VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shreya Soni (1BM22CS268)

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 **Shreya Soni (1BM22CS268),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

Github Link: https://github.com/SHREYASONI28/AI-LAB

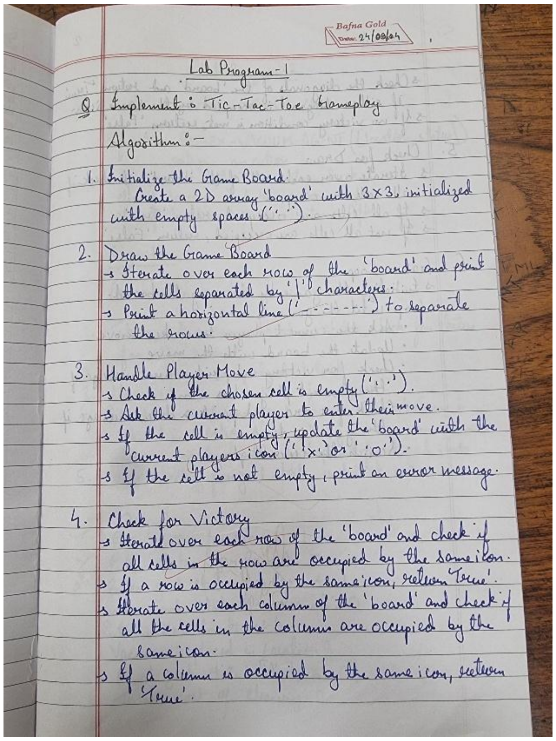
Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:

Tic –Tac –Toe Game



Bafna Gold Date: 24/09/24

Lab Program - 1

Q. Implement : Tic-Tac-Toe Gameplay

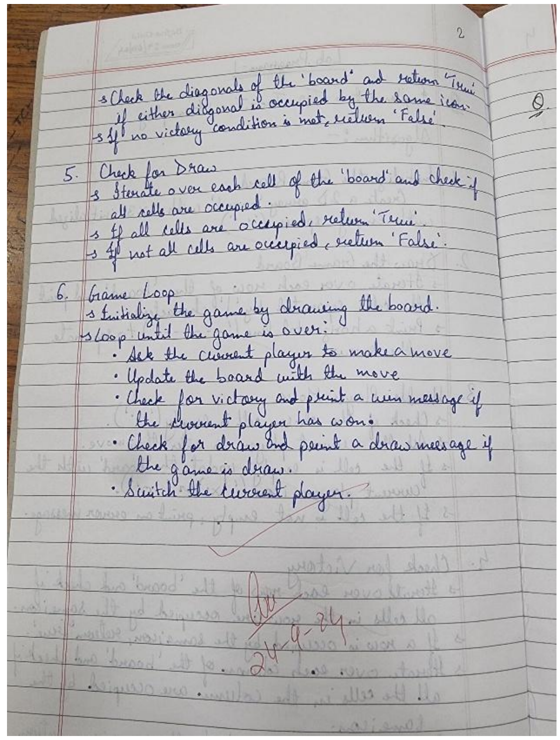
Algorithm :-

1. Initialize the Game Board
Create a 2D array 'board' with 3x3, initialized with empty spaces (' ').

2. Draw the Game Board
-> Iterate over each row of the 'board' and print the cells separated by '|' characters.
-> Print a horizontal line ('-----') to separate the rows.

3. Handle Player Move
-> Check if the chosen cell is empty (' ').
-> Ask the current player to enter their move.
-> If the cell is empty, update the 'board' with the current players icon ('x' or 'o').
-> If the cell is not empty, print an error message.

4. Check for Victory
-> Iterate over each row of the 'board' and check if all cells in the row are occupied by the same icon.
-> If a row is occupied by the same icon, return 'True'.
-> Iterate over each column of the 'board' and check if all the cells in the column are occupied by the same icon.
-> If a column is occupied by the same icon, return 'True'.



2

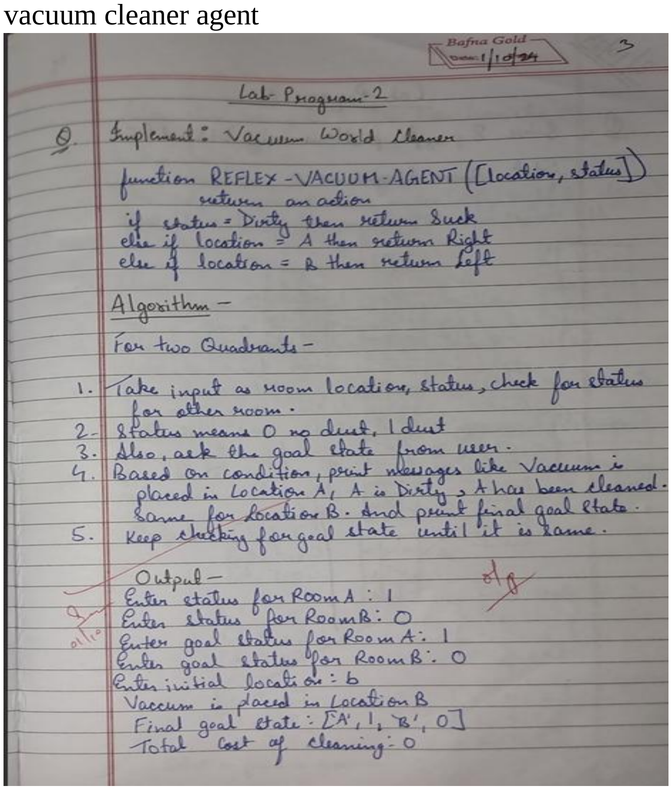
-> Check the diagonals of the 'board' and return 'True' if either diagonal is occupied by the same icon.
-> If no victory condition is met, return 'False'.

5. Check for Draw
-> Iterate over each cell of the 'board' and check if all cells are occupied.
-> If all cells are occupied, return 'True'.
-> If not all cells are occupied, return 'False'.

6. Game Loop
-> Initialize the game by drawing the board.
-> Loop until the game is over:
- Ask the current player to make a move
- Update the board with the move
- Check for victory and print a win message if the current player has won.
- Check for draw and print a draw message if the game is draw.
- Switch the current player.

24-9-24

vacuum cleaner agent



Bafna Gold Date: 1/10/24 3

Lab Program - 2

Q. Implement : Vacuum World Cleaner

function REFLEX-VACUUM-AGENT ([location, status])
returns an action
if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return Left

Algorithm -

For two Quadrants -

1. Take input as room location, status, check for status for other room.
2. Status means 0 no dust, 1 dust
3. Also, ask the goal state from user.
4. Based on condition, print messages like Vacuum is placed in Location A, A is Dirty, A has been cleaned. Same for Location B. And print final goal state.
5. Keep checking for goal state until it is same.

Output -
Enter status for Room A : 1
Enter status for Room B : 0
Enter goal status for Room A : 1
Enter goal status for Room B : 0
Enter initial location : b
Vaccum is placed in Location B
Final goal state : ['A', 1, 'B', 0]
Total cost of cleaning : 0

Code:

```
# Tic-Tac-Toe game
# The game board
board = [' ' for _ in range(9)]

# Function to draw the game board
def draw_board():
    row1 = '| {} | {} | {} |'.format(board[0], board[1], board[2])
    row2 = '| {} | {} | {} |'.format(board[3], board[4], board[5])
    row3 = '| {} | {} | {} |'.format(board[6], board[7], board[8])

    print()
    print(row1)
    print(row2)
    print(row3)
    print()

# Function to handle player move
def player_move(icon):
    if icon == 'X':
        number = 1
    elif icon == 'O':
        number = 2

    print("Your turn player {}".format(number))

    choice = int(input("Enter your move (1-9): ").strip())
    if board[choice - 1] == ' ':
        board[choice - 1] = icon
    else:
        print()
        print("That space is taken!")

# Function to check for a win
def is_victory(icon):
    if (board[0] == icon and board[1] == icon and board[2] == icon) or \
        (board[3] == icon and board[4] == icon and board[5] == icon) or \
        (board[6] == icon and board[7] == icon and board[8] == icon) or \
        (board[0] == icon and board[3] == icon and board[6] == icon) or \
        (board[1] == icon and board[4] == icon and board[7] == icon) or \
        (board[2] == icon and board[5] == icon and board[8] == icon) or \
        (board[0] == icon and board[4] == icon and board[8] == icon) or \
        (board[2] == icon and board[4] == icon and board[6] == icon):
        return True
    else:
        return False

# Function to check for a draw
```

```
def is_draw():
    if ' ' not in board:
        return True
    else:
        return False

# Function to handle the game loop
def play_game():
    draw_board()
    while True:
        player_move('X')
        draw_board()
        if is_victory('X'):
            print("Player 1 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break
        player_move('O')
        draw_board()
        if is_victory('O'):
            print("Player 2 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break

play_game()
```

OUTPUT:

```
|   |   |   |
|   |   |   |
|   |   |   |

Your turn player 1
Enter your move (1-9): 8

|   |   |   |
|   |   |   |
|   | X |   |

Your turn player 2
Enter your move (1-9): 1

| O |   |   |
|   |   |   |
|   | X |   |

Your turn player 1
Enter your move (1-9): 7

| O |   |   |
|   |   |   |
| X | X |   |

Your turn player 2
Enter your move (1-9): 9
```

```
Your turn player 1
Enter your move (1-9): 5

| O |   |   |
|   | X |   |
| X | X | O |

Your turn player 2
Enter your move (1-9): 3

| O |   | O |
|   | X |   |
| X | X | O |

Your turn player 1
Enter your move (1-9): 2

| O | X | O |
|   | X |   |
| X | X | O |

Player 1 wins! Congratulations!
```

```
# Vacuum cleaner agent
def vacuum_cleaner_simulation():

    current_room = input("Enter current room either A or B: ").upper()
    room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
    room_B = int(input("Is Room B dirty? (yes:1/no:0): "))

    cost = 0

    def display_rooms():
        print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
        print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")

    print("\nInitial status of rooms:")
    display_rooms()
    print()

    while room_A == 1 or room_B == 1:
        if current_room == 'A' and room_A == 1:
            print("Cleaning Room A...")
            room_A = 0
            cost += 1
        elif current_room == 'B' and room_B == 1:
            print("Cleaning Room B...")
            room_B = 0
            cost += 1
        else:
            current_room = 'B' if current_room == 'A' else 'A'
            print(f"Moving to Room {current_room}...")
        print("Current status:")
        display_rooms()

    print(f"\nBoth rooms are now clean! Total cost: {cost}")

vacuum_cleaner_simulation()


#For four quadrants
def vacuum_cleaner_simulation():
    current_room = input("Enter current room (A, B, C, or D): ").upper()
    room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
    room_B = int(input("Is Room B dirty? (yes:1/no:0): "))
    room_C = int(input("Is Room C dirty? (yes:1/no:0): "))
    room_D = int(input("Is Room D dirty? (yes:1/no:0): "))

    cost = 0
    count=2
```

```
def display_rooms():
    print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
    print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")
    print(f"Room C: {'Clean' if room_C == 0 else 'Dirty'}")
    print(f"Room D: {'Clean' if room_D == 0 else 'Dirty'}")

print("\nInitial status of rooms:")
display_rooms()
print()

while room_A == 1 or room_B == 1 or room_C == 1 or room_D == 1:
    if count==0:
      print("Vacuum is recharging")
      count=2
    else:
      if current_room == 'A' and room_A == 1:
          print("Cleaning Room A...")
          room_A = 0
          cost += 1
          count-=1
      elif current_room == 'B' and room_B == 1:
          print("Cleaning Room B...")
          room_B = 0
          cost += 1
          count-=1
      elif current_room == 'C' and room_C == 1:
          print("Cleaning Room C...")
          room_C = 0
          cost += 1
          count-=1
      elif current_room == 'D' and room_D == 1:
          print("Cleaning Room D...")
          room_D = 0
          cost += 1
          count-=1
      else:
          if current_room == 'A':
              current_room = 'B'
          elif current_room == 'B':
              current_room = 'C'
          elif current_room == 'C':
              current_room = 'D'
          else:
              current_room = 'A'
          print(f"Moving to Room {current_room}...")

print("\nCurrent status:")
display_rooms()
```

```
    print(f"\nAll rooms are now clean! Total cost: {cost}")

vacuum_cleaner_simulation()
```

OUTPUT:

```
Enter current room either A or B: 1
Is Room A dirty? (yes:1/no:0): 0
Is Room B dirty? (yes:1/no:0): 0

Initial status of rooms:
Room A: Clean
Room B: Clean


Both rooms are now clean! Total cost: 0
Enter current room (A, B, C, or D): 4
Is Room A dirty? (yes:1/no:0): 1
Is Room B dirty? (yes:1/no:0): 0
Is Room C dirty? (yes:1/no:0): 1
Is Room D dirty? (yes:1/no:0): 0
```

```
Initial status of rooms:
Room A: Dirty
Room B: Clean
Room C: Dirty
Room D: Clean

Moving to Room A...
Cleaning Room A...
Moving to Room B...
Moving to Room C...
Cleaning Room C...

Current status:
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Clean

All rooms are now clean! Total cost: 2
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

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

Algorithm using DFS -

→ Set the goal configuration of the puzzle.
→ Locate the empty space(s).
→ Generate valid states by moving adjacent tiles into empty space.
→ If the current state matches the goal state, return the path.
→ If no solution is found, backtrack.
→ Prompt the user for the initial configuration.
→ Convert the input to a 2D matrix.
→ Call the DFS function with the initial state and track visited states.

Iterative deepening search algorithm.

4

22.1.2024

Lab Program-4

Q. Implement Iterative Deepening depth first Search algorithm.

Algorithm -

function ITERATIVE-DEEPENING-SEARCH (problem) returns a solution, or failure
for depth = 0 to ∞ do
result ← Depth-Limited-Search (problem, depth)
if result ≠ cutoff then return result

OR

1. For each child of the current node.
2. If it is the target node, return
3. If the current maximum depth is reached, return
4. Set the current node to this node and go back to 1.
5. After having gone through all children, go to the next child of the parent (the next sibling)
6. After having gone through all children of the start node, increase the maximum depth and go back to 1.
7. If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

State Space for graph

Initial State: A , Goal State: D

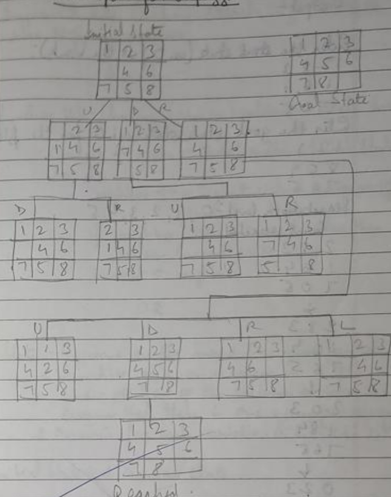
Solution Path: ['A', 'B', 'D']

Output :

Enter the initial state: A
Enter the goal state: D
Enter the adjacency list for the graph (neighbors of each node)
Type done when finished.
Enter node (or 'done' to finish): A
Enter neighbors of A separated by spaces: B C
: B
: A D
: C
: A
: D
: B
Done

Exploring depth: 0
Exploring depth: 1
Exploring depth: 2
Solution Path: ['A', 'B', 'D']

State Space for 8-puzzle



Output -

Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0 5
Enter the goal state state (use 0 for the blank):
1 2 3
8 0 4
7 6 5
Search depth level: 0, 1, 2, 3, 4, 5
Goal reached
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

CODE:

```
# 8 puzzle problems using Depth First Search (DFS)
 from collections import deque

 def dfs(start, max_depth):
    stack = deque([(start, [start], 0)])  # (node, path, level)
    visited = set([start])
    all_moves = []
    while stack:
        node, path, level = stack.pop()
        all_moves.append((path, level))
        if level < max_depth:
            for next_node in get_neighbors(node):
                if next_node not in visited:
                    visited.add(next_node)
                    stack.append((next_node, path + [next_node], level + 1))
    return all_moves

 def get_neighbors(node):
    neighbors = []
    for i in range(9):
        if node[i] == 0:
            x, y = i // 3, i % 3
            for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < 3 and 0 <= ny < 3:
                    n = list(node)
                    n[i], n[nx * 3 + ny] = n[nx * 3 + ny], n[i]
                    neighbors.append(tuple(n))
            break
    return neighbors

 def print_board(board):
    board = [board[i:i+3] for i in range(0, 9, 3)]
    for row in board:
        print(" | ".join(str(x) for x in row))
        print("---------")

 def main():
    start = tuple(int(x) for x in input("Enter the initial state (space-separated): ").split())
    max_depth = 10  # maximum depth to search
    all_moves = dfs(start, max_depth)
    if all_moves:
        print("All possible moves:")
        for i, (path, level) in enumerate(all_moves):
            print(f"Move {i+1}:")
            for j, node in enumerate(path):
                print(f"Step {j}:")
```

```
            print_board(node)
            print()
        print(f"Number of moves: {level}")
        print()
    else:
        print("No solution found.")

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

OUTPUT:

```
4 | 2 | 8
---------
3 | 6 | 7
---------

Step 6:
0 | 1 | 5
---------
4 | 2 | 8
---------
3 | 6 | 7
---------

Step 7:
4 | 1 | 5
---------
0 | 2 | 8
---------
3 | 6 | 7
---------

Step 8:
4 | 1 | 5
---------
2 | 0 | 8
---------
3 | 6 | 7
---------

Step 9:
4 | 1 | 5
---------
2 | 8 | 0
---------
3 | 6 | 7
---------

Step 10:
4 | 1 | 0
---------
2 | 8 | 5
---------
3 | 6 | 7
---------

Number of moves: 10
```

```
# Iterative deepening search algorithm
from copy import deepcopy
```

```
# Directions for moving the blank space (0): up, down, left, right
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                # Swap the blank with the adjacent tile
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors

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

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None
```

```
def iterative_deepening_search(start_state, goal_state):
    depth = 0
    while True:
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
        depth += 1

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

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

    return start_state, goal_state

def main():
    # Get the start and goal states from the user
    start_board, goal_board = get_user_input()

    # Create PuzzleState objects for start and goal
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    # Perform iterative deepening search
    result = iterative_deepening_search(start_state, goal_state)

    # Display the result
    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found.")
```

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

OUTPUT:

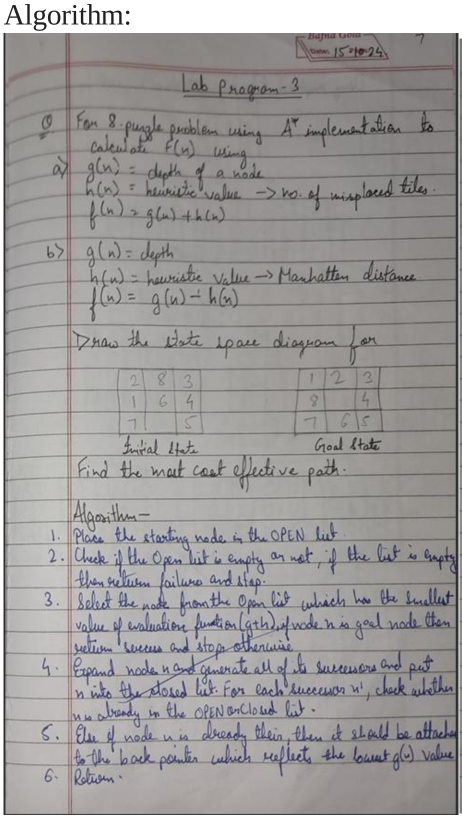
```
Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0 5
Enter the goal state (use 0 for the blank):
1 2 3
8 0 4
7 6 5

Searching at depth level: 0

Searching at depth level: 1

Searching at depth level: 2

Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5
```

```
Goal reached!
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 3

Implement A* search algorithm.

Algorithm:



Date: 15-10-24

Lab Program-3

Q For 8 puzzle problem using A* implementation to calculate F(n) using

a) g(n) = depth of a node
h(n) = heuristic value -> no. of misplaced tiles.
f(n) = g(n) + h(n)

b) g(n) = depth
h(n) = heuristic Value -> Manhatten distance
f(n) = g(n) + h(n)

Draw the state space diagram for

2	8	3
1	6	4
7		5

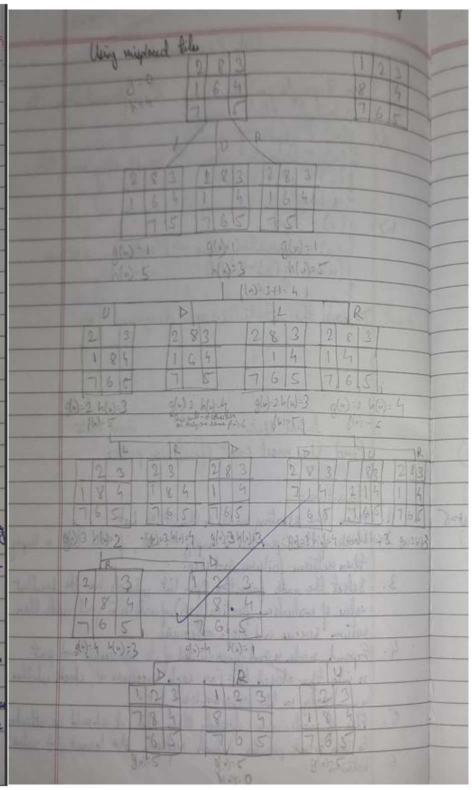
Initial State

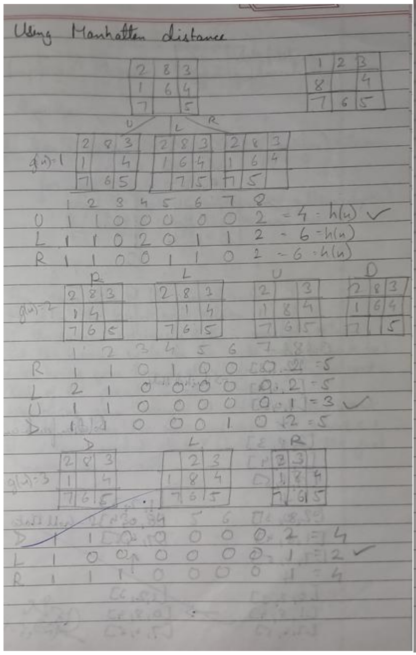
1	2	3
8		4
7	6	5

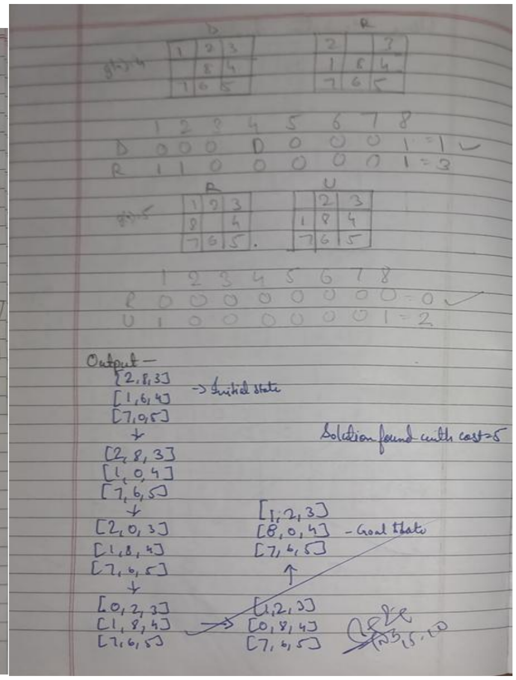
Goal State

Find the most cost effective path.

Algorithm–
1. Place the starting node in the OPEN list.
2. Check if the Open list is empty or not, if the list is empty then return failure and stop.
3. Select the node from the Open list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop otherwise
4. Expand node n and generate all of its successors and put n into the closed list. For each successor n', check whether n' is already in the OPEN or Closed list.
5. Else if node n is already there, then it should be attached to the back pointer which reflects the lowest g(n) value.
6. Return.







CODE:

```
#A* 8-PUZZLE NO. OF MISPLACED TILES
import heapq

def misplaced_tile(state, goal_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, goal):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tile(start, goal), 0, start))

    g_score = {start: 0}
    came_from = {}

    visited = set()

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

        if current == goal:
            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, goal)  # 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()
```

```python
def get_state_from_user(prompt):

    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
    goal_state = get_state_from_user("Goal state")

    print("\nInitial State:")
    print_state(start_state)

    print("\nGoal State:")
    print_state(goal_state)

    solution, cost = a_star(start_state, goal_state)

    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("\nNo solution found.")
```

OUTPUT:

```
Enter the initial state:
Initial state row 1 (space-separated): 2 8 3
Initial state row 2 (space-separated): 1 6 4
Initial state row 3 (space-separated): 7 0 5

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 8 0 4
Goal state row 3 (space-separated): 7 6 5

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


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

```
Solution found with cost: 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)
```

#A* 8-PUZZLE MANHATTEN DISTANCE

```
import heapq

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

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

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, goal):
    open_list = []
```

```
    heapq.heappush(open_list, (0 + manhattan_distance(start, goal), 0, start))

    g_score = {start: 0}
    came_from = {}

    visited = set()

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

        if current == goal:
            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, goal)

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

    return None, None

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

def get_state_from_user(prompt):
    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
    goal_state = get_state_from_user("Goal state")
```

```
    print("\nInitial State:")
    print_state(start_state)

    print("\nGoal State:")
    print_state(goal_state)

    solution, cost = a_star(start_state, goal_state)

    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("\nNo solution found.")
```

OUTPUT:

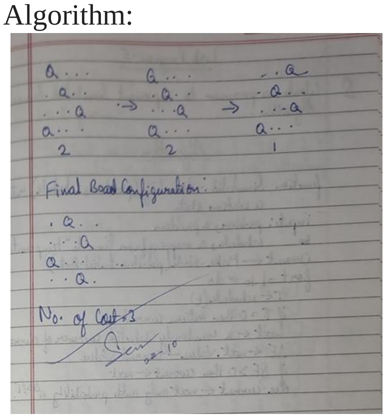
```
Enter the initial state:
Initial state row 1 (space-separated): 2 8 3
Initial state row 2 (space-separated): 1 6 4
Initial state row 3 (space-separated): 7 0 5

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 8 0 4
Goal state row 3 (space-separated): 7 6 5

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


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

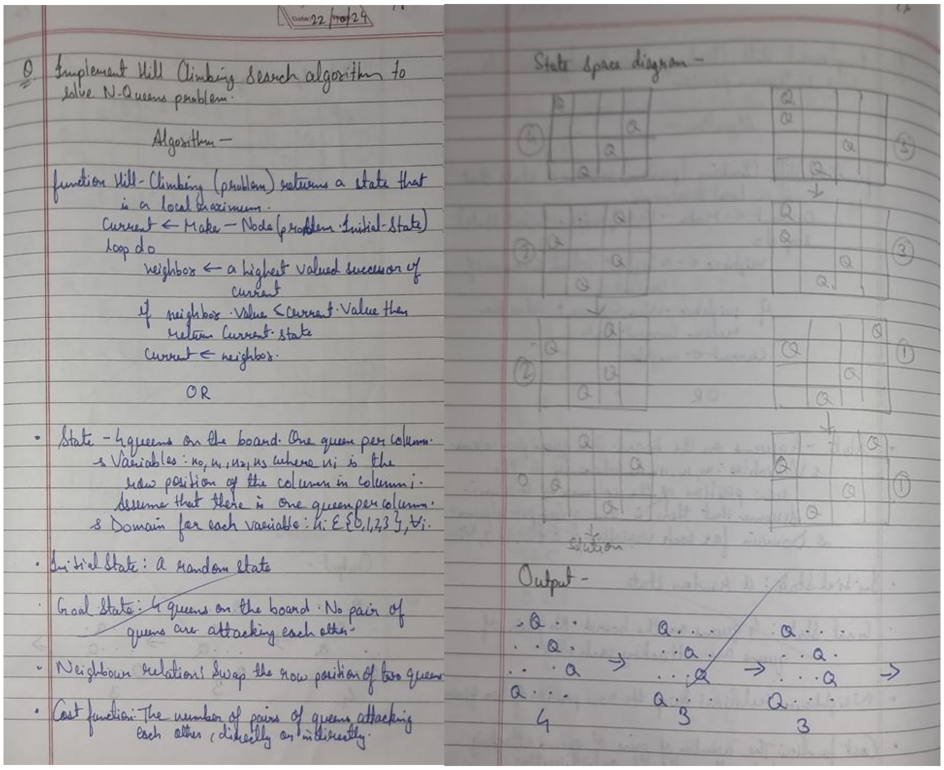
```
Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(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:




Date: 22/10/24

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

Algorithm –

function Hill-Climbing (problem) returns a state that is a local maximum.
Current ← Make – Node (problem.Initial-State)
loop do
neighbor ← a highest valued successor of current
if neighbor.Value < current.Value then return Current.state
Current ← neighbor.

OR

- State – 4 queens on the board. One queen per column.
  - Variables: x_0, x_1, x_2, x_3 where x_i is the row position of the column in column i. Assume that there is one queen per column.
  - Domain for each variable: $x_i \in \{0,1,2,3\}, \forall i$.
- Initial State: a random state
- Goal State: 4 queens on the board. No pair of queens are attacking each other.
- Neighbour relation: Swap the row position of two queens
- Cost function: The number of pairs of queens attacking each other, directly or indirectly.

CODE:

```
#HILL CLIMBING N-QUEENS
import random

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def hill_climbing(n):
    cost=0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)

        while True:
            # Generate neighbors by moving each queen to a different position
            found_better = False
            for i in range(n):
                for j in range(n):
```

```
            if j != current_board[i]:  # Only consider different positions
                neighbor_board = list(current_board)
                neighbor_board[i] = j
                neighbor_conflicts = calculate_conflicts(neighbor_board)
                if neighbor_conflicts < current_conflicts:
                    print_board(current_board)
                    print(current_conflicts)
                    print_board(neighbor_board)
                    print(neighbor_conflicts)
                    current_board = neighbor_board
                    current_conflicts = neighbor_conflicts
                    cost+=1
                    found_better = True
                    break
        if found_better:
            break

    # If no better neighbor found, stop searching
    if not found_better:
        break

  # If a solution is found (zero conflicts), return the board
  if current_conflicts == 0:
    return current_board, current_conflicts, cost

def print_board(board):
  n = len(board)
  for i in range(n):
    row = ['.'] * n
    row[board[i]] = 'Q'  # Place a queen
    print(' '.join(row))
  print()
print("===============")
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
```
OUTPUT:

```
. . . Q
. . Q .
. Q . .
Q . . .

6
Q . . .
. . Q .
. Q . .
Q . . .

4
Q . . .
. . Q .
. Q . .
Q . . .

4
Q . . .
. . . Q
. Q . .
Q . . .

2
Q . . .
. . . Q
. Q . .
```

```
4
Q . . .
. . . Q
. Q . .
Q . . .

2
Q . . .
. . . Q
. Q . .
Q . . .

2
Q . . .
. . . Q
. Q . .
. . Q .

1
. . Q .
. Q . .
Q . . .
. . . Q

4
. Q . .
. Q . .
Q . . .
. . . Q

3
. Q . .
. Q . .
Q . . .
```

```
3
. Q . .
Q . . .
Q . . .
. . . Q

2
. Q . .
Q . . .
Q . . .
. . . Q

2
. Q . .
. . Q .
Q . . .
. . . Q

1
. . Q .
. Q . .
Q . . .
. . . Q

4
. Q . .
. Q . .
Q . . .
. . . Q

3
. Q . .
. Q . .
```

```
1
Q . . .
. Q . .
. . Q .
. . . Q

6
. Q . .
. Q . .
. . Q .
. . . Q

4
. Q . .
. Q . .
. . Q .
. . . Q

4
. Q . .
Q . . .
. . Q .
. . . Q

2
Final Board Configuratior
. Q . .
. . . Q
Q . . .
. . Q .

Number of Cost: 26
```

Program 5

Simulated Annealing to Solve 8-Queens problem.

CODE:

```
#sIMULATED ANNEALING 8-QUEENS
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int)  # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1
```

```
    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun  # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

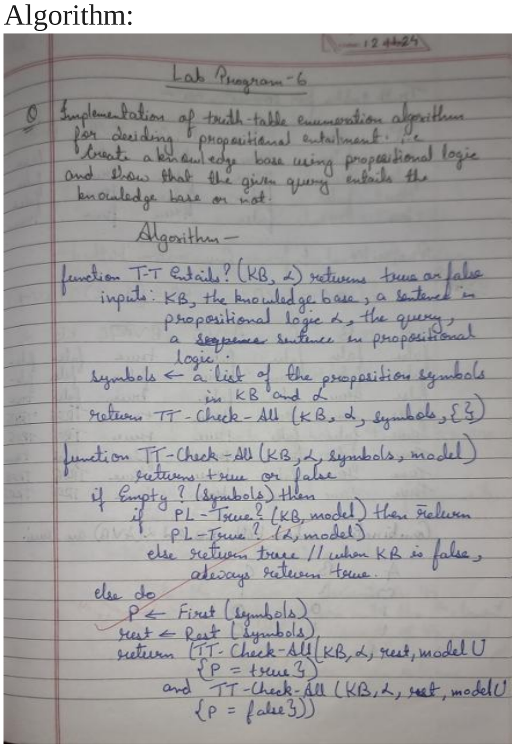
OUTPUT:

```
The best position found is: [6 3 1 7 5 0 2 4]
The number of queens that are not attacking each other is: 8
```

Program 6

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

Algorithm:



12-11-24

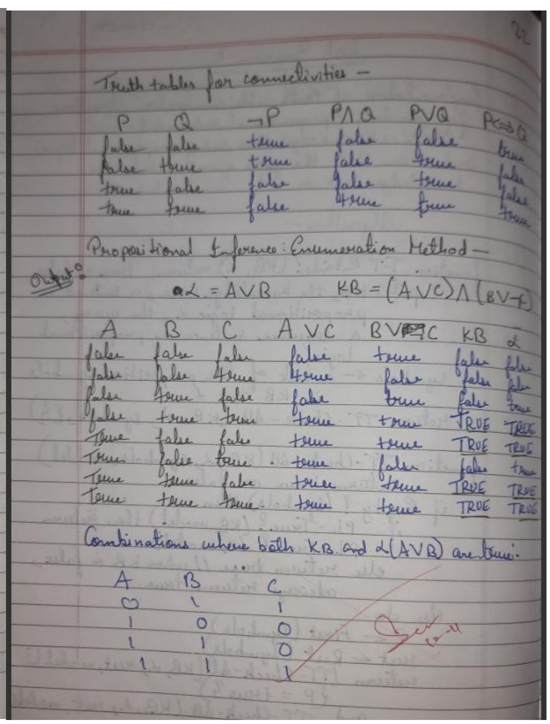
Lab Program-6

Q Implementation of truth-table enumeration algorithm for deciding propositional entailment. i.e. Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm –

function T-T Entails? (KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic α, the query, a ~~sequence~~ sentence in propositional logic.
symbols ← a list of the proposition symbols in KB and α
return TT-Check-All (KB, α, symbols, { })

function TT-Check-All (KB, α, symbols, model) returns true or false
if Empty? (symbols) then
if PL-True? (KB, model) then return PL-True? (α, model)
else return true // when KB is false, always return true.
else do
P ← First (symbols)
rest ← Rest (symbols)
return (TT-Check-All (KB, α, rest, model ∪ {P = true})
and TT-Check-All (KB, α, rest, model ∪ {P = false}))



22

Truth tables for connectivities –

P	Q	¬P	P∧Q	P∨Q	P⇔Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

Propositional Inference: Enumeration Method –

Output:

α = A∨B   KB = (A∨C)∧(B∨¬C)

A	B	C	A∨C	B∨¬C	KB	α
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

Combinations where both KB and α(A∨B) are true:

A	B	C
0	1	1
1	0	0
1	1	0
1	1	1

CODE:

```
#propositional logic

import itertools
```

```
# Define symbols in the KB and query
symbols = ['A', 'B', 'C']

# Define the Knowledge Base (KB) as separate components
A_or_C = lambda A, B, C: A or C
B_or_not_C = lambda A, B, C: B or not C

# Combine the components to define KB
KB = lambda A, B, C: A_or_C(A, B, C) and B_or_not_C(A, B, C)

# Define the Query (alpha)
query = lambda A, B, C: A or B

# Function to print the truth tables
def print_truth_tables(symbols, A_or_C, B_or_not_C, KB, query):
    # Full truth table
    print(f"{'A':<6}{'B':<6}{'C':<6}{'A∨C':<8}{'B∨¬C':<8}{'KB':<8}{'α (A∨B)':<8}")
    print("-" * 56)

    # List to store combinations where both KB and α are true
    both_true = []

    # Generate all possible truth assignments for symbols
    for values in itertools.product([False, True], repeat=len(symbols)):
        # Create a dictionary for the current truth assignment
        assignment = dict(zip(symbols, values))

        # Evaluate each part of the table based on the current assignment
        A_val = assignment['A']
        B_val = assignment['B']
        C_val = assignment['C']
        A_or_C_val = A_or_C(A_val, B_val, C_val)
        B_or_not_C_val = B_or_not_C(A_val, B_val, C_val)
        KB_val = KB(A_val, B_val, C_val)
        query_val = query(A_val, B_val, C_val)

        # Print each row of the truth table
        print(f"{str(A_val):<6}{str(B_val):<6}{str(C_val):<6}"
              f"{str(A_or_C_val):<8}{str(B_or_not_C_val):<8}"
              f"{str(KB_val):<8}{str(query_val):<8}")

        # Store combinations where both KB and α are true
        if KB_val and query_val:
            both_true.append(assignment)

    # Table for combinations where both KB and α are true
    print("\nCombinations where both KB and α (A∨B) are true:")
```

```
    print(f"{'A':<6}{'B':<6}{'C':<6}")
    print("-" * 18)
    for assignment in both_true:
        print(f"{assignment['A']:<6}{assignment['B']:<6}{assignment['C']:<6}")

# Run the function to print the truth tables
print_truth_tables(symbols, A_or_C, B_or_not_C, KB, query)
```

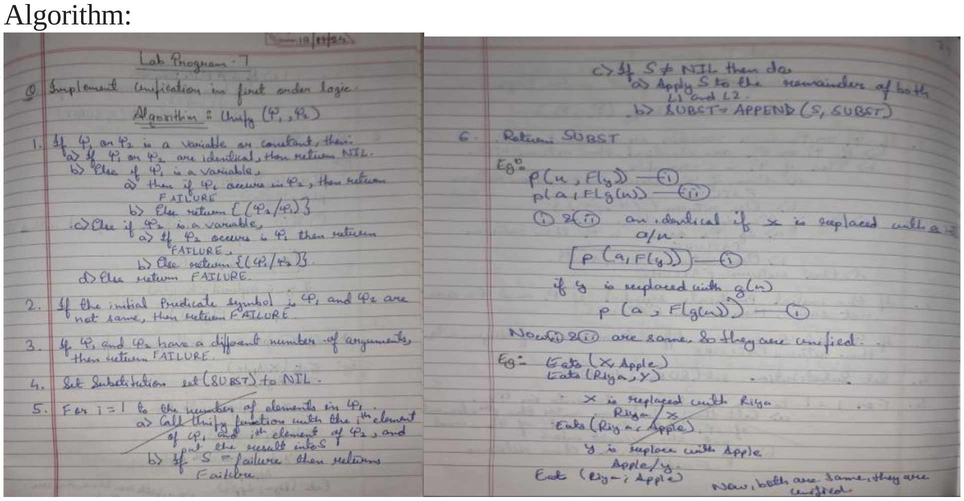
OUTPUT:

```
A     B     C     AVC     BV¬C    KB      α (AVB)
--------------------------------------------------------
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

Combinations where both KB and α (AVB) are true:
A     B     C
------------------
0     1     1
1     0     0
1     1     0
1     1     1
```

Program 7

Implement unification in first order logic.

Algorithm:



19/11/24

Lab Program - 7

Q Implement unification in first order logic.

Algorithm : Unify (Ψ_1, Ψ_2)

1. If Ψ_1 or Ψ_2 is a variable or constant, then:
   a) If Ψ_1 or Ψ_2 are identical, then return NIL.
   b) Else if Ψ_1 is a variable,
      a) then if Ψ_1 occurs in Ψ_2, then return FAILURE
      b) Else return {(Ψ_2/Ψ_1)}
   c) Else if Ψ_2 is a variable,
      a) If Ψ_2 occurs in Ψ_1 then return FAILURE.
      b) Else return {(Ψ_1/Ψ_2)}.
   d) Else return FAILURE.
2. If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.
3. If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.
4. Set Substitution set (SUBST) to NIL.
5. For i = 1 to the number of elements in Ψ_1.
   a) Call Unify function with the i^{th} element of Ψ_1 and i^{th} element of Ψ_2, and put the result into S
   b) If S = failure then returns Failure
   c) If S ≠ NIL then do
      a) Apply S to the remainder of both L1 and L2.
      b) SUBST = APPEND (S, SUBST)
6. Return SUBST

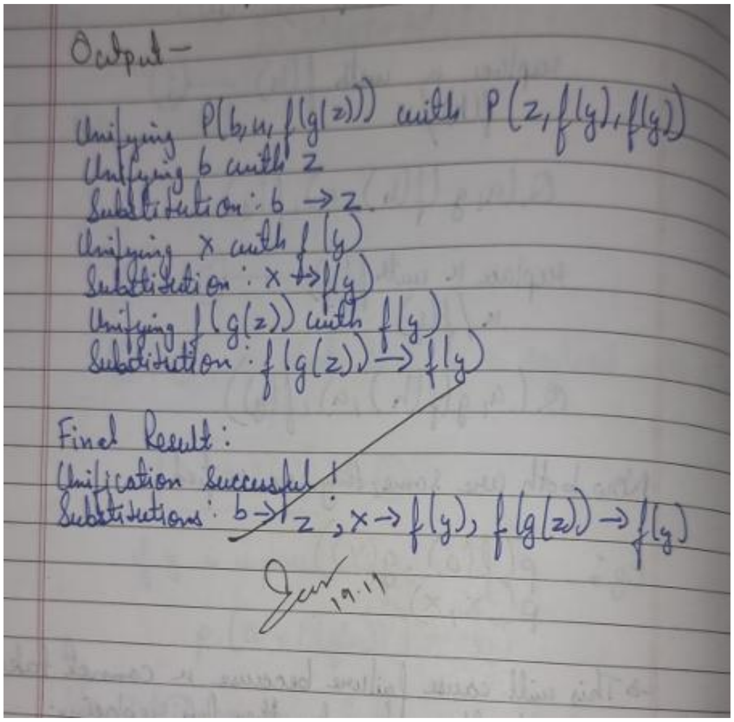
Eg:
p(x, F(y)) — (i)
p(a, F(g(x))) — (ii)

(i) & (ii) are identical if x is replaced with a
a/x
$\boxed{p(a, F(y))}$ — (i)
if y is replaced with g(x)
p(a, F(g(x))) — (i)
Now (i) & (ii) are same. So they are unified.

Eg:- Eats (X, Apple)
Eats (Riya, Y)

X is replaced with Riya
Riya / X
Eats (Riya, Apple)

Y is replace with Apple
Apple/Y.
Eat (Riya, Apple)   Now, both are same, they are unified.


Output –
Unifying P(b,x,f(g(z))) with P(z,f(y),f(y))
Unifying b with z
Substitution: b → z
Unifying x with f(y)
Substitution: x → f(y)
Unifying f(g(z)) with f(y)
Substitution: f(g(z)) → f(y)

Final Result:
Unification Successful!
Substitutions: b → z, x → f(y), f(g(z)) → f(y)

19.11

CODE:

```
#unification first order logic

def unify(expr1, expr2):
    """
    Perform unification of two logical expressions.
    """
    print(f"Unifying {expr1} with {expr2}")
    if expr1 == expr2:
        print("Result: Identical terms, no substitution needed.")
        return []  # Return NIL if expressions are identical
    elif is_variable(expr1):
        return failure_if_occurs_check(expr1, expr2)
    elif is_variable(expr2):
        return failure_if_occurs_check(expr2, expr1)
    elif is_compound(expr1) and is_compound(expr2):
        if get_predicate(expr1) != get_predicate(expr2):
            print("Failure: Predicates do not match.")
            return "FAILURE"
        return unify_args(get_arguments(expr1), get_arguments(expr2))
    else:
        print("Failure: Incompatible terms.")
        return "FAILURE"

def unify_args(args1, args2):
    """
    Unify two lists of arguments.
    """
```

```
    if len(args1) != len(args2):
        print("Failure: Arguments have different lengths.")
        return "FAILURE"
    subst = []
    for a1, a2 in zip(args1, args2):
        s = unify(a1, a2)
        if s == "FAILURE":
            print(f"Failure: Could not unify {a1} with {a2}.")
            return "FAILURE"
        if s:
            subst.extend(s)
            args1 = apply_substitution(s, args1)
            args2 = apply_substitution(s, args2)
    return subst

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

def is_compound(expression):
    return isinstance(expression, str) and "(" in expression and ")" in expression

def get_predicate(expression):
    return expression.split("(")[0]

def get_arguments(expression):
    args_str = expression[expression.index("(") + 1 : expression.rindex(")")]
    return [arg.strip() for arg in args_str.split(",")]

def failure_if_occurs_check(variable, expression):
    if occurs_check(variable, expression):
        print(f"Failure: Occurs check failed for {variable} in {expression}.")
        return "FAILURE"
    print(f"Substitution: {variable} -> {expression}")
    return [(variable, expression)]

def occurs_check(variable, expression):
    if variable == expression:
        return True
    if is_compound(expression):
        return variable in get_arguments(expression)
    return False

def apply_substitution(subst, expression):
    if isinstance(expression, list):
        return [apply_substitution(subst, sub_expr) for sub_expr in expression]
    elif is_variable(expression):
        for var, value in subst:
            if expression == var:
```

```
            return value
    elif is_compound(expression):
        predicate = get_predicate(expression)
        arguments = get_arguments(expression)
        substituted_args = [apply_substitution(subst, arg) for arg in arguments]
        return f"{predicate}({', '.join(substituted_args)})"
    return expression

# Example usage:
expr1 = "P(f(a),g(Y))"
expr2 = "P(X,X)"

result = unify(expr1, expr2)

print("\nFinal Result:")
if result == "FAILURE":
    print("Unification failed!")
else:
    print("Unification successful!")
    print("Substitutions:", ', '.join(f"{var} -> {val}" for var, val in result))
```

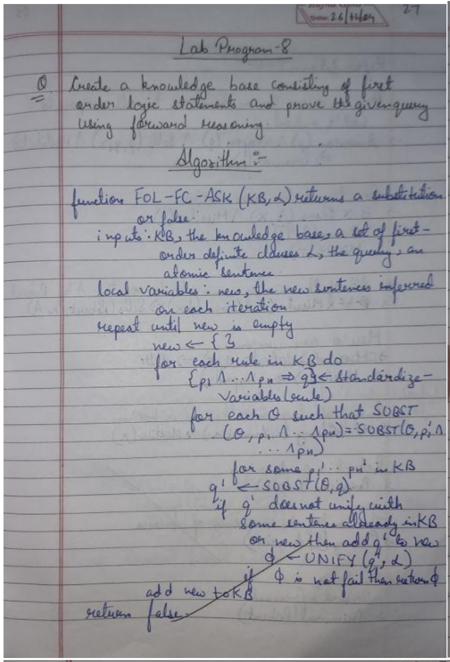
OUTPUT:

```
Unifying P(f(a),g(Y)) with P(X,X)
Unifying f(a) with X
Substitution: f(a) -> X
Unifying g(Y) with X
Failure: Incompatible terms.
Failure: Could not unify g(Y) with X.

Final Result:
Unification failed!
```

Program 8

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

Algorithm:



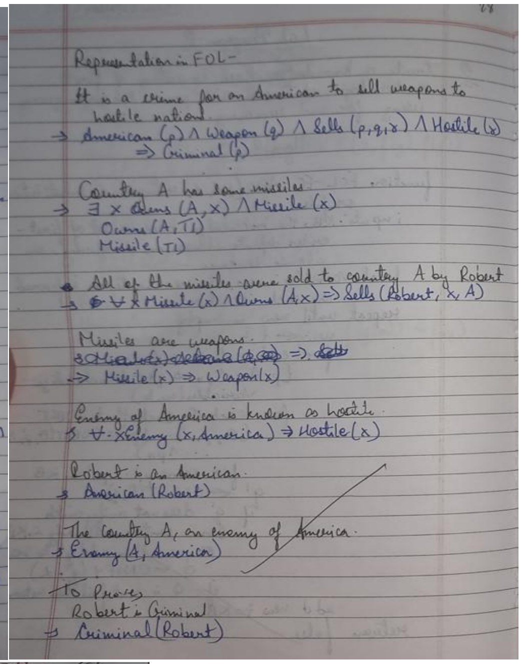
26/11/24 27

Lab Program-8

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

Algorithm :-

function FOL-FC-ASK (KB, α) returns a substitution or false
inputs: KB, the knowledge base, a set of first-order definite clauses α, the query, an atomic sentence
local variables: new, the new sentences inferred on each iteration
repeat until new is empty
new ← { }
for each rule in KB do
{p1 ∧ … ∧ pn ⇒ q} ← Standardize-Variables(rule)
for each θ such that SUBST(θ, p1 ∧ … ∧ pn) = SUBST(θ, p1' ∧ … ∧ pn')
for some p1' … pn' in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then add q' to new
φ ← UNIFY(q', α)
if φ is not fail then return φ
add new to KB
return false



28

Representation in FOL-

It is a crime for an American to sell weapons to hostile nation
→ American(p) ∧ Weapon(q) ∧ Sells(p,q,r) ∧ Hostile(r) ⇒ Criminal(p)

Country A has some missiles
→ ∃ x Owns(A, x) ∧ Missile(x)
Owns(A, T1)
Missile(T1)

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

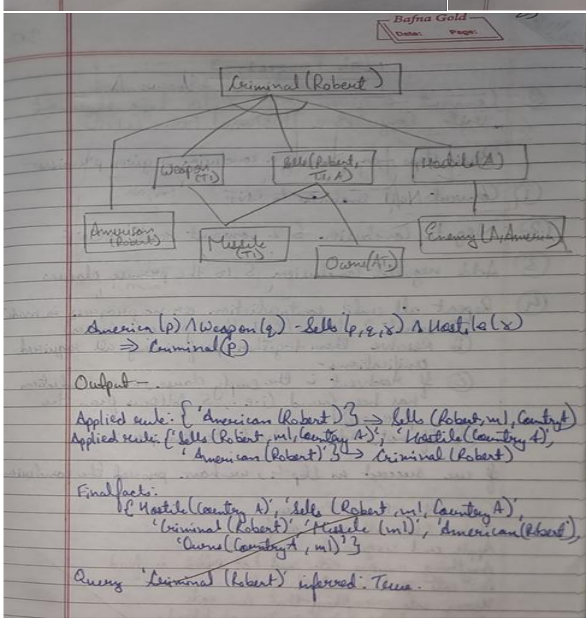
Missiles are weapons.
→ Missile(x) ⇒ Weapon(x)

Enemy of America is known as hostile.
→ ∀ x Enemy(x, America) ⇒ Hostile(x)

Robert is an American.
→ American(Robert)

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

To Prove,
Robert is Criminal
→ Criminal(Robert)



Bafna Gold
Date: Page:

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

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

Output -

Applied rule: {'American(Robert)'} → Sells(Robert, m1, CountryA)
Applied rule: {'Sells(Robert, m1, Country A)', 'Hostile(Country A)', 'American(Robert)'} → Criminal(Robert)

Final facts:
{'Hostile(Country A)', 'Sells(Robert, m1, Country A)', 'Criminal(Robert)', 'Missile(m1)', 'American(Robert)', 'Owns(Country A, m1)'}

Query 'Criminal(Robert)' inferred: True.

CODE:

```
#ForwardReasoning
class ForwardReasoning:
    def __init__(self, rules, facts):
        """
        Initializes the ForwardReasoning system.

        Parameters:
            rules (list): List of rules as tuples (condition, result),
```

```
                 where 'condition' is a set of facts.
            facts (set): Set of initial known facts.
        """
        self.rules = rules  # List of rules (condition -> result)
        self.facts = set(facts)  # Known facts

    def infer(self):
        """
        Applies forward reasoning to infer new facts based on rules and initial facts.

        Returns:
            set: Final set of facts after reasoning.
        """
        applied_rules = True

        while applied_rules:
            applied_rules = False
            for condition, result in self.rules:
                if condition.issubset(self.facts) and result not in self.facts:
                    self.facts.add(result)
                    applied_rules = True
                    print(f"Applied rule: {condition} -> {result}")
        return self.facts

# Define rules as (condition, result) where condition is a set
rules = [
        "American(Robert)",
    "Hostile(CountryA)",
    "Missile(m1)",
    "Owns(CountryA, m1)",
    "Owns(CountryA, m) ^ Missile(m) => Sells(Robert, m, CountryA)",
    "American(x) ^ Hostile(y) ^ Sells(x, z, y) => Criminal(x)"

]

# Define initial facts
facts = {"A", "D"}

# Initialize and run forward reasoning
reasoner = ForwardReasoning(rules, facts)
final_facts = reasoner.infer()

print("\nFinal facts:")
print(final_facts)
```

OUTPUT:

```
Applied rule: {'American(Robert)'} -> Sells(Robert, m1, CountryA)
Applied rule: {'Sells(Robert, m1, CountryA)', 'Hostile(CountryA)', 'American(Robert)'} -> Criminal(Robert)

Final facts:
{'Hostile(CountryA)', 'Sells(Robert, m1, CountryA)', 'Criminal(Robert)', 'Missile(m1)', 'American(Robert)', 'Owns(CountryA, m1)'}

Query 'Criminal(Robert)' inferred: True
```

Program 9

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

30

Lab Program-9

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

Basic steps for proving a conclusion S given premises—

Convert all sentences to CNF

Negate Conclusion S & convert result to CNF.

Add negated conclusion S to the premise clauses

Repeat all until contradiction or no progress is made.

(a) Select 2 clauses (call them parent clauses)
(b) Resolve them together, performing all required unifications.
(c) If resolvent is the empty clause, a contradiction has been found (i.e., S follows from the premises)
(d) If not, add resolvent to the premises.

If we succeed in Step-4, we have proved the conclusion.

Give the KB or Premises:
John likes all kind of food.
Apple and vegetables are food.
Anything anyone eats and not killed is food.
Anil eats peanuts and still alive.
Harry eats everything that Anil eats.
Anyone who is alive implies not killed.
Anyone who is not killed implies alive.
Prove John likes peanuts by resolution.

Date: Page:

(1) FOL-

(a) $\forall x: food(x) \rightarrow likes(John, x)$

(b) $food(Apple) \land food(Vegetables)$

(c) $\forall x \forall y\ eats(x,y) \land \neg killed(x) \rightarrow food(y)$

(d) $eats(Anil, peanuts) \land alive(Anil)$

(e) $\forall x: eats(Anil, x) \rightarrow eats(Harry, x)$

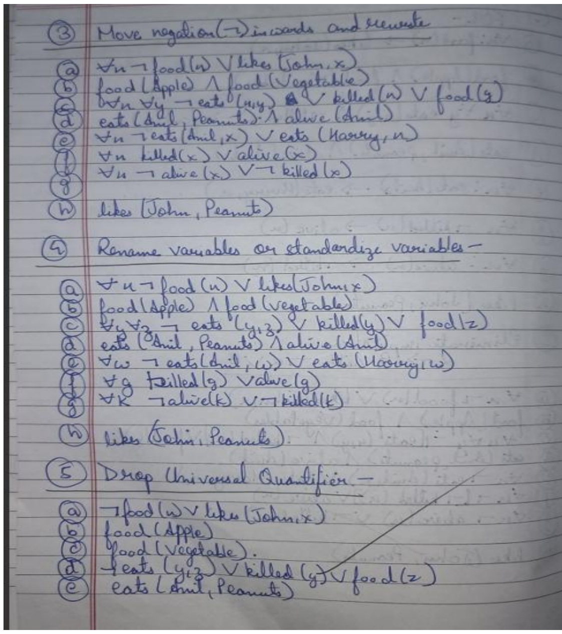
(f) $\forall x\ \neg killed(x) \rightarrow alive(x)$

(g) $\forall x: alive(x) \rightarrow \neg killed(x)$

(h) $likes(John, Peanuts)$

(2) Eliminate implication
$\alpha \rightarrow \beta$ with $\neg\alpha \lor \beta$

(a) $\forall x \neg food(x) \lor likes(John, x)$
(b) $food(Apple) \land food(Vegetables)$
(c) $\forall x \forall y\ \neg[eats(x,y) \land \neg killed(x)] \lor food(y)$
(d) $eats(Anil, peanuts) \land alive(Anil)$
(e) $\forall x \neg eats(Anil, x) \lor eats(Harry, x)$
(f) $\forall x \neg[\neg killed(x)] \lor alive(x)$
(g) $\forall x \neg alive(x) \lor \neg killed(x)$
(h) $likes(John, Peanuts)$



3. Move negation (¬) inwards and rewrite

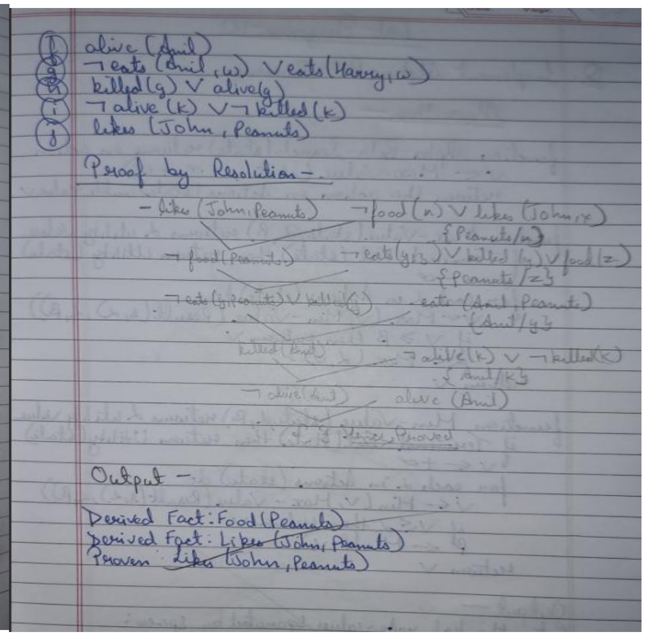
a. $\forall x\ \neg food(x) \lor likes(John, x)$
b. $food(Apple) \land food(Vegetable)$
c. $\forall x \forall y\ \neg eats(x, y) \lor killed(x) \lor food(y)$
d. $eats(Anil, Peanuts) \land alive(Anil)$
e. $\forall x\ \neg eats(Anil, x) \lor eats(Harry, x)$
f. $\forall x\ killed(x) \lor alive(x)$
g. $\forall x\ \neg alive(x) \lor \neg killed(x)$
h. $likes(John, Peanuts)$

4. Rename variables or standardize variables -

a. $\forall x\ \neg food(x) \lor likes(John, x)$
b. $food(Apple) \land food(vegetable)$
c. $\forall y \forall z\ \neg eats(y, z) \lor killed(y) \lor food(z)$
d. $eats(Anil, Peanuts) \land alive(Anil)$
e. $\forall w\ \neg eats(Anil, w) \lor eats(Harry, w)$
f. $\forall g\ killed(g) \lor alive(g)$
g. $\forall k\ \neg alive(k) \lor \neg killed(k)$
h. $likes(John, Peanuts)$.

5. Drop Universal Quantifier -

a. $\neg food(x) \lor likes(John, x)$
b. $food(Apple)$
c. $food(Vegetable)$.
d. $\neg eats(y, z) \lor killed(y) \lor food(z)$
e. $eats(Anil, Peanuts)$



f. $alive(Anil)$
g. $\neg eats(Anil, w) \lor eats(Harry, w)$
h. $killed(g) \lor alive(g)$
i. $\neg alive(k) \lor \neg killed(k)$
j. $likes(John, Peanuts)$

Proof by Resolution -

$\neg likes(John, Peanuts)$ $\neg food(x) \lor likes(John, x)$ {Peanuts/x}
$\neg food(Peanuts)$ $\neg eats(y, z) \lor killed(y) \lor food(z)$ {Peanuts/z}
$\neg eats(y, Peanuts) \lor killed(y)$ $eats(Anil, Peanuts)$ {Anil/y}
$killed(Anil)$ $\neg alive(k) \lor \neg killed(k)$ {Anil/k}
$\neg alive(Anil)$ $alive(Anil)$
{} Hence, Proved

Output -

Derived Fact: Food(Peanuts)
Derived Fact: Likes(John, Peanuts)
Proven Likes(John, Peanuts)

CODE:

```
#FOL to CNF
# Knowledge Base (KB)
facts = {
    "Eats(Anil, Peanuts)": True,
    "not Killed(Anil)": True,
    "Food(Apple)": True,
    "Food(Vegetables)": True,
}

rules = [
    # Rule: Food(X) :- Eats(Y, X) and not Killed(Y)
    {"conditions": ["Eats(Y, X)", "not Killed(Y)"], "conclusion": "Food(X)"},
    # Rule: Likes(John, X) :- Food(X)
    {"conditions": ["Food(X)"], "conclusion": "Likes(John, X)"},
]

# Query
query = "Likes(John, Peanuts)"

# Helper function to substitute variables in a rule
def substitute(rule_part, substitutions):
    for var, value in substitutions.items():
        rule_part = rule_part.replace(var, value)
    return rule_part

# Function to resolve the query
def resolve_query(facts, rules, query):
    working_facts = facts.copy()
    while True:
        new_facts_added = False
```

```
        for rule in rules:
            conditions = rule["conditions"]
            conclusion = rule["conclusion"]

            # Try all substitutions for variables (X, Y) in the rules
            for entity in ["Apple", "Vegetables", "Peanuts", "Anil", "John"]:
                substitutions = {"X": "Peanuts", "Y": "Anil"}  # Fixed for this problem
                resolved_conditions = [substitute(cond, substitutions) for cond in conditions]
                resolved_conclusion = substitute(conclusion, substitutions)

                # Check if all conditions are true
                if all(working_facts.get(cond, False) for cond in resolved_conditions):
                    if resolved_conclusion not in working_facts:
                        working_facts[resolved_conclusion] = True
                        new_facts_added = True
                        print(f"Derived Fact: {resolved_conclusion}")

        if not new_facts_added:
            break

    # Check if the query is resolved
    return working_facts.get(query, False)

# Run the resolution process
if resolve_query(facts, rules, query):
    print(f"Proven: {query}")
else:
    print(f"Not Proven: {query}")
```

OUTPUT:

```
Derived Fact: Food(Peanuts)
Derived Fact: Likes(John, Peanuts)
Proven: Likes(John, Peanuts)
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

3/12/24 34

Lab-Program-10

Q. Implement Alpha-Beta Pruning.

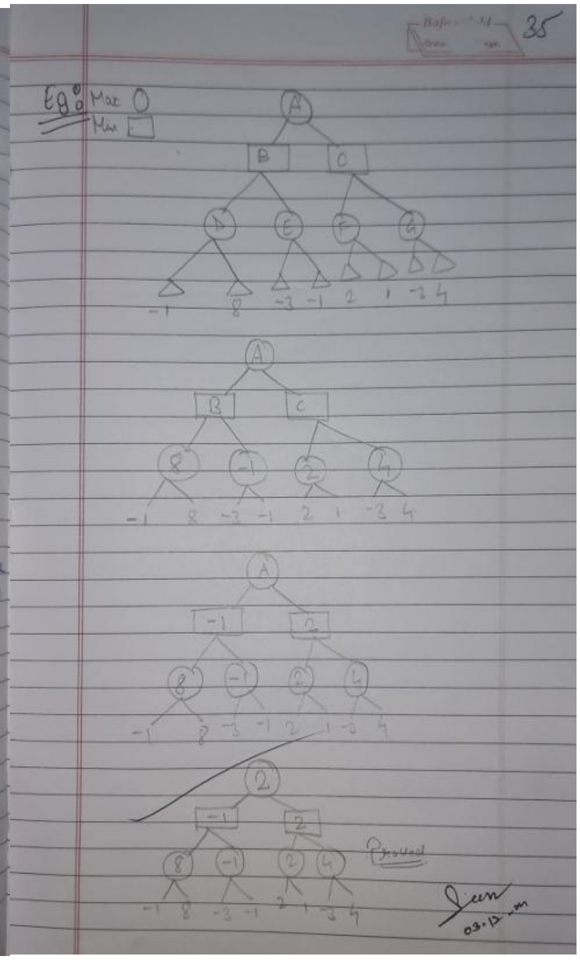
Algorithm—

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(s, 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(s, a), α, β))
if v ≤ α then return v
β ← Min(β, v)
return v

Output—
Enter the leaf node values separated by spaces:
-1 8 -3 -1 2 1 -3 4
Optimal value Calculated using Minimax: 2



CODE:

```
#Alpha-Beta Pruning
import math

def minimax(depth, index, maximizing_player, values, alpha, beta):
    # Base case: when we've reached the leaf nodes
    if depth == 0:
        return values[index]

    if maximizing_player:
        max_eval = float('-inf')
        for i in range(2):  # 2 children per node
            eval = minimax(depth - 1, index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:  # Beta cutoff
                break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):  # 2 children per node
```

```
            eval = minimax(depth - 1, index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:  # Alpha cutoff
                break
        return min_eval

# Accept values from the user
leaf_values = list(map(int, input("Enter the leaf node values separated by spaces: ").split()))

# Check if the number of values is a power of 2
if math.log2(len(leaf_values)) % 1 != 0:
    print("Error: The number of leaf nodes must be a power of 2 (e.g., 2, 4, 8, 16).")
else:
    # Calculate depth of the tree
    tree_depth = int(math.log2(len(leaf_values)))

    # Run Minimax with Alpha-Beta Pruning
    optimal_value = minimax(depth=tree_depth, index=0, maximizing_player=True, values=leaf_values,
alpha=float('-inf'), beta=float('inf'))

    print("Optimal value calculated using Minimax:",optimal_value)
```

OUTPUT:

```
Enter the leaf node values separated by spaces: -1 8 -3 -1 2 1 -3 4
Optimal value calculated using Minimax: 2
```