VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Artificial Intelligence (23CS5PCAIN)

Submitted by

TANMAY BHARADWAJ (1BM22CS303)

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 **Tanmay Bharadwaj (1BM22CS303),** 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 of the above mentioned subject and the work prescribed for the said degree.

Prof. Prameetha Pai Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

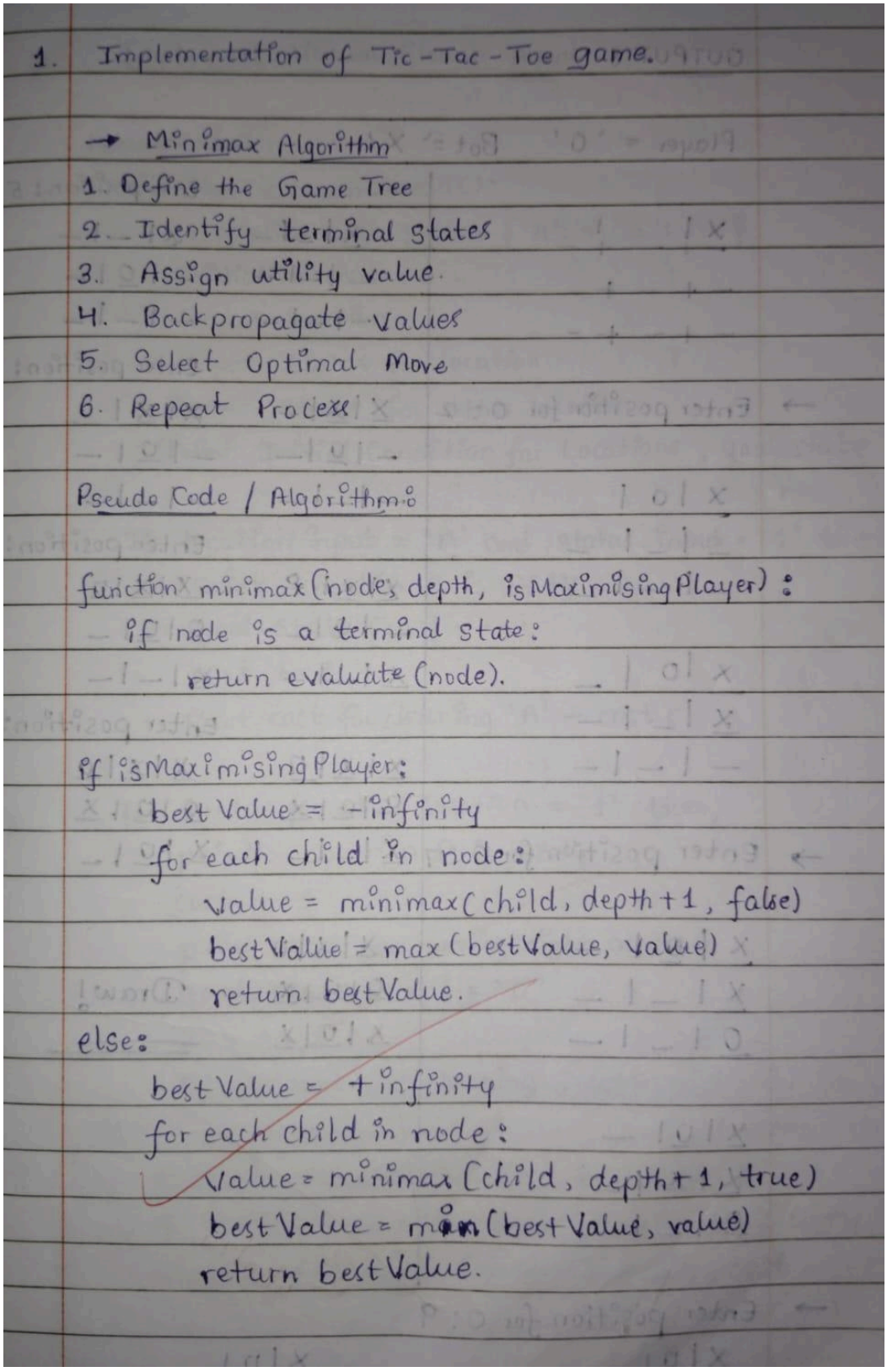
Index

Github Link:
https://github.com/TanmayBj23/AI_LAB

Program 1:

Implement Tic –Tac –Toe Game.

Algorithm/Pseudo Code:



1. Implementation of Tic-Tac-Toe game.

→ Minimax Algorithm

1. Define the Game Tree
2. Identify terminal states
3. Assign utility value.
4. Backpropagate values
5. Select Optimal Move
6. Repeat Process

Pseudo Code / Algorithm:

```
function minimax(node, depth, isMaximisingPlayer):
    if node is a terminal state:
        return evaluate(node).

if isMaximisingPlayer:
    bestValue = -infinity
    for each child in node:
        value = minimax(child, depth+1, false)
        bestValue = max(bestValue, value)
    return bestValue.
else:
    bestValue = +infinity
    for each child in node:
        value = minimax(child, depth+1, true)
        bestValue = min(bestValue, value)
    return bestValue.
```

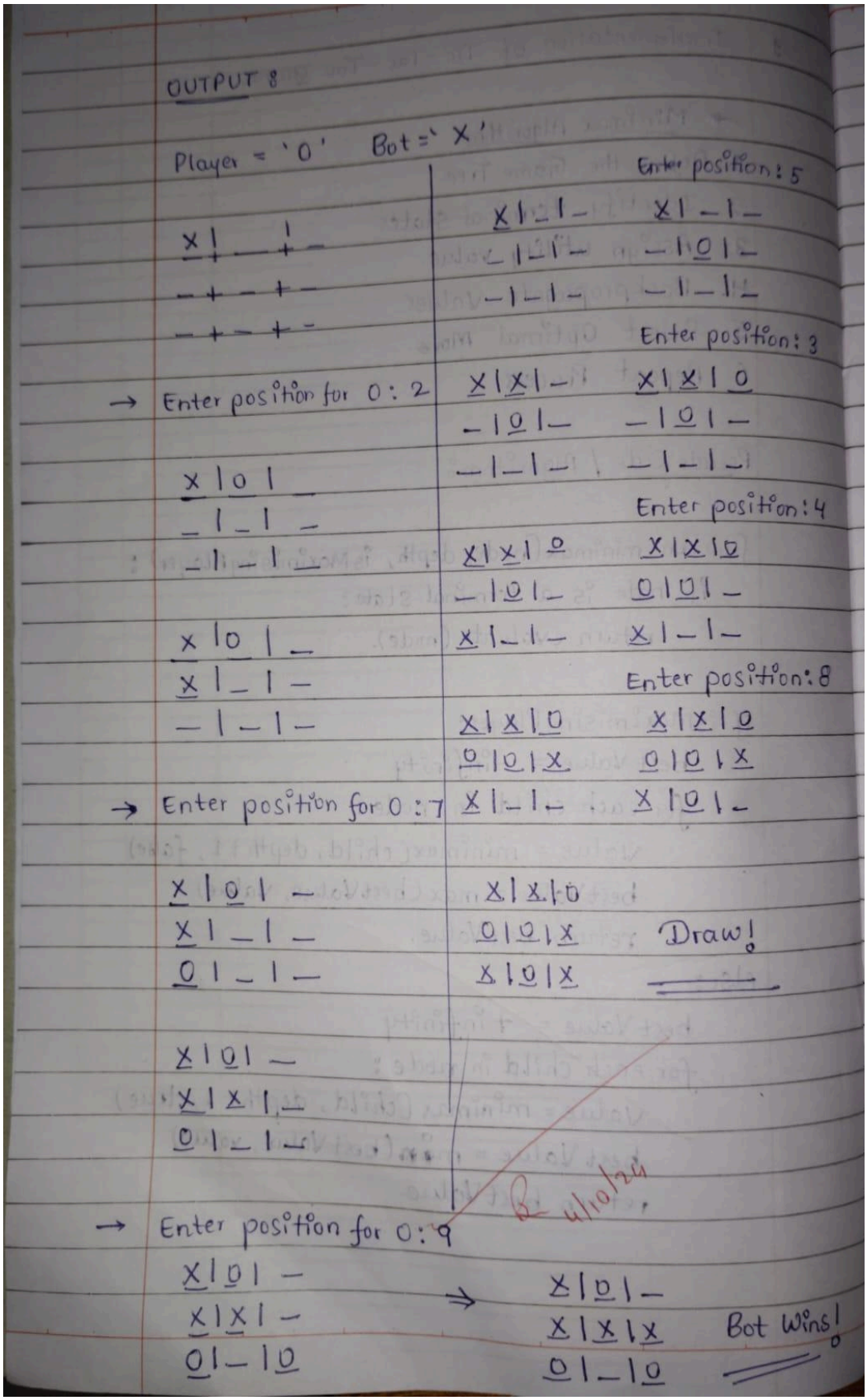
Code:

```
board={1:' ',2:' ',3:' ',
      4:' ',5:' ',6:' ',
      7:' ',8:' ',9:' '
}

def printBoard(board):
   print(board[1]+'|'+board[2]+'|'+board[3])
   print('-+-+-')
   print(board[4] + '|' + board[5] + '|' + board[6])
   print('-+-+-')
   print(board[7] + '|' + board[8] + '|' + board[9])
   print('\n')

def spaceFree(pos):
   if(board[pos]==' '):
      return True
   else:
      return False

def checkWin():
   if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
      return True
   elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
      return True
   elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
      return True
   elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
      return True
   elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
      return True
   elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
      return True
   elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
      return True
   elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
      return True
   else:
      return False

def checkMoveForWin(move):
   if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
      return True
   elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
      return True
   elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
      return True
   elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
      return True
```

```
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True


def insertLetter(letter, position):
    if (spaceFree(position)):
        board[position] = letter
        printBoard(board)

        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return

    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)
        return

player = 'O'
bot ='X'

def playerMove():
    position=int(input('Enter position for O:'))
    insertLetter(player, position)
    return

def compMove():
    bestScore=-1000
    bestMove=0
    for key in board.keys():
```

```
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, False)
            board[key] = ' '
            if (score > bestScore):
                bestScore = score
                bestMove = key

    insertLetter(bot, bestMove)
    return


def minimax(board, isMaximizing):
    if (checkMoveForWin(bot)):
        return 1
    elif (checkMoveForWin(player)):
        return -1
    elif (checkDraw()):
        return 0

    if isMaximizing:
        bestScore = -1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                if (score > bestScore):
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                if (score < bestScore):
                    bestScore = score
        return bestScore

while not checkWin():
    compMove()
    playerMove()
```

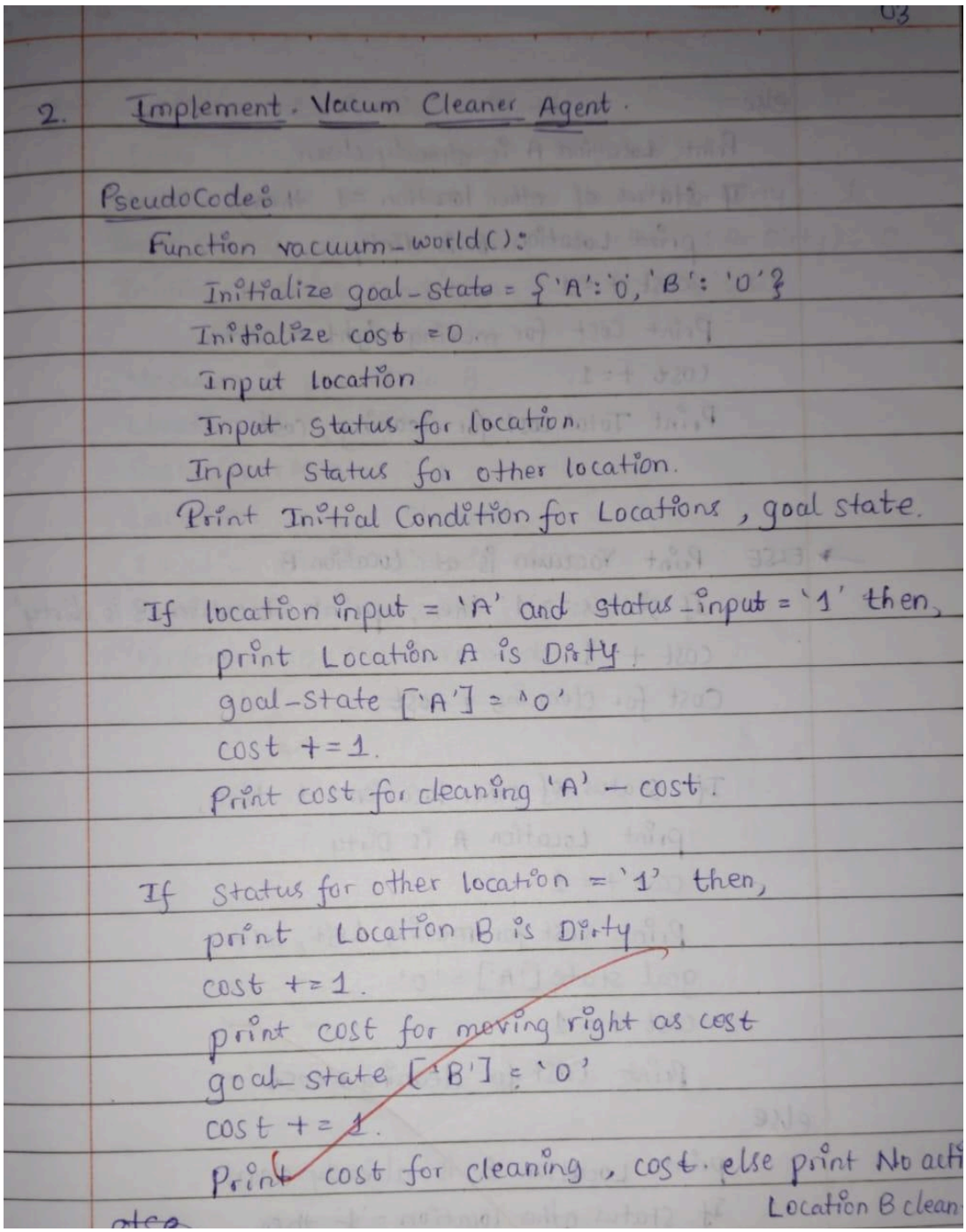
Output:


OUTPUT :
Player = '0' Bot = 'X'
Enter position: 5
Enter position: 3
Enter position for 0: 2
Enter position: 4
Enter position: 8
Enter position for 0: 7
Draw!
Enter position for 0: 9
Bot wins!

Program 2:

Implement a vacuum cleaner agent.

Algorithm/Pseudo Code:



2. Implement. Vacum Cleaner Agent.

PseudoCode:

```
Function vacuum_world():
    Initialize goal_state = {'A': '0', 'B': '0'}
    Initialize cost = 0.
    Input location
    Input status for location.
    Input status for other location.
    Print Initial Condition for Locations, goal state.

If location input = 'A' and status_input = '1' then,
    print Location A is Dirty
    goal_state ['A'] = '0'
    cost += 1.
    Print cost for cleaning 'A' — cost.

If status for other location = '1' then,
    print Location B is Dirty
    cost += 1.
    print cost for moving right as cost
    goal_state ['B'] = '0'
    cost += 1.
    Print cost for cleaning, cost. else print No acti
ates                                    Location B clean
```

else
Print Location A is already clean.
If status of other location =1 then,
print Location B is dirty
cost + = 1
Print Cost for moving right = cost.
cost += 1
Print Total cost for cleaning, cost.

→ ELSE Print Vacuum is at Location B.
If status = = '1' then, print 'Location B is dirty'
cost += 1
Cost for cleaning, cost.

If status of other location = 1 then,
print Location A is Dirty.
cost + = 1
Print Cost for moving Left, cost
goal state ['A'] = '0'
cost + = 1
Print Cost for cleaning, cost
else
print Location B is already clean.
If status other location = '1' then
Print Location A is dirty
cost + = 1.
print Cost for moving Left, cost.
goal_state ['A'] = '0'.
cost + = 1
Print Cost for clean, cost.
Print Performance Measurement, cost

Code:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input + " (0 for Clean, 1 for Dirty): ")
    status_input_complement = input("Enter status of other room (0 for Clean, 1 for Dirty): ")
    print("Initial Location Condition: " + str(goal_state))
    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1
                print("COST for moving RIGHT: " + str(cost))
                goal_state['B'] = '0'
                cost += 1
                print("COST for SUCK: " + str(cost))
                print("Location B has been Cleaned.")
            else:
                print("No action: " + str(cost))
                print("Location B is already clean.")
        if status_input == '0':
            print("Location A is already clean.")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving RIGHT to Location B.")
                cost += 1
                print("COST for moving RIGHT: " + str(cost))
                goal_state['B'] = '0'
                cost += 1
                print("Cost for SUCK: " + str(cost))
                print("Location B has been Cleaned.")
            else:
                print("No action: " + str(cost))
                print("Location B is already clean.")
    else:
        print("Vacuum is placed in Location B")
        if status_input == '1':
            print("Location B is Dirty.")
            goal_state['B'] = '0'
            cost += 1
            print("COST for CLEANING: " + str(cost))
```

```
            print("Location B has been Cleaned.")
            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to Location A.")
                cost += 1
                print("COST for moving LEFT: " + str(cost))
                goal_state['A'] = '0'
                cost += 1
                print("COST for SUCK: " + str(cost))
                print("Location A has been Cleaned.")
            else:
                print("Location A is already clean.")
        if status_input == '0':
            print("Location B is already clean.")
            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to Location A.")
                cost += 1
                print("COST for moving LEFT: " + str(cost))
                goal_state['A'] = '0'
                cost += 1
                print("Cost for SUCK: " + str(cost))
                print("Location A has been Cleaned.")
            else:
                print("No action: " + str(cost))
                print("Location A is already clean.")
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

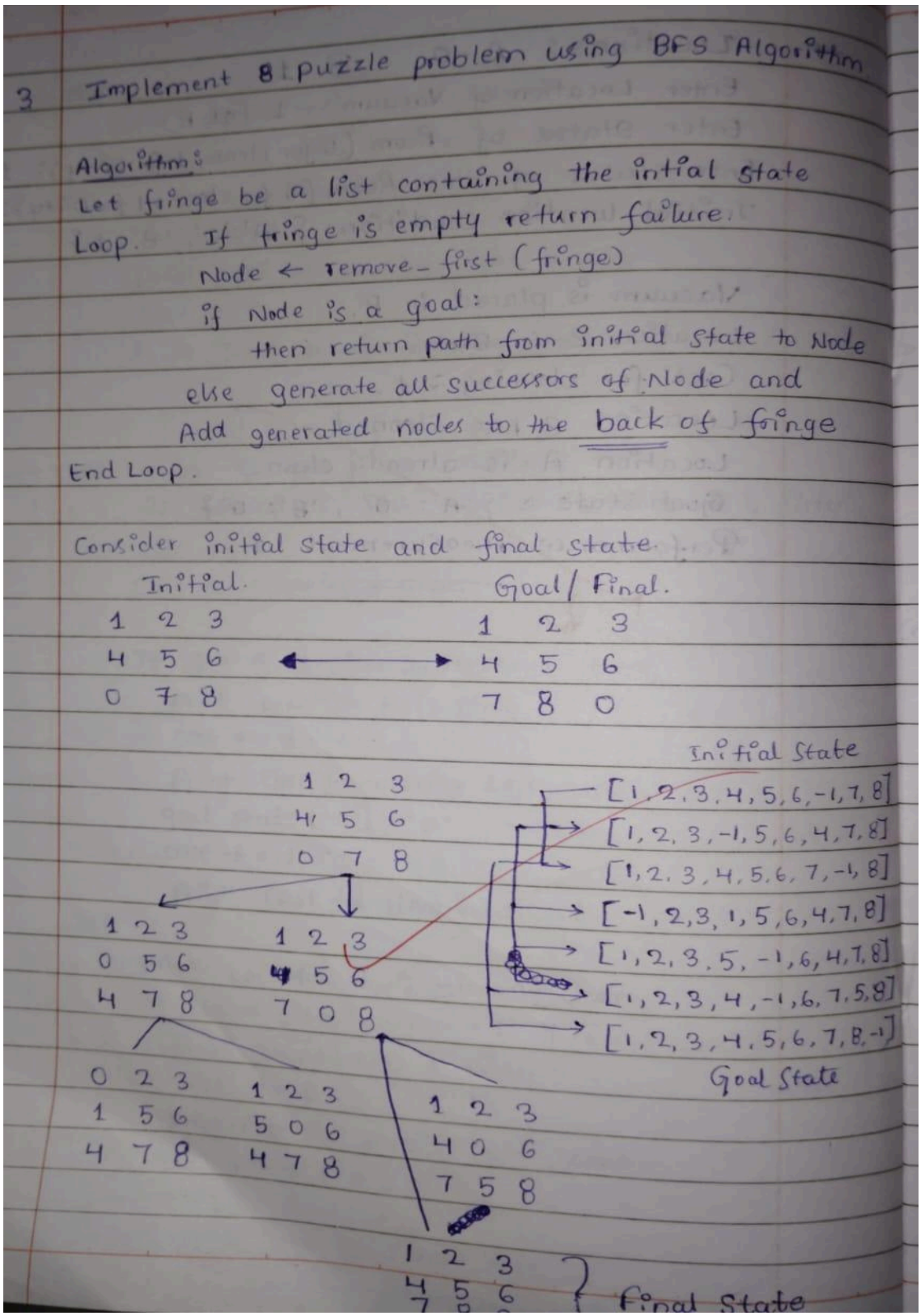
Output:

```
Enter Location of Vacuum: A
Enter status of A (0 for Clean, 1 for Dirty): 0
Enter status of other room (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is already clean.
Location B is Dirty.
Moving RIGHT to Location B.
COST for moving RIGHT: 1
Cost for SUCK: 2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

Program 3:

Solve 8 puzzle problems using BFS and DFS techniques.

Algorithm / PseudoCode:



11. Implement 8 puzzle problem using DFS algorithm.

Algorithm:

Let fringe be a list containing the initial state.
Loop If fringe is empty return failure
Node ← remove-first (fringe)
if Node is a goal:
then return path from initial state to Node.
else generate all successors of Node and
Add generated nodes to the front of fringe.
End Loop.

Initial State

1 2 3
4 5 6
0 7 8.

1 2 3
0 5 6
4 7 8

0 2 3
1 5 6
4 7 8

2 0 3
1 5 6
4 7 8

⋮

1 2 3
4 5 6
7 8 0

} Final State Success.

18/10/2024

Code:

```
#BFS Algorithm

from collections import deque
def solve_8puzzle_dfs(initial_state):
  def find_blank(state):
    for row in range(3):
      for col in range(3):
        if state[row][col] == 0:
          return row, col
  def get_neighbors(state):
    row, col = find_blank(state)
    neighbors = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
    for dr, dc in directions:
      new_row, new_col = row + dr, col + dc
      if 0 <= new_row < 3 and 0 <= new_col < 3:
        new_state = [r[:] for r in state]
        new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col],
new_state[row][col]
        neighbors.append(new_state)
    return neighbors


  goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
  stack = [(initial_state, [])]
  visited = set()
  while stack:
    current_state, path = stack.pop()
    state_tuple = tuple(map(tuple, current_state))
    if state_tuple in visited:
      continue
    visited.add(state_tuple)
    if current_state == goal_state:
      return path + [current_state]
    for neighbor in get_neighbors(current_state):
      stack.append((neighbor, path + [current_state]))
  return None

initial_state = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
solution = solve_8puzzle_dfs(initial_state)
if solution:
  print("Solution found:")
  for state in solution:
    for row in state:
      print(row)
    print()
else:
  print("No solution found.")
```

#DFS Algorithm

```
from collections import deque
def solve_8puzzle_dfs(initial_state):
  def find_blank(state):
    for row in range(3):
      for col in range(3):
        if state[row][col] == 0:
          return row, col
  def get_neighbors(state):
    row, col = find_blank(state)
    neighbors = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right
    for dr, dc in directions:
      new_row, new_col = row + dr, col + dc
      if 0 <= new_row < 3 and 0 <= new_col < 3:
          new_state = [r[:] for r in state]
          new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col],
new_state[row][col]
          neighbors.append(new_state)
    return neighbors

  goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
  stack = [(initial_state, [])]
  visited = set()
  while stack:
    current_state, path = stack.pop()
    state_tuple = tuple(map(tuple, current_state))
    if state_tuple in visited:
      continue
    visited.add(state_tuple)
    if current_state == goal_state:
      return path + [current_state]
    for neighbor in get_neighbors(current_state):
        stack.append((neighbor, path + [current_state]))
  return None

initial_state = [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
solution = solve_8puzzle_dfs(initial_state)
if solution:
  print("Solution found:")
  for state in solution:
    for row in state:
      print(row)
    print()
else:
  print("No solution found.")
```

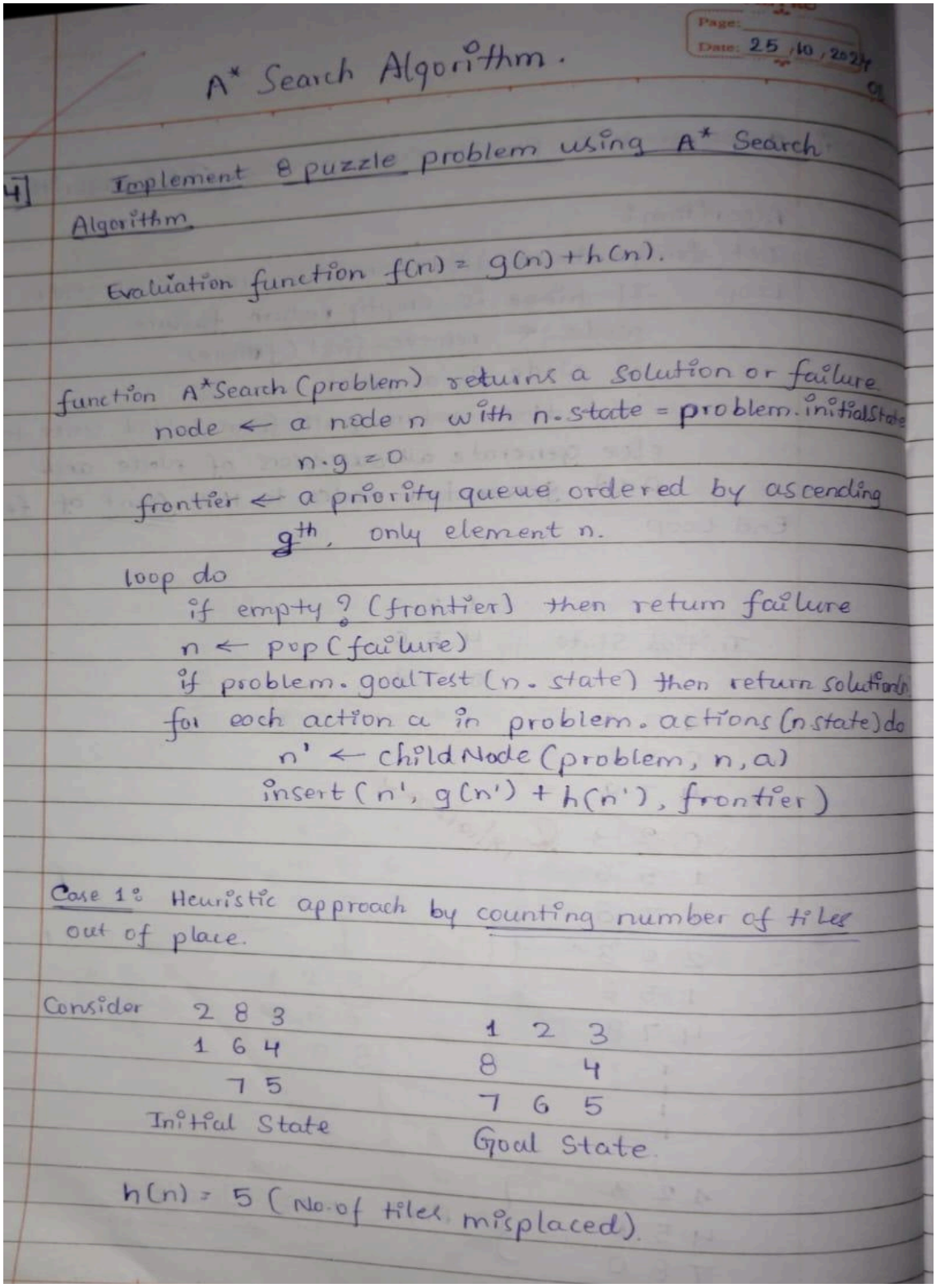
Output:

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

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

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

Program 4:

Implement A* search algorithm for 8 Puzzle Problem using heuristic approach: Misplaced tiles and Manhattan distance.

Algorithm/Pseudo Code:



A* Search Algorithm.

Page:
Date: 25/10/2024

4] Implement 8 puzzle problem using A* Search Algorithm

Evaluation function f(n) = g(n) + h(n).

function A*Search (problem) returns a solution or failure
node ← a node n with n.state = problem.initialState
n.g = 0
frontier ← a priority queue ordered by ascending g^th, only element n.
loop do
if empty? (frontier) then return failure
n ← pop (failure)
if problem.goalTest (n.state) then return solution(n)
for each action a in problem.actions (n.state) do
n' ← childNode (problem, n, a)
insert (n', g(n') + h(n'), frontier)

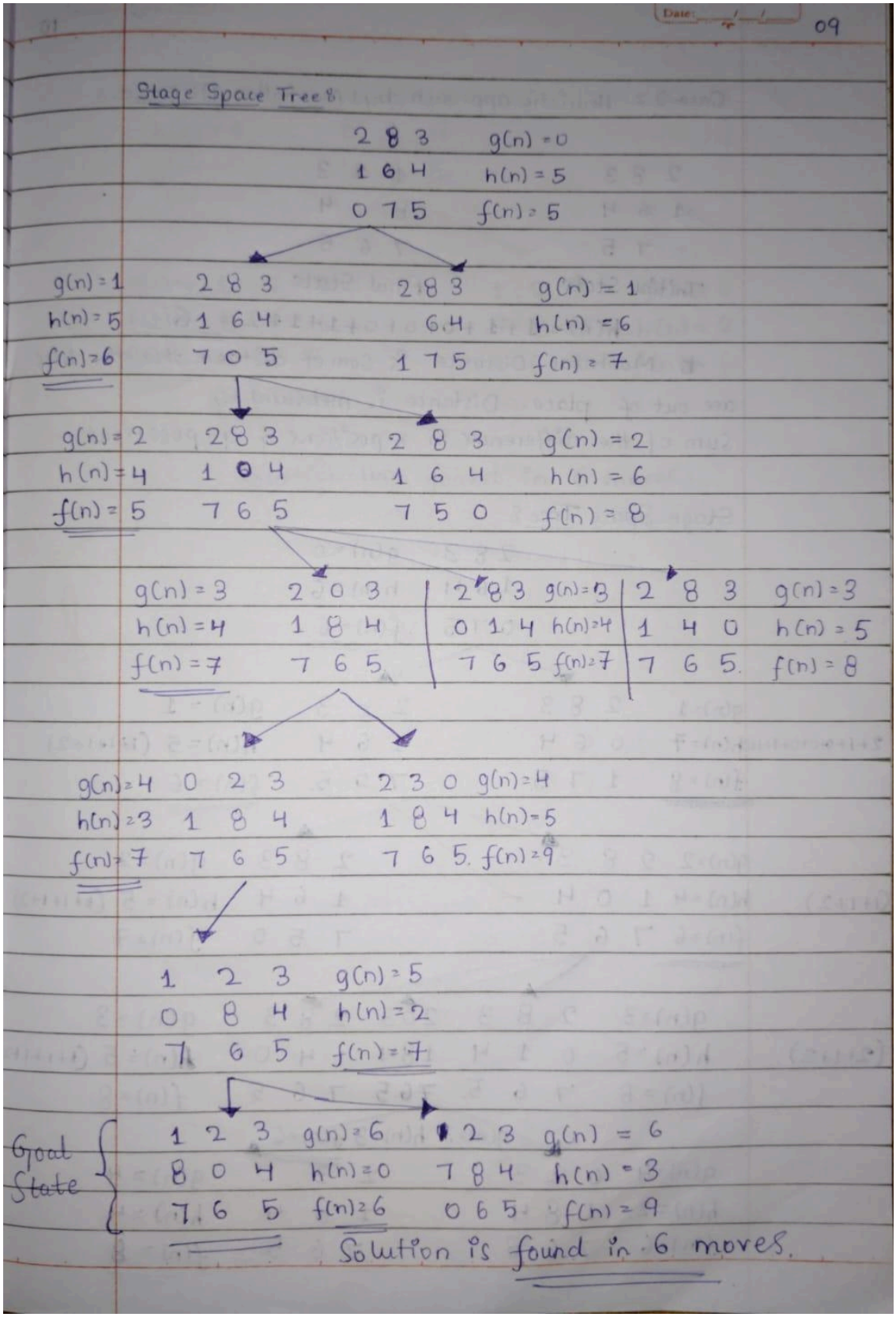
Case 1: Heuristic approach by counting number of tiles out of place.

Consider

2 8 3	1 2 3
1 6 4	8 4
7 5	7 6 5
Initial State	Goal State

h(n) = 5 (No. of tiles misplaced).

Stage Space Tree:



Solution is found in 6 moves.

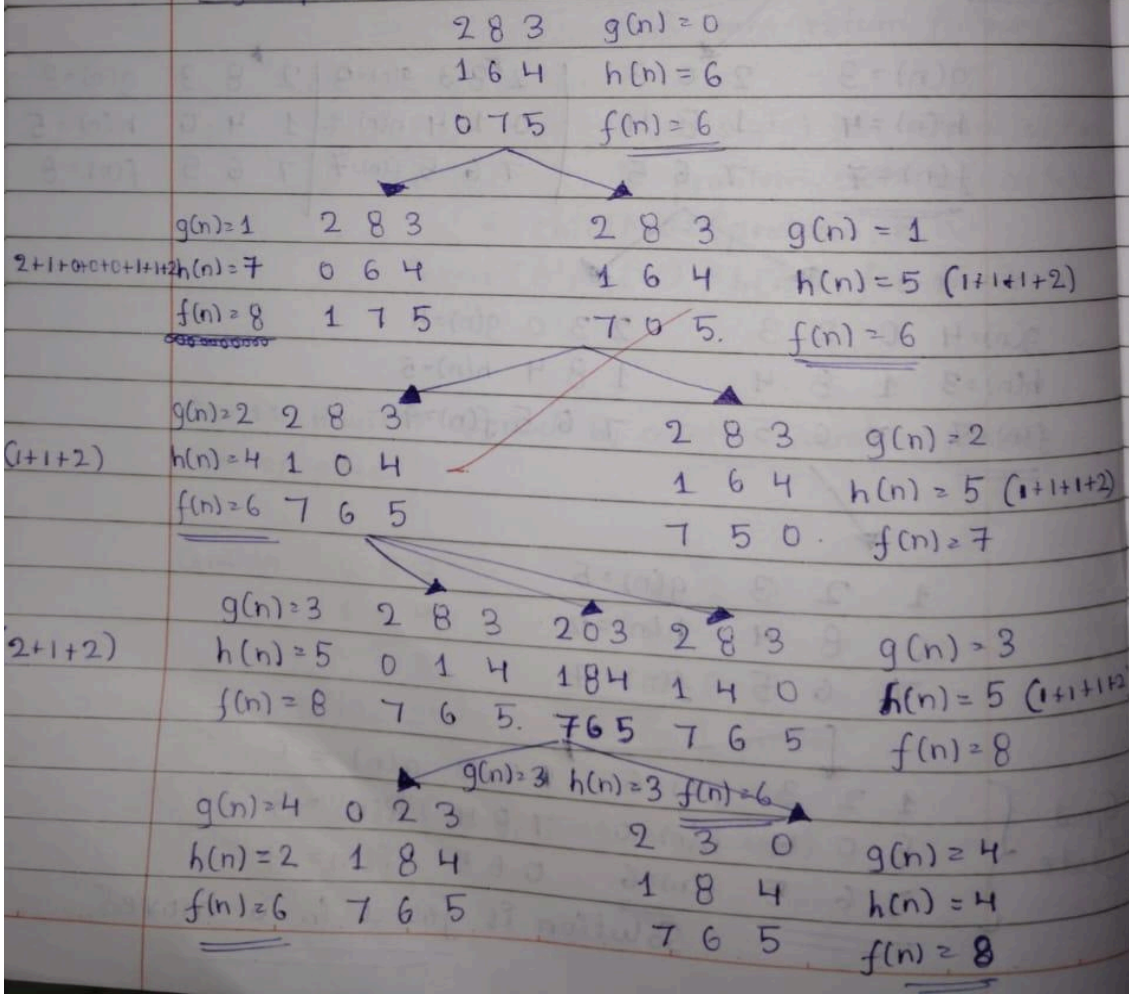
Case 2 : Heuristic approach by Manhatten Distance.

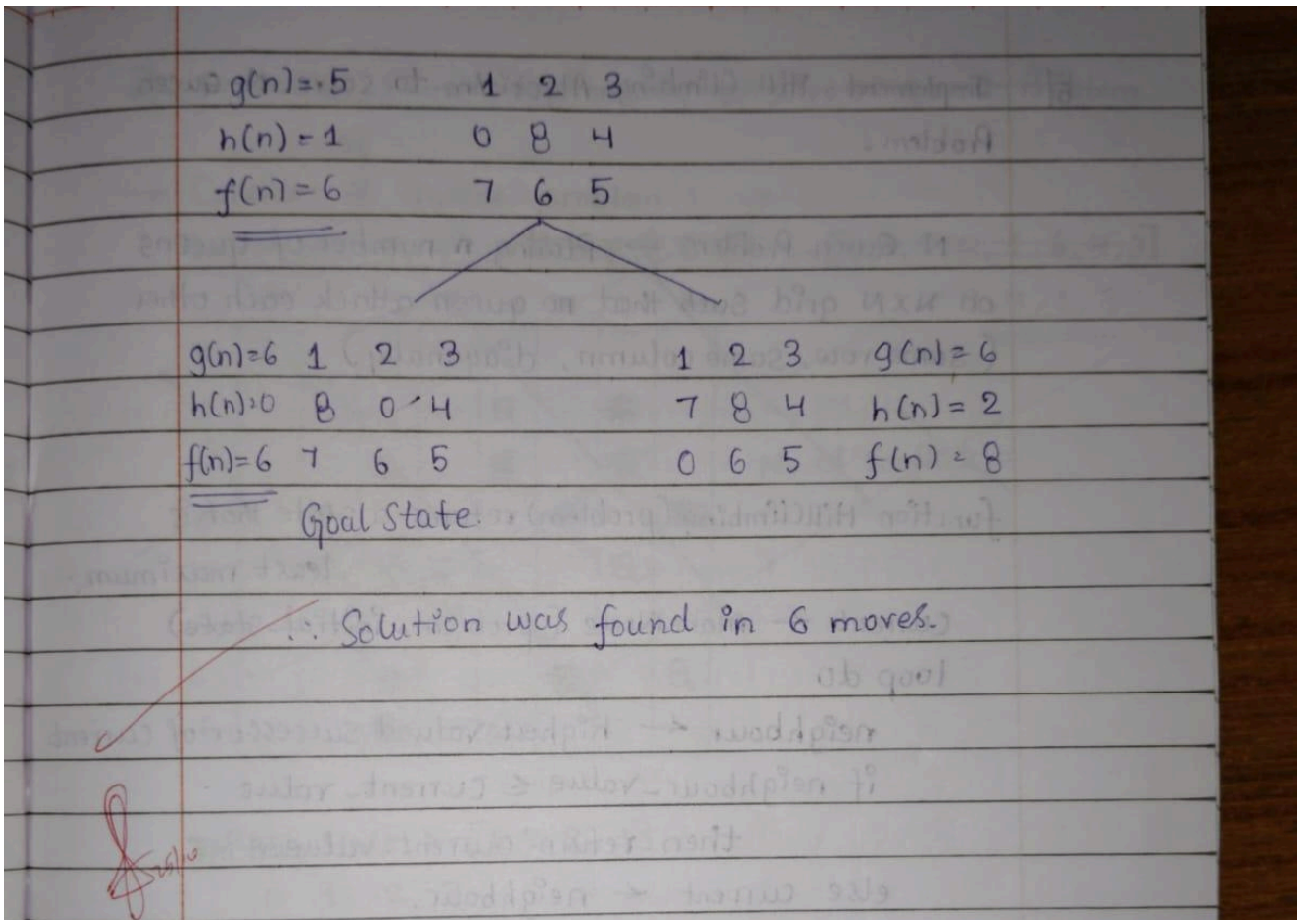
2 8 3	1 2 3
1 6 4	8 4
7 5	7 6 5
Initial State	Final State

$h(n) = 1+1+0+0+0+1+1+2 = 6$

Manhattan Distance is sum of distance that the tiles are out of place. Distance is measured by sum of the differences in x-positions & y-positions.

Stage Space Tree :





Code:

```
# Heuristic function: number of misplaced tiles
import heapq
# Define the goal state
GOAL_STATE = [[1, 2, 3],
              [8, 0, 4],
              [7, 6, 5]]

# Utility function to find the position of the blank tile
def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def h(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
```

```
# Generate possible moves (Up, Down, Left, Right)
def get_neighbors(state):
    neighbors = []
    x, y = find_blank_position(state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

# A* search algorithm
def a_star_search(initial_state):
    open_list = []
    heapq.heappush(open_list, (h(initial_state), 0, initial_state, []))  # (f(n), g(n), state, path)
    closed_set = set()
    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)
        if current_state == GOAL_STATE:
            return path + [current_state]  # Return the path to the goal
        closed_set.add(tuple(tuple(row) for row in current_state))  # Add to closed set
        for neighbor in get_neighbors(current_state):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple in closed_set:
                continue
            heapq.heappush(open_list, (g + 1 + h(neighbor), g + 1, neighbor, path + [current_state]))
    return None  # No solution

# Print the puzzle state
def print_state(state):
    for row in state:
        print(row)
    print()

# Define the initial state
initial_state = [[2, 8, 3],
                 [1, 6, 4],
                 [0, 7, 5]]

# Run the A* search
solution_path = a_star_search(initial_state)
# Print the solution path
if solution_path:
    print("Solution found in", len(solution_path)-1, "moves:")
    for step in solution_path:
        print_state(step)
else:
    print("No solution found.")
```

```
# Heuristic function: Manhattan distance
import heapq
# Define the goal state
GOAL_STATE = [[1, 2, 3],
              [8, 0, 4],
              [7, 6, 5]]
# Utility function to find the position of the blank tile
def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
def h(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_x, goal_y = divmod(state[i][j] - 1, 3)
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance
# Generate possible moves (Up, Down, Left, Right)
def get_neighbors(state):
    neighbors = []
    x, y = find_blank_position(state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(new_state)
    return neighbors
# A* search algorithm
def a_star_search(initial_state):
    open_list = []
    heapq.heappush(open_list, (h(initial_state), 0, initial_state, []))  # (f(n), g(n), state, path)
    closed_set = set()
    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)
        if current_state == GOAL_STATE:
            return path + [current_state]  # Return the path to the goal
        closed_set.add(tuple(tuple(row) for row in current_state))  # Add to closed set
        for neighbor in get_neighbors(current_state):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple in closed_set:
                continue
            heapq.heappush(open_list, (g + 1 + h(neighbor), g + 1, neighbor, path + [current_state]))
    return None  # No solution
```

```
# Print the puzzle state
def print_state(state):
    for row in state:
        print(row)
    print()
# Define the initial state
initial_state = [[2, 8, 3],
                 [1, 6, 4],
                 [0, 7, 5]]
# Run the A* search
solution_path = a_star_search(initial_state)
# Print the solution path
if solution_path:
    print("Solution found in", len(solution_path)-1, "moves:")
    for step in solution_path:
        print_state(step)
else:
    print("No solution found.")
```

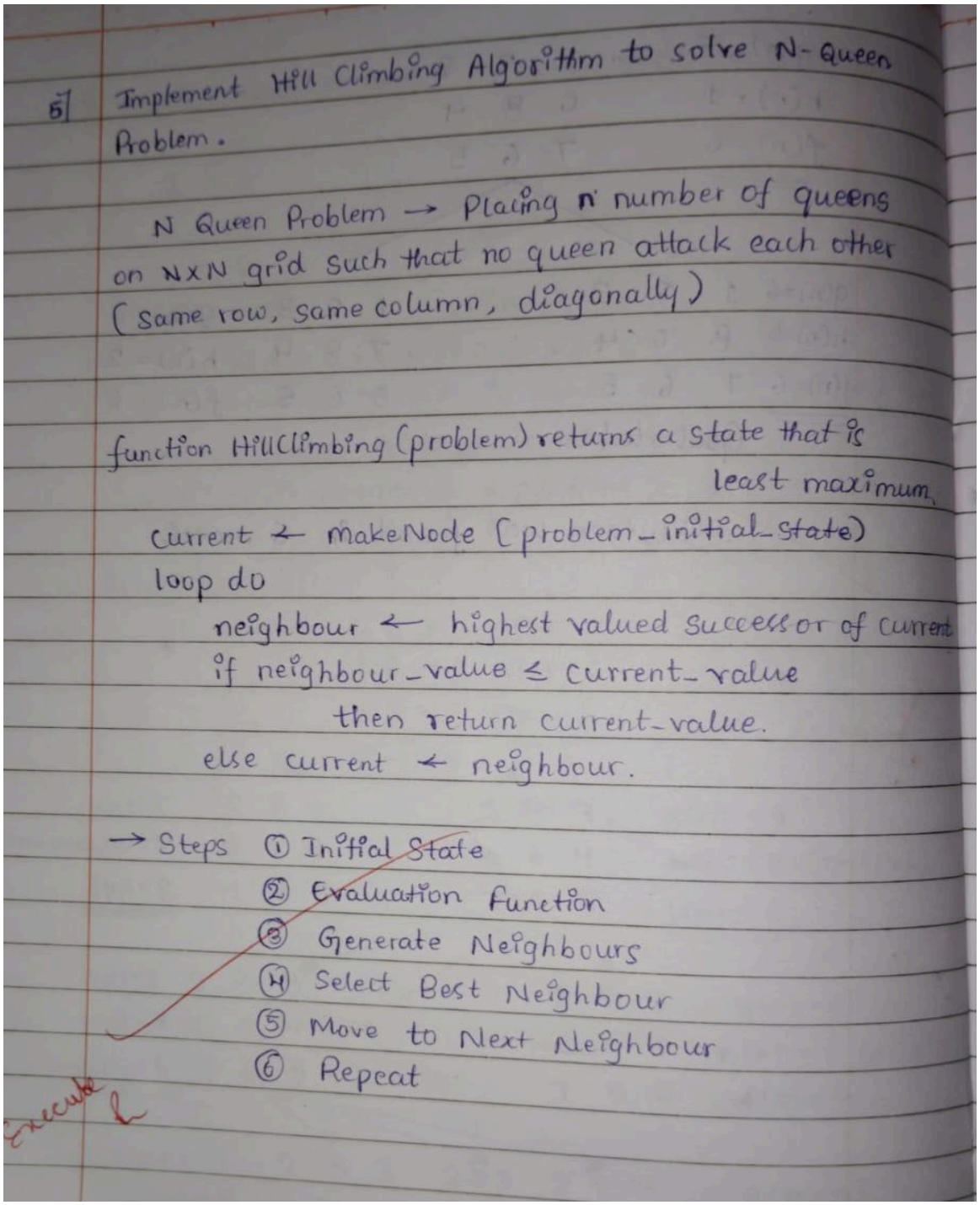
Output:

```
Solution found in 6 moves:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

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

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

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

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

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

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

Program 5:

Implement Hill Climbing Algorithm to solve N Queens Problem.

Algorithm/Pseudo Code:

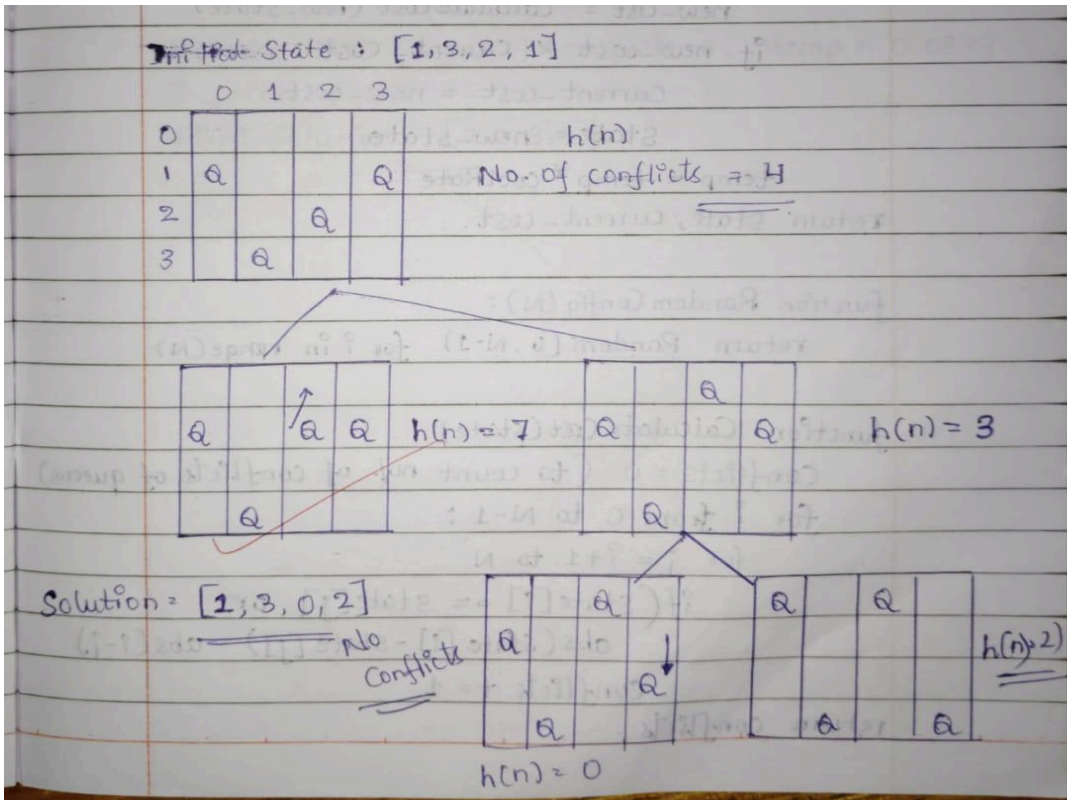


Code:

```
import random
def calculate_cost(state):
    conflicts = 0
    N = len(state)
    for i in range(N):
        for j in range(i + 1, N):
            # Check if queens i and j are attacking each other
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def generate_neighbors(state):
    neighbors = []
    N = len(state)
    for col in range(N):
        # Try moving the queen in column `col` to all other rows
        for row in range(N):
            if state[col] != row:  # Avoid moving to the same position
                new_state = state[:]
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors
def hill_climbing(N):
     # Step 1: Initialize a random state
    state = [random.randint(0, N - 1) for _ in range(N)]
    print(f"Initial State: {state}")
    current_cost = calculate_cost(state)
    print(f"Initial Conflicts: {current_cost}"
    # Step 2: Loop until a solution is found or no improvement is possible
    while current_cost > 0:
        # Generate neighbors
        neighbors = generate_neighbors(state)
        # Step 3: Select the best neighbor (with the least conflicts)
        best_neighbor = None
        best_cost = float('inf')
        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:
                best_cost = cost
                best_neighbor = neighbor
        # If no better neighbor is found, we've reached a local minimum
        if best_cost >= current_cost:
            print("No better neighbor found. Stopping.")
            return None  # Return None indicating failure (local minimum reached)

        # Step 4: Move to the best neighbor
        state = best_neighbor
        current_cost = best_cost
        print(f"New State: {state}")
```

```
        print(f"New Conflicts: {current_cost}")
    # If the loop terminates with zero conflicts, we have found a solution
    print(f"Solution found: {state}")
    return state
# Example usage
N = 4
 # Number of queens (8-Queens problem)
solution = hill_climbing(N)
if solution:
    print(f"Final Solution: {solution}")
else:
    print("No solution found.")
```

Output:

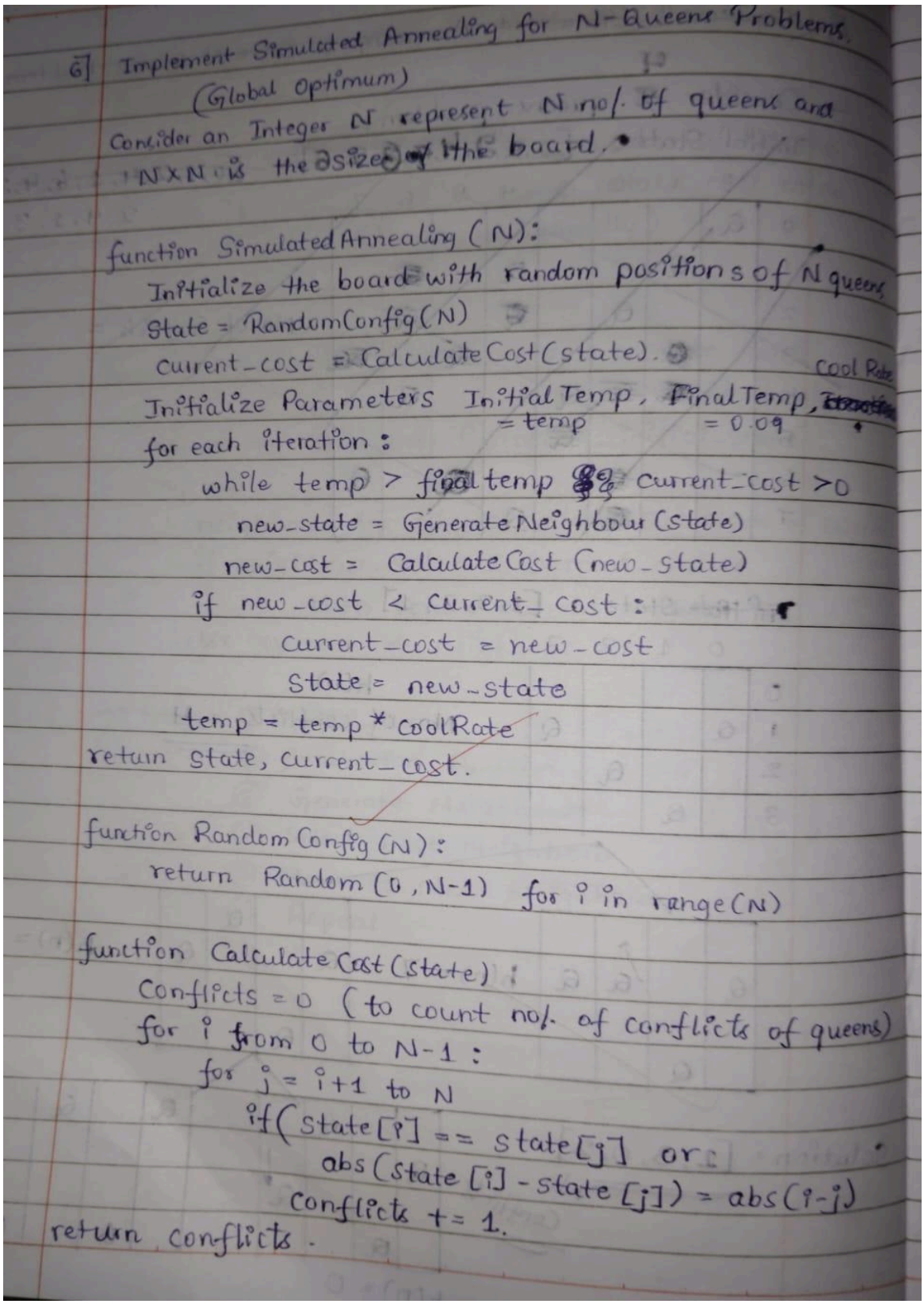



```
Initial State: [2, 2, 2, 1]
Initial Conflicts: 4
New State: [2, 0, 2, 1]
New Conflicts: 2
New State: [2, 0, 3, 1]
New Conflicts: 0
Solution found: [2, 0, 3, 1]
Final Solution: [2, 0, 3, 1]
```

Program 6:

Write a program to implement a Simulated Annealing Algorithm.

Algorithm/Pseudo Code:

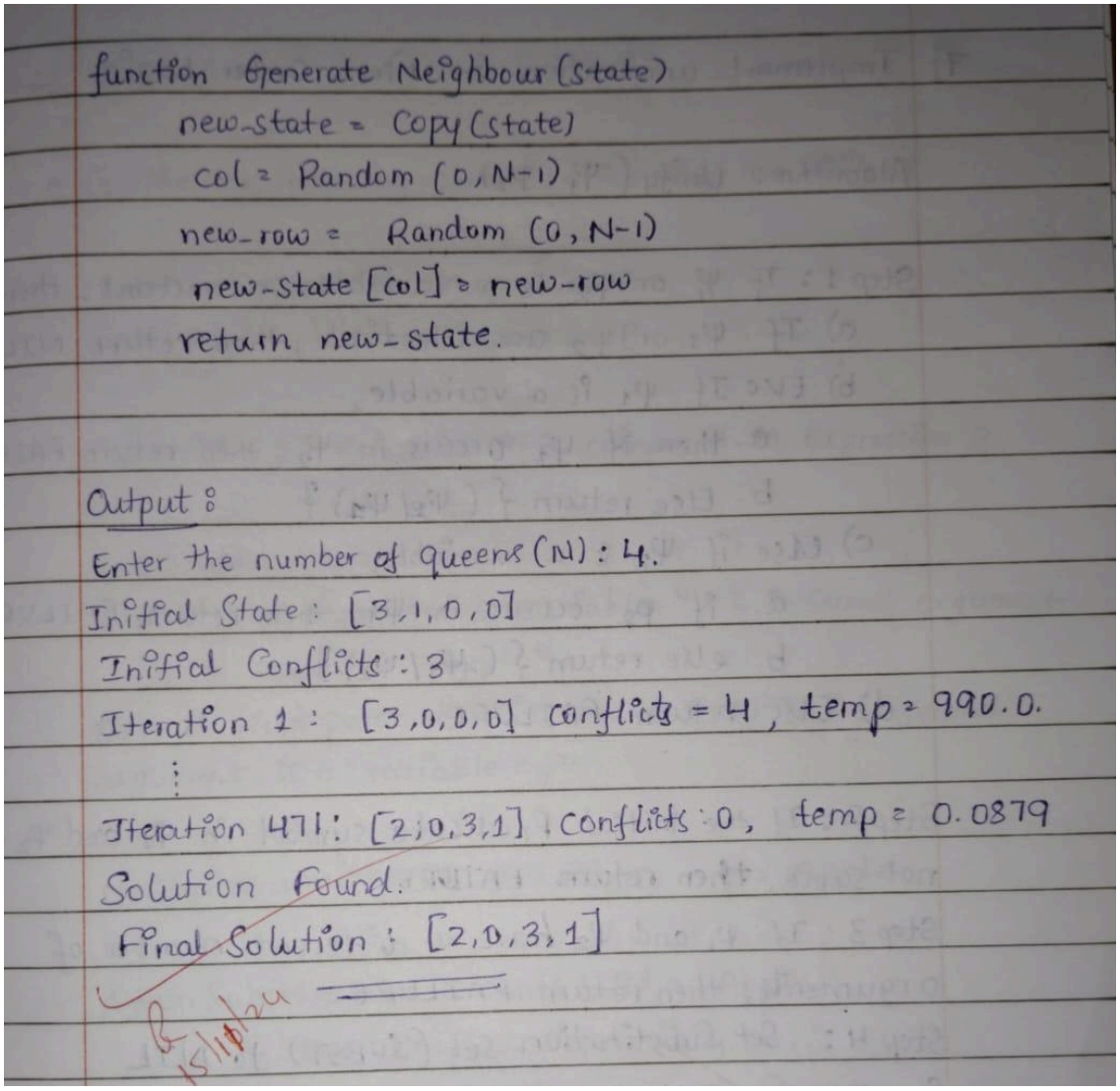


6] Implement Simulated Annealing for N-Queens Problems.
(Global Optimum)
Consider an Integer N represent N no/. of queens and
N X N is the size of the board.

function SimulatedAnnealing (N):
Initialize the board with random positions of N queens
State = RandomConfig(N)
current_cost = CalculateCost(state).
Initialize Parameters InitialTemp = temp, FinalTemp = 0.09, CoolRate
for each iteration:
while temp > final temp & current_cost > 0
new_state = GenerateNeighbour(state)
new_cost = CalculateCost(new_state)
if new_cost < current_cost:
current_cost = new_cost
State = new_state
temp = temp * coolRate
return state, current_cost.

function RandomConfig(N):
return Random(0, N-1) for i in range(N)

function CalculateCost(state):
Conflicts = 0 (to count no/. of conflicts of queens)
for i from 0 to N-1:
for j = i+1 to N
if(State[i] == state[j] or
abs(state[i] - state[j]) = abs(i-j)
Conflicts += 1.
return conflicts.



Code:

```
import random
import math

def calculate_cost(state):
    conflicts = 0
    N = len(state)

    for i in range(N):
        for j in range(i + 1, N):
            # Check if queens i and j are attacking each other
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
```

```
def generate_neighbor(state):
    N = len(state)
    new_state = state[:]

    # Randomly select a column to move a queen
    col = random.randint(0, N - 1)

    # Randomly select a new row for the queen in that column (avoid current row)
    new_row = random.randint(0, N - 1)
    while new_row == new_state[col]:
        new_row = random.randint(0, N - 1)

    # Move the queen to the new row
    new_state[col] = new_row

    return new_state

def simulated_annealing(N, initial_temp, cooling_rate, max_iterations):
    """
    Simulated Annealing algorithm to solve the N-Queens problem.
    N: Number of queens (board size)
    initial_temp: Initial temperature
    cooling_rate: Rate at which temperature decreases
    max_iterations: Maximum number of iterations to run
    """
    # Step 1: Initialize the state with a random configuration of queens
    state = [random.randint(0, N - 1) for _ in range(N)]
    current_cost = calculate_cost(state)

    # Step 2: Start with an initial temperature
    temp = initial_temp
    best_state = state[:]
    best_cost = current_cost
    print(f"Initial State: {state}")
    print(f"Initial Conflicts: {current_cost}")

    # Step 3: Main loop
    for iteration in range(max_iterations):
        # Generate a neighboring state
        neighbor = generate_neighbor(state)
        neighbor_cost = calculate_cost(neighbor)
        # If the neighbor is better, accept it
        if neighbor_cost < current_cost:
            state = neighbor
            current_cost = neighbor_cost
        else:
            # If the neighbor is worse, accept it with a certain probability
            delta_cost = neighbor_cost - current_cost
            acceptance_prob = math.exp(-delta_cost / temp)
```

```
            if random.random() < acceptance_prob:
                state = neighbor
                current_cost = neighbor_cost
        # Keep track of the best solution found
        if current_cost < best_cost:
            best_state = state[:]
            best_cost = current_cost
        # Cool down the temperature
        temp *= cooling_rate
        # Print the current state and its conflicts
        if iteration % (max_iterations // 10) == 0 or current_cost == 0:
            print(f"Iteration {iteration + 1}: {state}, Conflicts: {current_cost}, Temperature: {temp}")
        # Stop if the solution is found (zero conflicts)
        if current_cost == 0:
            print(f"Solution found: {state}")
            return state
    print("Reached maximum iterations without finding a solution.")
    return best_state

# Example usage
N = int(input("Enter the number of queens (N): "))  # Number of queens (board size)
initial_temp = 1000  # Initial temperature
cooling_rate = 0.99  # Cooling rate
max_iterations = 10000  # Maximum iterations
# Run the Simulated Annealing algorithm
solution = simulated_annealing(N, initial_temp, cooling_rate, max_iterations)
if solution:
    print(f"Final Solution: {solution}")
else:
    print("No solution found.")
```

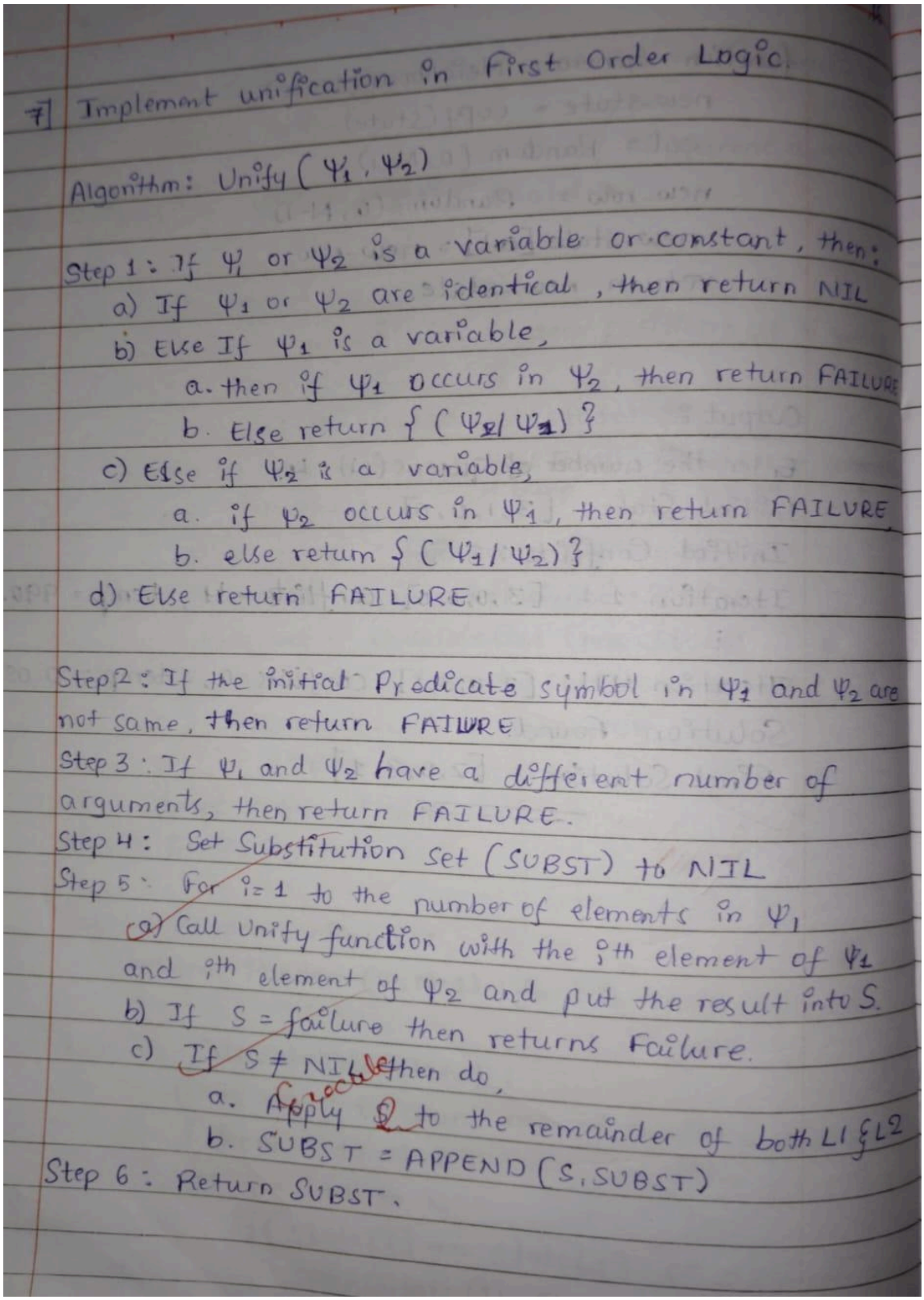
Output:

```
Enter the number of queens (N): 4
Initial State: [3, 1, 0, 0]
Initial Conflicts: 3
Iteration 1: [3, 0, 0, 0], Conflicts: 4, Temperature: 990.0
Iteration 471: [2, 0, 3, 1], Conflicts: 0, Temperature: 8.793801444488377
Solution found: [2, 0, 3, 1]
Final Solution: [2, 0, 3, 1]
```

Program 7:

Implement unification in first order logic.

Algorithm/Pseudo Code:



7] Implement unification in First Order Logic.

Algorithm: Unify (Ψ_1, Ψ_2)

Step 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

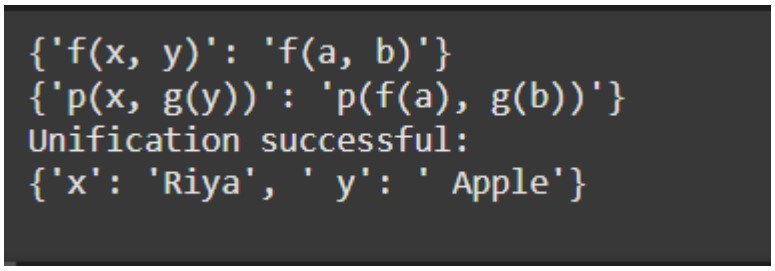
Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE
Step 3: If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.
Step 4: Set Substitution set (SUBST) to NIL
Step 5: For i= 1 to the number of elements in Ψ_1
a) Call Unify function with the ith element of Ψ_1 and ith 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 & L2
b. SUBST = APPEND (S, SUBST)
Step 6: Return SUBST.

Code:

```
def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    # Step 1: If x and y are the same, return the current substitutions
    if x == y:
        return subst
    # If x is a variable
    if is_variable(x):
        return unify_var(x, y, subst)
    # If y is a variable
    if is_variable(y):
        return unify_var(y, x, subst)
    # If x and y are compound expressions
    if is_compound(x) and is_compound(y):
        if get_predicate(x) != get_predicate(y):
            return "FAILURE"
        return unify(get_args(x), get_args(y), subst)

    # If x and y are lists
    if isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return "FAILURE"
        if not x and not y:
            return subst
        return unify(x[1:], y[1:], unify(x[0], y[0], subst))

    # If x and y are constants or cannot be unified
    return "FAILURE"

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    if x in subst:
        return unify(var, subst[x], subst)
    if occurs_check(var, x):
        return "FAILURE"
    subst[var] = x
    return subst

def occurs_check(var, x):
    if var == x:
        return True
    if isinstance(x, list):
        return any(occurs_check(var, arg) for arg in x)
    return False

def is_variable(x):
    return isinstance(x, str) and x.islower()  # Variables are lowercase strings
```

```
def is_compound(x):
    return isinstance(x, str) and '(' in x and ')' in x

def get_predicate(x):
    return x.split('(')[0]

def get_args(x):
    return x[x.index('(') + 1:x.index(')')].split(',')

# Test cases
x1 = "f(x, y)"
x2 = "f(a, b)"
print(unify(x1, x2))  # Expected: {'x': 'a', 'y': 'b'}

x3 = "p(x, g(y))"
x4 = "p(f(a), g(b))"
print(unify(x3, x4))  # Expected: {'x': 'f(a)', 'y': 'b'}

expression_a = "Eats(x, Apple)"
expression_b = "Eats(Riya, y)"

substitution = unify(expression_a, expression_b)

# Print the result
if substitution == "FAILURE":
    print("Unification failed.")
else:
    print("Unification successful:")
    print(substitution)
```

Output:



Output:

Consider the following phrases, which must be unified.

Eats(x, Apple) is an expression A
Eats(Riya, y) is an expression B

Comparision : Expression A is compared to Expression B.

Substitution Variable :
Exp A first parameter is variable "x" & Second argument is a constant "Apple"
Exp B first parameter is constant "Riya" & Second argument is a variable "y".

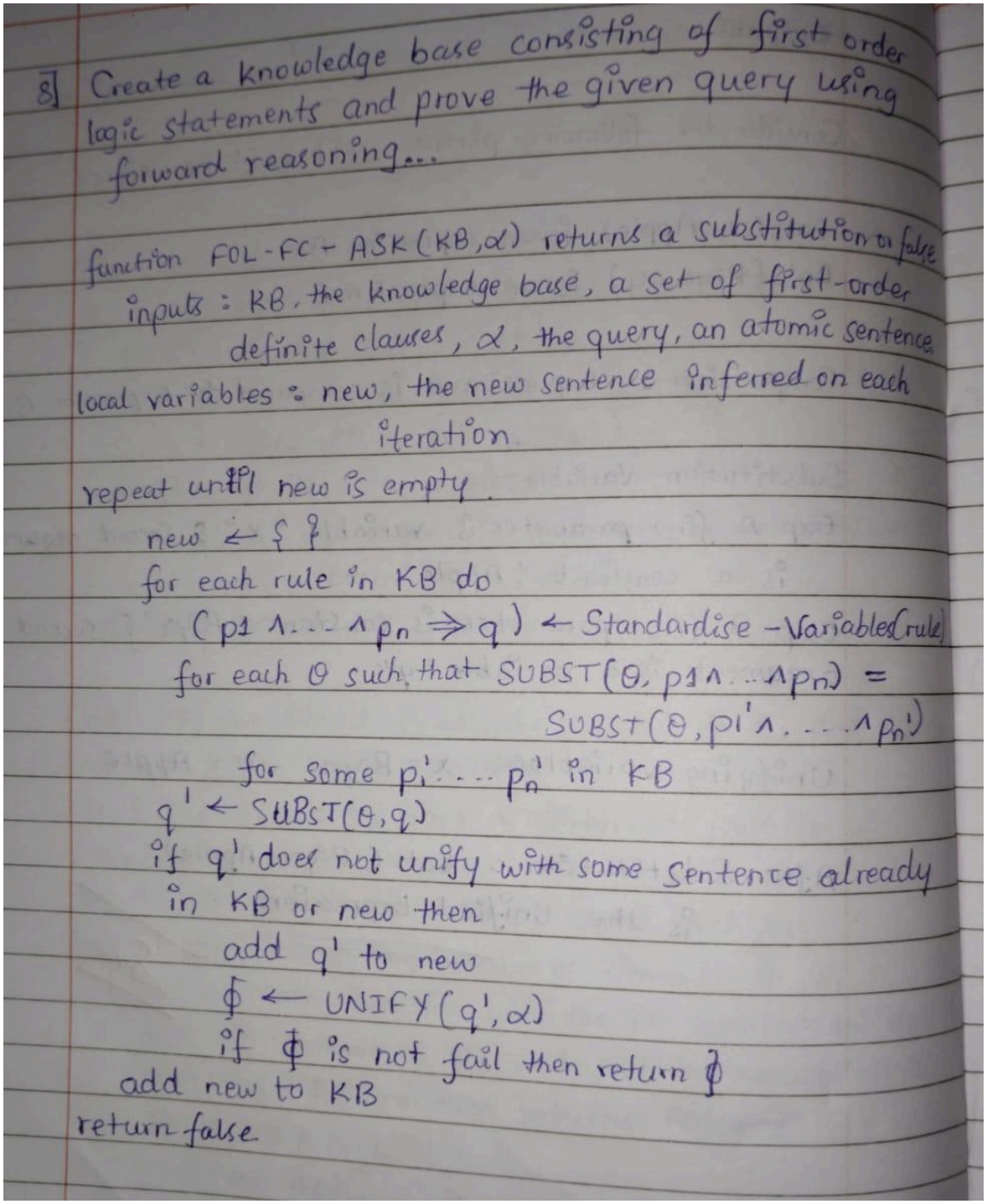
Unifying variables : x = Riya y = Apple.

After Substitution : Eats (Riya , Apple)
is the Unified Expression.

Program 8:

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

Algorithm/Pseudo Code:



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

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 sentence inferred on each iteration
repeat until new is empty
  new $\leftarrow$ { }
  for each rule in KB do
    $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ Standardise-Variables(rule)
    for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p_1' \wedge \ldots \wedge p_n'$)
      for some $p_1' \ldots p_n'$ in KB
    $q' \leftarrow$ SUBST(θ, q)
    if q' does not unify with some sentence already in KB or new then
      add q' to new
      $\phi \leftarrow$ UNIFY(q', α)
      if ϕ is not fail then return ϕ
  add new to KB
return false

→ KB = [
"American (Robert)"
"Enemy (A, America)",
"Missile (T1),
"Owns (A, T1)",
"Missile(x) ⇒ Weapon(x)"
"Weapon(x) ∧ Sells (Robert, x, A) ∧ Hostile(A) ⇒ Criminal (Robert)"
"Enemy (x, America) ⇒ Hostile(x)",
"Sells (Robert, T1, A)",
]

query = "Criminal (Robert)"

⇒ The query 'Criminal (Robert)' can be proved.

- Missile(x) ⇒ Weapon(x).
  Substitute x = T1 Inference: Weapon(T1)

- Enemy (x, America) ⇒ Hostile(x).
  Fact in KB: Enemy (A, America).
  Substitute x = A. Inference: Hostile(A)

- Weapon(x) ∧ Sells(p, x, r) ∧ Hostile(r) ⇒ Criminal(p).
  Substitute x = T1, p = Robert, r = A
  Weapon (T1), Sells (Robert, T1, A), Hostile(A)
  Inference: Criminal (Robert).

9) Every Surgeon has a lawyer.
∀p (Occupation(p, Surgeon) → ∃l (Occupation(l, lawyer) ∧ Customer(p, l))

Code:

```
def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    # Step 1: If x and y are the same, return the current substitutions
    if x == y:
        return subst
    # If x is a variable
    if is_variable(x):
        return unify_var(x, y, subst)
    # If y is a variable
    if is_variable(y):
        return unify_var(y, x, subst)
    # If x and y are compound expressions
    if is_compound(x) and is_compound(y):
        if get_predicate(x) != get_predicate(y):
            return "FAILURE"
        return unify(get_args(x), get_args(y), subst)

    # If x and y are lists
    if isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return "FAILURE"
        if not x and not y:
            return subst
        return unify(x[1:], y[1:], unify(x[0], y[0], subst))

    # If x and y are constants or cannot be unified
    return "FAILURE"

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    if x in subst:
        return unify(var, subst[x], subst)
    if occurs_check(var, x):
        return "FAILURE"
    subst[var] = x
    return subst

def occurs_check(var, x):
    if var == x:
        return True
    if isinstance(x, list):
        return any(occurs_check(var, arg) for arg in x)
    return False

def is_variable(x):
    return isinstance(x, str) and x.islower()  # Variables are lowercase strings
```

```
def is_compound(x):
    return isinstance(x, str) and '(' in x and ')' in x

def get_predicate(x):
    return x.split('(')[0]

def get_args(x):
    return x[x.index('(') + 1:x.index(')')].split(',')

# Test cases
x1 = "f(x, y)"
x2 = "f(a, b)"
print(unify(x1, x2))  # Expected: {'x': 'a', 'y': 'b'}

x3 = "p(x, g(y))"
x4 = "p(f(a), g(b))"
print(unify(x3, x4))  # Expected: {'x': 'f(a)', 'y': 'b'}

expression_a = "Eats(x, Apple)"
expression_b = "Eats(Riya, y)"

substitution = unify(expression_a, expression_b)

# Print the result
if substitution == "FAILURE":
    print("Unification failed.")
else:
    print("Unification successful:")
    print(substitution)
# Define the Knowledge Base (KB)
kb = []
# Facts
kb.append(("fact", "American", ["Robert"]))
kb.append(("fact", "Enemy", ["A", "America"]))
kb.append(("fact", "Missile", ["T1"]))
# Rules
kb.append(("rule", ["Missile(x)"], "Weapon(x)"))  # All missiles are weapons
kb.append(("rule", ["Weapon(x)", "Sells(Robert, x, A)", "Hostile(A)"], "Criminal(Robert)"))  #
Selling weapons to hostiles implies criminality
kb.append(("rule", ["Enemy(x, America)"], "Hostile(x)"))  # Enemies of America are hostile
kb.append(("fact", "Sells", ["Robert", "T1", "A"]))  # Robert sells T1 to A

# Forward Reasoning Function
def forward_reasoning(kb, query):
    inferred = set()  # Already inferred facts
    while True:
        new_inferred = set()
        for entry in kb:
            if entry[0] == "fact":
```

```
            # Add atomic facts to inferred
            fact_name = entry[1]
            fact_args = tuple(entry[2])
            new_inferred.add((fact_name, fact_args))

        elif entry[0] == "rule":
            # Check if the rule's premise is satisfied
            premises = entry[1]
            conclusion = entry[2]

            # Evaluate premises
            premises_satisfied = True
            substitutions = {}
            for premise in premises:
                satisfied = False
                for fact in inferred:
                    fact_name, fact_args = fact
                    if match(premise, fact_name, fact_args, substitutions):
                        satisfied = True
                        break
                if not satisfied:
                    premises_satisfied = False
                    break

            # If all premises are satisfied, infer the conclusion
            if premises_satisfied:
                conclusion_name, conclusion_args = parse_predicate(conclusion)
                conclusion_args = [
                    substitutions.get(arg, arg) for arg in conclusion_args
                ]
                new_inferred.add((conclusion_name, tuple(conclusion_args)))

        # If no new facts are inferred, stop
        if not new_inferred.difference(inferred):
            break
        inferred.update(new_inferred)

        # Check if the query is inferred
        query_name, query_args = parse_predicate(query)
        if (query_name, tuple(query_args)) in inferred:
            return True

    return False

# Helper functions for matching and parsing
def match(premise, fact_name, fact_args, substitutions):
    premise_name, premise_args = parse_predicate(premise)
    if premise_name != fact_name or len(premise_args) != len(fact_args):
        return False
    for p_arg, f_arg in zip(premise_args, fact_args):
```

```
        if p_arg.islower():  # Variable
            substitutions[p_arg] = f_arg
        elif p_arg != f_arg:  # Constant mismatch
            return False
    return True

def parse_predicate(predicate):
    name, args = predicate.split("(")
    args = args[:-1].split(",")  # Remove closing ")"
    return name, args

# Query
query = "Criminal(Robert)"

# Check if the query can be proved
if forward_reasoning(kb, query):
    print(f"The query {query} can be proved.")
else:
    print(f"The query {query} cannot be proved.")
```

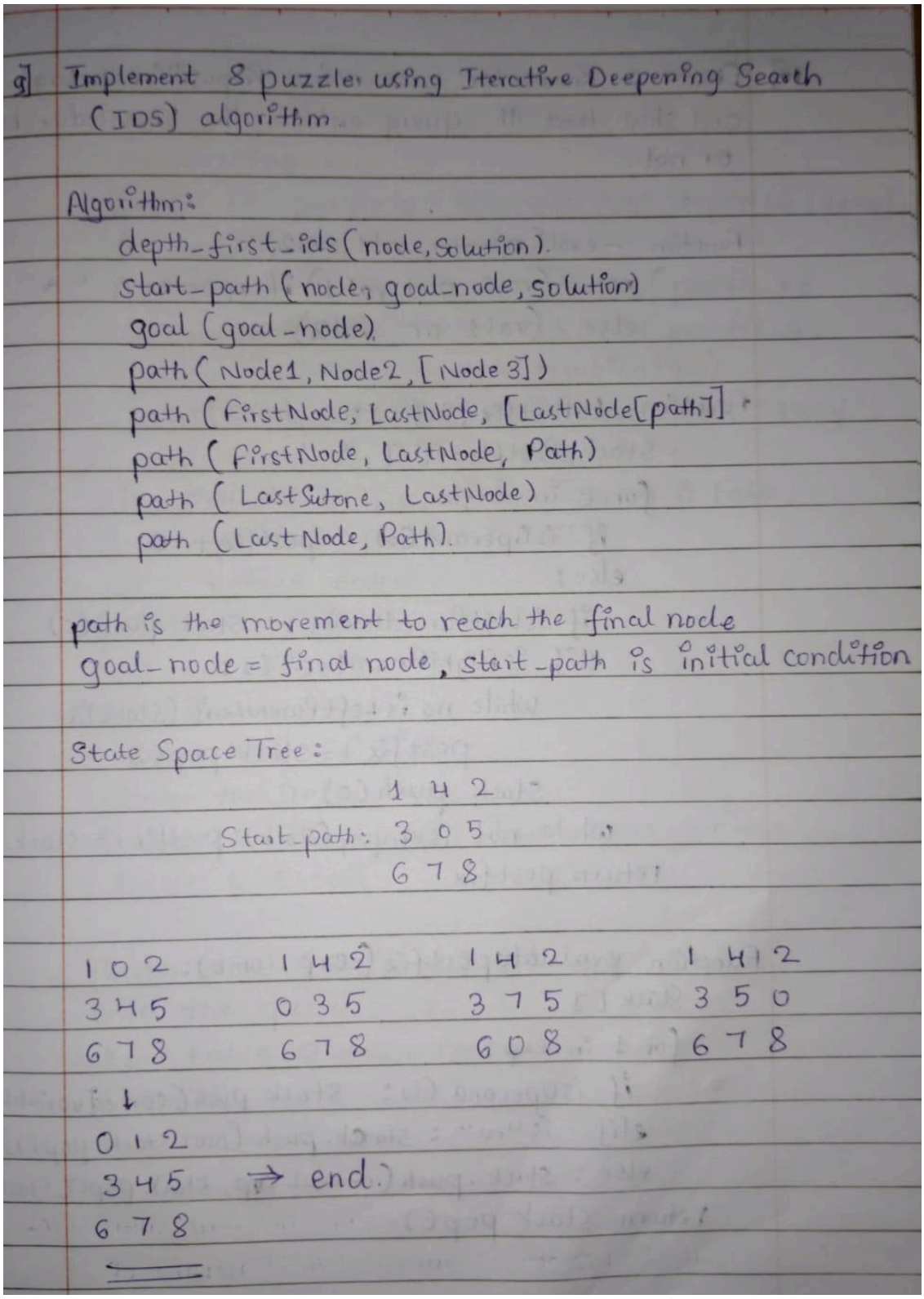
Output:

```
The query Criminal(Robert) cannot be proved.
```

Program 9:

Implement Iterative deepening search algorithm.

Algorithm/Pseudo Code:



9] Implement 8 puzzle using Iterative Deepening Search (IDS) algorithm

Algorithm:

depth_first_ids(node, Solution).
start_path (node, goal_node, solution)
goal (goal_node).
path (Node1, Node2, [Node 3])
path (FirstNode, LastNode, [LastNode[path]]
path (FirstNode, LastNode, Path)
path (LastSutone, LastNode)
path (LastNode, Path).

path is the movement to reach the final node
goal_node = final node, start_path is initial condition

State Space Tree:

1 4 2
Start_path: 3 0 5
6 7 8

1 0 2	1 4 2	1 4 2	1 4 2
3 4 5	0 3 5	3 7 5	3 5 0
6 7 8	6 7 8	6 0 8	6 7 8
↓			
0 1 2			
3 4 5 ⇒ end.			
6 7 8			

Code:

```
from collections import deque
def is_valid_position(x, y):
    return 0 <= x < 3 and 0 <= y < 3
def iterative_deepening_search(initial_state, goal_state):
    def dfs_limited(state, depth):
        if state == goal_state:
            return []
        if depth == 0:
            return None
        empty_x, empty_y = [(x, y) for x in range(3) for y in range(3) if state[x][y] == 0][0]
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in moves:
            new_x, new_y = empty_x + dx, empty_y + dy

            if is_valid_position(new_x, new_y):
                new_state = [list(row) for row in state]
                new_state[empty_x][empty_y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[empty_x][empty_y]
                new_state = tuple(tuple(row) for row in new_state)
                result = dfs_limited(new_state, depth - 1)
                if result is not None:
                    return [(new_state, (new_x, new_y))] + result
        return None

    depth = 0
    initial_state = tuple(tuple(row) for row in initial_state)
    goal_state = tuple(tuple(row) for row in goal_state)
    while True:
        result = dfs_limited(initial_state, depth)
        if result is not None:
            return result
        depth += 1
# Example usage
def print_state(state):
    for row in state:
        print(" ".join(map(str, row)))
    print()

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

```
solution = iterative_deepening_search(initial_state, goal_state)
if solution:
    print("Solution found!")
    current_state = tuple(tuple(row) for row in initial_state)
    print("Initial State:")
    print_state(current_state)

    for new_state, move in solution:
        print(f"Move empty tile to position {move}:")
        print_state(new_state)
else:
    print("No solution exists.")
```

Output:

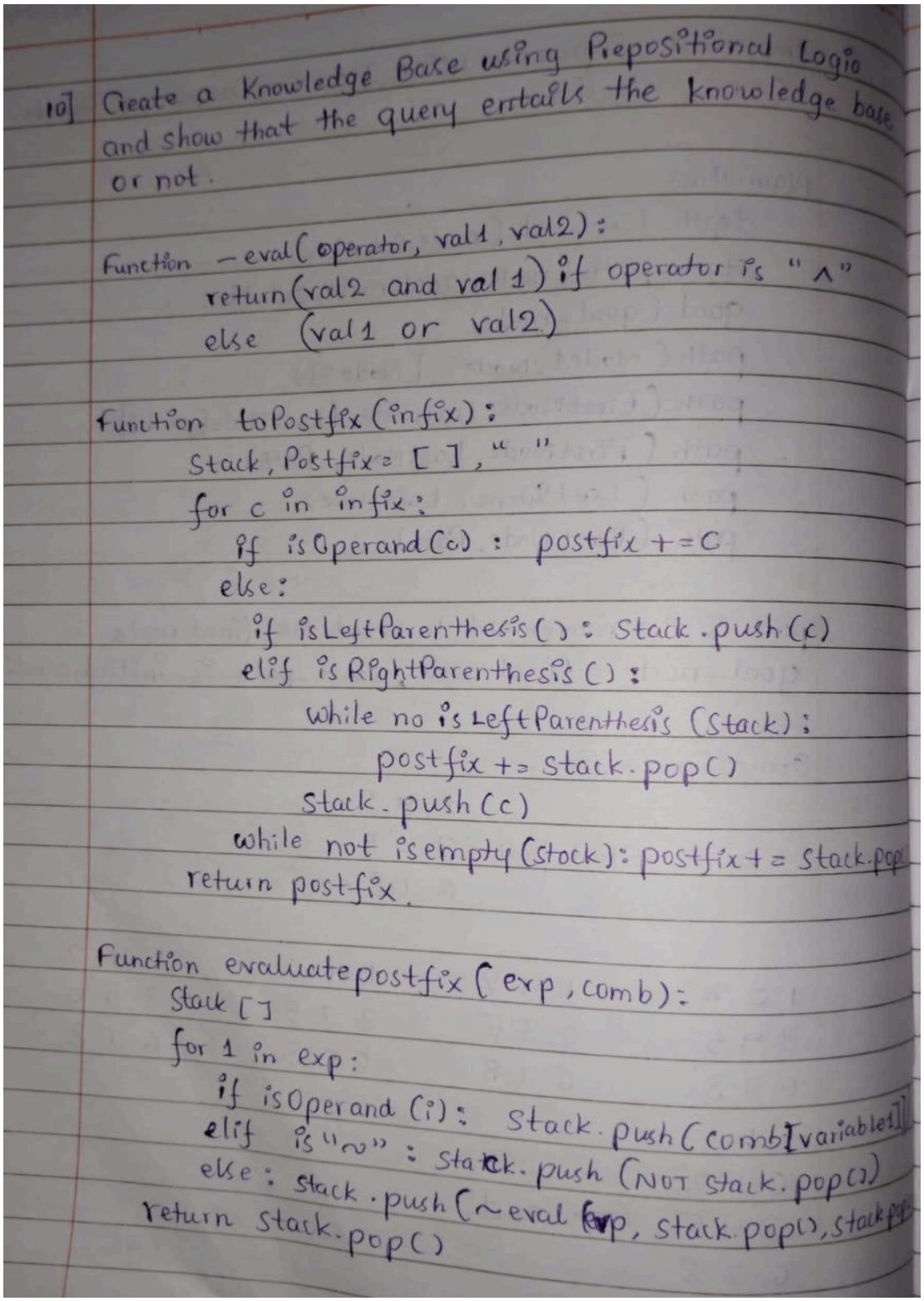
```
Solution found!
Initial State:
1 2 3
4 0 5
6 7 8

Move empty tile to position (1, 2):
1 2 3
4 5 0
6 7 8

Move empty tile to position (2, 2):
1 2 3
4 5 8
6 7 0

Move empty tile to position (2, 1):
1 2 3
4 5 8
6 0 7

Move empty tile to position (2, 0):
1 2 3
4 5 8
0 6 7

Move empty tile to position (1, 0):
1 2 3
0 5 8
4 6 7
```

```
Move empty tile to position (2, 2):
1 2 3
5 6 8
4 7 0

Move empty tile to position (1, 2):
1 2 3
5 6 0
4 7 8

Move empty tile to position (1, 1):
1 2 3
5 0 6
4 7 8

Move empty tile to position (1, 0):
1 2 3
0 5 6
4 7 8

Move empty tile to position (2, 0):
1 2 3
4 5 6
0 7 8

Move empty tile to position (2, 1):
1 2 3
4 5 6
7 0 8

Move empty tile to position (2, 2):
1 2 3
4 5 6
7 8 0
```

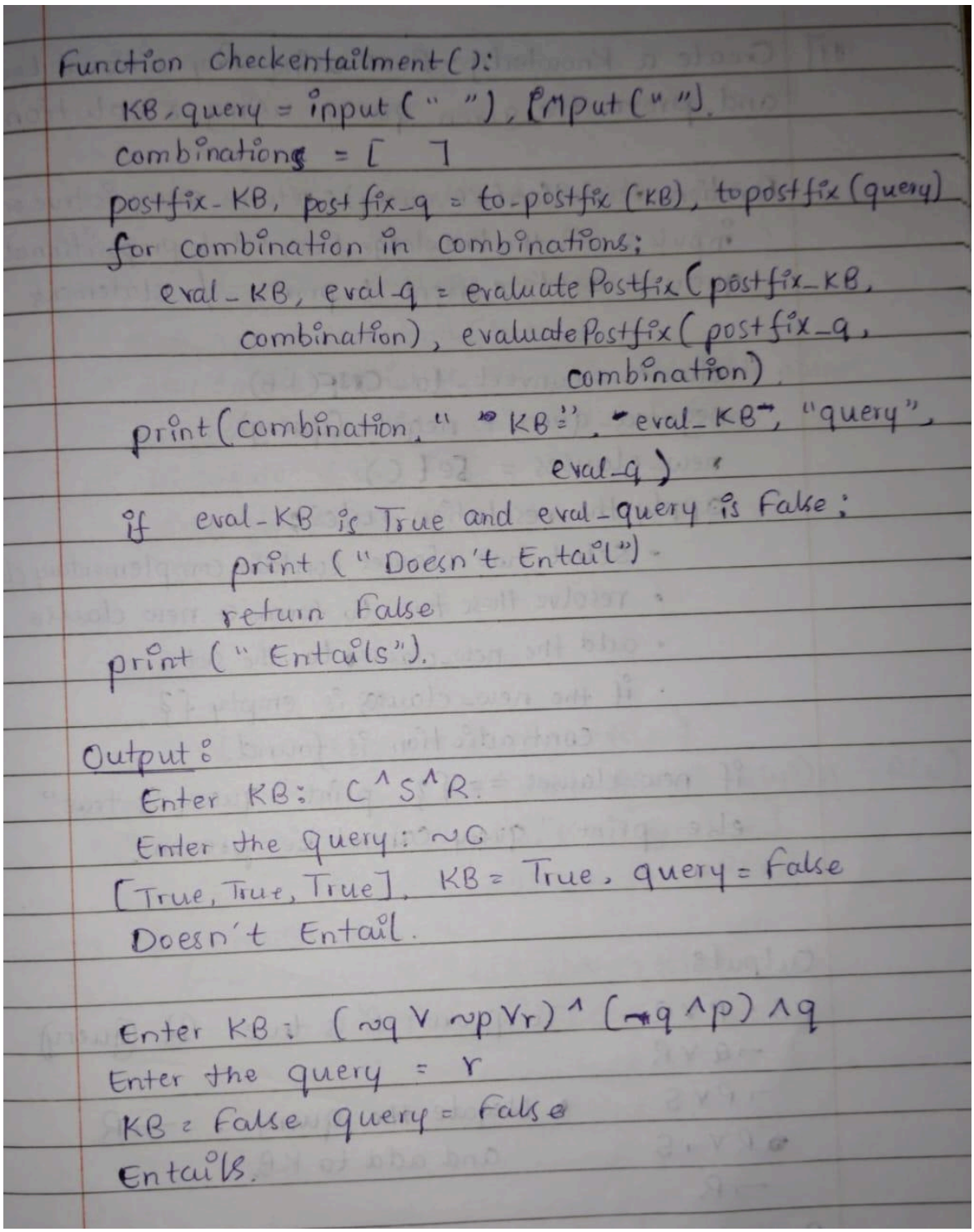
Program 10:

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

Algorithm/Pseudo Code:



```
10] Create a Knowledge Base using Propositional Logic
and show that the query entails the knowledge base
or not.

Function - eval(operator, val1, val2):
    return(val2 and val1) if operator is "^"
    else (val1 or val2)

Function toPostfix(infix):
    Stack, Postfix = [ ], " "
    for c in infix:
      if isOperand(c) : postfix += c
      else:
        if isLeftParenthesis() : Stack.push(c)
        elif isRightParenthesis():
            while no isLeftParenthesis(Stack):
                postfix += stack.pop()
            Stack.push(c)
      while not isempty(stock): postfix += stack.pop
    return postfix.

Function evaluatepostfix(exp, comb):
    Stack [ ]
    for i in exp:
      if isOperand(i): Stack.push(comb[variable1]
      elif is "~" : stack.push(NOT stack.pop())
      else: stack.push(~eval(exp, stack.pop(), stack.pop
    return stack.pop()
```



Code:

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

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

    def add_rule(self, rule):
        self.rules.append(rule)
    def entails(self, query):
        inferred = set()
        new_facts = True
        while new_facts:
            new_facts = False
            for rule in self.rules:
                for fact in self.facts:
                    inferred_fact = rule(fact, self.facts)
                    if inferred_fact and inferred_fact not in self.facts and inferred_fact not in inferred:
                        inferred.add(inferred_fact)
                        new_facts = True
        return query in inferred

# Define propositional logic rules
def rule_crime_law(fact, facts):
    if fact == "American(Robert)" and "Sells(Robert, Weapons, CountryA)" in facts and
"Hostile(CountryA)" in facts:
        return "Criminal(Robert)"
    return None
# Initialize the knowledge base
kb = KnowledgeBase()
# Add facts to the knowledge base
kb.add_fact("American(Robert)")
kb.add_fact("Sells(Robert, Weapons, CountryA)")
kb.add_fact("Hostile(CountryA)")
# Add rules to the knowledge base
kb.add_rule(rule_crime_law)
# Define the query
query = "Criminal(Robert)"
# Check if the query is entailed
if kb.entails(query):
    print(f"The query '{query}' is entailed by the knowledge base.")
else:
    print(f"The query '{query}' is NOT entailed by the knowledge base.")
```

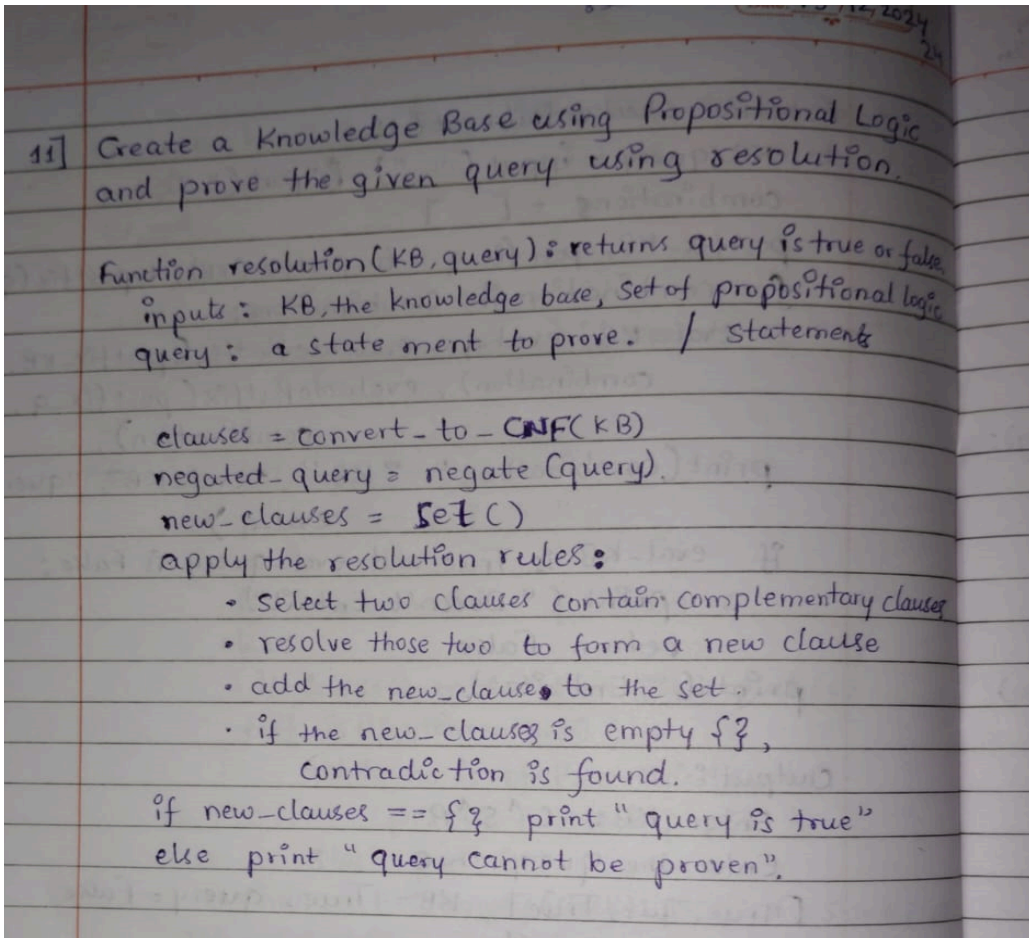
Output:

```
The query 'Criminal(Robert)' is entailed by the knowledge base.
```

Program 11:

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

Algorithm/Pseudo Code:



Code:

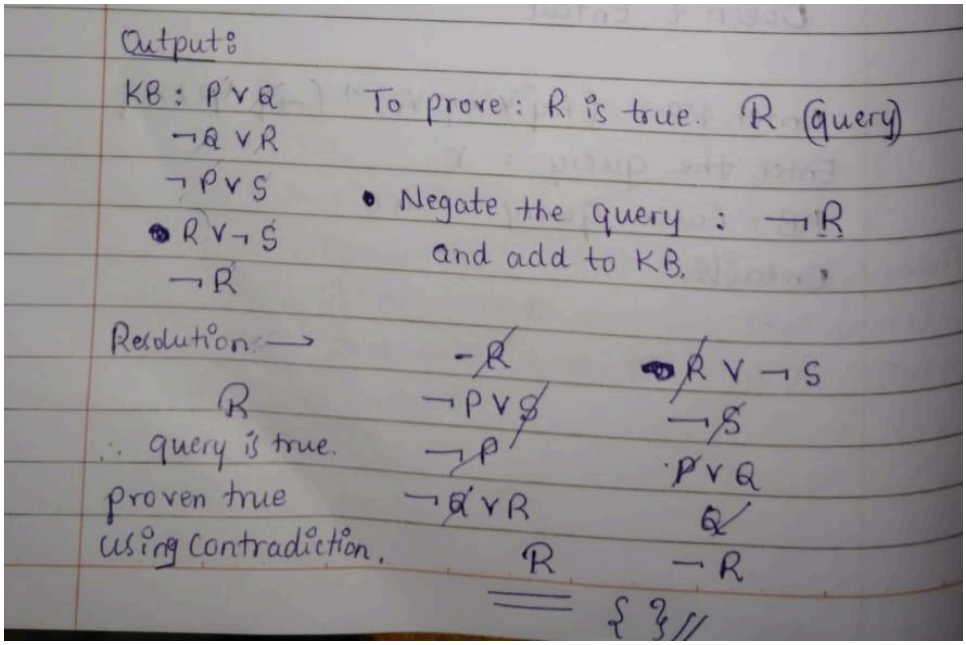
```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

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

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

    def entails(self, query):
        inferred = set()
```

```
        new_facts = True

        while new_facts:
            new_facts = False
            for rule in self.rules:
                for fact in self.facts:
                    inferred_fact = rule(fact, self.facts)
                    if inferred_fact and inferred_fact not in self.facts and inferred_fact not in inferred:
                        inferred.add(inferred_fact)
                        new_facts = True

        return query in inferred

# Define propositional logic rules
def rule_crime_law(fact, facts):
    if fact == "American(Robert)" and "Sells(Robert, Weapons, CountryA)" in facts and
"Hostile(CountryA)" in facts:
        return "Criminal(Robert)"
    return None
# Initialize the knowledge base
kb = KnowledgeBase()
# Add facts to the knowledge base
kb.add_fact("American(Robert)")
kb.add_fact("Sells(Robert, Weapons, CountryA)")
kb.add_fact("Hostile(CountryA)")

# Add rules to the knowledge base
kb.add_rule(rule_crime_law)

# Define the query
query = "Criminal(Robert)"

# Check if the query is entailed
if kb.entails(query):
    print(f"The query '{query}' is entailed by the knowledge base.")
else:
    print(f"The query '{query}' is NOT entailed by the knowledge base.")def parse_clause(clause):
    """
    Parse a clause string into a set of literals.
    """
    return set(clause.split(" ∨ "))

def resolve(clause1, clause2):
    """
    Resolve two clauses if they contain complementary literals.
    Returns a new clause or None if they cannot be resolved.
    """
    for literal in clause1:
        if literal.startswith("¬") and literal[1:] in clause2:
            new_clause = (clause1 - {literal}) | (clause2 - {literal[1:]})
```

```
            return new_clause
        elif not literal.startswith("¬") and f"¬{literal}" in clause2:
            new_clause = (clause1 - {literal}) | (clause2 - {f"¬{literal}"})
            return new_clause
    return None


def resolution(kb, query):
    """
    Perform resolution to prove if the query is true or not.
    :param kb: List of clauses in the knowledge base (each clause is a set of literals).
    :param query: The query to prove (string).
    :return: True if the query is proven, False otherwise.
    """
    # Step 1: Convert KB and query to CNF
    clauses = [parse_clause(clause) for clause in kb]
    negated_query = {f"¬{query}"}
    clauses.append(negated_query)

    # Step 2: Apply resolution rules
    new_clauses = set()
    while True:
        n = len(clauses)
        for i in range(n):
            for j in range(i + 1, n):
                resolvent = resolve(clauses[i], clauses[j])
                if resolvent is not None:
                    if not resolvent:  # Empty clause found
                        return True
                    new_clauses.add(frozenset(resolvent))

        # Check if new clauses are added
        if not new_clauses.difference(set(map(frozenset, clauses))):
            break
        clauses.extend(map(set, new_clauses))

    return False


if __name__ == "__main__":
    # Example Knowledge Base (KB)
    kb = [
        "P ∨ Q",
        "¬Q ∨ R",
        "¬P ∨ S",
        "R ∨ ¬S"
    ]
    query = "R"
```

```
# Prove the query
    if resolution(kb, query):
        print("Query is true.")
    else:
        print("Query cannot be proven.")
```

Output:

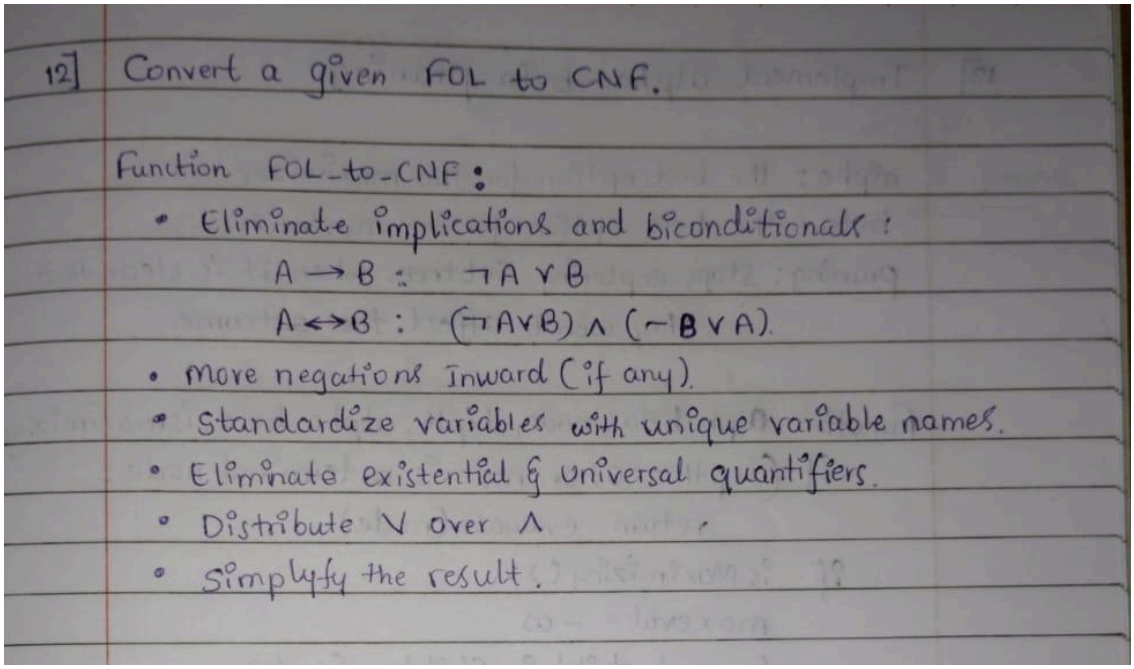



```
Query is true.
```

Program 12:

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

Algorithm/Pseudo Code:

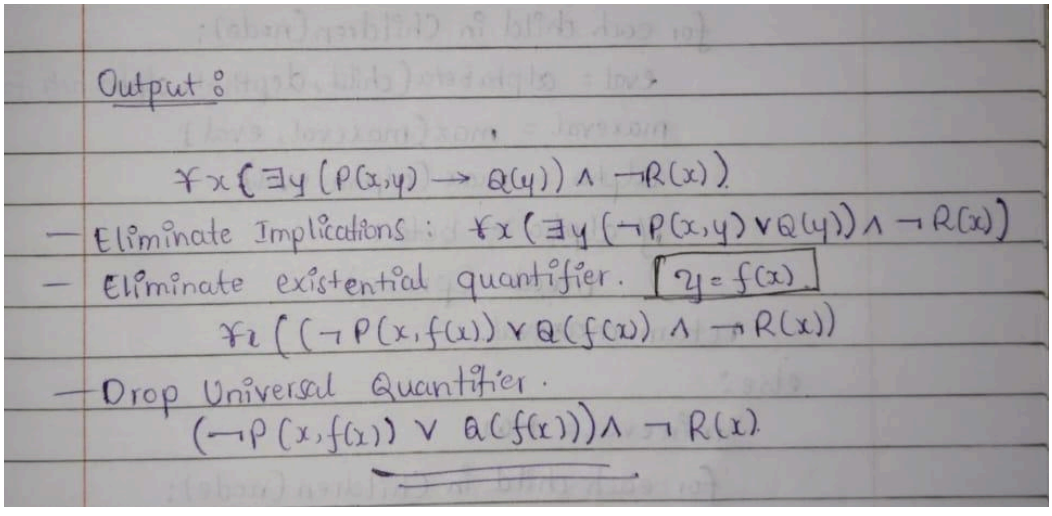


Code:

```
from sympy.logic.boolalg import Or, And, Not, Implies
from sympy import symbols, Function, Predicate, AppliedPredicate

# Define symbols and Skolem functions
x, y = symbols("x y")  # Variables
f = Function("f")      # Skolem function
# Define predicates
P = Predicate("P", x)
Q = Predicate("Q", x)
A = Predicate("A", x)
R = Predicate("R", x)
# Helper function to eliminate implications
def eliminate_implications(formula):
    """Eliminate implications (A → B ≡ ¬A ∨ B)."""
    return formula.replace(
        lambda x: isinstance(x, Implies),
        lambda x: Or(Not(x.args[0]), x.args[1])
    )
# Helper function to replace existential quantifiers with Skolem functions
def eliminate_existential_quantifier(formula):
    """Replace existential quantifier ∃ with Skolem functions."""
    # Assume y is existentially quantified and replace it with f(x) (Skolemization)
    return formula.subs(y, f(x))
```

```
# Helper function to drop universal quantifiers
def drop_universal_quantifiers(formula):
    """Drop universal quantifiers, as they are implied in CNF."""
    # Universal quantifiers are implicit in CNF
    return formula
# Main function to convert FOL to CNF
def fol_to_cnf(formula):
    """Convert a First-Order Logic formula to Conjunctive Normal Form (CNF)."""
    # Step 1: Eliminate implications
    formula = eliminate_implications(formula)
    # Step 2: Eliminate existential quantifier (Skolemization)
    formula = eliminate_existential_quantifier(formula)
    # Step 3: Drop universal quantifiers
    formula = drop_universal_quantifiers(formula)
    # Step 4: Distribute OR over AND manually for this example
    cnf = And(
        Or(Not(AppliedPredicate(P, x, f(x))), AppliedPredicate(Q, f(x))),  # First clause
        Or(Not(AppliedPredicate(A, x)), Not(AppliedPredicate(R, x)))      # Second clause
    )
    return cnf

# Original FOL formula
# ∀x (∃y (P(x, y) → Q(y)) ∧ (A(x) → ¬R(x)))
fol_formula = And(
    Implies(AppliedPredicate(P, x, y), AppliedPredicate(Q, y)),  # First implication
    Implies(AppliedPredicate(A, x), Not(AppliedPredicate(R, x)))  # Second implication
)
# Convert to CNF
cnf_formula = fol_to_cnf(fol_formula)
# Output the CNF formula
print("CNF Formula:", cnf_formula)
```

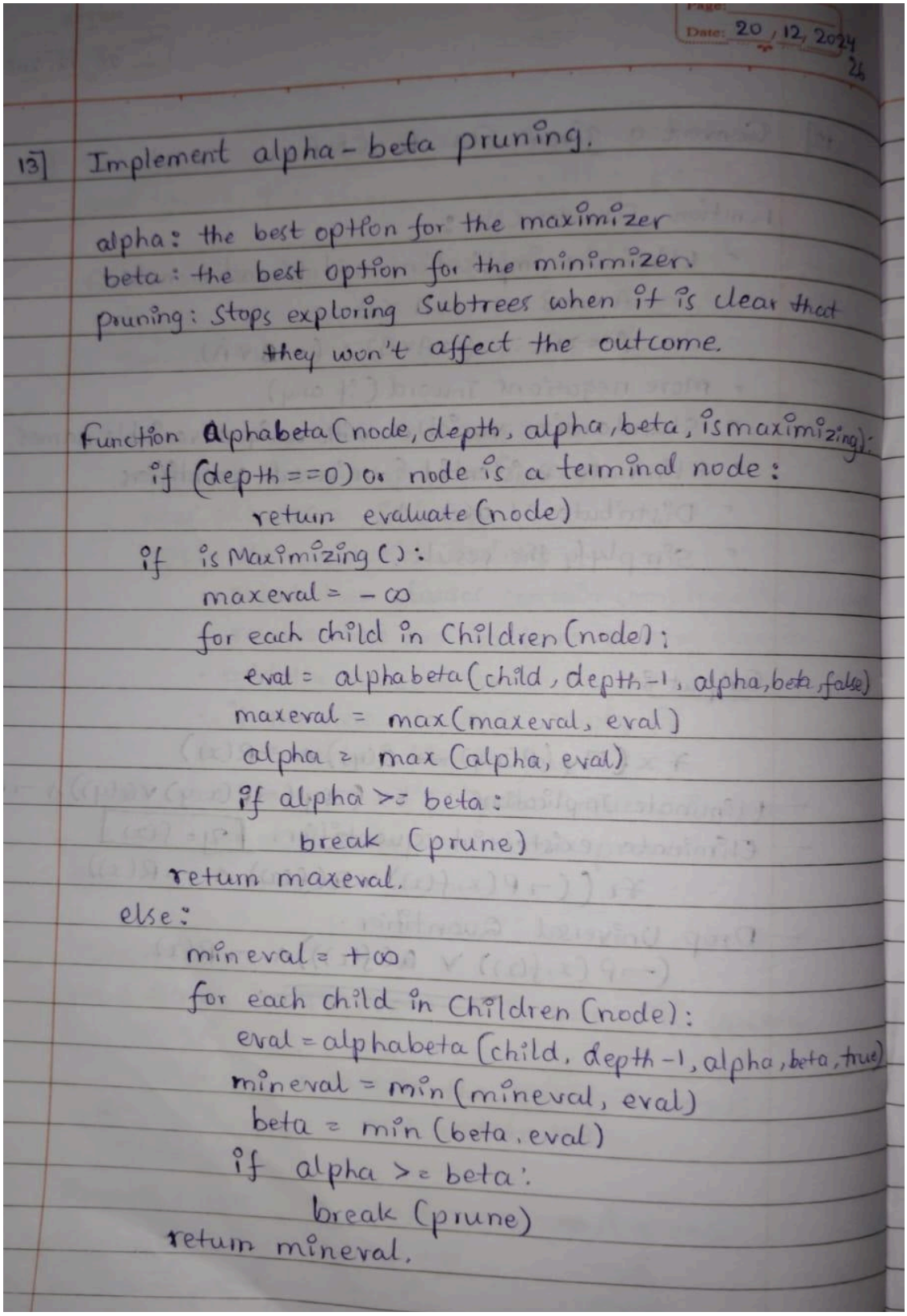
Output:



Program 13:

Implement Alpha-Beta Pruning.

Algorithm/Pseudo Code:



13] Implement alpha-beta pruning.

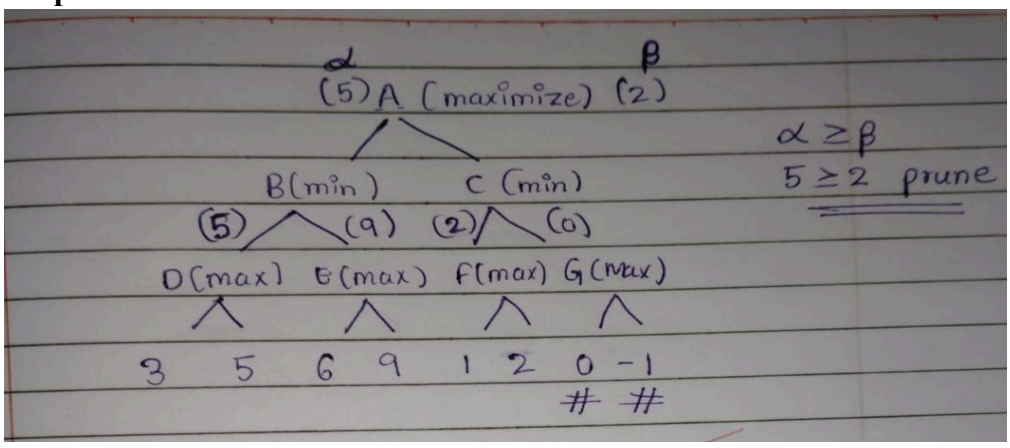
alpha: the best option for the maximizer
beta: the best option for the minimizer.
pruning: Stops exploring subtrees when it is clear that they won't affect the outcome.

```
Function alphabeta(node, depth, alpha, beta, ismaximizing):
    if (depth==0) or node is a terminal node:
        return evaluate(node)
    if isMaximizing():
        maxeval = -∞
        for each child in Children(node):
            eval = alphabeta(child, depth-1, alpha, beta, false)
            maxeval = max(maxeval, eval)
            alpha = max(alpha, eval)
            if alpha >= beta:
                break (prune)
        return maxeval.
    else:
        mineval = +∞
        for each child in Children(node):
            eval = alphabeta(child, depth-1, alpha, beta, true)
            mineval = min(mineval, eval)
            beta = min(beta, eval)
            if alpha >= beta:
                break (prune)
        return mineval.
```

Code:

```
# Alpha-Beta Pruning Implementation
def alpha_beta_pruning(depth, node_index, maximizing_player, values, alpha, beta):
    # Base case: Leaf node
    if depth == 3:  # Assuming depth of 3 for the provided tree
        return values[node_index]
    if maximizing_player:
        max_eval = float('-inf')
        for i in range(2):  # Each node has 2 children
            eval_value = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval_value)
            alpha = max(alpha, eval_value)
            # Prune the branch
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):  # Each node has 2 children
            eval_value = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval_value)
            beta = min(beta, eval_value)
            # Prune the branch
            if beta <= alpha:
                break
        return min_eval
# Example Usage
if __name__ == "__main__":
    # Values at the leaf nodes of the game tree
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    alpha = float('-inf')
    beta = float('inf')
    result = alpha_beta_pruning(0, 0, True, values, alpha, beta)
    print("The optimal value is:", result)
```

Output:




```
The optimal value is: 5
```