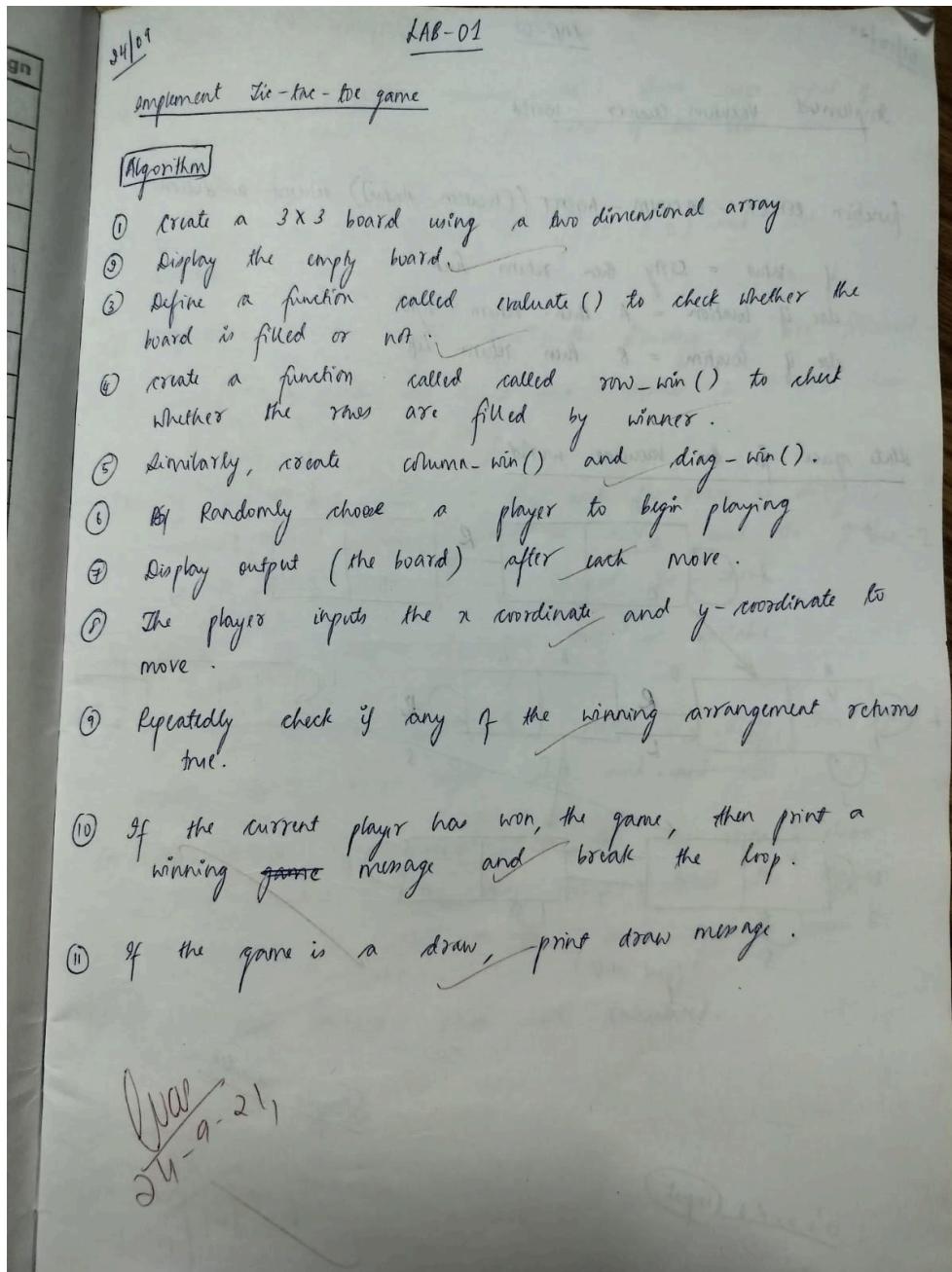
Github Link:

[https://github.com/SanjanaSuresh30/AI\\_LAB](https://github.com/SanjanaSuresh30/AI_LAB)

## Program 1

Implement Tic - Tac - Toe Game

**Algorithm:**



## Code:

```
import numpy as np

def create_board():
    return np.array([[" ", " ", " "],
                    [" ", " ", " "],
                    [" ", " ", " "]))

def coordinates(board, player):
    i, j, cn = (-1, -1, 0)

    player_symbol = 'X' if player == 1 else 'O'

    while (i < 0 or i > 2 or j < 0 or j > 2) or (board[i][j] != " "):
        if cn > 0:
            print("Wrong Input! Try Again")

        print("Player {}'s turn ({})".format(player, player_symbol))

        i = int(input("x-coordinates (1-3): ")) - 1

        j = int(input("y-coordinates (1-3): ")) - 1

        cn += 1

    board[i][j] = player_symbol

    return board

def row_win(board, player):
    player_symbol = 'X' if player == 1 else 'O'

    for x in range(len(board)):
        if all(board[x, y] == player_symbol for y in range(len(board))):
            return True

    return False
```

```

def col_win(board, player):

    player_symbol = 'X' if player == 1 else 'O'

    for x in range(len(board)):

        if all(board[y][x] == player_symbol for y in range(len(board))):

            return True

    return False


def diag_win(board, player):

    player_symbol = 'X' if player == 1 else 'O'

    if all(board[x][x] == player_symbol for x in range(len(board))):

        return True

    if all(board[x][len(board)-1-x] == player_symbol for x in
range(len(board))):

        return True

    return False


def evaluate(board):

    winner = 0

    for player in [1, 2]:

        if (row_win(board, player) or

            col_win(board, player) or

            diag_win(board, player)):

            winner = player

```

```

if np.all(board != " ") and winner == 0:

    winner = -1

return winner


def play_game():

    board, winner, counter = create_board(), 0, 1

    print(board)

    while winner == 0:

        for player in [1, 2]:

            board = coordinates(board, player)

            print("Board after " + str(counter) + " move:")

            print(board)

            counter += 1

            winner = evaluate(board)

            if winner != 0:

                break

    return winner


winner = play_game()

if winner == -1:

    print("It's a draw!")

else:

    print("Winner is Player {} ({})".format(winner, 'X' if winner == 1 else
'0'))

```

## Output:

```
*** [[ ' ' ' ' ' ]]  
[ ' ' ' ' ' ]  
[ ' ' ' ' ' ]]  
Player 1's turn (X)  
x-coordinates (1-3): 4  
y-coordinates (1-3): 1  
Wrong Input! Try Again  
Player 1's turn (X)  
x-coordinates (1-3): 
```

---

```
→ [[ ' ' ' ' ' ]]  
[ ' ' ' ' ' ]  
[ ' ' ' ' ' ]]  
Player 1's turn (X)  
x-coordinates (1-3): 1  
y-coordinates (1-3): 1  
Board after 1 move:  
[['X' ' ' ' ' ]]  
[ ' ' ' ' ' ]  
[ ' ' ' ' ' ]]  
Player 2's turn (O)  
x-coordinates (1-3): 1  
y-coordinates (1-3): 2  
Board after 2 move:  
[['X' 'O' ' ' ' ]]  
[ ' ' ' ' ' ]  
[ ' ' ' ' ' ]]  
Player 1's turn (X)  
x-coordinates (1-3): 2  
y-coordinates (1-3): 2  
Board after 3 move:  
[['X' 'O' ' ' ' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' ' ' ' ]]  
Player 2's turn (O)  
x-coordinates (1-3): 1  
y-coordinates (1-3): 3  
Board after 4 move:  
[['X' 'O' 'O' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' ' ' ' ]]  
Player 1's turn (X)  
x-coordinates (1-3): 3  
y-coordinates (1-3): 3  
Board after 5 move:  
[['X' 'O' 'O' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' ' ' X' ]]  
Winner is Player 1 (X)
```

```
▶ [[ ' ' ' ' ]]  
▶ [ ' ' ' ' ]  
▶ [ ' ' ' ' ]]
```

Player 1's turn (X)

x-coordinates (1-3): 1

y-coordinates (1-3): 2

Board after 1 move:

```
[[ ' ' 'X' ' ' ]]  
[ ' ' ' ' ' ]  
[ ' ' ' ' ' ]]
```

Player 2's turn (O)

x-coordinates (1-3): 1

y-coordinates (1-3): 1

Board after 2 move:

```
[[ 'O' 'X' ' ' ]]  
[ ' ' ' ' ' ]  
[ ' ' ' ' ' ]]
```

Player 1's turn (X)

x-coordinates (1-3): 2

y-coordinates (1-3): 2

Board after 3 move:

```
[[ 'O' 'X' ' ' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' ' ' ' ]]
```

Player 2's turn (O)

x-coordinates (1-3): 3

y-coordinates (1-3): 2

Board after 4 move:

```
[[ 'O' 'X' ' ' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' 'O' ' ' ]]
```

Player 1's turn (X)

x-coordinates (1-3): 3

y-coordinates (1-3): 3

Board after 5 move:

```
[[ 'O' 'X' ' ' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' 'O' 'X' ]]
```

Player 2's turn (O)

x-coordinates (1-3): 1

y-coordinates (1-3): 3

Board after 6 move:

```
[[ 'O' 'X' 'O' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' 'O' 'X' ]]
```

Player 1's turn (X)

x-coordinates (1-3): 2

y-coordinates (1-3): 1

Board after 7 move:

```
[[ 'O' 'X' 'O' ]]  
[ 'X' 'X' ' ' ]  
[ ' ' 'O' 'X' ]]
```

Player 2's turn (O)

Board after 6 move:

```
[[ 'O' 'X' 'O' ]]  
[ ' ' 'X' ' ' ]  
[ ' ' 'O' 'X' ]]
```

Player 1's turn (X)

x-coordinates (1-3): 2

y-coordinates (1-3): 1

Board after 7 move:

```
[[ 'O' 'X' 'O' ]]  
[ 'X' 'X' ' ' ]  
[ ' ' 'O' 'X' ]]
```

Player 2's turn (O)

x-coordinates (1-3): 2

y-coordinates (1-3): 3

Board after 8 move:

```
[[ 'O' 'X' 'O' ]]  
[ 'X' 'X' 'O' ]  
[ ' ' 'O' 'X' ]]
```

Player 1's turn (X)

x-coordinates (1-3): 3

y-coordinates (1-3): 1

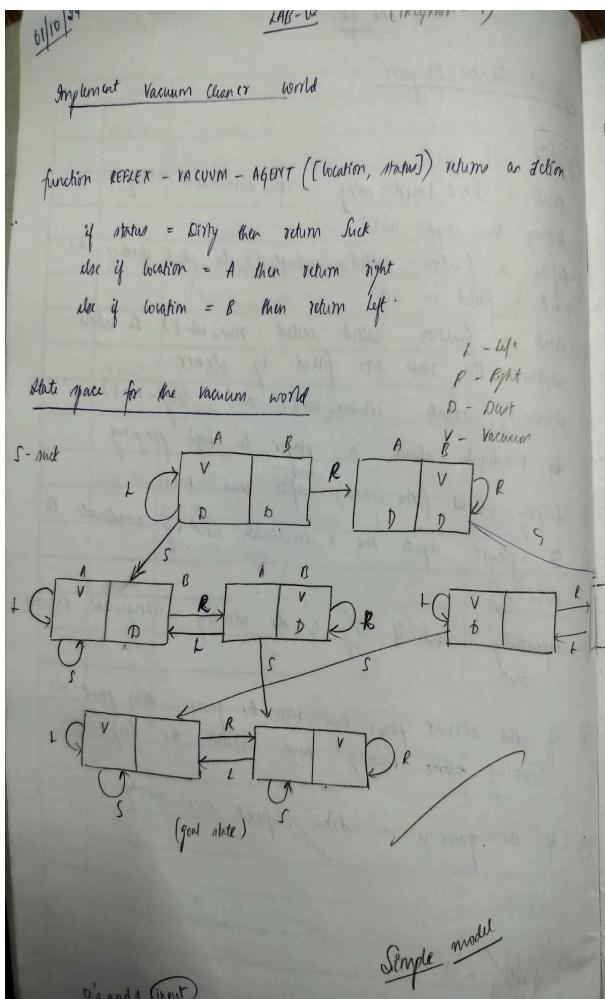
Board after 9 move:

```
[[ 'O' 'X' 'O' ]]  
[ 'X' 'X' 'O' ]  
[ 'X' 'O' 'X' ]]
```

It's a draw!

## Implement vacuum cleaner agent

### Algorithm:



\* important

- Define a function called `vacuum_world()` where user input of the location of the vacuum, the status of the room and status of other room.
- The goal state is defined as `[A, 0, B, 0]` and cost is initialized to 0.
- Depending on the user's input, the following steps are followed-
  - Location is specified as A/B and status is 0 when clean, 1 when room is dirty.
  - If the input location is A and the room is dirty, then the vacuum has to suck and the goal state for room A is changed to zero.
  - Increment the cost variable
  - If the other room's status is dirty, then the vacuum has to move right and suck. ~~and goal~~
- If the input location is A and the room is clean, then depending on the other room's status the vacuum will remain in room A or move right to room B.
- Goal state attained. and cost calculated.

Goal

### Code:

```

#Enter LOCATION A/B in captial letters

#Enter Status 0/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum_world():
  
```

```

# initializing goal_state

# 0 indicates Clean and 1 indicates Dirty

goal_state = {'A': '0', 'B': '0'}

cost = 0


location_input = input("Enter Location of Vacuum") #user_input of
location vacuum is placed

status_input = input("Enter status of " + location_input) #user_input if
location is dirty or clean

status_input_complement = input("Enter status of other room")

print("Initial Location Condition" + str(goal_state))

if location_input == 'A':

    # Location A is Dirty.

    print("Vacuum is placed in Location A")

    if status_input == '1':

        print("Location A is Dirty.")

        # suck the dirt and mark it as clean

        goal_state['A'] = '0'

        cost += 1                      #cost for suck

        print("Cost for CLEANING A " + str(cost))

        print("Location A has been Cleaned.")

    if status_input_complement == '1':

```

```

# if B is Dirty

print("Location B is Dirty.")

print("Moving right to the Location B. ")

cost += 1                      #cost for moving right

print("COST for moving RIGHT" + str(cost))

# suck the dirt and mark it as clean

goal_state['B'] = '0'

cost += 1                      #cost for suck

print("COST for SUCK " + str(cost))

print("Location B has been Cleaned. ")

else:

    print("No action" + str(cost))

    # suck and mark clean

    print("Location B is already clean.")


if status_input == '0':


    print("Location A is already clean ")

    if status_input_complement == '1':# if B is Dirty

        print("Location B is Dirty.")

        print("Moving RIGHT to the Location B. ")

        cost += 1                  #cost for moving right

        print("COST for moving RIGHT " + str(cost))

        # suck the dirt and mark it as clean

        goal_state['B'] = '0'

```

```

cost += 1                      #cost for suck

print("Cost for SUCK" + str(cost))

print("Location B has been Cleaned. ")

else:

    print("No action " + str(cost))

    print(cost)

    # suck and mark clean

    print("Location B is already clean.")


else:

print("Vacuum is placed in location B")

# Location B is Dirty.

if status_input == '1':

    print("Location B is Dirty.")

    # suck the dirt and mark it as clean

    goal_state['B'] = '0'

    cost += 1 # cost for suck

    print("COST for CLEANING " + str(cost))

    print("Location B has been Cleaned.")


if status_input_complement == '1':

    # if A is Dirty

    print("Location A is Dirty.")

    print("Moving LEFT to the Location A. ")

```

```

        cost += 1 # cost for moving right

        print("COST for moving LEFT" + str(cost))

        # suck the dirt and mark it as clean

        goal_state['A'] = '0'

        cost += 1 # cost for suck

        print("COST for SUCK " + str(cost))

        print("Location A has been Cleaned.")


else:

    print(cost)

    # suck and mark clean

    print("Location B is already clean.")


if status_input_complement == '1': # if A is Dirty

    print("Location A is Dirty.")

    print("Moving LEFT to the Location A. ")

    cost += 1 # cost for moving right

    print("COST for moving LEFT " + str(cost))

    # suck the dirt and mark it as clean

    goal_state['A'] = '0'

    cost += 1 # cost for suck

    print("Cost for SUCK " + str(cost))

    print("Location A has been Cleaned. ")

else:

```

```

        print("No action " + str(cost))

        # suck and mark clean

        print("Location A is already clean.")




# done cleaning

print("GOAL STATE: ")

print(goal_state)

print("Performance Measurement: " + str(cost))

vacuum_world()

```

## Output:

```

Enter Location of VacuumA
Enter status of A0
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean
No action 0
0
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 0

```

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

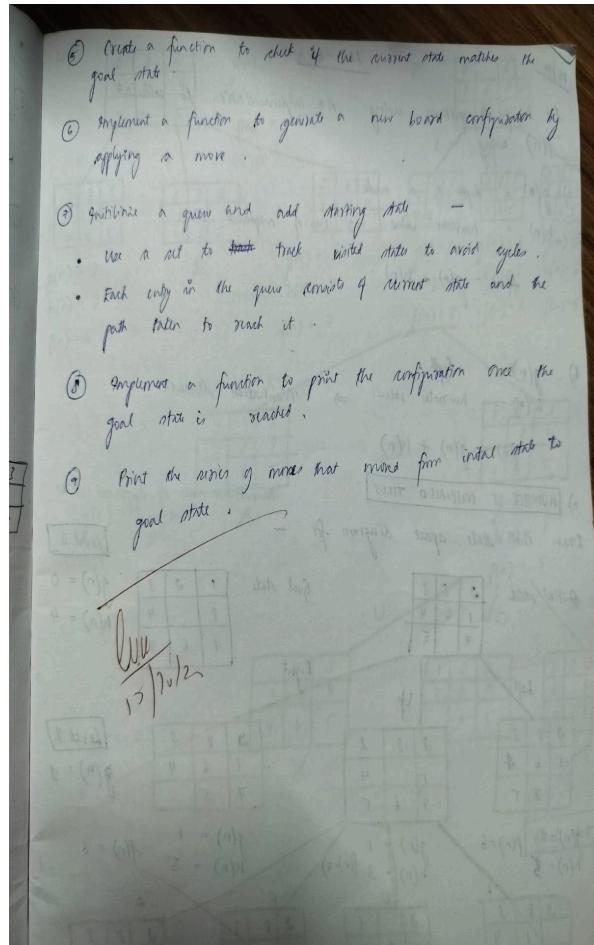
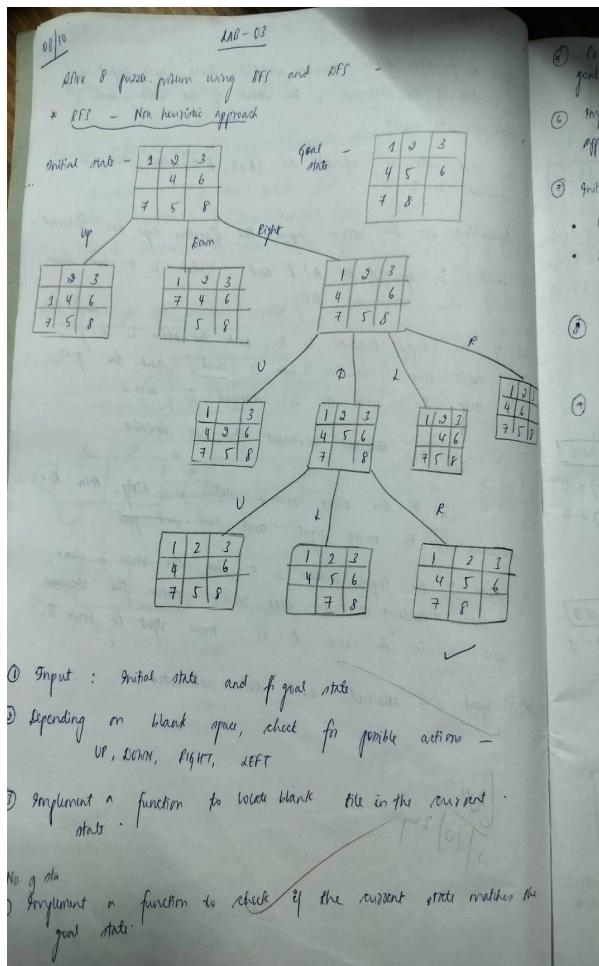
```
Enter Location of VacuumA
Enter status of A1
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
No action1
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
```

```
Enter Location of VacuumA
Enter status of A0
Enter status of other room1
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

## Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

### Algorithm:



### Code:

```
import numpy as np

def bfs(src, target):
    queue = [(src, None)] # State and last move
    visited = set()
```

```

state_count = 0 # Initialize state count

while queue:

    state, last_move = queue.pop(0)

    state_tuple = tuple(state) # Convert state to tuple for set
operations

    if state_tuple not in visited:

        visited.add(state_tuple)

        state_count += 1 # Increment the state count

        print_board(state)

        if last_move:

            print(f"Current move: {last_move}\n")

    if state == target:

        print("Goal state achieved!")

        break

for move, direction in possible_moves(state):

    if tuple(move) not in visited:

        queue.append((move, direction))

print(f"Total unique states explored: {state_count}")

```

```

def possible_moves(state):
    b = state.index(0)

    directions = []

    if b not in [0, 1, 2]: directions.append('u')

    if b not in [6, 7, 8]: directions.append('d')

    if b not in [0, 3, 6]: directions.append('l')

    if b not in [2, 5, 8]: directions.append('r')

    return [(gen(state, d, b), d) for d in directions]

def gen(state, direction, b):
    temp = state.copy()

    if direction == 'u': temp[b], temp[b - 3] = temp[b - 3], temp[b]
    if direction == 'd': temp[b], temp[b + 3] = temp[b + 3], temp[b]
    if direction == 'l': temp[b], temp[b - 1] = temp[b - 1], temp[b]
    if direction == 'r': temp[b], temp[b + 1] = temp[b + 1], temp[b]

    return temp

def print_board(state):
    board = np.array(state).reshape(3, 3)
    print(board)

```

```

# Initial configuration and target configuration

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

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

# Run BFS to solve the puzzle

bfs(src, target)

```

## Output:

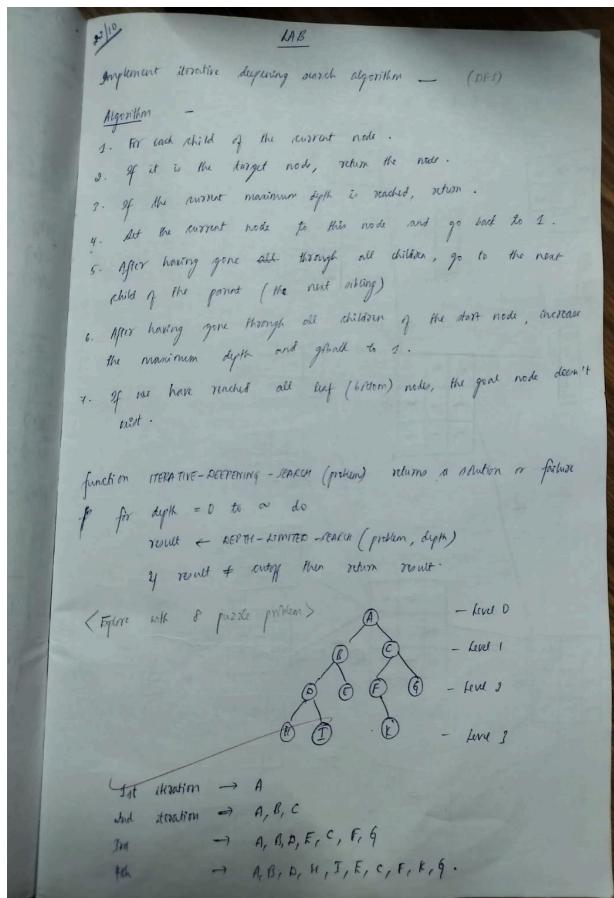
```

[[1 2 3]
 [0 4 6]
 [7 5 8]]
[[0 2 3]
 [1 4 6]
 [7 5 8]]
Current move: u
[[1 2 3]
 [7 4 6]
 [0 5 8]]
Current move: d
[[1 2 3]
 [4 0 6]
 [7 5 8]]
Current move: r
[[2 0 3]
 [1 4 6]
 [7 5 8]]
Current move: r
[[1 2 3]
 [7 4 6]
 [5 0 8]]
Current move: r
[[1 0 3]
 [4 2 6]
 [7 5 8]]
Current move: u
[[1 2 3]
 [4 5 6]
 [7 0 8]]
Current move: d
[[1 2 3]
 [4 6 0]
 [7 5 8]]
Current move: r
[[2 4 3]
 [1 0 6]
 [7 5 8]]
Current move: d
[[2 3 0]
 [1 4 6]
 [7 5 8]]
Current move: r
[[1 2 3]
 [7 0 6]
 [5 4 8]]
Current move: u
[[1 2 3]
 [7 4 6]
 [5 8 0]]
Current move: r
[[0 1 3]
 [4 2 6]
 [7 5 8]]
Current move: l
[[1 3 0]
 [4 2 6]
 [7 5 8]]
Current move: r
[[1 2 3]
 [4 5 6]
 [0 7 8]]
Current move: l
[[1 2 3]
 [4 5 6]
 [7 8 0]]
Current move: r
Goal state achieved!
Total unique states explored: 17

```

## Implement Iterative deepening search algorithm

### Algorithm:



### Code:

```
import copy

class Node:

    def __init__(self, state, parent=None, action=None, depth=0):

        self.state = state

        self.parent = parent

        self.action = action
```

```

    self.depth = depth

def __lt__(self, other):
    return self.depth < other.depth

def expand(self):
    children = []

    row, col = self.find_blank()

    possible_actions = []

    if row > 0: # Can move the blank tile up
        possible_actions.append('Up')

    if row < 2: # Can move the blank tile down
        possible_actions.append('Down')

    if col > 0: # Can move the blank tile left
        possible_actions.append('Left')

    if col < 2: # Can move the blank tile right
        possible_actions.append('Right')

    for action in possible_actions:
        new_state = copy.deepcopy(self.state)

        if action == 'Up':
            new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col], new_state[row][col]

        elif action == 'Down':

```

```

        new_state[row][col], new_state[row + 1][col] = new_state[row
+ 1][col], new_state[row][col]

    elif action == 'Left':

        new_state[row][col], new_state[row][col - 1] =
new_state[row][col - 1], new_state[row][col]

    elif action == 'Right':

        new_state[row][col], new_state[row][col + 1] =
new_state[row][col + 1], new_state[row][col]

    children.append(Node(new_state, self, action, self.depth + 1))

    return children


def find_blank(self):

    for row in range(3):

        for col in range(3):

            if self.state[row][col] == 0:

                return row, col

    raise ValueError("No blank tile found")


def depth_limited_search(node, goal_state, limit):

    if node.state == goal_state:

        return node

    if node.depth >= limit:

        return None

    for child in node.expand():

```

```

        result = depth_limited_search(child, goal_state, limit)

    if result is not None:

        return result

    return None


def iterative_deepening_search(initial_state, goal_state, max_depth):

    for depth in range(max_depth):

        result = depth_limited_search(Node(initial_state), goal_state, depth)

        if result is not None:

            return result

    return None


def print_solution(node):

    path = []

    while node is not None:

        path.append((node.action, node.state))

        node = node.parent

    path.reverse()

    for action, state in path:

        if action:

            print(f"Action: {action}")

            for row in state:

                print(row)

```

```

print()

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

max_depth = 5

solution = iterative_deepening_search(initial_state, goal_state, max_depth)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")

```

## Output:

```

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

Action: Right
[1, 2, 3]
[4, 0, 6]
[7, 5, 8] initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Action: Down
[1, 2, 3]
[4, 5, 6]
[7, 0, 8] max_depth = 3
solution = iterative_deepening_search(initial_state, goal_state, max_depth)

Action: Right
[1, 2, 3]
[4, 5, 6]
[7, 8, 0] if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")

Solution not found.

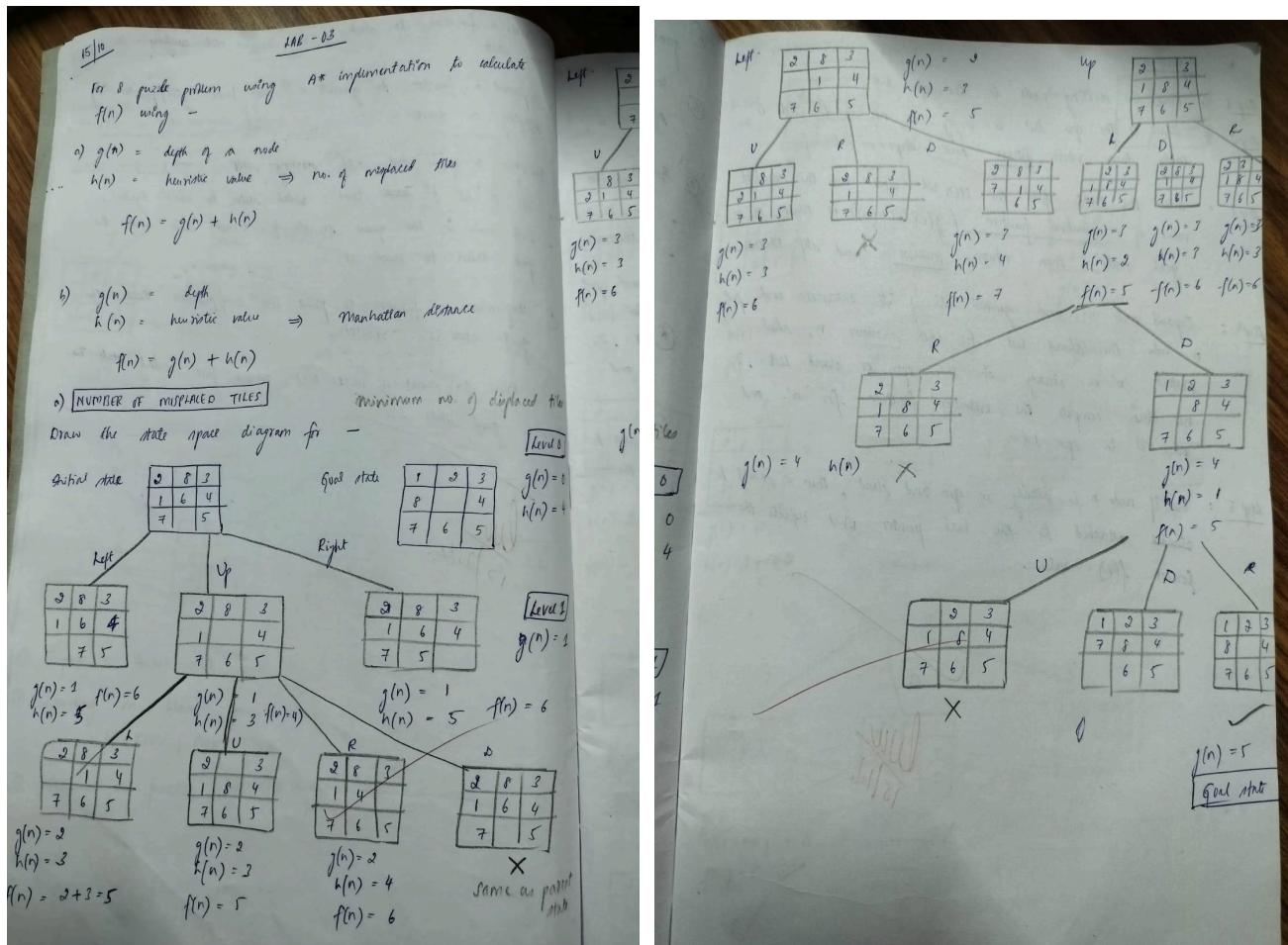
```

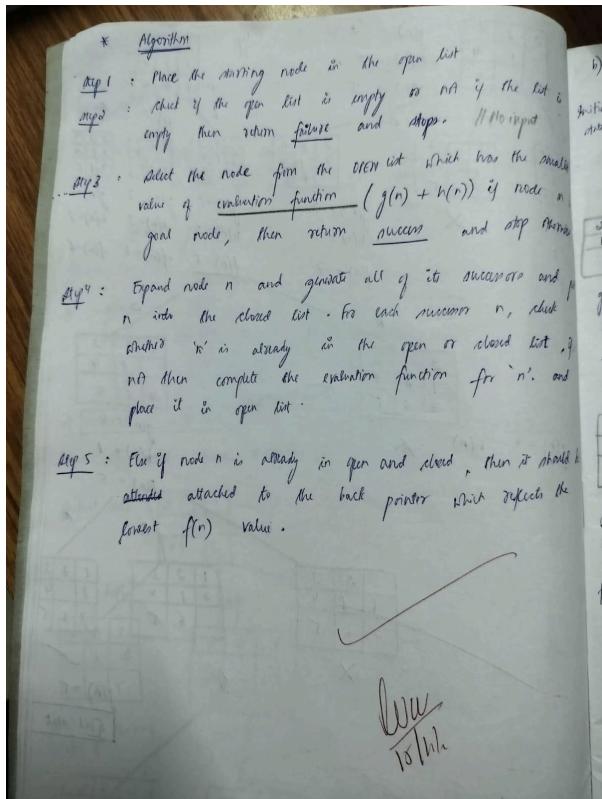
## Program 3

Implement A\* search algorithm using

- a) Number of misplaced tiles

**Algorithm:**





## Code:

```

import heapq

GOAL_STATE = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

def misplaced_tile(state):
    misplaced = 0

    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1

    return misplaced

```

```

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

def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny],
            new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))

    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

```

```

def a_star(start):

    open_list = []

    heapq.heappush(open_list, (0 + misplaced_tile(start), 0, start))

    g_score = {start: 0}

    came_from = {}

    visited = set()

    while open_list:

        _, g, current = heapq.heappop(open_list)

        if current == GOAL_STATE:

            path = reconstruct_path(came_from, current)

            return path, g

        visited.add(current)

        for neighbor in generate_neighbors(current):

            if neighbor in visited:

                continue

            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):

                came_from[neighbor] = current

                g_score[neighbor] = tentative_g

                f_score = tentative_g + misplaced_tile(neighbor) # f(n) =
g(n) + h(n)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

```

```

    return None, None

def print_state(state):

    for row in state:

        print(row)

    print()

if __name__ == "__main__":

    start_state = ((2, 8, 3),
                   (1, 6, 4),
                   (7, 0, 5))

    print("Initial State:")

    print_state(start_state)

    print("Goal State:")

    print_state(GOAL_STATE)

    solution, cost = a_star(start_state)

    if solution:

        print(f"Goal state achieved with cost: {cost}")

        print("Steps:")

        for step in solution:

            print_state(step)

    else:

        print("No solution found.")

```

Output:

```

Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

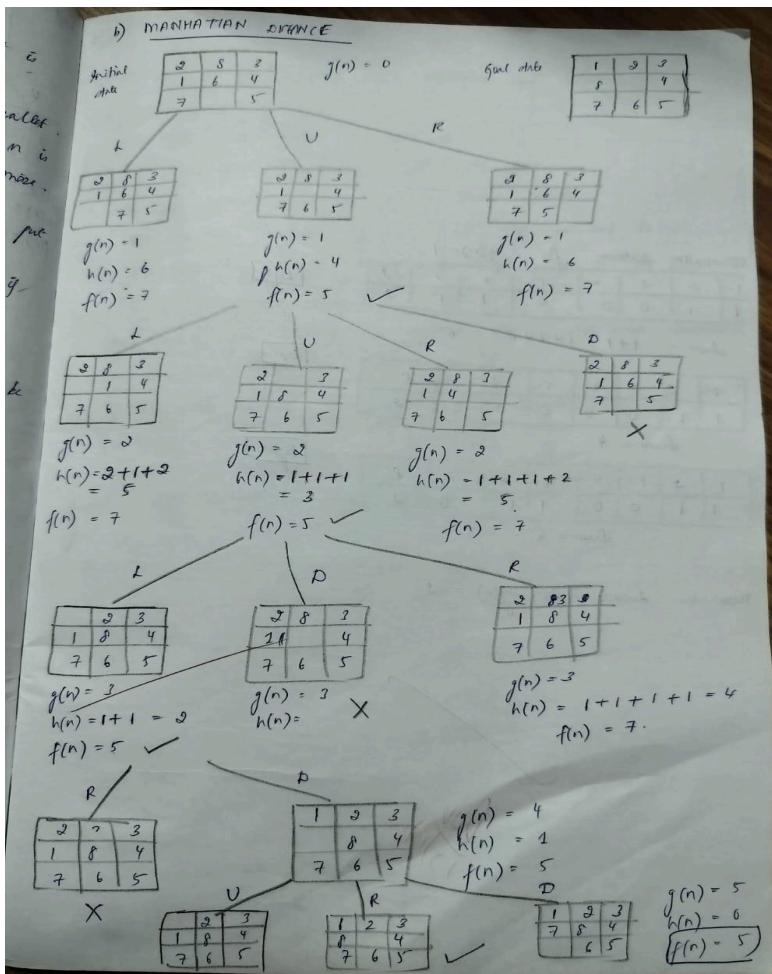
Goal State:
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
(2, 8, 3)
(1, 8, 4)
(7, 6, 5)
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

```

b) Manhattan Distance

**Algorithm:**



## Code:

```
import heapq

GOAL_STATE = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

def manhattan_distance(state):
    distance = 0

    for i in range(3):
        for j in range(3):
            value = state[i][j]

            if value != 0:
                goal_x, goal_y = divmod(value - 1, 3)
                distance += abs(goal_x - i) + abs(goal_y - j)

    return distance

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

def generate_neighbors(state):
    neighbors = []

    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
```

```

for dx, dy in directions:

    nx, ny = x + dx, y + dy

    if 0 <= nx < 3 and 0 <= ny < 3:

        new_state = [list(row) for row in state]

        new_state[x][y], new_state[nx][ny] = new_state[nx][ny],
new_state[x][y]

        neighbors.append(tuple(tuple(row) for row in new_state))

return neighbors

def reconstruct_path(came_from, current):

    path = [current]

    while current in came_from:

        current = came_from[current]

        path.append(current)

    path.reverse()

    return path

def a_star(start):

    open_list = []

    heapq.heappush(open_list, (manhattan_distance(start), 0, start))

    g_score = {start: 0}

    came_from = { }

```

```

visited = set()

while open_list:

    f, g, current = heapq.heappop(open_list)

    if current == GOAL_STATE:
        path = reconstruct_path(came_from, current)
        return path, g

    visited.add(current)

    for neighbor in generate_neighbors(current):
        if neighbor in visited:
            continue

        tentative_g = g_score[current] + 1

        if tentative_g < g_score.get(neighbor, float('inf')):
            came_from[neighbor] = current
            g_score[neighbor] = tentative_g
            f_score = tentative_g + manhattan_distance(neighbor)

            heapq.heappush(open_list, (f_score, tentative_g, neighbor))

return None, None

```

```

def print_state(state):

    for row in state:

        print(row)

    print()

if __name__ == "__main__":

    start_state = ((2, 8, 3),
                   (1, 6, 4),
                   (7, 0, 5))

    print("Initial State:")

    print_state(start_state)

    print("Goal State:")

    print_state(GOAL_STATE)

    solution, cost = a_star(start_state)

    if solution:

        print(f"Goal state achieved with cost: {cost}")

        print("Steps:")

        for step in solution:

            print_state(step)

    else: print("No solution found.")

```

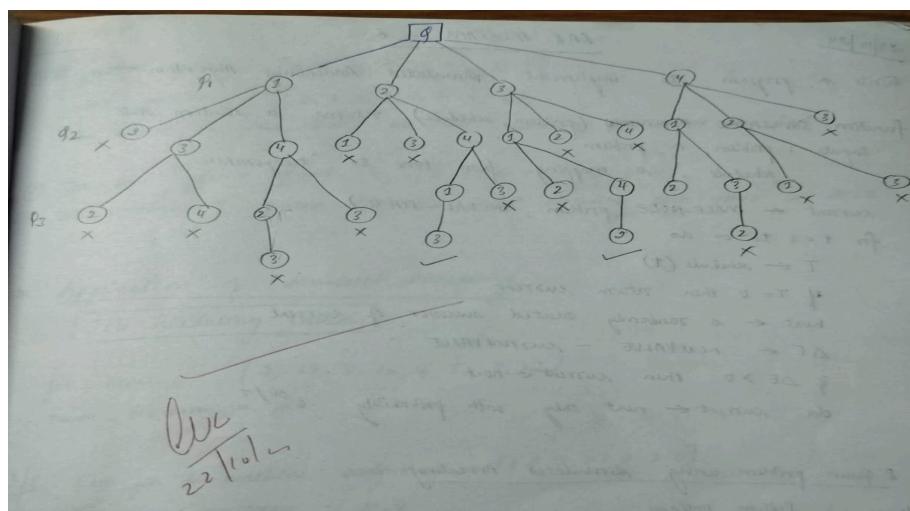
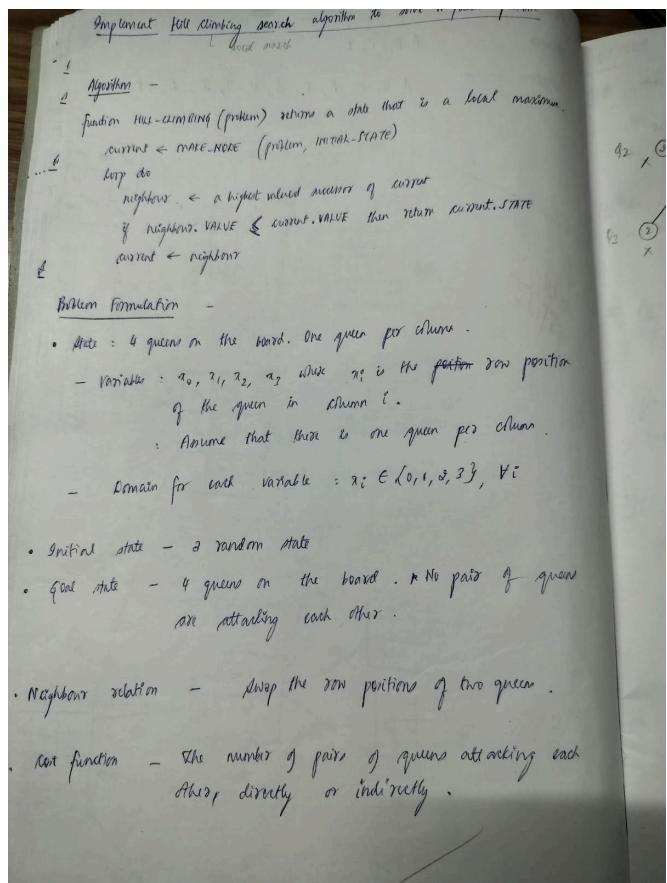
Output:

```
ps ➔ Initial State:  
      (2, 8, 3)  
      (1, 6, 4)  
      (7, 0, 5)  
  
      Goal State:  
      (1, 2, 3)  
      (8, 0, 4)  
      (7, 6, 5)  
  
      Goal state achieved with cost using Manhattan Distance: 5  
      Steps:  
      (2, 8, 3)  
      (1, 6, 4)  
      (7, 0, 5)  
  
      (2, 8, 3)  
      (1, 0, 4)  
      (7, 6, 5)  
  
      (2, 0, 3)  
      (1, 8, 4)  
      (7, 6, 5)  
  
      (0, 2, 3)  
      (1, 8, 4)  
      (7, 6, 5)  
  
      (1, 2, 3)  
      (0, 8, 4)  
      (7, 6, 5)  
  
      (1, 2, 3)  
      (8, 0, 4)  
      (7, 6, 5)
```

## Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

**Algorithm:**



## 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(n, max_iterations=1000):

    """Perform hill climbing search to solve the N-Queens problem."""

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

    current_cost = calculate_cost(current_state)

    for iteration in range(max_iterations):

        if current_cost == 0: # Found a solution
            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: # No improvement found
            print(f"Local maximum reached at iteration {iteration}.\n"
            "Restarting...")
            return None # Restart with a new random state

        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

# Get user-defined input for the number of queens

```

```

try:

    n = int(input("Enter the number of queens (N): "))

    if n <= 0:

        raise ValueError("N must be a positive integer.")

except ValueError as e:

    print(e)

n = 4 # Default to 4 if input is invalid

solution = None

# Keep trying until a solution is found

while solution is None:

    solution = hill_climbing(n)

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

```

## Output:

→ Enter the number of queens (N): 4  
 Iteration 0: Current state: [1, 3, 2, 0], Cost: 1  
 Local maximum reached at iteration 1. Restarting...  
 Iteration 0: Current state: [1, 1, 2, 0], Cost: 2  
 Iteration 1: Current state: [1, 3, 2, 0], Cost: 1  
 Local maximum reached at iteration 2. Restarting...  
 Iteration 0: Current state: [1, 3, 0, 3], Cost: 1  
 Iteration 1: Current state: [1, 3, 0, 2], Cost: 0  
 Solution found: [1, 3, 0, 2]

## Program 5

Simulated Annealing to Solve 8-Queens problem

**Algorithm:**

39/10/24

SAR PROGRAM - 6

1 Write a program to implement simulated Annealing Algorithm.

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

3     Input : problem, a problem

4         schedule, a mapping from time to "temperature"

5     current  $\leftarrow$  MAKE-NODE (problem, INITIAT-STATE)

6     for t=1 to  $\infty$  do

7         T  $\leftarrow$  schedule (t)

8         if T = 0 then return current

9         next  $\leftarrow$  a randomly selected neighbor of current

10          $\Delta E \leftarrow$  nextVALUE - currentVALUE

11         if  $\Delta E > 0$  then current  $\leftarrow$  next

12         else current  $\leftarrow$  next only with probability  $e^{\Delta E/T}$

\* 8 Queen problem using Simulated Annealing

minpac - Python package

minpac for simulated annealing

① Import the minpac and numpy libraries

②

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

OUTPUT

③ The initial position = [4, 6, 1, 5, 0, 3, 7]

best position = [0, 6, 4, 7, 1, 3, 5, 8]

The number of queen that are not attacking each other is 8=8

④ The best position found is [0 5 7 1 3 0 6 4]

The number of queen that are not attacking each other is: 8=8

\* Application of Simulated Annealing

[Job Scheduling Problem]

Job - time = [3, 14, 4, 16, 6, 5, 7, 12]

num - machine = 2

⑤ Best job-to-machine assignment = [1 8 0 12 0 1 0]

Minimum makespan : 21.0

Machine 1 jobs : [(3,4), (5,5), (7,12)] Total time : 21

Machine 2 jobs : [(0,3), (3,16), (6,7)] Total time : 21

Machine 3 jobs : [(6,14), (4,6)] Total time : 20

Job	Machine 1	Machine 2	Machine 3
1	3	0	6
2	4	16	14
3	5	3	7
4	7	12	6
5			1
6			2
7			3
8			4

## Code:

```
import mlrose_hiive as mlrose

import numpy as np


def queens_max(position):

    n = len(position)

    attacks = 0

    for i in range(n):

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

            # Check if queens attack each other

            if position[i] == position[j] or abs(position[i] - position[j]) == j - i:

                attacks += 1

    # The fitness is the total number of pairs of queens minus the number of attacks

    return (n * (n - 1) // 2) - attacks

# Define the custom fitness function

objective = mlrose.CustomFitness(queens_max)

# Set up the optimization problem

problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True,
max_val=8)
```

```
T = mlrose.ExpDecay()

# Define the initial position

initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])

# Run the simulated annealing algorithm

best_state, best_fitness, _ = mlrose.simulated_annealing(problem, schedule=T,
max_attempts=500, max_iters=5000, init_state=initial_position)

print('The best position found is:', best_state)

print('The number of queens that are not attacking each other is:',
best_fitness)
```

### Output:

```
→ The best position found is: [2 5 7 1 3 0 6 4]
The number of queens that are not attacking each other is: 28.0
```

## Program 6

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

### Algorithm:

WEEK - 07

17/11  
Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not

Algorithm :

function TT-ENTAILS? (KB,  $\alpha$ ) returns true or false

    Inputs : KB - the knowledge base, a sentence in propositional logic  
            $\alpha$  - the query, a sentence in propositional logic

    symbols  $\leftarrow$  a list of the proposition symbols in KB and  $\alpha$ .  
     return TT-CHECK-ALL (KB,  $\alpha$ , symbols, {})

function TT-CHECK-ALL (KB,  $\alpha$ , symbols, model) returns true or false

    if EMPTY? (symbols) then  
         if KB-TRUE? (KB, model) then return KB-TRUE? ( $\alpha$ , model)  
         else return true // when KB is false, always return true.  
     else do  
         p  $\leftarrow$  FIRST (symbols)  
         rest  $\leftarrow$  REST (symbols)  
         return (TT-CHECK-ALL (KB,  $\alpha$ , rest, model  $\cup \{p = \text{true}\}$ )  
               and  
               TT-CHECK-ALL (KB,  $\alpha$ , rest, model  $\cup \{p = \text{false}\}$ ))

Truth table

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Rightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	false	true
true	false	false	false	true	false	false
true	true	false	true	true	true	true

$\wedge$  - and  $\Rightarrow$  implies if P then Q  
 $\vee$  - or

The given

$\alpha = A \vee B$   
 $KB = (A \vee C) \wedge (B \vee C)$

Checking  $KB \models \alpha$

A	B	C	$A \vee C$	$B \vee C$	KB	$\alpha$
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

( $\alpha$ , model) is true

See SCD 11/11/2023

$P \Rightarrow Q$

True  
 True  
 True

44

**Code:**

```
from itertools import product

def pl_true(sentence, model):
    """Evaluates if a sentence is true in a given model."""
    if isinstance(sentence, str):
        return model.get(sentence, False)
    elif isinstance(sentence, tuple) and len(sentence) == 2: # NOT operation
        operator, operand = sentence
        if operator == "NOT":
            return not pl_true(operand, model)
    elif isinstance(sentence, tuple) and len(sentence) == 3:
        operator, left, right = sentence
        if operator == "AND":
            return pl_true(left, model) and pl_true(right, model)
        elif operator == "OR":
            return pl_true(left, model) or pl_true(right, model)
        elif operator == "IMPLIES":
            return not pl_true(left, model) or pl_true(right, model)
        elif operator == "IFF":
            return pl_true(left, model) == pl_true(right, model)

def tt_entails(kb, alpha, symbols):
    """Checks if KB entails alpha using truth-table enumeration."""

```

```

all_models = product([False, True], repeat=len(symbols))

valid_models = []

for values in all_models:

    model = dict(zip(symbols, values))

    kb_value = pl_true(kb, model)

    alpha_value = pl_true(alpha, model)

    if kb_value: # If KB is true in this model

        if not alpha_value: # If KB is true but α is not, entailment
fails

            return False, None

    else:

        valid_models.append(model)

return True, valid_models

```

```

def print_truth_table(kb, alpha, symbols):

    """Generates and prints the truth table for KB and α."""

    headers = ["A      ", "B      ", "C      ", "A ∨ C ", "B ∨ ¬C ", "KB",
", "α      "]

    print(" | ".join(headers))

    print("-" * (len(headers) * 9)) # Separator line

    # Generate all combinations of truth values

```

```

for values in product([False, True], repeat=len(symbols)):

    model = dict(zip(symbols, values))

    # Evaluate sub-expressions and main expressions

    a_or_c = pl_true(("OR", "A", "C"), model)

    b_or_not_c = pl_true(("OR", "B", ("NOT", "C")), model)

    kb_value = pl_true(kb, model)

    alpha_value = pl_true(alpha, model)

    # Print the truth table row

    row = values + (a_or_c, b_or_not_c, kb_value, alpha_value)

    row_str = " | ".join(str(v).ljust(7) for v in row)

    # Highlight rows where both KB and α are true

    if kb_value and alpha_value:

        print(f"\033[92m{row_str}\033[0m") # Green color for rows where
KB and α are true

    else:

        print(row_str)

# Define the knowledge base and query

symbols = ["A", "B", "C"]

kb = ("AND", ("OR", "A", "C"), ("OR", "B", ("NOT", "C")))

alpha = ("OR", "A", "B")

```

```

# Print the truth table

print_truth_table(kb, alpha, symbols)

# Run the truth-table entailment check

entailment, models = tt_entails(kb, alpha, symbols)

# Print the result

print("\nResult:")

if entailment:

    print("KB entails  $\alpha$ .")

    print("The values of A, B, C for which KB and  $\alpha$  are true:")

    for model in models:

        print(model)

else:

    print("KB does not entail  $\alpha$ .")

```

Output:

A	B	C	$A \vee C$	$B \vee \neg C$	KB	$\alpha$
False	False	False	False	True	False	False
False	False	True	True	False	False	False
False	True	False	False	True	False	True
False	True	True	True	True	True	True
True	False	False	True	True	True	True
True	False	True	True	False	False	True
True	True	False	True	True	True	True
True	True	True	True	True	True	True

Result:  
 KB entails  $\alpha$ .  
 The values of A, B, C for which KB and  $\alpha$  are true:  
 {'A': False, 'B': True, 'C': True}  
 {'A': True, 'B': False, 'C': False}  
 {'A': True, 'B': True, 'C': False}  
 {'A': True, 'B': True, 'C': True}

## Program 7

Implement unification in first order logic.

**Algorithm:**

LAB PROGRAM - 7

Implement unification in First Order Logic

Algorithm :  $\text{Unify } (\psi_1, \psi_2)$

Step 1 : If  $\psi_1$  or  $\psi_2$  is a variable or constant, then :

- If  $\psi_1$  or  $\psi_2$  are identical, then return NIL.
- Else if  $\psi_1$  is a variable,
  - then if  $\psi_1$  occurs in  $\psi_2$ , then return FAILURE.
  - Else return  $\{\psi_2 / \psi_1\}$ .
- Else if  $\psi_2$  is a variable,
  - if  $\psi_2$  occurs in  $\psi_1$ , then return FAILURE,
  - Else return  $\{\psi_1 / \psi_2\}$ .
- Else return FAILURE.

Step 2 : If the inline predicate symbol in  $\psi_1$  and  $\psi_2$  are ~~not same~~, then return FAILURE.

Step 3 : If  $\psi_1$  and  $\psi_2$  have a different number of ~~arguments~~ arguments, then return FAILURE.

Step 4 : Set substitution set (SUBST) to NIL.

Step 5 : For  $i=1$  to the number of elements in  $\psi_1$ ,

- Call Unify function with the  $i$ th element of  $\psi_1$  and  $i$ th element of  $\psi_2$  and put the result into  $s$ .
- If  $s = \text{failure}$ , then return failure.
- If  $s \neq \text{NIL}$ , then do -
  - Apply  $s$  to the remainder of both L1 and L2.
  - $\text{SUBST} = \text{APPEND}(s, \text{SUBST})$ .

Step 6 : Return SUBST.

$\mathcal{G}: \mathcal{D} \xrightarrow{P(z, F(y))} \textcircled{1}$        $\textcircled{1}$  predicate  
 $P(z, f(g(z))) \rightarrow \textcircled{2}$ .       $\textcircled{2}$  by  $\eta$  expansion  
 $\textcircled{1}$  and  $\textcircled{2}$  are identical if  $y$  is replaced with 'z' in  $\textcircled{1}$   
 $P(z, F(y))$   
 $y$  is replaced with  $g(z)$ .  
 $P(z, F(g(z)))$

---

$\mathcal{I} \quad Q(z, g(z), f(y)) \rightarrow \textcircled{1}$   
 $Q(z, g(f(z)), z) \rightarrow \textcircled{2}$ .  
 Replace  $z$  in  $\textcircled{2}$  with  $f(z) \Rightarrow Q(z, g(f(z)), f(y))$   
 Replace  $f(y)$  in  $\textcircled{2}$  with  $z \Rightarrow Q(z, g(f(z)), z)$

---

q1)  $\psi_1 = P(f(a), f(x))$   
 $\psi_2 = P(x, x)$   
 due to the predicates  $f$  and  $g$  are distinct and not same,  
 they cannot be substituted as the same variable  $x$ .  
unification fail

q2)  $\psi_1 = P(b, z, f(g(z))) \rightarrow \textcircled{1}$   
 $\psi_2 = P(z, f(y), f(y)) \rightarrow \textcircled{2}$ .  
 Replace  $b$  with  $z$  in  $\textcircled{1}$        $P(z, z, f(g(z)))$   
 $z$  with  $f(y)$  in  $\textcircled{1}$        $P(z, f(y), f(g(z)))$   
 $g(z)$  with  $y$        $P(z, f(y), f(y))$

OUTPUT

Case 1 :  $P(f(a), f(b))$  and  $P(z, z)$ .  
 unification unsuccessful!

Case 2 :  $P(b, z, f(g(z)))$  and  $P(z, f(y), f(y))$   
 Replace  $b$  with  $z$   
 Replace  $z$  with  $f(y)$   
 Replace  $y$  with  $g(z)$   
 unification successful.

Date: [unclear]

## Code:

```
def unify(expr1, expr2, subst=None):  
  
    if subst is None:  
  
        subst = {}  
  
    # Apply substitutions to both expressions  
  
    expr1 = apply_substitution(expr1, subst)  
  
    expr2 = apply_substitution(expr2, subst)  
  
  
    # Base case: Identical expressions  
  
    if expr1 == expr2:  
  
        return subst  
  
  
    # If expr1 is a variable  
  
    if is_variable(expr1):  
  
        return unify_variable(expr1, expr2, subst)  
  
  
    # If expr2 is a variable  
  
    if is_variable(expr2):  
  
        return unify_variable(expr2, expr1, subst)  
  
  
    # If both are compound expressions (e.g., f(a), P(x, y))  
  
    if is_compound(expr1) and is_compound(expr2):  
  
        if expr1[0] != expr2[0] or len(expr1[1]) != len(expr2[1]):
```

```

        return None # Predicate/function symbols or arity mismatch

    for arg1, arg2 in zip(expr1[1], expr2[1]):

        subst = unify(arg1, arg2, subst)

        if subst is None:

            return None

    return subst

# If they don't unify

return None

def unify_variable(var, expr, subst):

    """Handle variable unification."""

    if var in subst: # Variable already substituted

        return unify(subst[var], expr, subst)

    if occurs_check(var, expr, subst): # Occurs-check

        return None

    subst[var] = expr

    return subst

def apply_substitution(expr, subst):

    """Apply the current substitution set to an expression."""

    if is_variable(expr) and expr in subst:

        return apply_substitution(subst[expr], subst)

    if is_compound(expr):

```

```

        return (expr[0], [apply_substitution(arg, subst) for arg in expr[1]])

    return expr


def occurs_check(var, expr, subst):
    """Check for circular references."""
    if var == expr:
        return True

    if is_compound(expr):
        return any(occurs_check(var, arg, subst) for arg in expr[1])

    if is_variable(expr) and expr in subst:
        return occurs_check(var, subst[expr], subst)

    return False


def is_variable(expr):
    """Check if the expression is a variable."""
    return isinstance(expr, str) and expr.islower()


def is_compound(expr):
    """Check if the expression is a compound expression."""
    return isinstance(expr, tuple) and len(expr) == 2 and isinstance(expr[1], list)

# Testing the algorithm with the given cases

if __name__ == "__main__":
    # Case 1: p(f(a), f(b)) and p(x, x)

```

```

expr1 = ("p", [ ("f", ["a"]), ("g", ["b"]) ])

expr2 = ("p", ["x", "x"])

result = unify(expr1, expr2)

print("Case 1 Result:", result)

# Case 2: p(b, x, f(g(z))) and p(z, f(y), f(y))

expr1 = ("p", ["b", "x", ("f", [ ("g", ["z"]) ]) ])

expr2 = ("p", ["z", ("f", ["y"]), ("f", ["y"]) ])

result = unify(expr1, expr2)

print("Case 2 Result:", result)

```

Output:

---

```

Case 1 Result: None
Case 2 Result: {'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 PROGRAM - 8

Given case study you should  
create a knowledge base consisting of first order logic statements and  
prove the given query using forward reasoning.

Algorithm : FORWARD REASONING

function FOL-FC-ASK ( $KB, \alpha$ ) returns a substitution or false

inputs :  $KB$ , the knowledge base, a set of first order definite clauses.  
 $\alpha$ , the query, an atomic sentence

local variables : new, the new sentences inferred on each iteration

repeat until new is empty

    new  $\leftarrow \{\}$

    for each rule in  $KB$  do

$(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-VARIABLES}(\text{rule})$

        for each  $\theta$  such that  $\text{SUBST}(\theta, p_1 \wedge \dots \wedge p_n) =$   
 $\text{SUBST}(\theta, p'_1 \wedge \dots \wedge p'_n)$

            for some  $p'_1, \dots, p'_n$  in  $KB$

$q' \leftarrow \text{SUBST}(\theta, q)$

                if  $q'$  does not unify with some sentence already in  
 $KB$  or new then

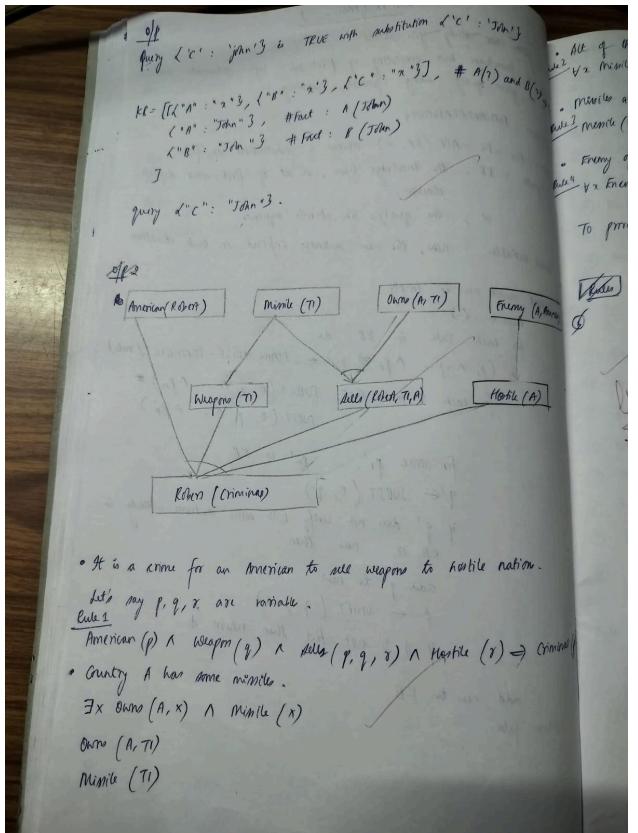
                    add  $q'$  to new

$\phi \leftarrow \text{UNIFY}(q', \alpha)$

                    if  $\phi$  is not fail then return  $\phi$

                add new to  $KB$

    return fail.



• All of the missiles were sold to country A by Robert.

$\forall x \text{ Missle}(x) \wedge \text{Doll}(A, x) \Rightarrow \text{Doll}(\text{Robert}, x, A)$ .

⇒ Q3:  $\{ \text{missile are weapons.} \}$

$\text{missile}(x) \Rightarrow \text{weapon}(x)$

• Enemy of America is known as hostile

$\forall x \text{ Enemy}(x, America) \Rightarrow \text{Hostile}(x)$

To prove: Robert is criminal

**Facts:**

- ① Robert is an American.
- ② Country A is an enemy of America.

Date: 26/11/2021

**Code:**

```
# Define the knowledge base with facts and rules

knowledge_base = [
    # Rule: Selling weapons to a hostile nation makes one a criminal
    {
        "type": "rule",
        "if": [
            {"type": "sells", "seller": "?X", "item": "?Z", "buyer": "?Y"},  

            {"type": "hostile_nation", "nation": "?Y"},  

            {"type": "citizen", "person": "?X", "country": "america"}
        ],
        "then": {"type": "criminal", "person": "?X"}
    },
    # Facts
    {"type": "hostile_nation", "nation": "CountryA"},  

    {"type": "sells", "seller": "Robert", "item": "missiles", "buyer": "CountryA"},  

    {"type": "citizen", "person": "Robert", "country": "america"}
]

# Forward chaining function

def forward_reasoning(kb, query):
    inferred = [] # Track inferred facts
    while True:
        new_inferences = []
        for rule in [r for r in kb if r["type"] == "rule"]:
```

```

conditions = rule["if"]
conclusion = rule["then"]
substitutions = {}

if match_conditions(conditions, kb, substitutions):
    inferred_fact = substitute(conclusion, substitutions)

    if inferred_fact not in kb and inferred_fact not in new_inferences:
        new_inferences.append(inferred_fact)

if not new_inferences:
    break

kb.extend(new_inferences)
inferred.extend(new_inferences)

return query in kb

```

```

# Helper to match conditions

def match_conditions(conditions, kb, substitutions):
    for condition in conditions:
        if not any(match_fact(condition, fact, substitutions) for fact in kb):
            return False

    return True

```

```

# Helper to match a single fact

def match_fact(condition, fact, substitutions):
    if condition["type"] != fact["type"]:
        return False

    for key, value in condition.items():
        if key == "type":

```

```

        continue

if isinstance(value, str) and value.startswith("?"): # Variable
    variable = value

if variable in substitutions:
    if substitutions[variable] != fact[key]:
        return False
    else:
        substitutions[variable] = fact[key]

elif fact[key] != value: # Constant
    return False

return True

# Substitute variables with their values

def substitute(conclusion, substitutions):
    result = conclusion.copy()

    for key, value in conclusion.items():
        if isinstance(value, str) and value.startswith("?"):
            result[key] = substitutions[value]

    return result

# Query: Is Robert a criminal?

query = {"type": "criminal", "person": "Robert"}  


```

```

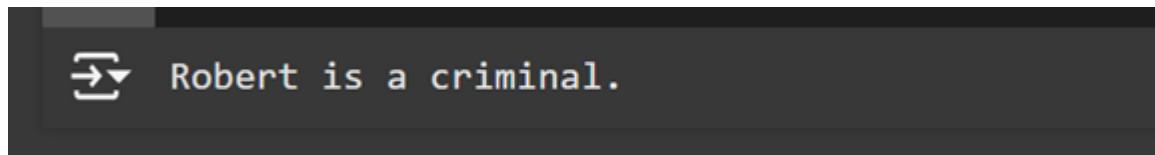
# Run the reasoning algorithm

if forward_reasoning(knowledge_base, query):
    print("Robert is a criminal.")

```

```
else:  
    print("Could not prove that Robert is a criminal.")
```

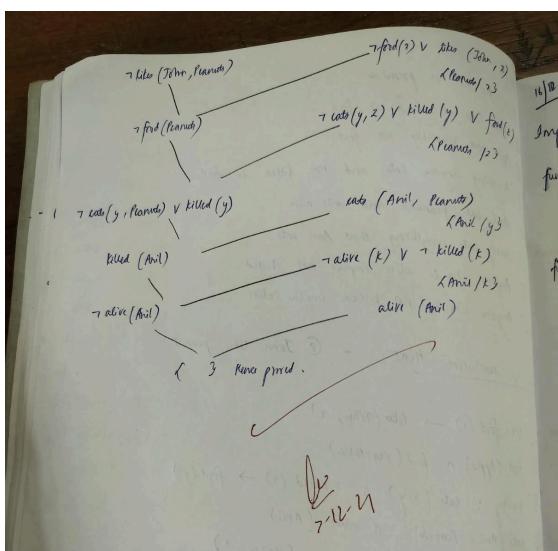
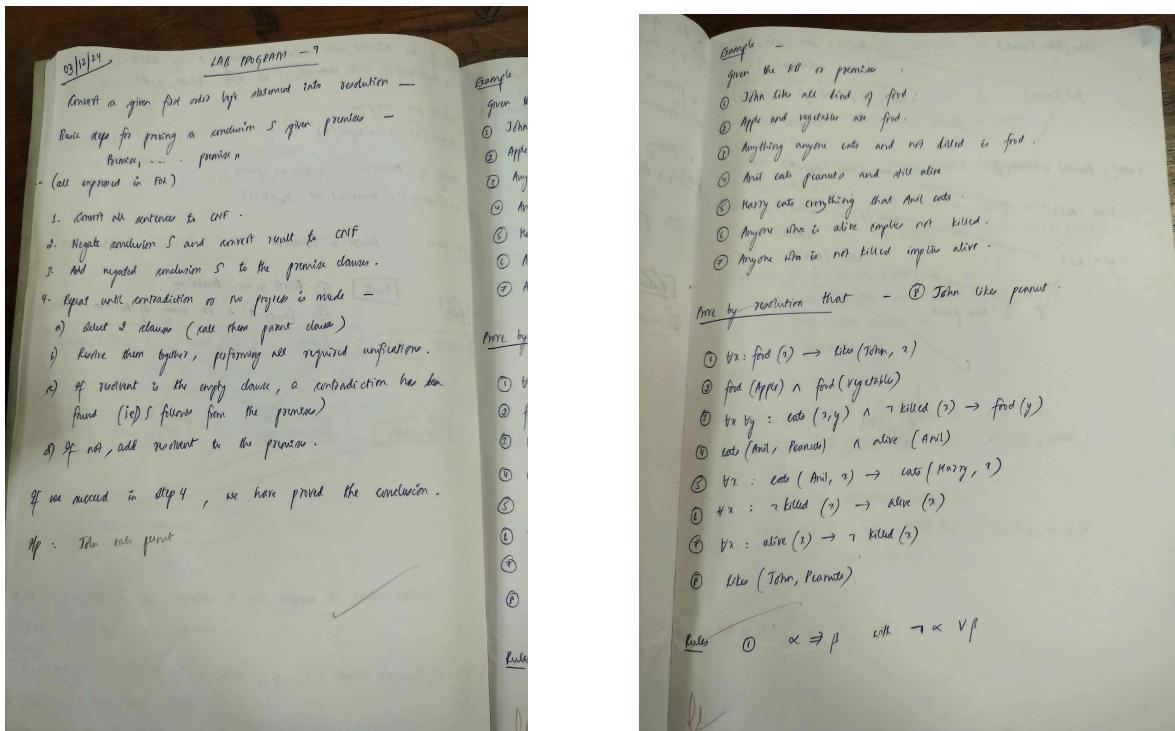
**Output:**



## Program 9

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

### Algorithm:



**Code:**

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)", # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)", # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)", # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)", # Rule: Alive implies not killed
    "not killed(X)": "alive(X)", # Rule: Not killed implies alive
}
```

```
# Function to evaluate if a predicate is true based on the KB
```

```
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]
```

```
# If it's a derived rule
```

```
if predicate in KB:
    rule = KB[predicate]
    if " and " in rule: # Handle conjunction
```

```

sub_preds = rule.split(" and ")
return all(resolve(sub.strip()) for sub in sub_preds)

elif " or " in rule: # Handle disjunction
    sub_preds = rule.split(" or ")
    return any(resolve(sub.strip()) for sub in sub_preds)

elif "not " in rule: # Handle negation
    sub_pred = rule[4:] # Remove "not "
    return not resolve(sub_pred.strip())

else: # Handle single predicate
    return resolve(rule.strip())

# If the predicate is a specific query (e.g., likes(John, Peanuts))

if "(" in predicate:
    func, args = predicate.split("(")
    args = args.strip(")").split(", ")
    if func == "food" and args[0] == "Peanuts":
        return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
    if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
        return resolve("food(Peanuts)")

# Default to False if no rule or fact applies

return False

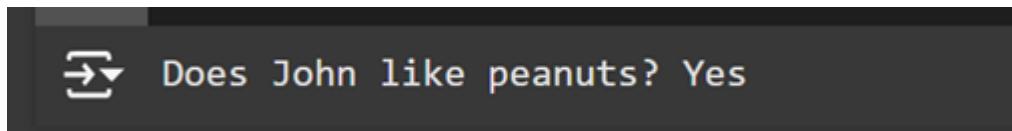
# Query to prove: John likes Peanuts

query = "likes(John, Peanuts)"
result = resolve(query)

```

```
# Print the result  
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

Output:



## Program 10

Implement Alpha-Beta Pruning.

**Algorithm:**

10/10/19

LAR PROGRAM - 10

Implement Alpha Beta Pruning —

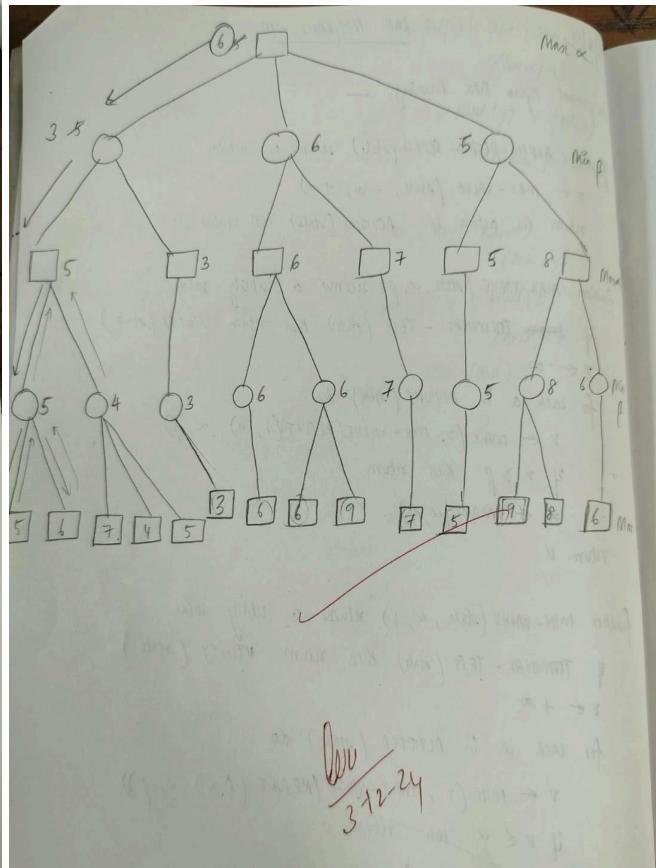
```

function ALPHA-BETA-SEARCH(state) returns an action
    v ← MAX-VALUE(state, -∞, +∞)
    return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← -∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(a), α, β))
        if v > β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← +∞
    for each a in ACTIONS(state) do
        v ← MIN(v, MAX-VALUE(RESULT(a), α, β))
        if v ≤ α then return v
        β ← MIN(β, v)
    return v

```



## Code:

```
import math

def minimax(node, depth, is_maximizing):
```

"""

Implement the Minimax algorithm to solve the decision tree.

Parameters:

node (dict): The current node in the decision tree, with the following structure:

```
{  
    'value': int,  
    'left': dict or None,  
    'right': dict or None  
}
```

depth (int): The current depth in the decision tree.

is\_maximizing (bool): Flag to indicate whether the current player is the maximizing player.

Returns:

int: The utility value of the current node.

"""

# Base case: Leaf node

if node['left'] is None and node['right'] is None:

```
    return node['value']
```

# Recursive case

if is\_maximizing:

```
    best_value = -math.inf
```

```
    if node['left']:
```

```

best_value = max(best_value, minimax(node['left'], depth + 1, False))

if node['right']:
    best_value = max(best_value, minimax(node['right'], depth + 1, False))

return best_value

else:

    best_value = math.inf

    if node['left']:
        best_value = min(best_value, minimax(node['left'], depth + 1, True))

    if node['right']:
        best_value = min(best_value, minimax(node['right'], depth + 1, True))

    return best_value

# Example usage

decision_tree = {

    'value': 5,

    'left': {

        'value': 6,

        'left': {

            'value': 7,

            'left': {

                'value': 4,

                'left': None,

                'right': None
            },
            'right': {

                'value': 5,
                'left': None,
            }
        }
    }
}

```

```
'right': None
}
},
'right': {
  'value': 3,
  'left': {
    'value': 6,
    'left': None,
    'right': None
  },
  'right': {
    'value': 9,
    'left': None,
    'right': None
  }
},
'right': {
  'value': 8,
  'left': {
    'value': 7,
    'left': {
      'value': 6,
      'left': None,
      'right': None
    },
    'right': None
  }
},
```

```

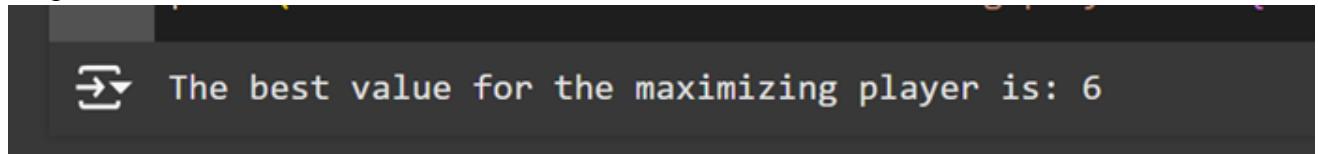
    'right': {
        'value': 9,
        'left': None,
        'right': None
    },
    'right': {
        'value': 8,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': None
    }
}

# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)

print(f"The best value for the maximizing player is: {best_value}")

```

Output:



```
The best value for the maximizing player is: 6
```