VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shankar shivappa pujar (1BM23CS309)

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
Aug 2025 to Dec 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 **Shankar shivappa pujar (1BM23CS309),** 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.

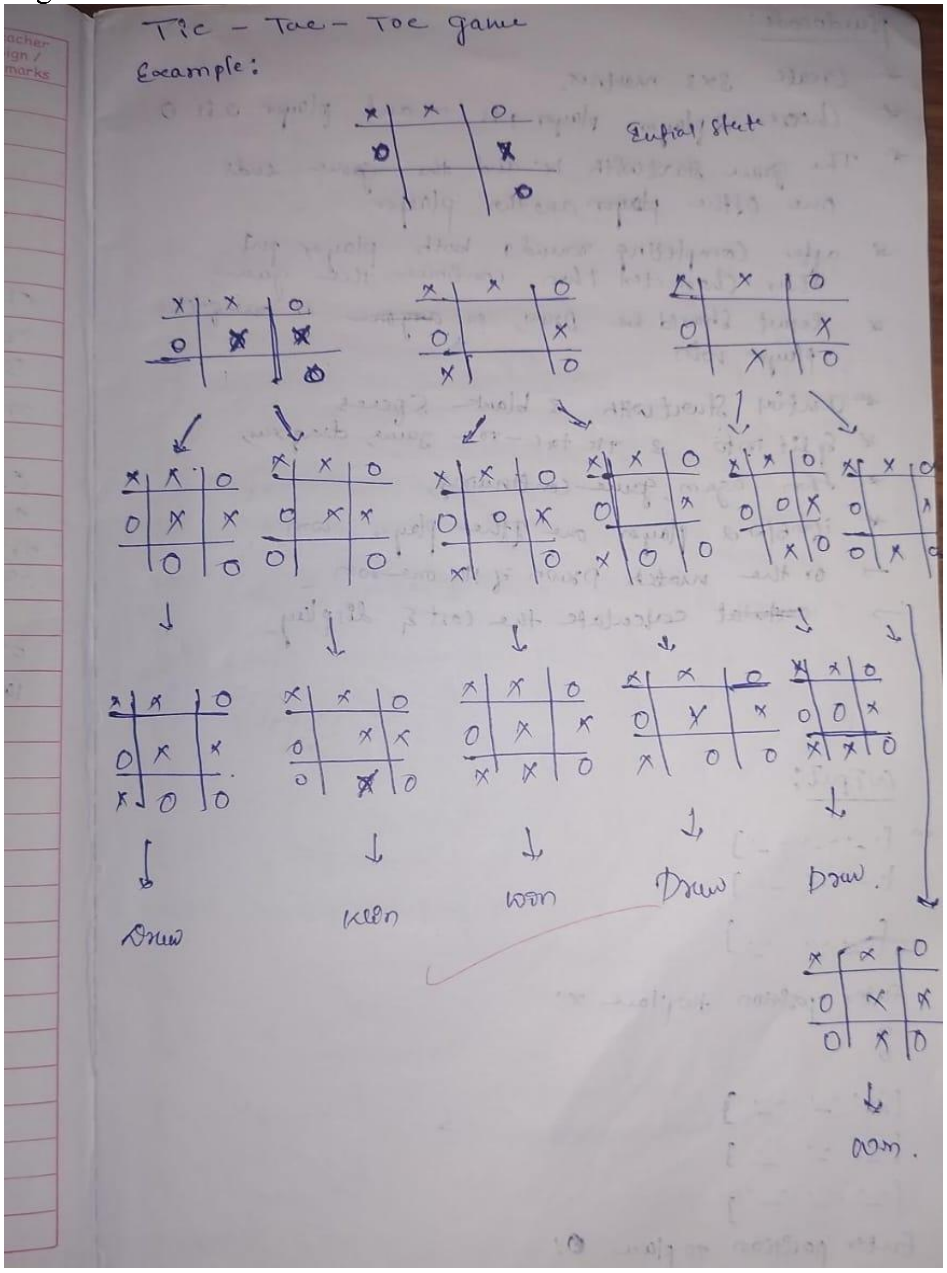
Lab faculty Seema Patil Name Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	04-12
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	13-19
3	14-10-2024	Implement A* search algorithm	20-25
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	26-29
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	30-31
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	32-35
7	2-12-2024	Implement unification in first order logic	36-39
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	40-42
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	43-47
10	16-12-2024	Implement Alpha-Beta Pruning.	48-50

Github Link:
https://github.com/shankar045/Shankar1BM23CS309AILAB

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



pseudocode:

```
* Create 3x3 matrix.
* choose 2 player. player 1 is 'x' and player 2 is 0
* The game start with x and the game ends one of the ~~player~~ another player.
* after Completing round 1 both player put their characters then, continue the game
* Result should be Draw, or ~~anyone~~ or one of the player won
* Initial Start with 3 blank Spaces
* Split into 3 tic tac-toe game, diagram,
* then again game continuing,
* if b/w 2 player one of the player won
→ or the match Drawn if No one won
→ ~~calculat~~ calculate the cost & display.
```

OUTPUT:

```
→ ['_','_','_']
  ['_','_','_']
  ['_','_','_']
Enter position to place x:
0
0
['x','_','_']
['_','_','_']
['_','_','_']
Enter position to place O:
```

5

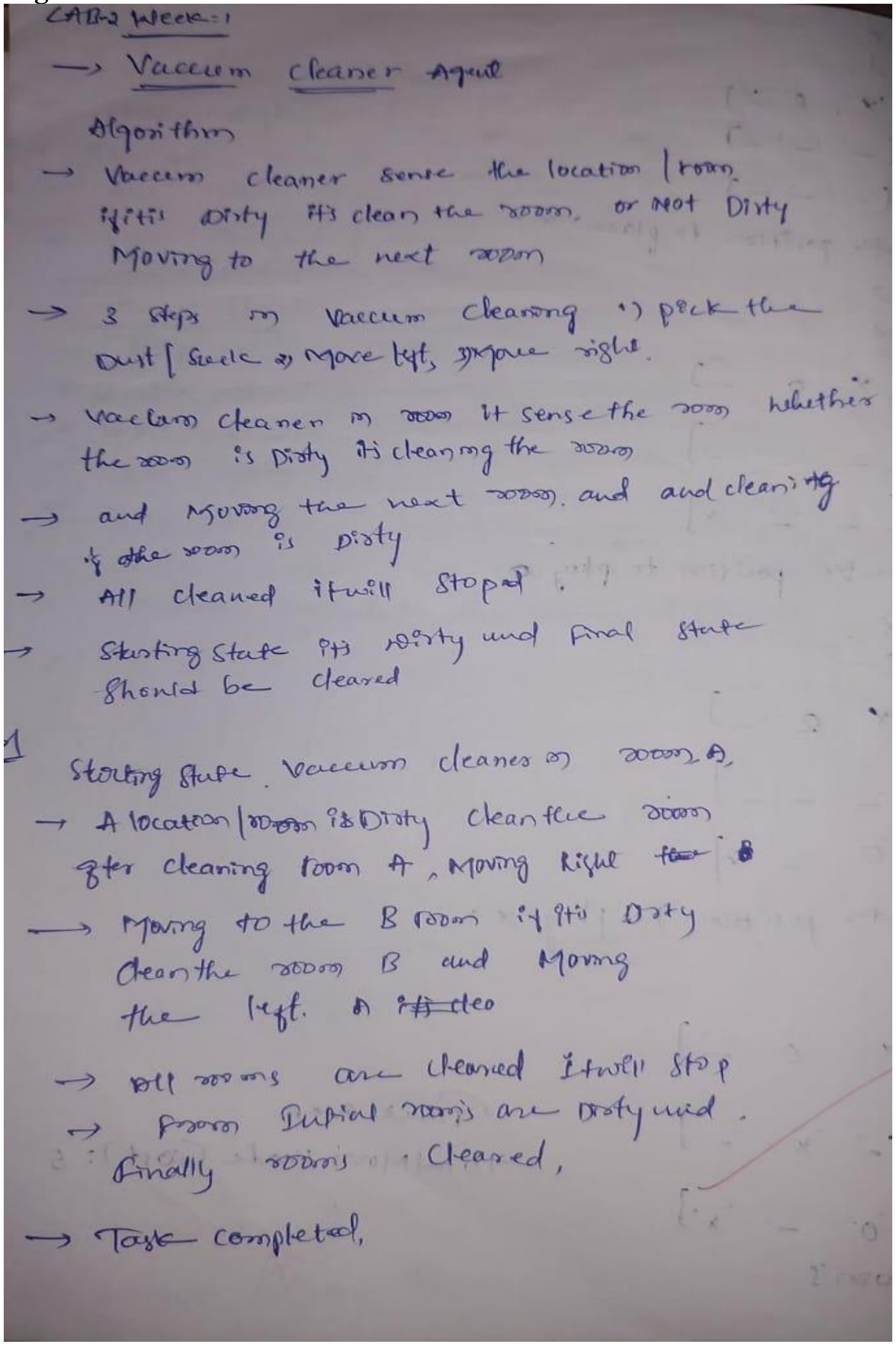
```
1
['x', 'o' '-']
['-' '-' '-']
['-' '-' '-']
enter position to place x:
2
2
[x  o  -]
[-  -  -]
[-  -  x]
Enter position to play o:
2
0
[x  o  -]
[-  -  -]
[o  -  x]
Enter position to play x:
1
1
[x  o  -]
[-  x  -]          Game over
                   Total Moves made (cost): 5.
[o  -  x]
x won's
```

[illegible]

Code:
```
print("Shankar ()")
def create_board():
    return [["-" for _ in range(3)] for _ in range(3)]

def display_board(board):
    for row in board:
        print(row)

def is_valid_move(board, row, col):
    return 0 <= row < 3 and 0 <= col < 3 and board[row][col] == "-"

def has_won(board, player):

    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or \
           all(board[j][i] == player for j in range(3)):
            return True

    if all(board[i][i] == player for i in range(3)) or \
       all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

def is_board_full(board):
    return all(cell != "-" for row in board for cell in row)

def tic_tac_toe():
    board = create_board()
    current_player = "X"
    move_count = 0

    while True:
        display_board(board)
        print(f"Enter position to place {current_player}:")

        try:
            row = int(input())
            col = int(input())
        except ValueError:
            print("Please enter valid integers for row and column.")
            continue

        if is_valid_move(board, row, col):
            board[row][col] = current_player
            move_count += 1

            if has_won(board, current_player):
                display_board(board)
                print(f"{current_player} wins!")
                print("Game Over")
                print(f"Total moves made (cost): {move_count}")
                break
            elif is_board_full(board):
                display_board(board)
                print("It's a draw!")
                print("Game Over")
                print(f"Total moves made (cost): {move_count}")
                break


            current_player = "O" if current_player == "X" else "X"
        else:
            print("Invalid move. Try again.")

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

Vaccum Cleaner Agent:

Algorithm:



LAB-2 Week:1

→ Vaccum Cleaner Agent

Algorithm

→ Vaccum cleaner sense the location | room
if it is Dirty it's clean the room, or Not Dirty
Moving to the next room

→ 3 Steps in Vaccum Cleaning 1) pick the
Dust | Suck 2) Move left, 3) Move right.

→ Vaccum cleaner in room it sense the room whether
the room is Dirty it's cleaning the room

→ and Moving the next room. and and cleaning
if the room is Dirty

→ All cleaned it will stoped .

→ Starting State it's Dirty and final state
should be cleared

1
Starting State. Vaccum cleaner in room A,

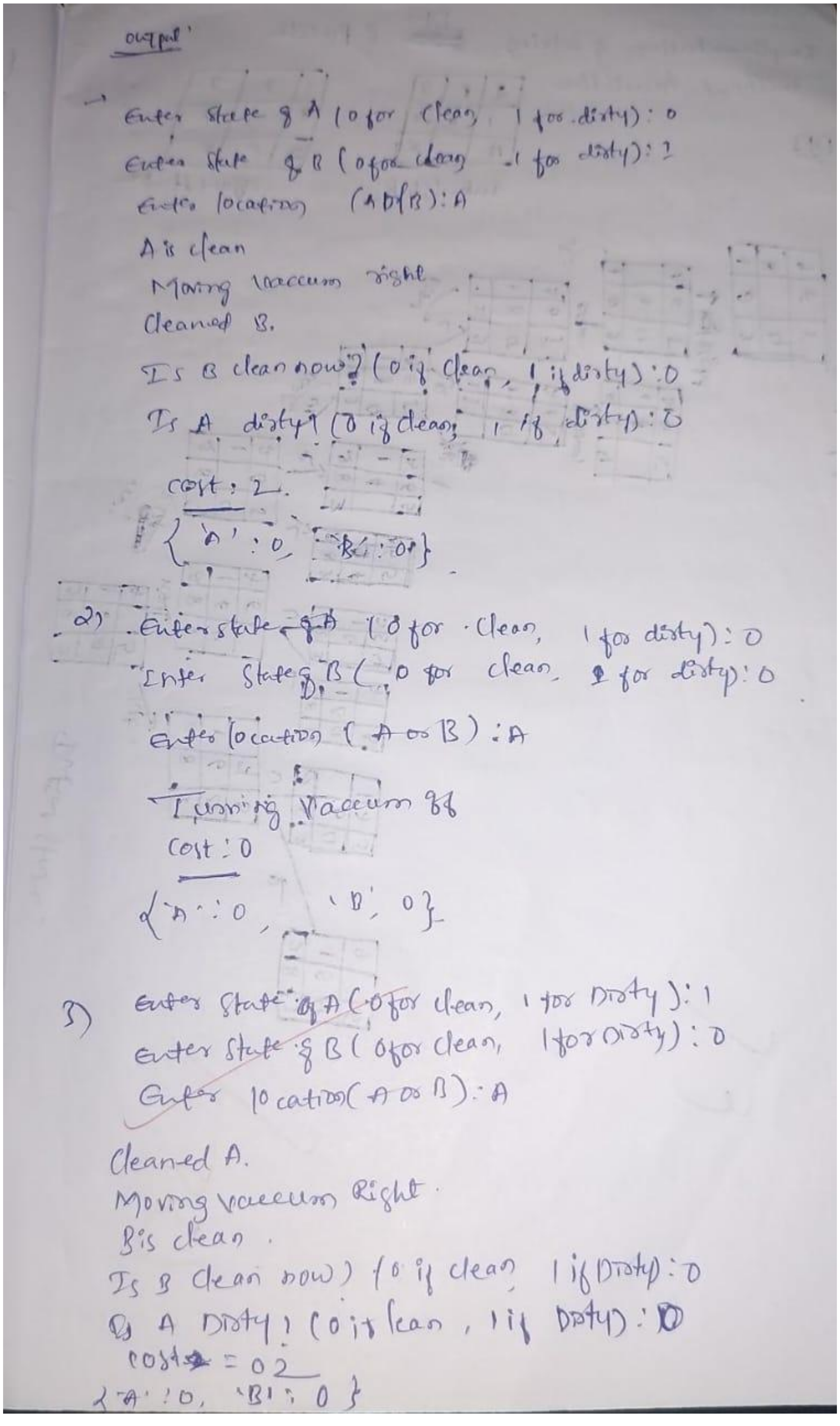
→ A location | room is Dirty clean the room
after cleaning room A, Moving Right ~~to~~ B

→ Moving to the B room if it is Dirty
Clean the room B and Moving
the left. A ~~its~~ cleo

→ All rooms are cleaned it will stop

→ from Initial room's are Dirty and.
finally rooms Cleared,

→ Task completed,



output:

```
1) Enter State of A (0 for clean, 1 for dirty): 0
   Enter State of B (0 for clean, 1 for dirty): 1
   Enter location (A or B): A
   A is clean
   Moving vaccum right
   Cleaned B.
   Is B clean now? (0 if clean, 1 if dirty): 0
   Is A dirty? (0 if clean, 1 if dirty): 0
   cost: 2.
   {'A': 0, 'B': 0}

2) Enter state of A (0 for clean, 1 for dirty): 0
   Enter State of B (0 for clean, 1 for dirty): 0
   Enter location (A or B): A
   Turning Vaccum off
   Cost: 0
   {'A': 0, 'B': 0}

3) Enter State of A (0 for clean, 1 for Dirty): 1
   Enter State of B (0 for clean, 1 for Dirty): 0
   Enter location (A or B): A
   Cleaned A.
   Moving vaccum Right.
   B is clean.
   Is B clean now) (0 if clean 1 if Dirty): 0
   Is A Dirty? (0 if clean, 1 if Dirty): 0
   cost = 02
   {'A': 0, 'B': 0}
```

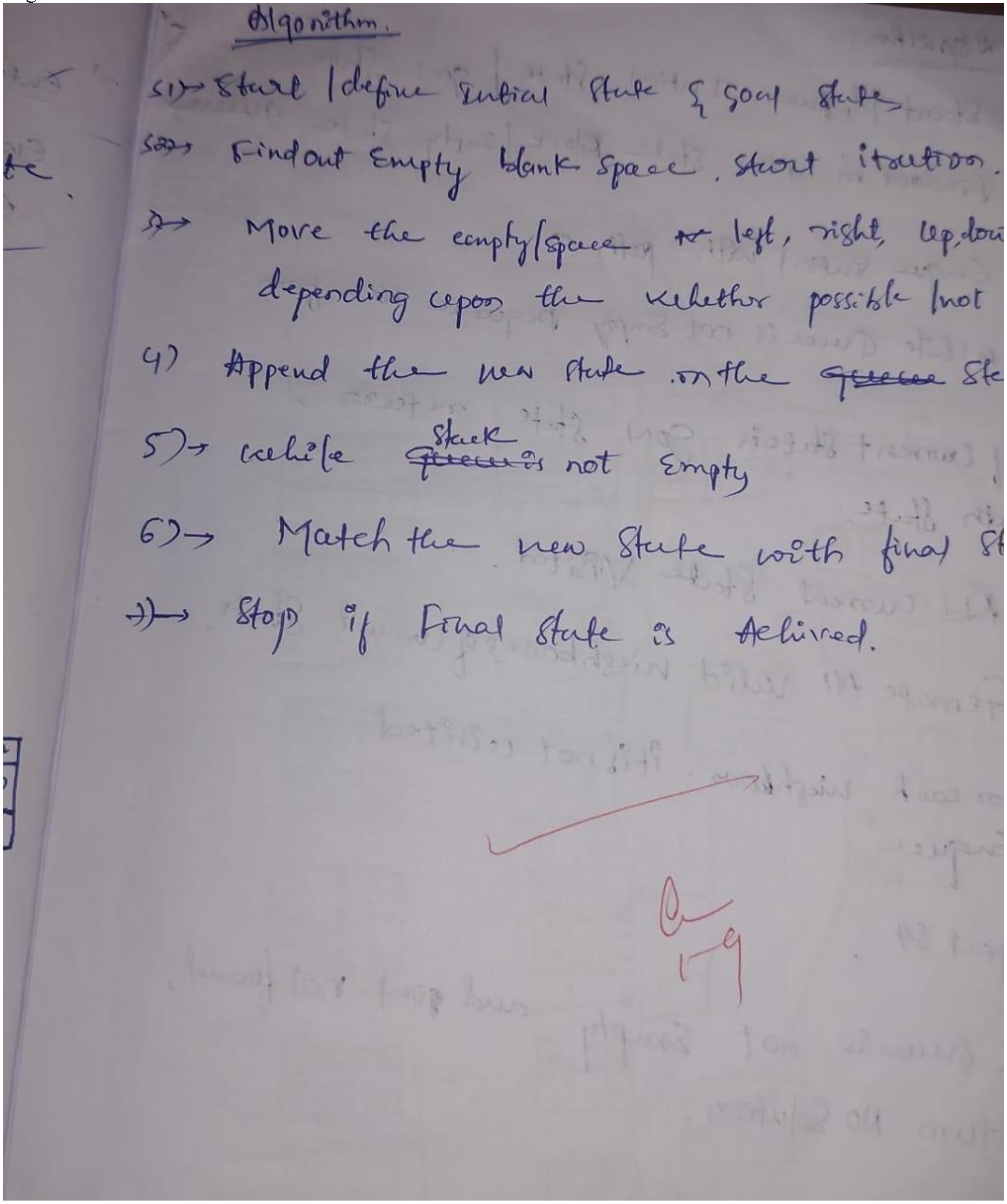
Code:
```
print("shanakr s pujar")
print("1BM23CS309")
def vacuum_cleaner_agent():
    # Take initial inputs
    state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    cost = 0
    states = {'A': state_A, 'B': state_B}

    if states['A'] == 0 and states['B'] == 0:
        print("Turning vacuum off")
        print(f"Cost: {cost}")
        print(states)
        return

    def clean_location(loc):
        nonlocal cost
        if states[loc] == 1:
            print(f"Cleaned {loc}.")
            states[loc] = 0
            cost += 1

    if location == 'A':
        if states['A'] == 1:
            clean_location('A')
        else:
            print("A is clean")
        print("Moving vacuum right")
        cost += 1
        if states['B'] == 1:
            clean_location('B')
        else:
            print("B is clean")

        print("Is B clean now? (0 if clean, 1 if dirty):", states['B'])
        print("Is A dirty? (0 if clean, 1 if dirty):", states['A'])

        if states['A'] == 1:
            print("Moving vacuum left")
            cost += 1
            clean_location('A')

    elif location == 'B':
        if states['B'] == 1:
            clean_location('B')
        else:
            print("B is clean")
        print("Moving vacuum left")
        cost += 1
        if states['A'] == 1:
            clean_location('A')
        else:
            print("A is clean")

        print("Is A clean now? (0 if clean, 1 if dirty):", states['A'])
        print("Is B dirty? (0 if clean, 1 if dirty):", states['B'])

        if states['B'] == 1:
            print("Moving vacuum right")
            cost += 1
            clean_location('B')

    print(f"Cost: {cost}")
    print(states)
vacuum_cleaner_agent()
```

Programa 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:



Algorithm.

S1)→ Start / define Initial State & Goal State

S2)→ Findout Empty blank space. Start iteration.

3)→ Move the empty/space ~~to~~ left, right, up, dow

depending upon the whether possible /not

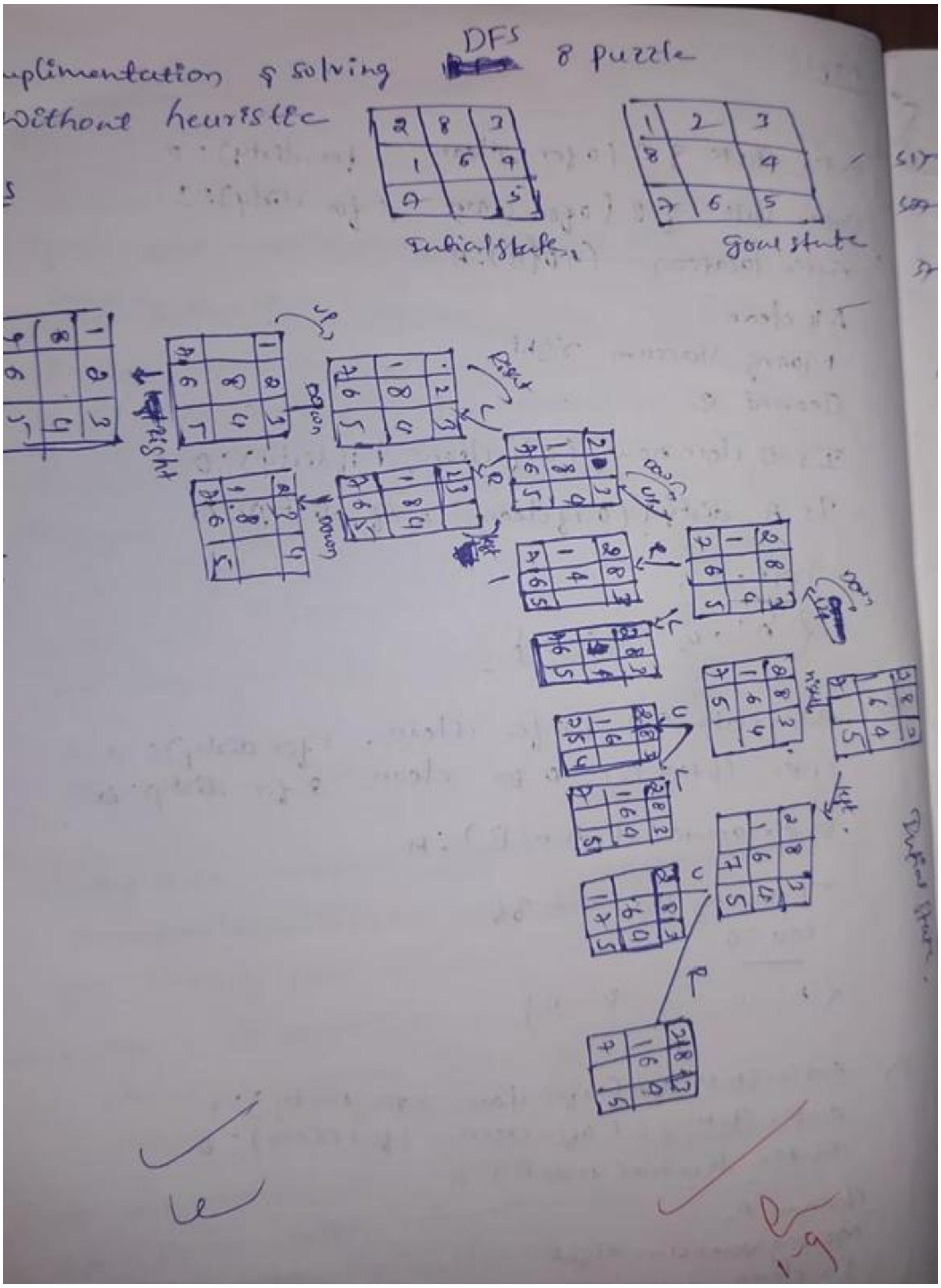
4) Append the new State on the ~~queue~~ St

5)→ while Stack ~~queue~~ is not Empty

6)→ Match the new State with final St

7)→ Stop if Final State is Achived.

[illegible]



uplimentation & solving DFS 8 puzzle

without heuristic

2	8	3
1	6	4
7		5

Initial state

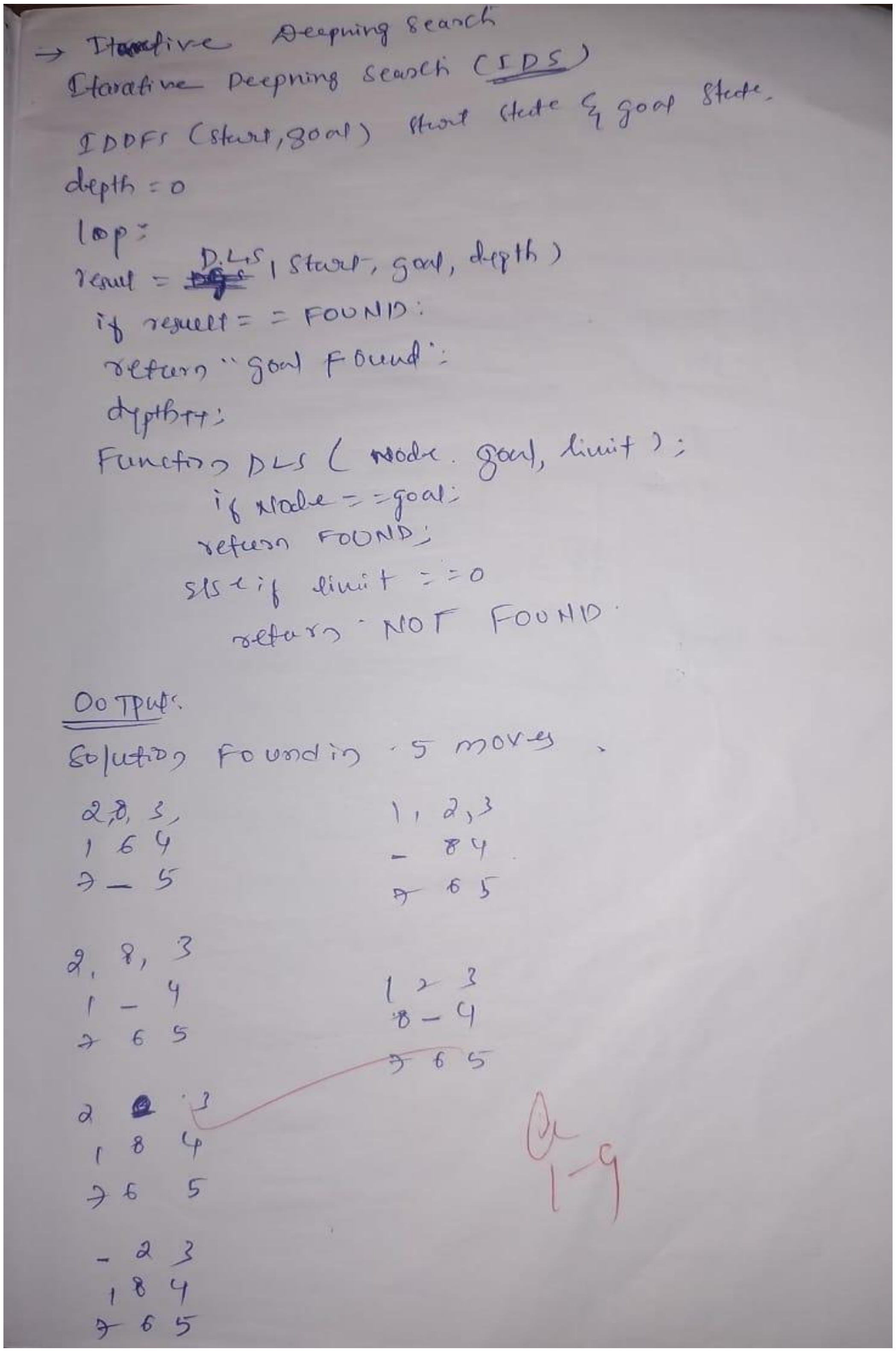
1	2	3
8		4
7	6	5

Goal state

Code:
```
from collections import deque

print("SHANKAR S PUJAR,1BM23CS309")

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

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

directions = {'Up': (-1, 0), 'Down': (1, 0), 'Left': (0, -1), 'Right': (0, 1)}

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

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    r1, c1 = pos1
    r2, c2 = pos2
    state_list[r1][c1], state_list[r2][c2] = state_list[r2][c2], state_list[r1][c1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    r, c = get_blank_pos(state)
    for move, (dr, dc) in directions.items():
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            new_state = swap_positions(state, (r, c), (nr, nc))
            neighbors.append((new_state, move))
    return neighbors

def dfs_8_puzzle(start, goal):
    stack = []
    visited = set()
    visited.add(start)
    stack.append((start, []))

    levels = []

    while stack:
        state, path = stack.pop()
        levels.append(state)

        if state == goal:
            return path, levels, len(visited)
        for neighbor, move in get_neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [move]))

    return None, levels, len(visited)

solution_path, level_states, total_visited = dfs_8_puzzle(initial_state, goal_state)

print(f"Solution length: {len(solution_path)} moves")
print("Solution moves:", solution_path)
print(f"Total states visited: {total_visited}\n")

print("States traversed:")
for i, state in enumerate(level_states):
    print(f"\nState {i+1}:")
    for row in state:
        print(row)
    print("---")
```

IDS

Algorithm:



→ ~~Itarative~~ Deepning search

Itarative Deepning search (IDS)

```
IDDFS (start, goal) start state & goal state.
depth = 0
loop:
result = D.L.S ~~DLS~~ ( start, goal, depth )
 if result = = FOUND:
 return "goal found":
 depth++;
 Function DLS ( node. goal, limit ):
        if node = = goal:
       return FOUND;
      else if limit = = 0
         return NOT FOUND.
```

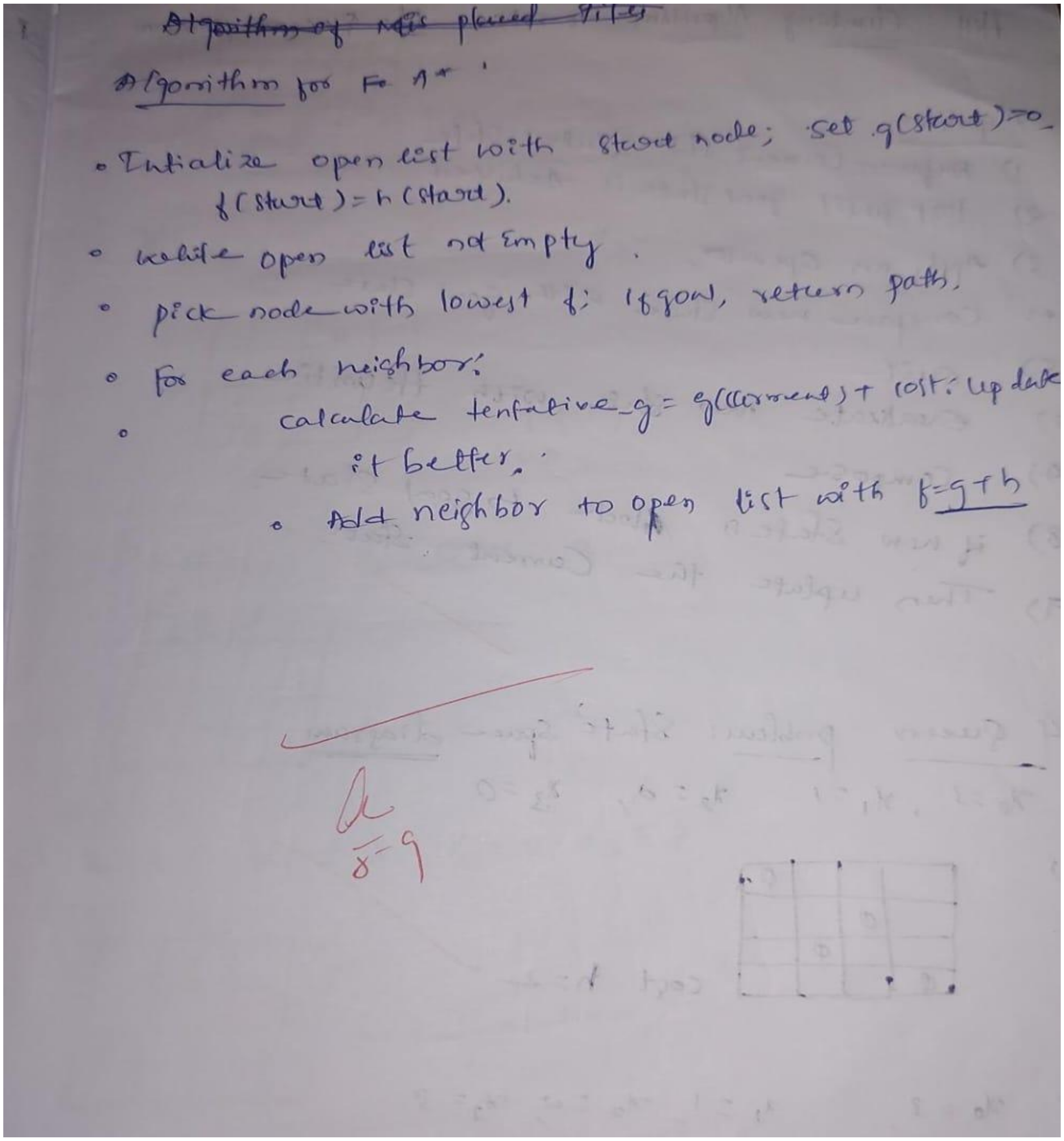
Output:

```
Solution found in 5 moves.

2, 8, 3,            1, 2, 3
1  6  4             _  8  4
7  _  5             7  6  5

2, 8, 3
1  _  4             1  2  3
7  6  5             8  _  4
                    7  6  5
2     3
1  8  4
7  6  5

_  2  3
1  8  4
7  6  5
```

[illegible] 1-9

Code:
```
goal_state = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

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

def swap_tiles(state, pos1, pos2):
    state_list = [list(row) for row in state]
    x1, y1 = pos1
    x2, y2 = pos2
    state_list[x1][y1], state_list[x2][y2] = state_list[x2][y2], state_list[x1][y1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    x, y = get_blank_pos(state)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            neighbors.append(swap_tiles(state, (x, y), (nx, ny)))
    return neighbors

def dls(state, goal, limit, visited, path):
    if state == goal:
        return path + [state]
    if limit <= 0:
        return None
    visited.add(state)
    for neighbor in get_neighbors(state):
        if neighbor not in visited:
            result = dls(neighbor, goal, limit - 1, visited, path + [state])
            if result is not None:
                return result
    visited.remove(state)
    return None

def iterative_deepening(start, goal, max_depth=50):
    for depth in range(max_depth):
        visited = set()
        result = dls(start, goal, depth, visited, [])
        if result is not None:
            return result
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else ' ' for x in row))
    print()

if __name__ == "__main__":
    start = ((2, 8, 3),
             (1, 6, 4),
             (7, 0, 5))

    solution = iterative_deepening(start, goal_state)

    if solution:
        print(f"Solution found in {len(solution) - 1} moves:\n")
        for step in solution:
            print_state(step)
    else:
        print("No solution found within depth limit.")
```

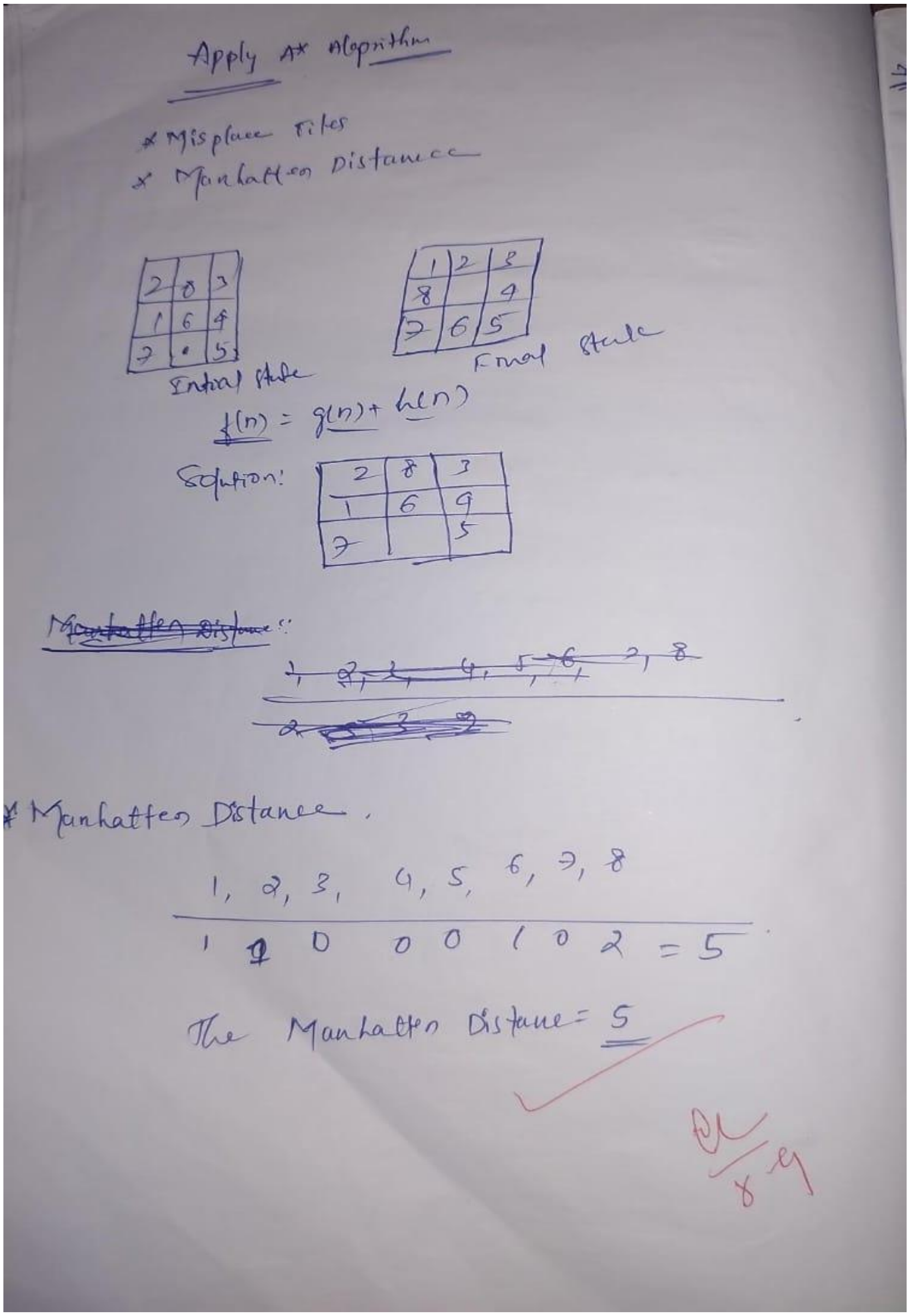
Program 3
Implement A* search algorithm
Algorithm:



~~Algorithm of Mis placed tiles~~

Algorithm for ~~Fo~~ A* :

- Intialize open list with start node; set g(start)=0
  f(start) = h(start).
- while open list not Empty .
- pick node with lowest f; if goal, return path.
- For each neighbor:
- calculate tentative_g = g(current) + cost; update it better.
- Add neighbor to open list with f=g+h



Apply A* Algorithm

* Misplace Tiles
* Manhatten Distance

2	8	3
1	6	4
7	•	5

Initial State

1	2	3
8		4
7	6	5

Final State

$f(n) = g(n) + h(n)$

Solution:

2	8	3
1	6	4
7		5

~~Manhatten Distance~~:

~~1, 2, 3, 4, 5, 6, 7, 8~~

~~1, 2, 1, 1, 1~~

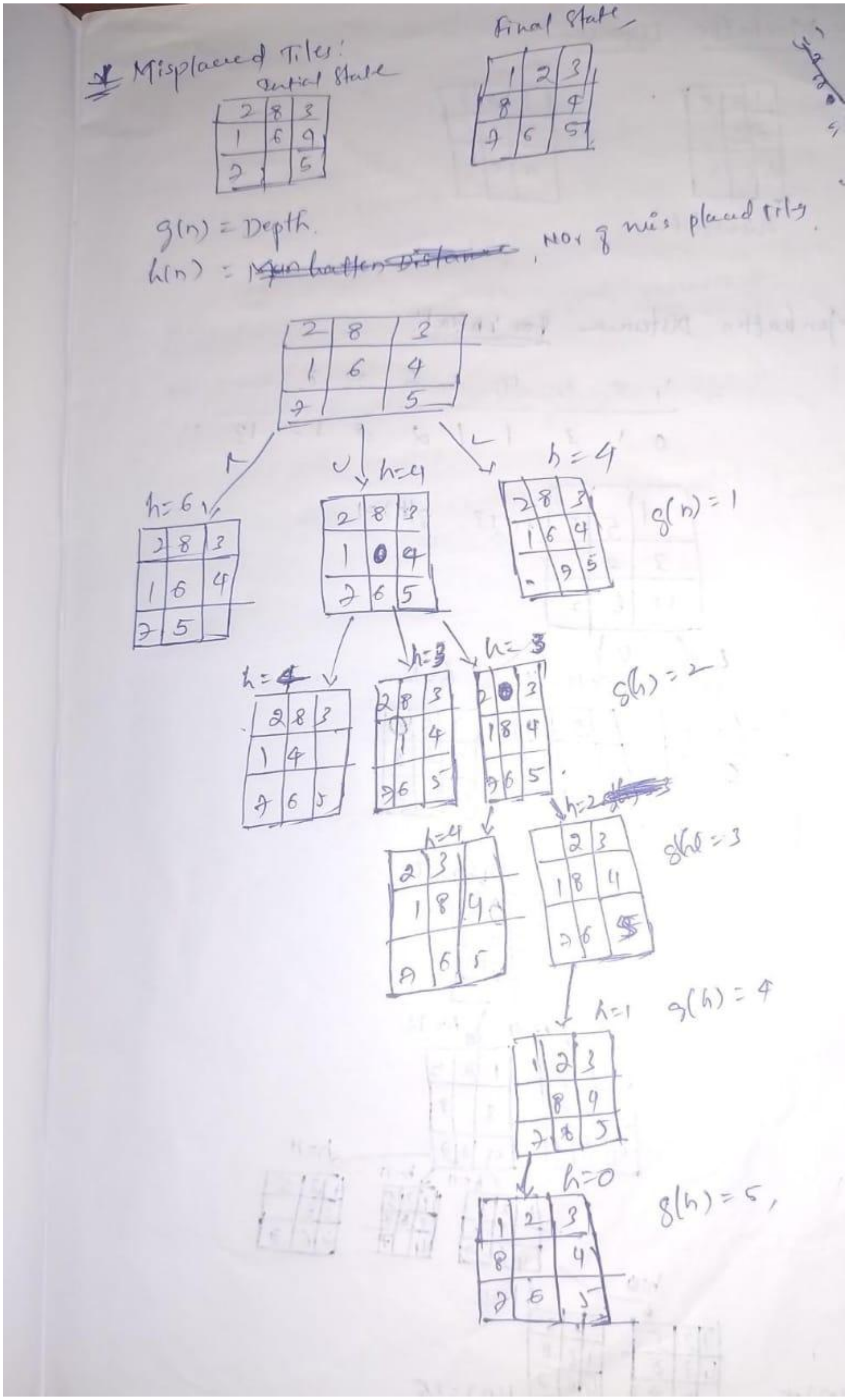
~~2 3 2~~

* Manhatten Distance.

$$\frac{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8}{1\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 2} = 5$$

The Manhatten Distance = 5

[illegible]



Misplaced Tiles:

Initial State

2	8	3
1	6	4
7		5

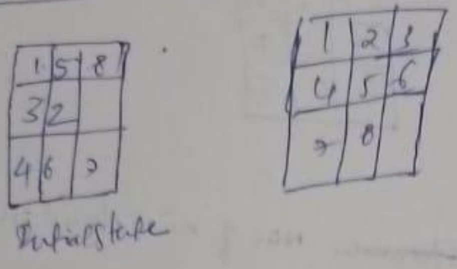
Final State

1	2	3
8		4
7	6	5

g(n) = Depth.

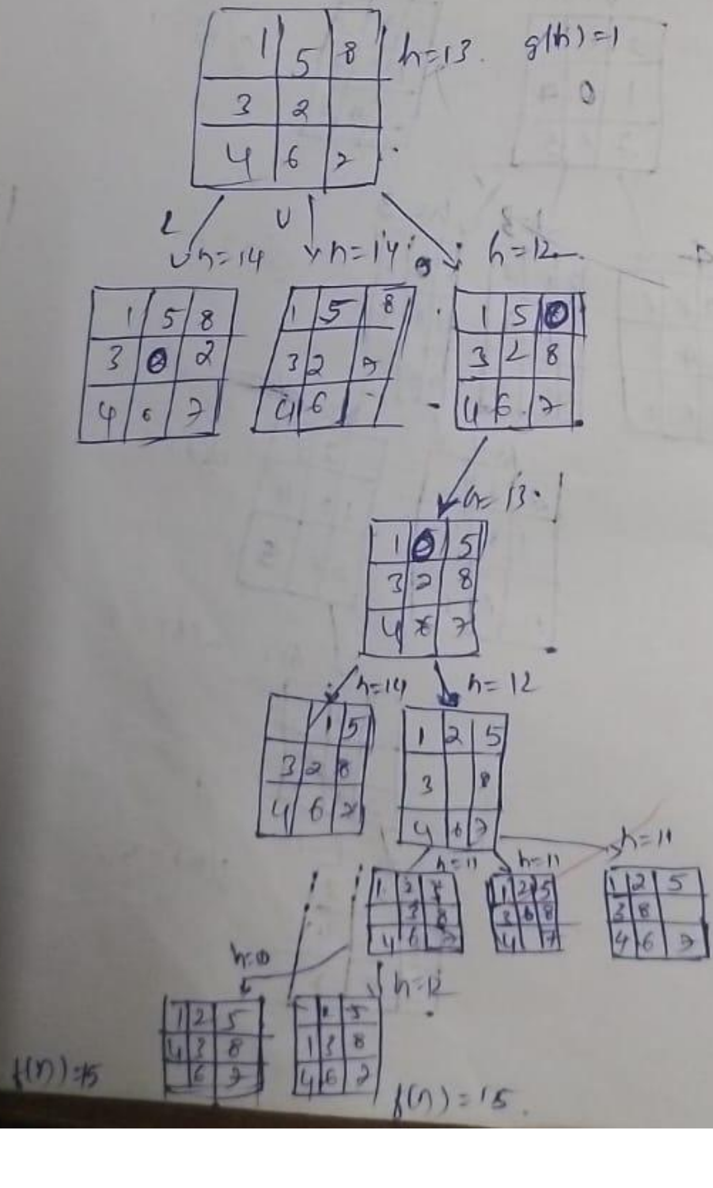
h(n) = ~~Manhattan Distance~~, No. of misplaced tiles.

* Manhatten Distance



Manhatten Distance for Initial:

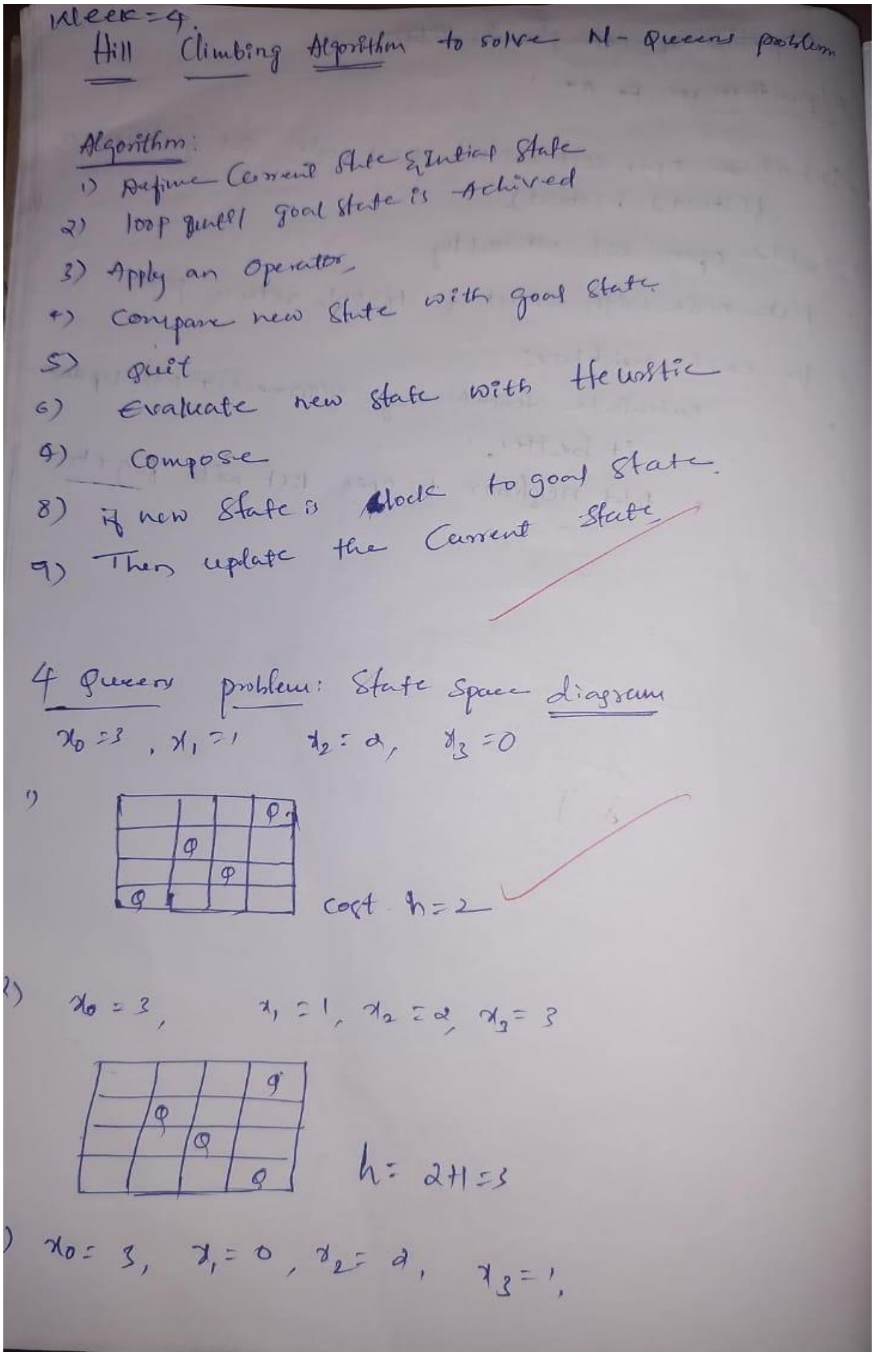
$$= \frac{1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7,\ 8,}{0\ \ 1\ \ 3\ \ 1\ \ 1\ \ 2\ \ 2\ \ 3} = 13,$$



Code:

```
import heapq
print("shankar.1BM23CS309")

def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                value = state[i][j]
                # Find the position of the value in the goal state
                for gi in range(3):
                    for gj in range(3):
                        if goal[gi][gj] == value:
                            goal_pos = (gi, gj)
                            break
                    else:
                        continue
                    break
                distance += abs(i - goal_pos[0]) + abs(j - goal_pos[1])
    return distance

def get_neighbors(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break
        else:
            continue
        break

    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    for dx, dy in moves:
        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 astar_search_manhattan(initial, goal):
    frontier = [(manhattan_distance(initial, goal), 0, initial)]
    explored = set()
    parent = {}
    cost = {initial: 0}

    while frontier:
        f, g, current = heapq.heappop(frontier)

        if current == goal:
            path = []
            while current in parent:
                path.append(current)
                current = parent[current]
            path.append(initial)
            return path[::-1]

        explored.add(current)

        for neighbor in get_neighbors(current):
            new_cost = cost[current] + 1
            if neighbor not in cost or new_cost < cost[neighbor]:
                cost[neighbor] = new_cost
                priority = new_cost + manhattan_distance(neighbor, goal)
                heapq.heappush(frontier, (priority, new_cost, neighbor))
                parent[neighbor] = current
    return None

def get_state_input(prompt):
    print(prompt)
    state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        state.append(row)
    return tuple(tuple(row) for row in state)

initial_state_m = get_state_input("Enter the initial state for Manhattan distance (3 rows of 3 numbers separated by spaces, use 0 for the blank):")
goal_state_m = get_state_input("Enter the goal state for Manhattan distance (3 rows of 3 numbers separated by spaces, use 0 for the blank):")

path_m = astar_search_manhattan(initial_state_m, goal_state_m)

if path_m:
    print("Solution found using Manhattan distance:")
    for step in path_m:
        for row in step:
            print(row)
        print()
else:
    print("No solution found using Manhattan distance.")
```

Program 4
Implement Hill Climbing search algorithm to solve N-Queens problem
Algorithm:



Week=4.

Hill Climbing Algorithm to solve N-Queens problem

Algorithm:
1) Define Current State & Intial State
2) loop until goal state is Achived
3) Apply an Operator,
4) Compare new State with goal State
5) quit
6) Evaluate new state with Heuristic
7) Compose
8) If new State is close to goal State.
9) Then update the Current State

4 Queens problem: State Space diagram

$x_0 = 3$, $x_1 = 1$ $x_2 = 2$, $x_3 = 0$

1)

			Q
	Q		
		Q	
Q			

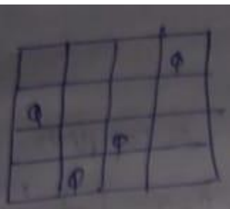
cost $h = 2$

2) $x_0 = 3$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$

			Q
	Q		
		Q	
			Q

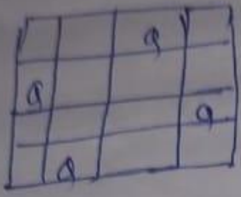
$h = 2+1 = 3$

3) $x_0 = 3$, $x_1 = 0$, $x_2 = 2$, $x_3 = 1$,



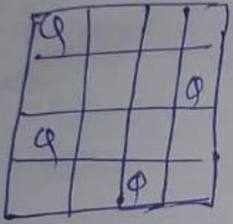
$h=1.$

4) $x_0 = 2 \quad x_1 = 0 \quad x_2 = 3 \quad x_3 = 1$



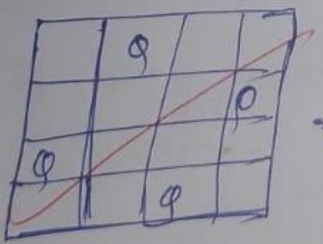
$\Rightarrow$ Solution
$h=0.$

5) $x_0 = 0 \quad x_1 = 3 \quad x_2 = 0 \quad x_3 = 2$



$h=1$

6) $x_0 = 1 \quad x_1 = 3 \quad x_2 = 0 \quad x_3 = 2,$



$\Rightarrow$ Solution
$h=0$

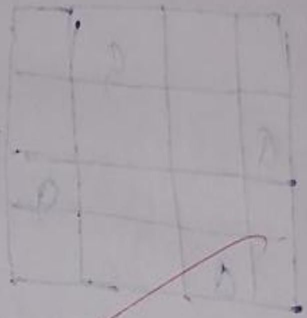
15-4-25

OUTPUT:

Intial board: [0, 3, 3, 1] with 2 conflicts.

Best ~~Found~~ · ~~St~~ Improved Found: [2, 0, 3, 1] with 0 conflicts.

Final Solution: Queens placed so no two attack each other

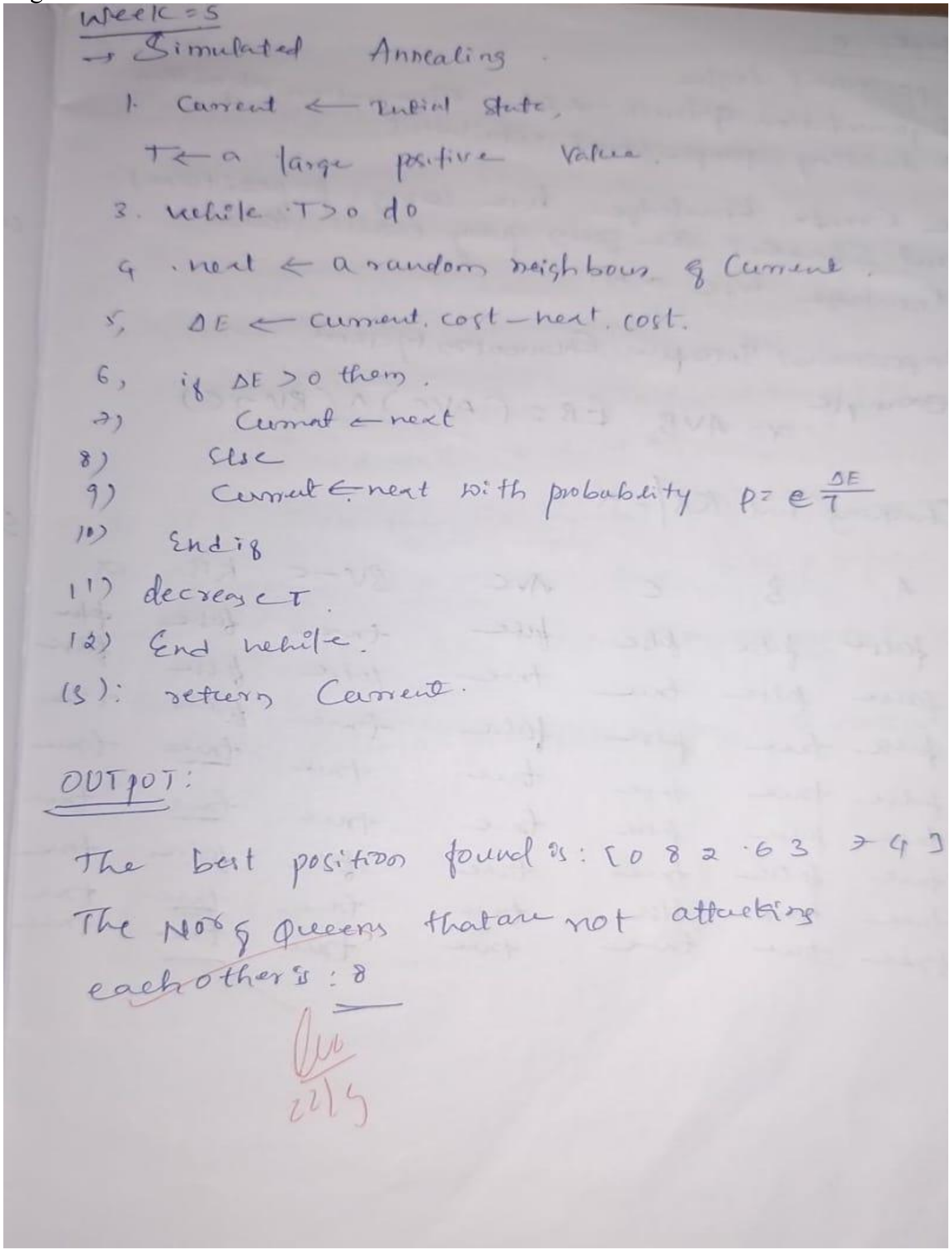


Code:
```
import random

def create_board():
    return [random.randint(0, 3) for _ in range(4)]

def calculate_conflicts(board):
    conflicts = 0
    for i in range(4):
        for j in range(i + 1, 4):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

def hill_climbing():
    board = create_board()
    print(f"Initial board: {board} with conflicts: {calculate_conflicts(board)}")

    while True:
        current_conflicts = calculate_conflicts(board)
        if current_conflicts == 0:
            return board
        next_board = None
        next_conflicts = float('inf')
        for i in range(4):
            temp_board = board[:]
            for j in range(4):
                if temp_board[i] != j:
                    temp_board[i] = j
                    temp_conflicts = calculate_conflicts(temp_board)
                    print(f"Board: {temp_board} with conflicts: {temp_conflicts}")
                    if temp_conflicts < next_conflicts:
                        next_conflicts = temp_conflicts
                        next_board = temp_board[:]
        if next_conflicts >= current_conflicts:
            return board
        board = next_board

solution = hill_climbing()
print(f"Final solution: {solution}")
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Week = 5

→ Simulated Annealing.

```
1. Current ← Initial State,
   T ← a large positive Value.
3. while T > 0 do
4.   next ← a random neighbour of Current.
5,   $\Delta E$ ← Current. cost − next. cost.
6,   if $\Delta E > 0$ then.
7)      Current ← next
8)   else
9)      Current ← next with probability $p = e^{\frac{\Delta E}{T}}$
10)  Endif
11) decrease T.
12) End while.
13): return Current.
```

OUTPUT:

```
The best position found is: [0 8 2 ·6 3 7 4 ]
The No of Queens that are not attacking
each other is : 8
```

[illegible]
22/5

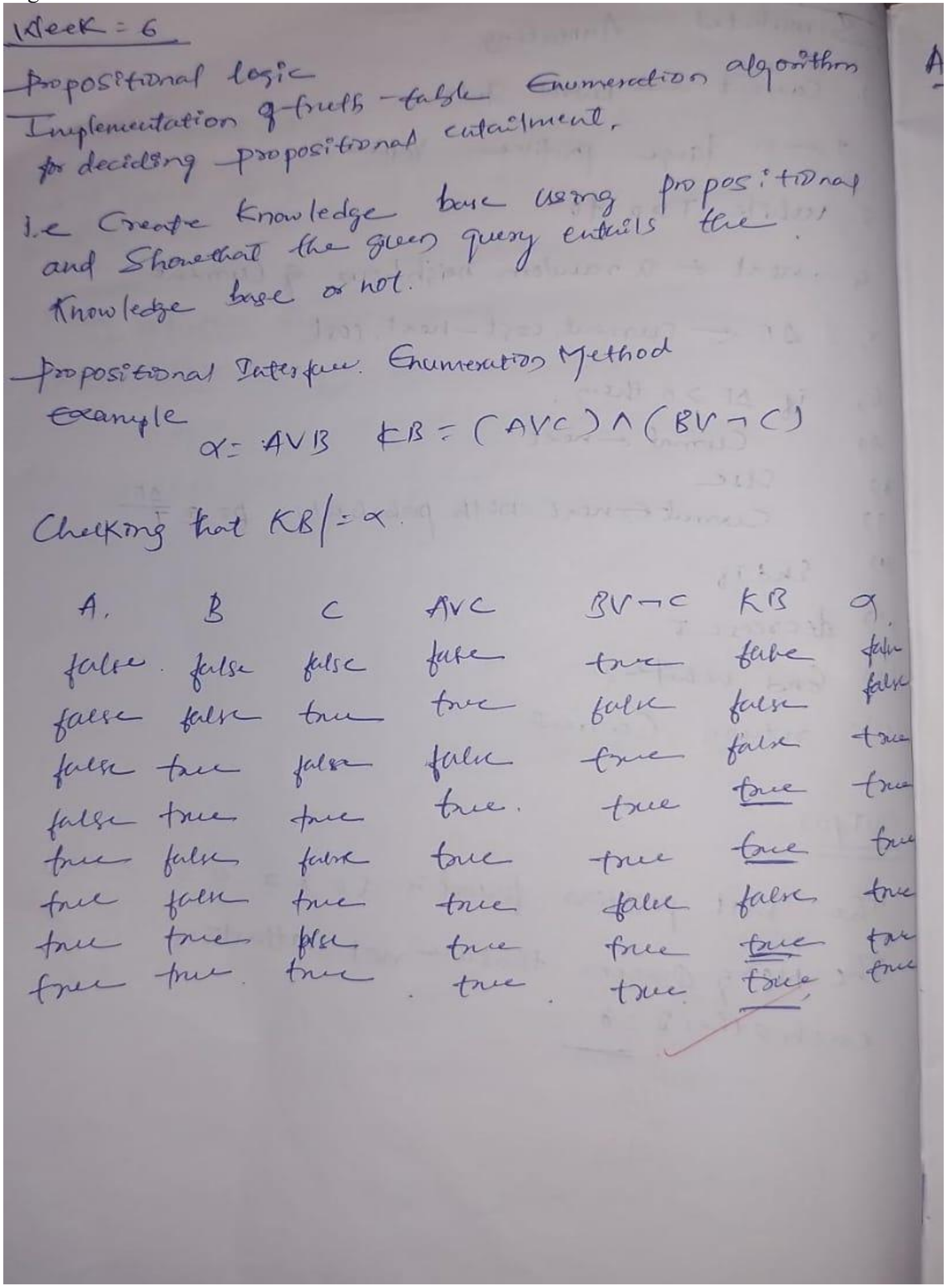
Code:

```
print("Shankar,1BM23CS309")
import math
import random

def objective_function(x):
   return x**2 + 10 * math.sin(x)

def simulated_annealing(objective, bounds, max_iterations, initial_temp, cooling_rate):
   # Random initial solution
   current = random.uniform(bounds[0], bounds[1])
   current_energy = objective(current)

   best = current
   best_energy = current_energy

   temp = initial_temp

   for i in range(max_iterations):
      candidate = current + random.uniform(-1, 1)  # small random move
      candidate = max(min(candidate, bounds[1]), bounds[0])  # keep inside bounds
      candidate_energy = objective(candidate)


      delta_e = candidate_energy - current_energy

      if delta_e < 0 or random.random() < math.exp(-delta_e / temp):
         current, current_energy = candidate, candidate_energy

        t
         if current_energy < best_energy:
            best, best_energy = current, current_energy

      temp = temp * cooling_rate


   return best, best_energy

bounds = [-10, 10]
max_iterations = 1000
initial_temp = 100.0
cooling_rate = 0.99
best_solution, best_value = simulated_annealing(objective_function, bounds,
                                     max_iterations, initial_temp, cooling_rate)
print("Best solution found: x =", best_solution)
print("Objective function value:", best_value)
```

Program 6

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

Algorithm:



Week = 6

- propositional logic

Implementation of truth-table Enumeration algorithm for deciding propositional entailment.

i.e Create knowledge base using propositional and Showthat the given query entails the knowledge base or not.

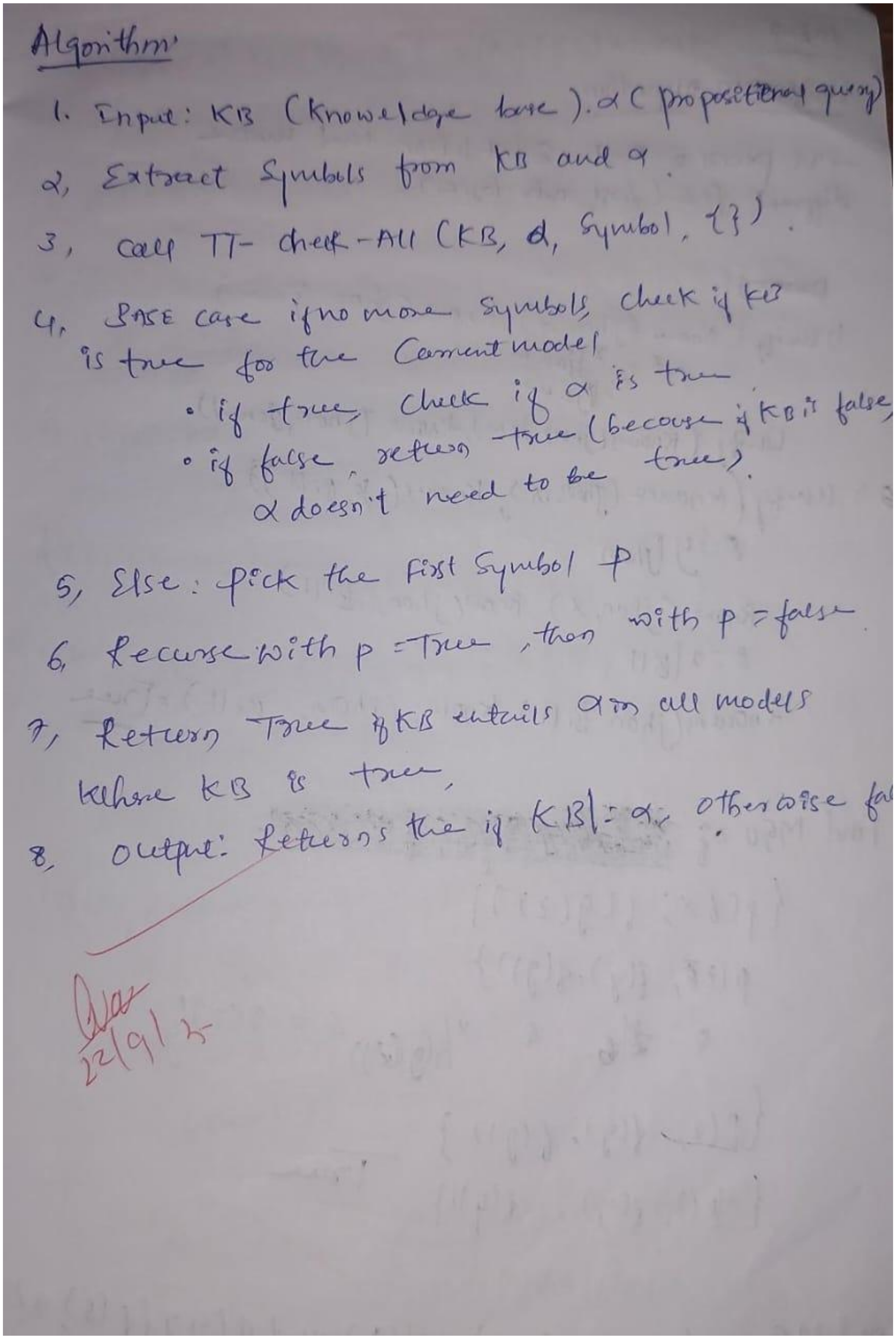
- propositional Interface Enumeration Method

Example

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

Checking that $KB \models \alpha$.

A.	B	C	$A \vee C$	$B \vee \neg C$	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



Algorithm:

1. Input: KB (Knowledge base). α (propositional query)
2, Extract Symbols from KB and α.
3, call TT- check-All (KB, α, Symbol, {}).
4, BASE case if no more symbols, check if KB is true for the Current model
    - if true, check if α is true
    - if false, return true (because if KB is false, α doesn't need to be true).
5, Else: pick the First Symbol P
6, Recurse with P = True, then with P = false.
7, Return True if KB entails α in all models where KB is true,
8, output: Returns the if $KB \models \alpha$, otherwise fal

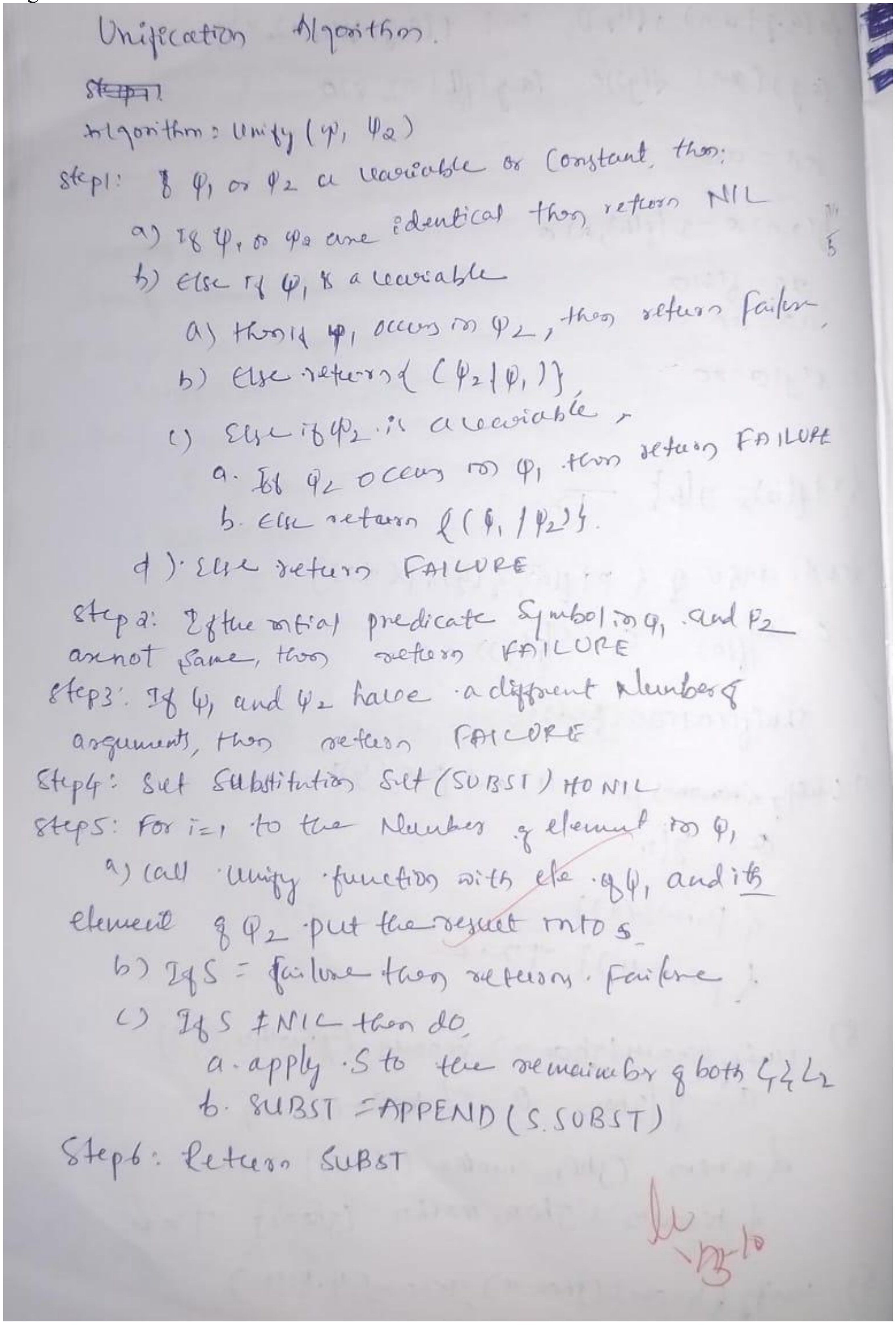
[illegible]
22/9/25

Code:

```
print("shankar,1  BM23CS309")
import itertools
import re

def pl_true(expr, model):
    """
    Evaluate a propositional logic expression in a given model.
    expr: string with symbols and logical operators (¬, ∧, ∨, →, ↔)
    model: dictionary mapping symbols to True/False
    """

    for sym, val in model.items():
        expr = re.sub(rf'\b{sym}\b', str(val), expr)


    expr = expr.replace("¬", " not ")
    expr = expr.replace("∧", " and ")
    expr = expr.replace("∨", " or ")
    expr = expr.replace("→", " <= ")
    expr = expr.replace("↔", " == ")

    P → Q = (not P) or Q
    while "<=" in expr:
        expr = re.sub(r'(True|False)\s*<=\s*(True|False)',
                      lambda m: str((not eval(m.group(1))) or eval(m.group(2))),
                      expr)


    return eval(expr)

def tt_entails(KB, alpha, symbols):
    return tt_check_all(KB, alpha, symbols, {})

def tt_check_all(KB, alpha, symbols, model):
    if not symbols:  # No more symbols → full model
        if pl_true(KB, model):
            return pl_true(alpha, model)
        else:
            return True
    else:
        P = symbols[0]
        rest = symbols[1:]
        return (tt_check_all(KB, alpha, rest, {**model, P: True}) and
                tt_check_all(KB, alpha, rest, {**model, P: False}))
```

```
KB = "(A ∨ B) ∧ (¬C)"
alpha = "A ∨ (B ∧ ¬C)"

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

result = tt_entails(KB, alpha, symbols)
print("Does KB ⊨ α ?", result)
```

Program 7:
Implement unification in first order logic
Algorithm:



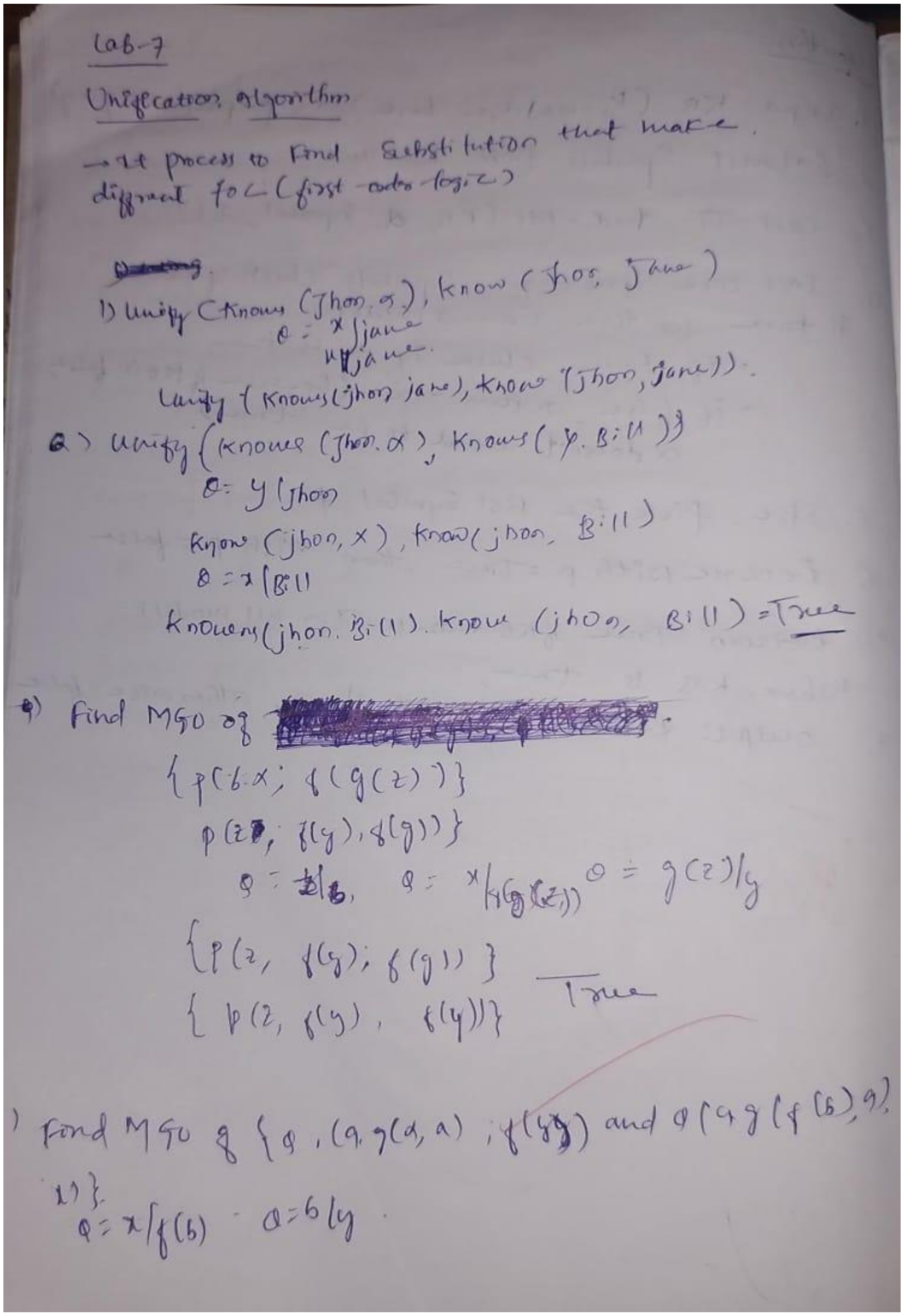
Unification Algorithm.

~~Step 1~~

Algorithm: Unify (Ψ_1, Ψ_2)

```
Step1: If Ψ1 or Ψ2 a variable or Constant, then:
    a) If Ψ1 or Ψ2 are identical then return NIL
    b) Else if Ψ1 is a variable
        a) then if Ψ1 occurs in Ψ2, then return failure,
        b) Else return{ (Ψ2 | Ψ1) },
    c) Else if Ψ2 is a variable,
        a. If Ψ2 occurs in Ψ1 then return FAILURE
        b. Else return {(Ψ1 | Ψ2)}.
    d). Else return FAILURE
Step2: If the initial predicate Symbol in Ψ1 and Ψ2
  are not same, then return FAILURE
Step3. If Ψ1 and Ψ2 have a different Number of
  arguments, then return FAILURE
Step4: Set Substitution Set (SUBST) to NIL
Step5: For i=1 to the Number of element in Ψ1,
    a) call Unify function with ele. of Ψ1 and ith
  element of Ψ2 put the result into s
    b) If S = failure then return. Failure
    c) If S ≠ NIL then do,
        a. apply S to the remainder of both L1 & L2
        b. SUBST = APPEND (S. SUBST)
Step6: Return SUBST
```

[illegible]



Lab-7

Unification algorithm

→ It process to Find Substitution that make different FOL (first order logic)

~~[illegible]~~

1) Unify (Knows (Jhon, x), know (Jhon, Jane)

$\theta = x / jane$

$x / jane$.

Unify (Knows(jhon jane), know (Jhon, jane)).

2) Unify (Knows (Jhon, x), Knows (y, Bill))

$\theta = y / jhon$

Know (jhon, x), know (jhon, Bill)

$\theta = x / Bill$

Knowns (jhon, Bill). know (jhon, Bill) = True

3) Find MGU of ~~[illegible]~~.

$\{ p(b, x; f(g(z)) \}$

$p(z, f(y), f(g)) \}$

$\theta = z/b, \quad \theta = x / f(g(z)) \quad \theta = g(z)/y$

$\{ p(z, f(y); f(g)) \}$ True

$\{ p(z, f(y), f(y)) \}$

) Find MGU of $\{ Q, (a, g(x, a) ; f(y)) \}$ and $Q(a, g(f(b), a)$, $x) \}$.

$\theta = x / f(b) \quad \theta = b / y$.

{o (a.g (x.a) , f(y)), and p(a,g(f(b) a), x)}

(a.g) (x.a) , f(y))θ = (a.g (f(b), a), x)θ

1) aθ = aθ.

2) g(x.a) θ = g (f(b), a) θ.

xθ = f(b)θ
aθ = aθ

3) f(y)θ = xθ

{x/f(b), y/b} True

3) find MGU of { p(f(a), g(y)), p(x, x)}

θ = x/f(a)   θ = x/g(y))

Unification False

(4) Unify { ~~knows~~ prime(11) and prime (y)}

θ = y/11

{prime(11)}

{prime(11)} True

5) Unify {knows (Jhon x) knows (y mother (y))}

θ = y/Jhon,   θ = x/ mother (Jhon)

{knows (Jhon, mother (Jhon))}

{knows (Jhon, mother (Jhon)} True

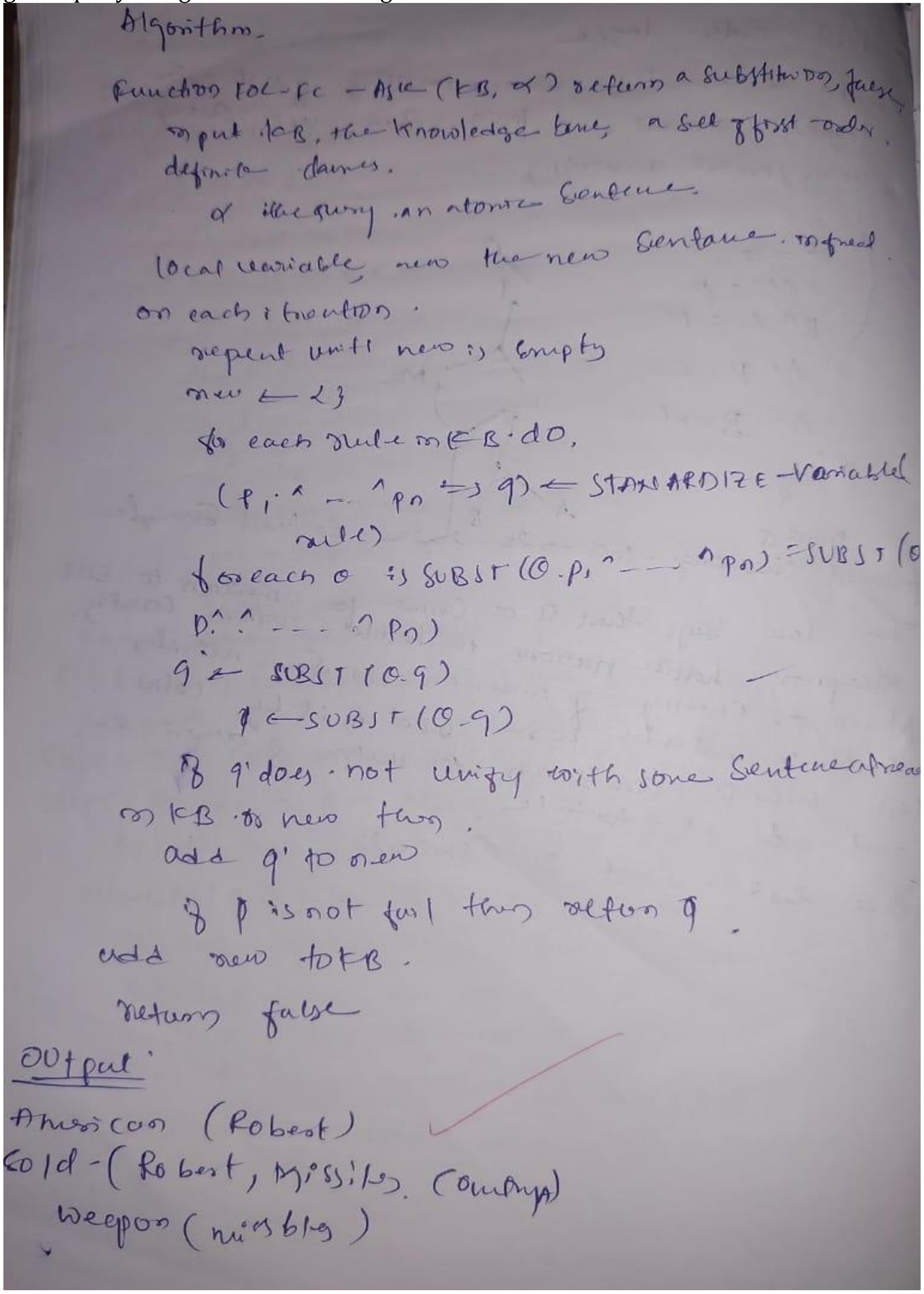
6) unify {knows (Jhon x), knows (y.Bill)

Code:
```
def unify(x, y, subs=None):
    if subs is None:
        subs = {}

    if isinstance(x, str) and x in subs:
        return unify(subs[x], y, subs)
    if isinstance(y, str) and y in subs:
        return unify(x, subs[y], subs)

    if x == y:
        return subs
    if isinstance(x, str) and x.isupper():
        subs[x] = y
        return subs

    if isinstance(y, str) and y.isupper():
        subs[y] = x
        return subs

    if isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
        for a, b in zip(x[1], y[1]):
            subs = unify(a, b, subs)
            if subs is None:
                return None
        return subs


    return None


expr1 = ('p', ['b', 'X', ('f', [('g', ['Z'])])])
expr2 = ('p', ['Z', ('f', ['Y']), ('f', ['Y'])])

mgu = unify(expr1, expr2)
print("Most General Unifier (MGU):", mgu)
```

Program 8

Algorithm:

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) return a substitution, false
    input KB, the knowledge base, a set of first order
    definite clauses.
        α the query, an atomic sentence.
  local variable, new the new sentence inferred
  on each iteration.
      repeat until new is empty
      new ← {}
        for each rule in KB do,
          (p1 ^ ... ^ pn ⇒ q) ← STANDARDIZE-Variable
              rule)
        for each θ is SUBST (θ.p1 ^ ... ^ pn) = SUBST (θ
        p1' ^ ... ^ pn')
      q' ← SUBST (θ.q)
          φ ← SUBST (θ.q)
        if q' does not unify with some sentence already
    in KB to new then.
      add q' to new
        if φ is not fail then return φ.
    add new to KB.
    return false
```

Output

```
American (Robert)
Sold-( Robert, Missiles, Country)
  Weapon ( missiles )
```

Enemy (country A, America)
Has (country A, Missiles)

Forward chaining proof (contd.)

$$\begin{tikzcd}
\boxed{\text{American}} \arrow[ddr] & \boxed{\text{Missiles}(T_1)} \arrow[d] \arrow[dr] & \boxed{\text{Owns}(A, T_1)} \arrow[d] & \boxed{\text{Enemy}(A, \text{Americ}} \arrow[d] \\
 & \boxed{\text{Weapons}(T_1)} \arrow[d] & \boxed{\text{Sells}(\text{Robert}, T_1, A))} \arrow[dl] & \boxed{\text{Hostile}(A} \arrow[dll] \\
 & \boxed{\text{Robert}(\text{Criminal})} & &
\end{tikzcd}$$

American(p) $\wedge$ Weapon
Sells(p, q, r) $\wedge$ Hostile(r

$\Rightarrow$ Criminal(p)

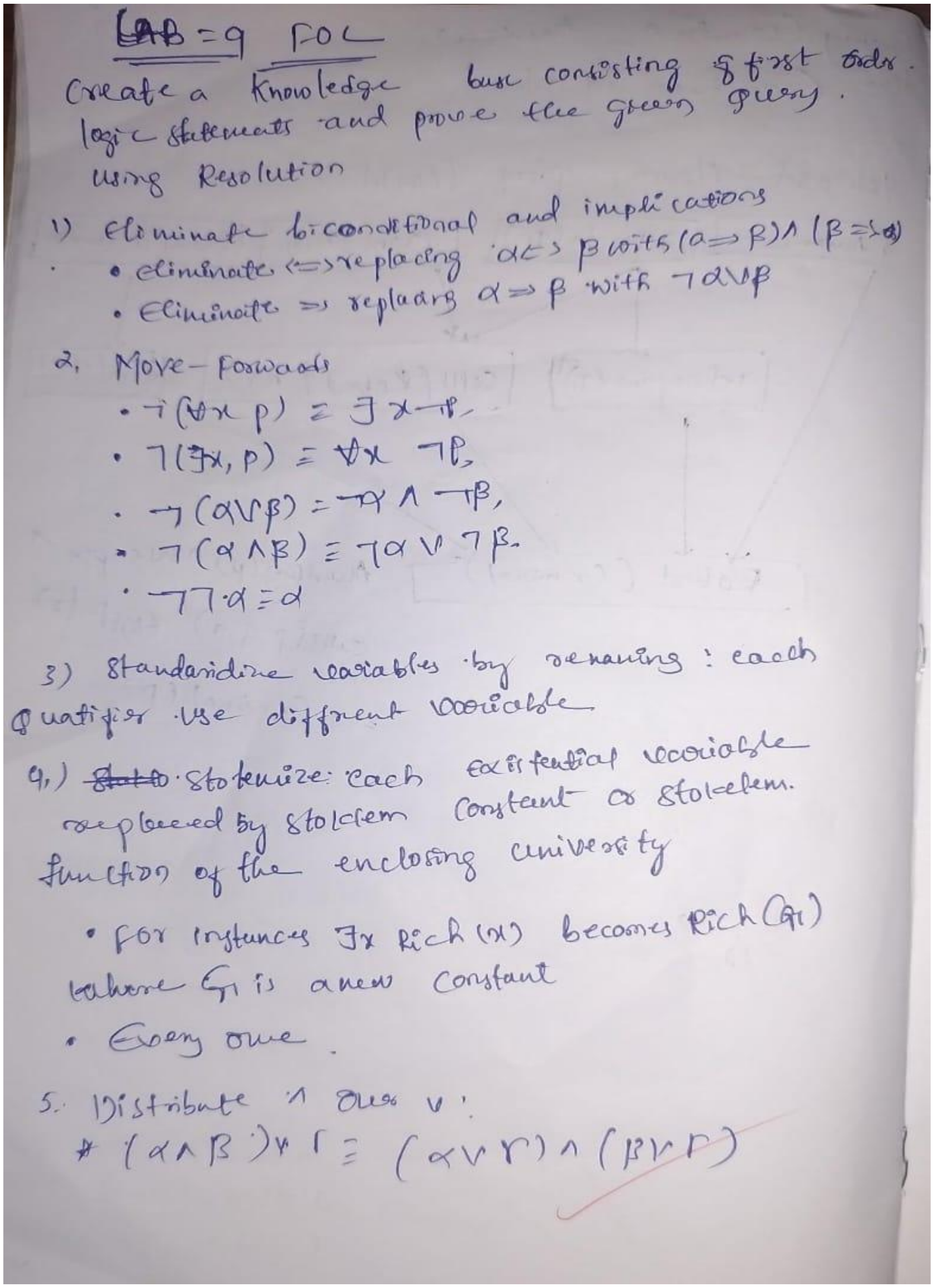
[illegible] 15-10

Code:
```
facts = set()
rules = []

facts.update(["American(West)",
              "Missile(M1)",
              "Owns(Nono, M1)",
              "Enemy(Nono, America)"])

def missile_is_weapon():
    if "Missile(M1)" in facts:
        facts.add("Weapon(M1)")

def enemy_is_hostile():
    if "Enemy(Nono, America)" in facts:
        facts.add("Hostile(Nono, America)")

def sells_relation():
    if "Missile(M1)" in facts and "Owns(Nono, M1)" in facts:
        facts.add("Sells(West, M1, Nono)")

def criminal_rule():
    if ("American(West)" in facts and
        "Weapon(M1)" in facts and
        "Sells(West, M1, Nono)" in facts and
        "Hostile(Nono, America)" in facts):
        facts.add("Criminal(West)")

rules = [missile_is_weapon, enemy_is_hostile, sells_relation, criminal_rule]

new = True
while new:
    new = False
    before = len(facts)
    for r in rules:
        r()
    after = len(facts)
    if after > before:
        new = True

for f in facts:
    print(f)

if "Criminal(West)" in facts:
    print("\nColonel West is a Criminal.")
else:
    print("\nCould not prove West is a Criminal.")
```

Program 9

Algorithm:

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



LAB = 9 FOL

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

1) Eliminate biconditional and implications
- eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)$
- Eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \lor \beta$

2. Move-forwards
- $\neg(\forall x\ p) \equiv \exists x \neg p$,
- $\neg(\exists x, p) \equiv \forall x\ \neg p$,
- $\neg(\alpha \lor \beta) \equiv \neg \alpha \land \neg \beta$,
- $\neg(\alpha \land \beta) \equiv \neg \alpha \lor \neg \beta$.
- $\neg\neg \alpha \equiv \alpha$

3) Standardize variables by renaming : each Quatifier use diffrent variable

4.) ~~Stalo~~ Stokenize: each existential variable replaced by Stolcem Constant or Stolcem. function of the enclosing university
- for instances $\exists x$ Rich(x) becomes Rich(G_1) where G_1 is anew Constant
- Every one.

5. Distribute $\land$ over $\lor$:
- $(\alpha \land \beta) \lor r \equiv (\alpha \lor r) \land (\beta \lor r)$

Resolution in FOL
Basic steps for proving a conclusion S given premisis premisis
(all expressed in FOL)
1) Convert all sentence to CNF
2) Negate conclusion S of convert result to CNF
3. add negated conclusion S to the premisis clauses
4. Repeat until contradiction or no progress is made
a. Select 2 clauses (call them parent clauses)
b. Reslove them together performing all required unification.
c, if resolvent is the Empty clause a contradiction has been formed (i.e follows from the premisis.
d, If not add resolvent to the premisis
If we succeed in step is neither proved the conclusion.

Proof by Resolution:
* Given the KB or premisis
* John likes all kind of food.
* Apple & Vegetable 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 By Resolution that: John likes peanuts.

f alive (Anil)

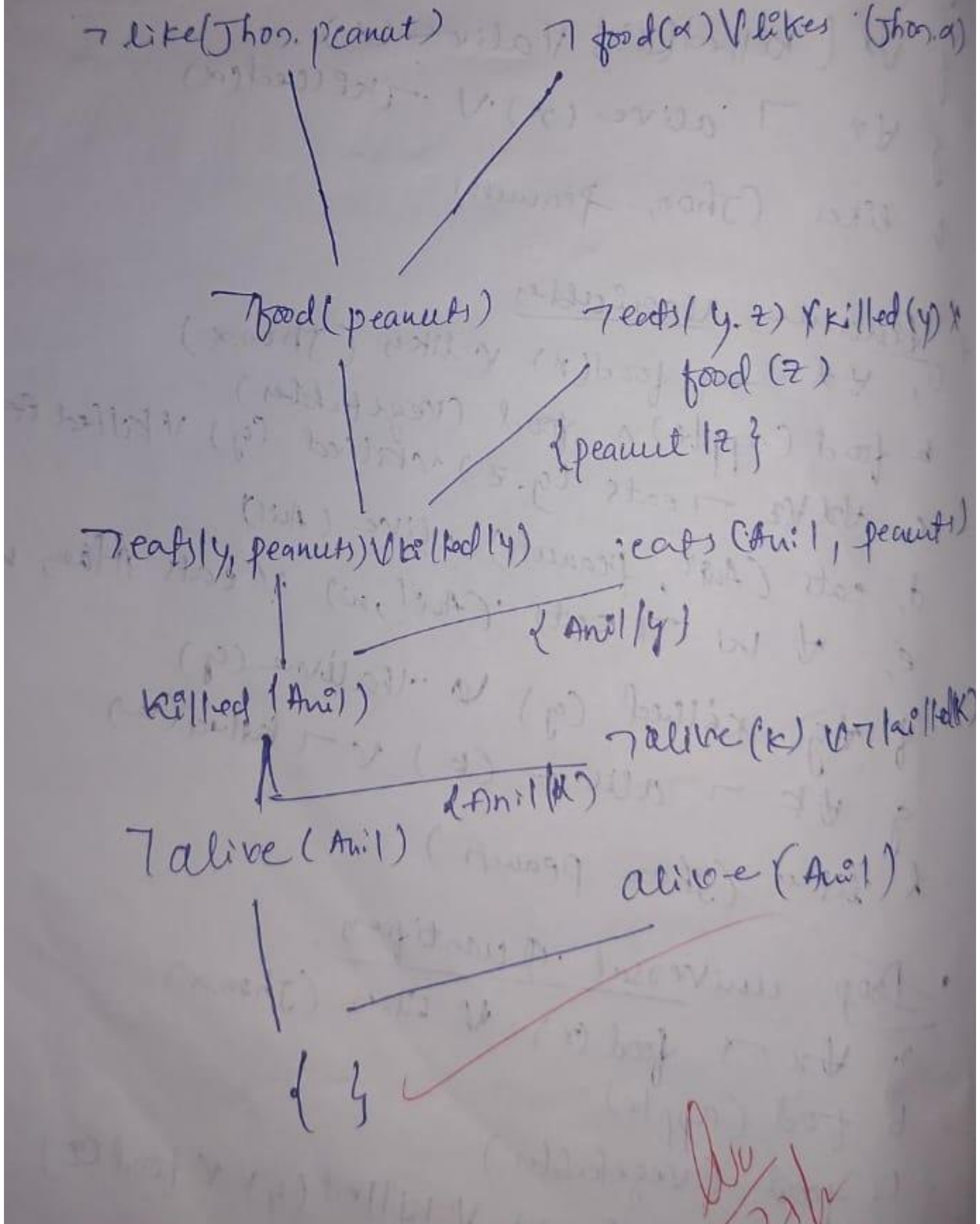
g ¬ eats (Anil, w) ∨ eats (Harry, w))

h killed(g) ∨ alive(g)

i ¬alive(k) ∨ ¬killed(k)

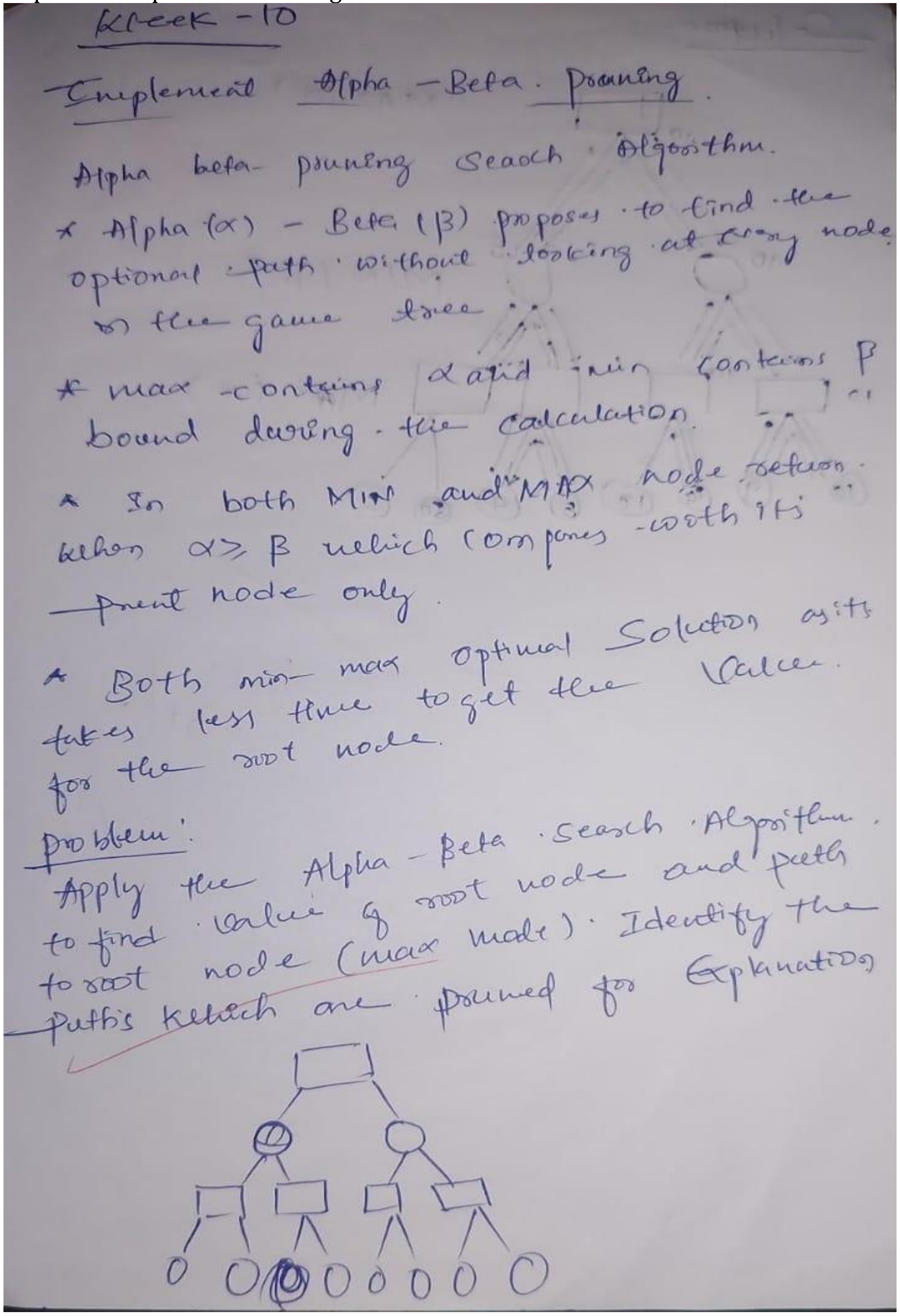
j. likes (Jhon peanuts)

OUTPOT



Code:
```
from itertools import combinations

def negate(literal):
    if literal.startswith('~'):
        return literal[1:]
    else:
        return '~' + literal

def is_complementary(a, b):
    return a == negate(b)

def resolve(ci, cj):
    for lit in ci:
        for lit2 in cj:
            if is_complementary(lit, lit2):
                new_clause = list(set(ci + cj))
                new_clause.remove(lit)
                new_clause.remove(lit2)
                return list(set(new_clause))
    return None


def resolution(clauses):
    new = set()
    while True:
        pairs = list(combinations(clauses, 2))
        for (ci, cj) in pairs:
            resolvent = resolve(ci, cj)
            if resolvent == []:
                print("Contradiction found ⇒ Proof successful ")
                return True
            if resolvent is not None:
                new.add(tuple(sorted(resolvent)))
        if new.issubset(set(map(tuple, clauses))):
            print("No new clauses ⇒ Proof failed ")
            return False
        for c in new:
            if list(c) not in clauses:
                clauses.append(list(c))

clauses = [
    ['~Food(x)', 'Likes(John,x)'],
    ['Food(Apple)'],
    ['Food(Vegetable)'],
    ['~Eats(x,y)', 'Killed(x)', 'Food(y)'],
    ['Eats(Anil,Peanut)'],
    ['Alive(Anil)'],
    ['~Eats(Anil,x)', 'Eats(Harry,x)'],
    ['~Alive(x)', '~Killed(x)'],
    ['Killed(x)', 'Alive(x)'],
    ['~Likes(John,Peanut)']
]

resolution(clauses)
```

Program 10
Algorithm:
Implement Alpha-Beta Pruning.



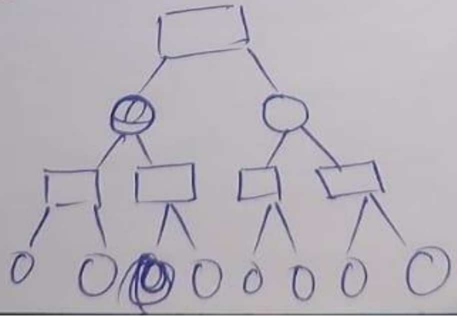
Week - 10

Implement Alpha - Beta . Pruning.

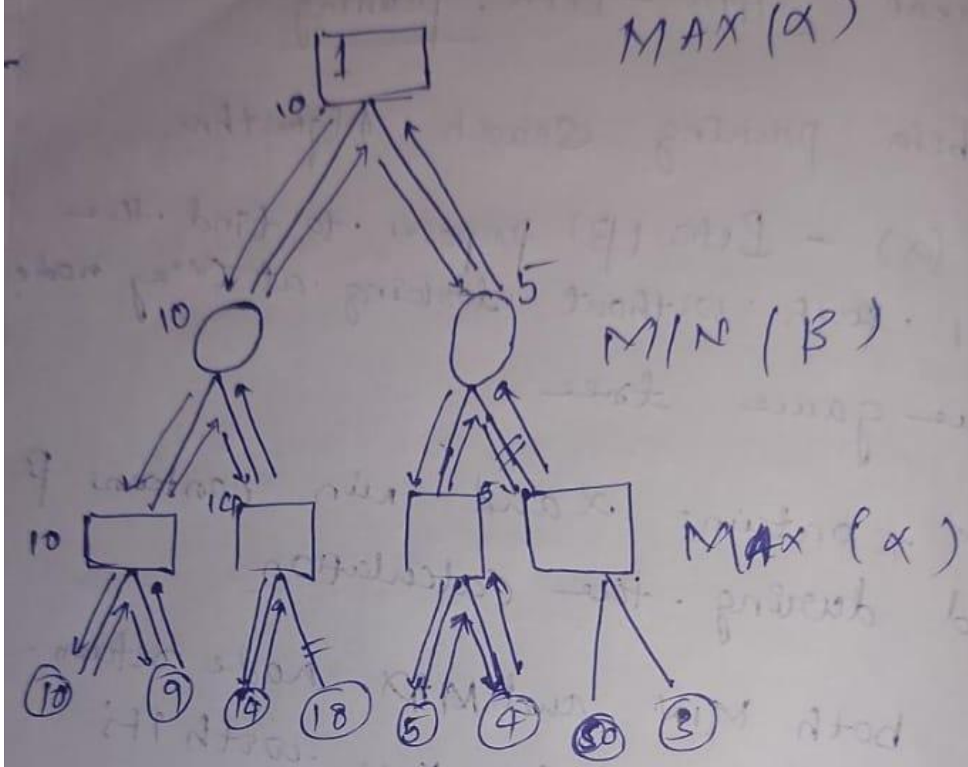
Alpha beta pruning Search Algorithm.

* Alpha (α) - Beta (β) proposes to find the optional path without looking at every node in the game tree.
* max contains α and min contains β bound during the calculation.
* In both MIN and MAX node return when $\alpha \geq \beta$ which compares with its prent node only.
* Both min-max optimal Solution as its takes less time to get the value for the root node.

problem:
Apply the Alpha-Beta Search Algorithm to find value of root node and path to root node (max node). Identify the paths which are pruned for Explanation



Solution.



Code:

```
import math

def alpha_beta(depth, node_index, is_maximizing, values, alpha, beta,
max_depth):
    indent = "    " * depth
    if depth == max_depth:
        print(f"{indent}Leaf node[{node_index}] = {values[node_index]}")
        return values[node_index]

    if is_maximizing:
        best = -math.inf
        print(f"{indent}MAX node (α={alpha}, β={beta})")
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta,
max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            print(f"{indent}    MAX updating α={alpha}")
            if beta <= alpha:
                print(f"{indent}    β cut-off (β={beta}, α={alpha}) ")
                break
        return best
    else:
        best = math.inf
        print(f"{indent}MIN node (α={alpha}, β={beta})")
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta,
max_depth)
            best = min(best, val)
            beta = min(beta, best)
            print(f"{indent}    MIN updating β={beta}")
            if beta <= alpha:
                print(f"{indent}    α cut-off (β={beta}, α={alpha}) ")
                break
        return best

values = [10, 9, 14, 18, 5, 4, 50, 3]
max_depth = int(math.log2(len(values)))

print("Leaf Nodes:", values)
result = alpha_beta(0, 0, True, values, -math.inf, math.inf, max_depth)
print("\nOptimal value:", result)
```