VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Tarun S Sunadoli(1WA23CS015)

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 **Tarun S Sunadoli (1WA23CS015),** 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.

Dr. K. R. Mamatha Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Tarun-619/AI-lab-1WA23CS015

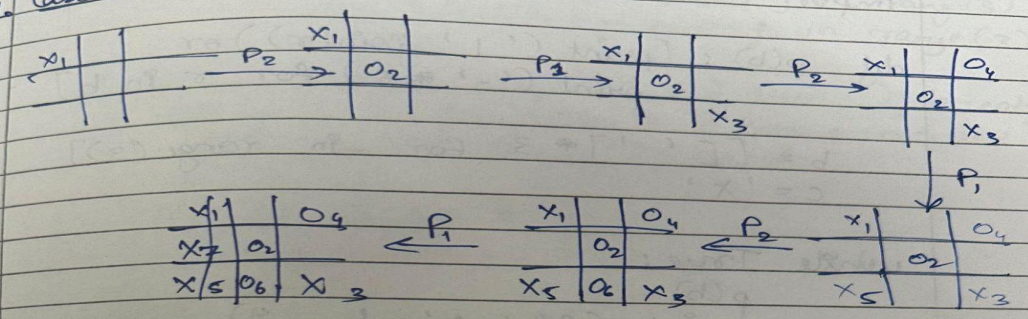
Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:

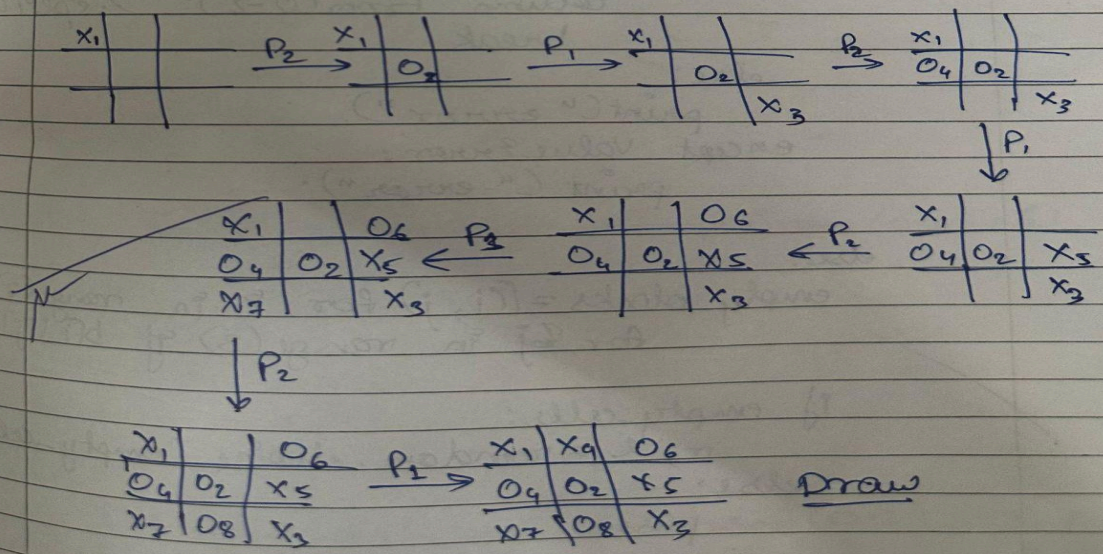
LAB - 1 → Tic tac toe URBAN EDGE 19, 8, 25

1. Case - I → when



P1 wins

2. Case - II → draw



Draw

code:

```
import random

def p(b): [print('|'.join(r)) or
           print('-'*6) for r in b]

b = [[' ']*3 for _ in range(3)]
c = 'X'

while True:
    p(b)
    print(f"{c}'s turn")

    if c=='X':
        while True:
            try:
                r,l = map(int,input("row and
                column from(0-2): ").split())
                break
            else:
                print("error")
            except ValueError:
                print("error").

    else:
        empty_blocks = [(i,j) for i in range(3)
            for j in range(3) if b[i][j]==' ']

        if empty_calls:
            r,l = random.choice(empty_calls)
        else:
            pass

    b[r][l]=c
    w = lambda p: any(all(b[i][j]==p for j in range(3)) or
                      all(b[j][i]==p for j in range(3))

    if w(c): p(b); print(f"{c} win"); break
    if all(e!=' ' for row in b for e in row):
        p(b); print("draw"); break

    c='XO'[c=='X']
```

output:

```
  |  |
  |  |
  |  |
Enter row and column from(0-2) 0 1
  |X|
  |  |
  |  |
O's turn
  |X|O
  |  |
  |  |
Enter row and column from(0-2): 1 1
  |X|O
  |X|
  |  |
O's turn
  |X|O
  |X|O
  |  |
Enter row and col from(0-2): 2 1
  |X|O
  |X|O
  |X|
X wins
```

Learn
19/2/25

Code:

tic tac toe

```
def p(b): [print('|'.join(r)) or print('-'*6) for r in b]
b=[[' ']*3 for _ in range(3)]
c='X'
while True:
    p(b)
    print(f"{c}'s turn")
    try:
        r=int(input("Row: "))
        l=int(input("Col: "))
    except: continue
    if 0<=r<3 and 0<=l<3 and b[r][l]==' ':
        b[r][l]=c
        w=lambda p: any(all(b[i][j]==p for j in range(3))or all(b[j][i]==p for j in range(3))for i in range(3))or
all(b[i][i]==p for i in range(3))or all(b[i][2-i]==p for i in range(3))
        if w(c): p(b); print(f"{c} wins"); break
        if all(e!=' 'for row in b for e in row): p(b); print("draw hoo gaya bhaiii"); break
        c='XO'[c=='X']
```

op:

```
O's turn
Row: 0
Col: 1
X|O|
------
 | |
------
 | |
------
X's turn
Row: 1
Col: 0
X|O|
------
X| |
------
 | |
------
O's turn
Row: 2
Col: 2
X|O|
------
X| |
------
 | |O
------
X's turn
Row: 2
Col: 0
X|O|
------
X| |
------
X| |O
------
X wins
```

vacuum cleaner

```python
def vacuum_cleaner_agent():
    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()
```

op:

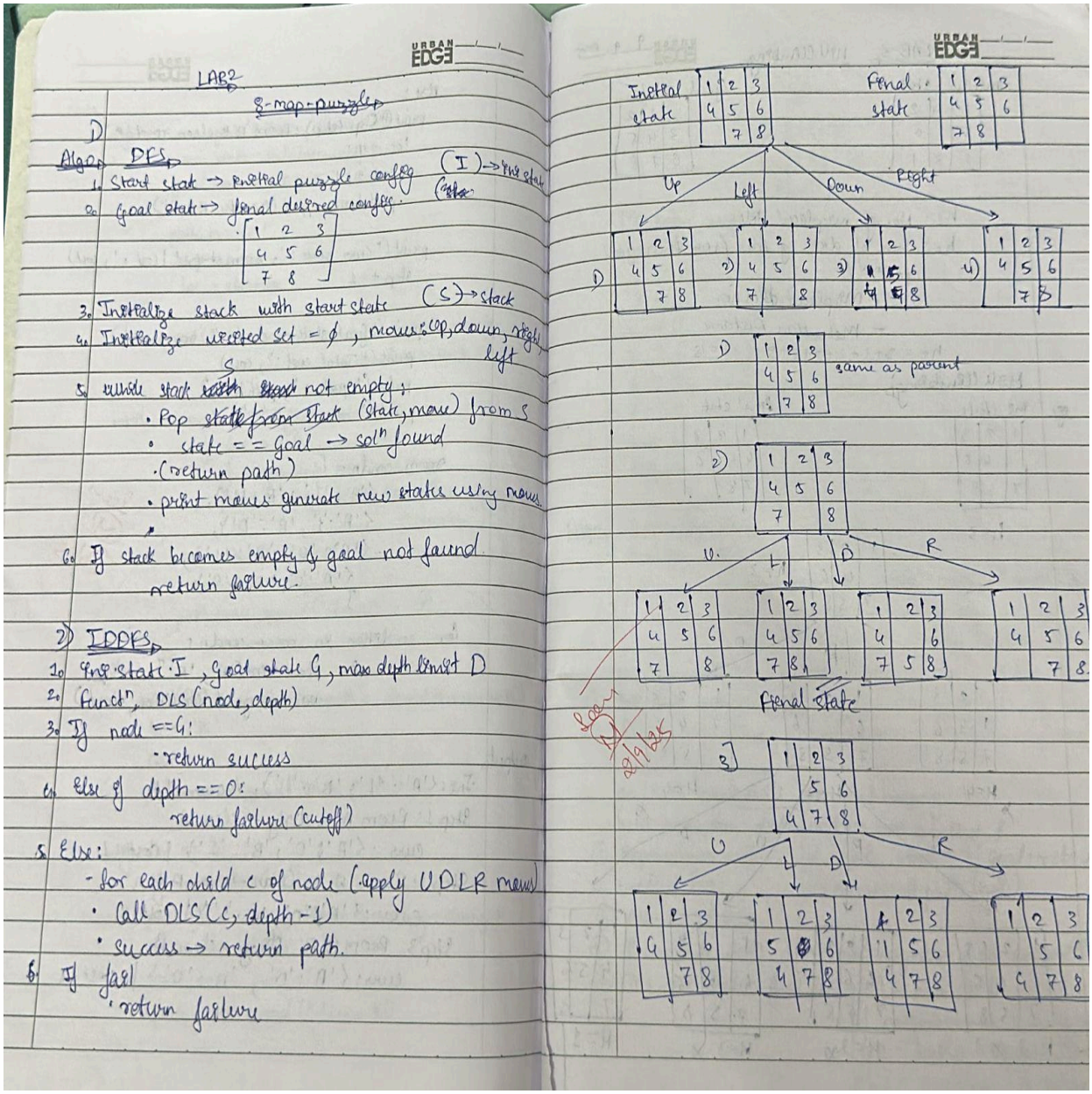
```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
Cost: 2
{'A': 0, 'B': 0}
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:



Code:

8 map puzzle DFS

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

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

def neighbors(s):
    x,y=find_blank(s)
    moves=[(1,0),(-1,0),(0,1),(0,-1)]
    result=[]
    for dx,dy in moves:
        nx,ny=x+dx,y+dy
        if 0<=nx<3 and 0<=ny<3:
            ns=[row[:] for row in s]
            ns[x][y],ns[nx][ny]=ns[nx][ny],ns[x][y]
            result.append(ns)
    return result

def dfs(start):
    stack=[(start,[start])]
    visited=set()
    while stack:
        state,path=stack.pop()
        if state==goal: return path
        visited.add(str(state))
        for nxt in neighbors(state):
            if str(nxt) not in visited:
                stack.append((nxt,path+[nxt]))
    return None

start=[[1,2,3],[4,0,6],[7,5,8]]
sol=dfs(start)
if sol:
    for step in sol:
        for row in step: print(row)
        print("----")
else:
```

```
    print("No solution")
```

op:

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

8 map puzzle IDDFS

```
def dls(state,limit):
    if state==goal: return [state]
    if limit<=0: return None
    for nxt in neighbors(state):
        res=dls(nxt,limit-1)
        if res: return [state]+res
    return None

def iddfs(start,max_depth=10):
    for depth in range(max_depth+1):
        result=dls(start,depth)
        if result: return result
    return None
sol=iddfs(start,6)
if sol:
    for step in sol:
        for row in step: print(row)
        print("----")
else:
    print("No solution within max depth")
```

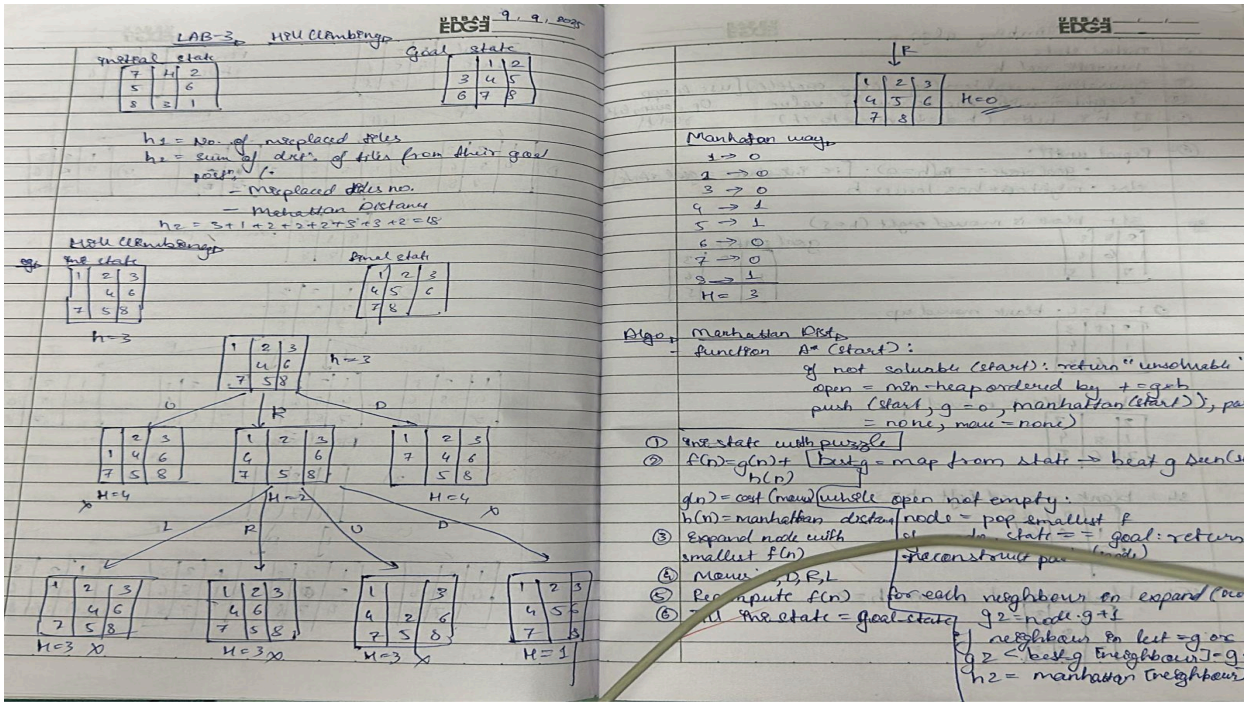
op:

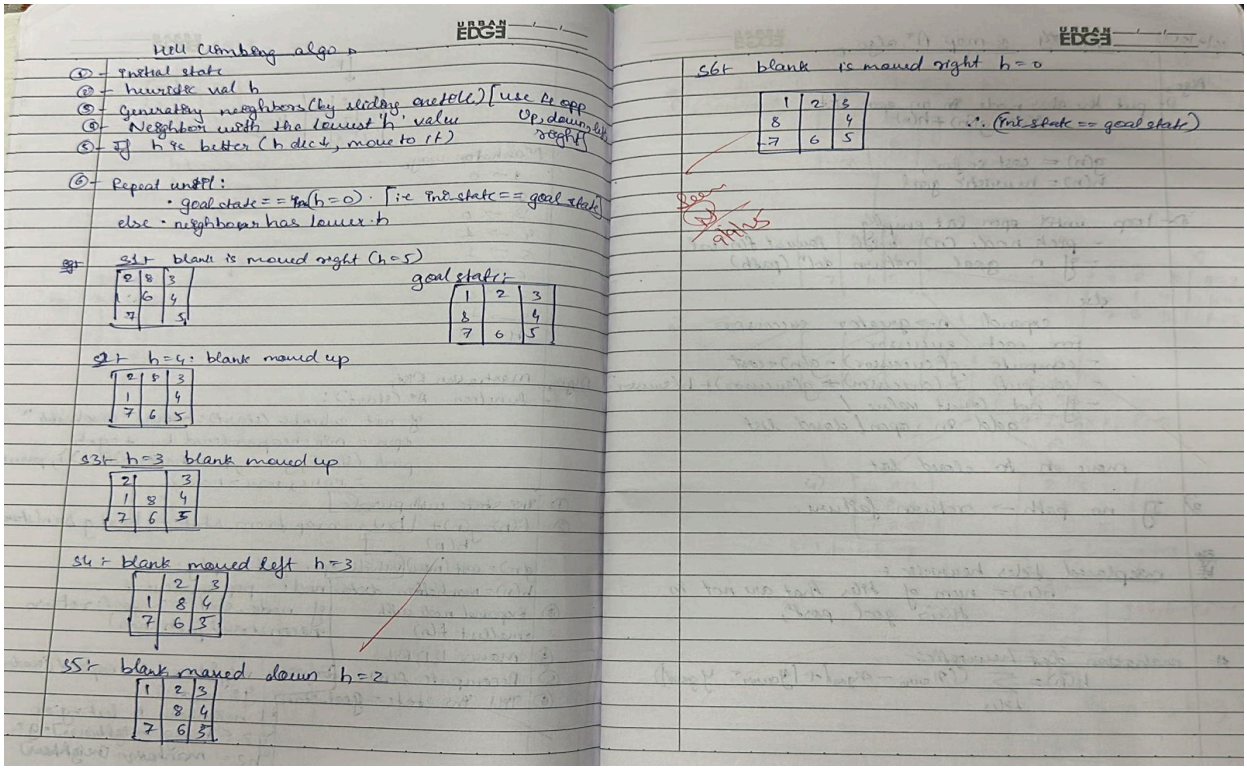
```
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
----
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
----
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
----
```

Program 3

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





Code:

```
import copy,random

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

def manhattan(s):
    pos={goal[r][c]:(r,c) for r in range(3) for c in range(3)}
    return sum(abs(i-pos[s[i][j]][0])+abs(j-pos[s[i][j]][1])
               for i in range(3) for j in range(3) if s[i][j]!=0)

def misplaced(s):
    return sum(s[i][j]!=0 and s[i][j]!=goal[i][j]
               for i in range(3) for j in range(3))

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

def neighbors(s):
    x,y=find_blank(s)
    moves=[(1,0),(-1,0),(0,1),(0,-1)]
    result=[]
    for dx,dy in moves:
        nx,ny=x+dx,y+dy
        if 0<=nx<3 and 0<=ny<3:
            ns=copy.deepcopy(s)
            ns[x][y],ns[nx][ny]=ns[nx][ny],ns[x][y]
            result.append(ns)
    return result

def hill_climbing(start,hfunc):
    current=start
    steps=0
    while True:
        cur_h=hfunc(current)
        print(f"Step {steps}, h={cur_h}")
       for row in current: print(row)
```

```
        print("----")
        if cur_h==0: return current
        neigh=neighbors(current)
        if not neigh: return current
        nxt=min(neigh,key=hfunc)
        if hfunc(nxt)>=cur_h: return current
        current=nxt; steps+=1
start=[[2,8,3],[1,6,4],[7,0,5]]
hill_climbing(start,manhattan)
```

op:

```
Step 0, h=5
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
----
Step 1, h=4
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
----
Step 2, h=3
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
----
Step 3, h=2
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
----
Step 4, h=1
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]
----
Step 5, h=0
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
----
Out[ ]:  [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
```

Programme 4:

Implement A* Search Algorithm

Algorithm:

16/9/2025 LAB-#4 8-map A* algo

Algo:

1) put the start node in an open list
$f(n) = g(n) + h(n)$

$g(n)$ = cost so far,
$h(n)$ = heuristic goal

2) loop until open list empty
- pick node (n) with lowest f(n) val
- if n goal return sol" (path)

else
expand n → greator successor
for each successor
- compute g(successor) = g(n) + cost
- compute f(successor) = g(successor) + h(successor)
- if not (same value)
add in open/closed list

move n to closed list

3) If no path → return failure.

misplaced tiles heuristic :
h(n) = num of tiles that are not in their goal posⁿ,

manhattan dist heuristics
$h(n) = \sum_{\text{tiles}} (|x_{curr} - x_{goal}| + |y_{curr} - y_{goal}|)$

level 0 (g=0, h=4, f=4)

1) misplaced

2	8	3
1	6	4
7		5

2) up

2	8	3
1		4
7	6	5

Left → X

2	8	3
—	1	4
7	6	5

Down

up

3)

2	—	3
1	8	4
7	6	5

g=2, h=3, f=5

L R

4)

—	2	3
1	8	4
7	6	5

g=3, h=2, f=7

2	3	—
1	8	—
7	6	5

f=5, h=4, g=3

Down

5)

1	2	3
—	8	4
7	6	5

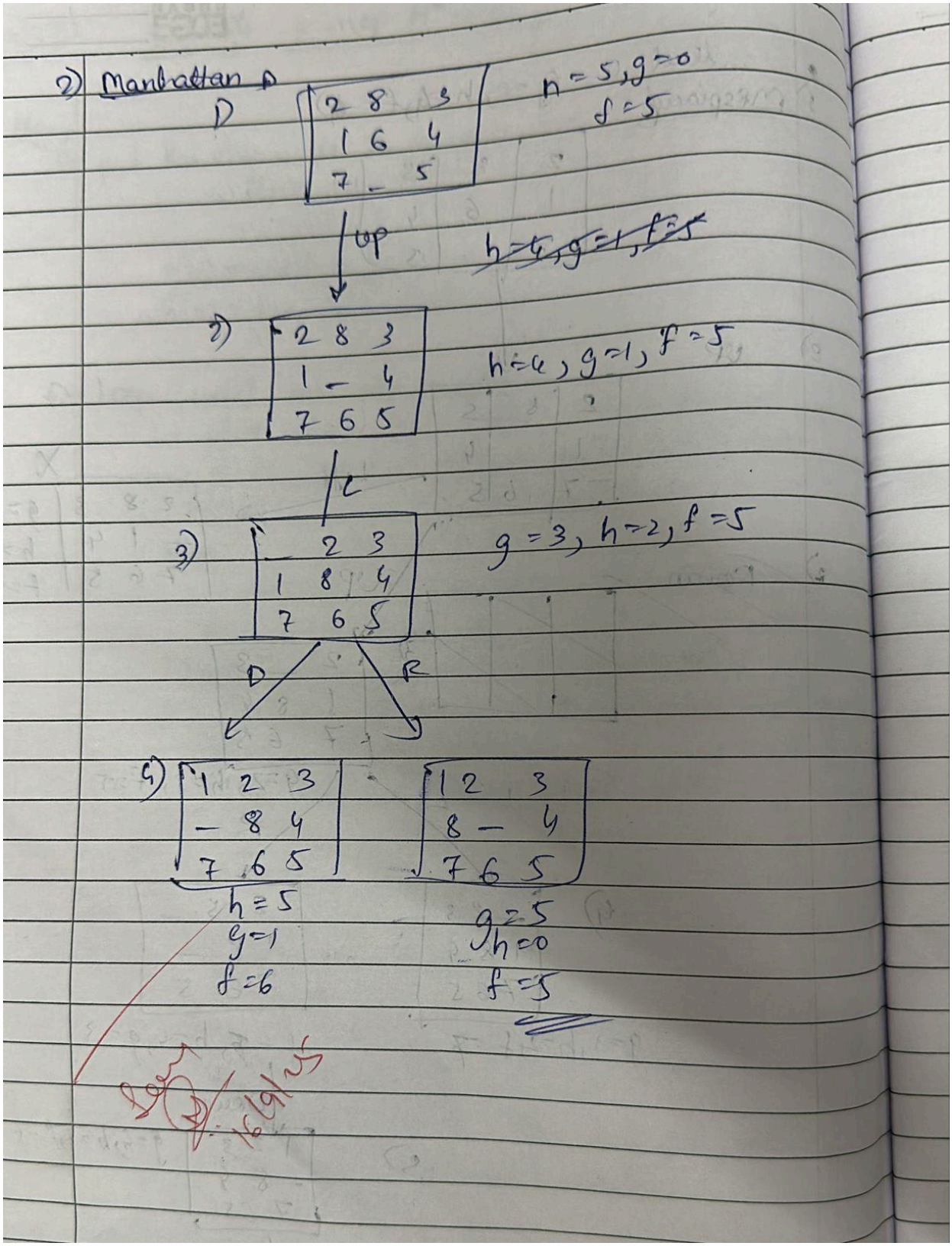
g=4, h=1, f=

R

6)

1	2	3
8	—	4
7	6	5

g=5, h=0


2) Manhattan
D
2 8 3
1 6 4
7 _ 5
h=5, g=0
f=5
up
h=4, g=1, f=5
2)
2 8 3
1 _ 4
7 6 5
h=4, g=1, f=5
L
3)
_ 2 3
1 8 4
7 6 5
g=3, h=2, f=5
D
R
4)
1 2 3
_ 8 4
7 6 5
h=5
g=1
f=6
1 2 3
8 _ 4
7 6 5
g=5
h=0
f=5

Code:

8-map A* Manhattan Distance

```
import heapq

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

def manhattan_distance(board, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = board[i][j]
            if tile != 0:

                for gi in range(3):
                    for gj in range(3):
                        if goal[gi][gj] == tile:
                            distance += abs(i - gi) + abs(j - gj)
                            break
    return distance

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

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

def board_to_tuple(board):
    return tuple(tuple(row) for row in board)

def a_star_search(start, goal):
   open_list = []
    start_h = manhattan_distance(start, goal)
    heapq.heappush(open_list, (start_h, 0, start, [start]))
    closed_set = set()

    while open_list:
        f, g, current, path = heapq.heappop(open_list)
```

```
        if current == goal:
            return path

        closed_set.add(board_to_tuple(current))

        for neighbor in get_neighbors(current):
            neighbor_tuple = board_to_tuple(neighbor)
            if neighbor_tuple in closed_set:
                continue
            g_new = g + 1
            h_new = manhattan_distance(neighbor, goal)
            f_new = g_new + h_new
            heapq.heappush(open_list, (f_new, g_new, neighbor, path + [neighbor]))

    return None

def get_user_input(prompt_text):
    print(f"\nEnter the {prompt_text} (use 0 for the blank tile).")
    board = []
    nums_seen = set()
    for i in range(3):
        while True:
            try:
                row = input(f"Enter row {i + 1} (e.g., 1 2 3): ").strip().split()
                if len(row) != 3:
                    raise ValueError
                row_int = [int(x) for x in row]
                for n in row_int:
                    if n < 0 or n > 8 or n in nums_seen:
                        raise ValueError
                    nums_seen.add(n)
                board.append(row_int)
                break
            except ValueError:
                print("Invalid input! Enter three unique integers from 0 to 8, no duplicates.")
    return board

def print_board(board):
    for row in board:
        print(' '.join(str(x) for x in row))


initial_state = get_user_input("initial state")
goal_state = get_user_input("goal state")

print("\nSolving the puzzle using A* search with Manhattan Distance heuristic...\n")
solution = a_star_search(initial_state, goal_state)

if solution:
```

```
    print(f"Solution found in {len(solution) - 1} moves:\n")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        print_board(state)
        print()
else:
    print("No solution found.")
```

Op:

```
Enter the initial state (use 0 for the blank tile).
Enter row 1 (e.g., 1 2 3): 1 2 3
Enter row 2 (e.g., 1 2 3): 0 4 6
Enter row 3 (e.g., 1 2 3): 7 5 8

Enter the goal state (use 0 for the blank tile).
Enter row 1 (e.g., 1 2 3): 1 2 3
Enter row 2 (e.g., 1 2 3): 4 5 6
Enter row 3 (e.g., 1 2 3): 7 8 0

Solving the puzzle using A* search with Manhattan Distance heuristic...

Solution found in 3 moves:

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

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

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

Step 3:
1 2 3
4 5 6
7 8 0
```

8-map A* Misplaced tiles

```
import heapq

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

def heuristic(board, goal):
    count = 0
    for i in range(3):
```

```
            for j in range(3):
                if board[i][j] != 0 and board[i][j] != goal[i][j]:
                    count += 1
    return count

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

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

def board_to_tuple(board):
    return tuple(tuple(row) for row in board)

def a_star_search(start, goal):
    open_list = []
    start_h = heuristic(start, goal)
    heapq.heappush(open_list, (start_h, 0, start, [start]))
    closed_set = set()

    while open_list:
        f, g, current, path = heapq.heappop(open_list)
        if current == goal:
            return path

        closed_set.add(board_to_tuple(current))

        for neighbor in get_neighbors(current):
            neighbor_tuple = board_to_tuple(neighbor)
            if neighbor_tuple in closed_set:
                continue
            g_new = g + 1
            h_new = heuristic(neighbor, goal)
            f_new = g_new + h_new
            heapq.heappush(open_list, (f_new, g_new, neighbor, path + [neighbor]))

    return None
```

```
def get_user_input(prompt_text):
    print(f"\nEnter the {prompt_text} (use 0 for the blank tile).")
    board = []
    nums_seen = set()
    for i in range(3):
        while True:
            try:
                row = input(f"Enter row {i + 1} (e.g., 1 2 3): ").strip().split()
                if len(row) != 3:
                    raise ValueError
                row_int = [int(x) for x in row]
                for n in row_int:
                    if n < 0 or n > 8 or n in nums_seen:
                        raise ValueError
                    nums_seen.add(n)
                board.append(row_int)
                break
            except ValueError:
                print("Invalid input! Enter three unique integers from 0 to 8, no duplicates.")
    return board

def print_board(board):
    for row in board:
        print(' '.join(str(x) for x in row))
initial_state = get_user_input("initial state")
goal_state = get_user_input("goal state")

print("\nSolving the puzzle using A* search...\n")
solution = a_star_search(initial_state, goal_state)

if solution:
    print(f"Solution found in {len(solution) - 1} moves:\n")
    for step, state in enumerate(solution):
        print(f"Step {step}:")
        print_board(state)
        print()
else:
    print("No solution found.")
```

op:

```
Enter the initial state (use 0 for the blank tile).
Enter row 1 (e.g., 1 2 3): 1 2 3
Enter row 2 (e.g., 1 2 3): 0 4 6
Enter row 3 (e.g., 1 2 3): 7 5 8

Enter the goal state (use 0 for the blank tile).
Enter row 1 (e.g., 1 2 3): 1 2 3
Enter row 2 (e.g., 1 2 3): 4 5 6
Enter row 3 (e.g., 1 2 3): 7 8 0

Solving the puzzle using A* search...

Solution found in 3 moves:

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

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

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

Step 3:
1 2 3
4 5 6
7 8 0
```

Programme 5:

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

LAB - 5 ▷ Proposition logic ▷ URBAN EDGE 15.10.2025

Algo ▷

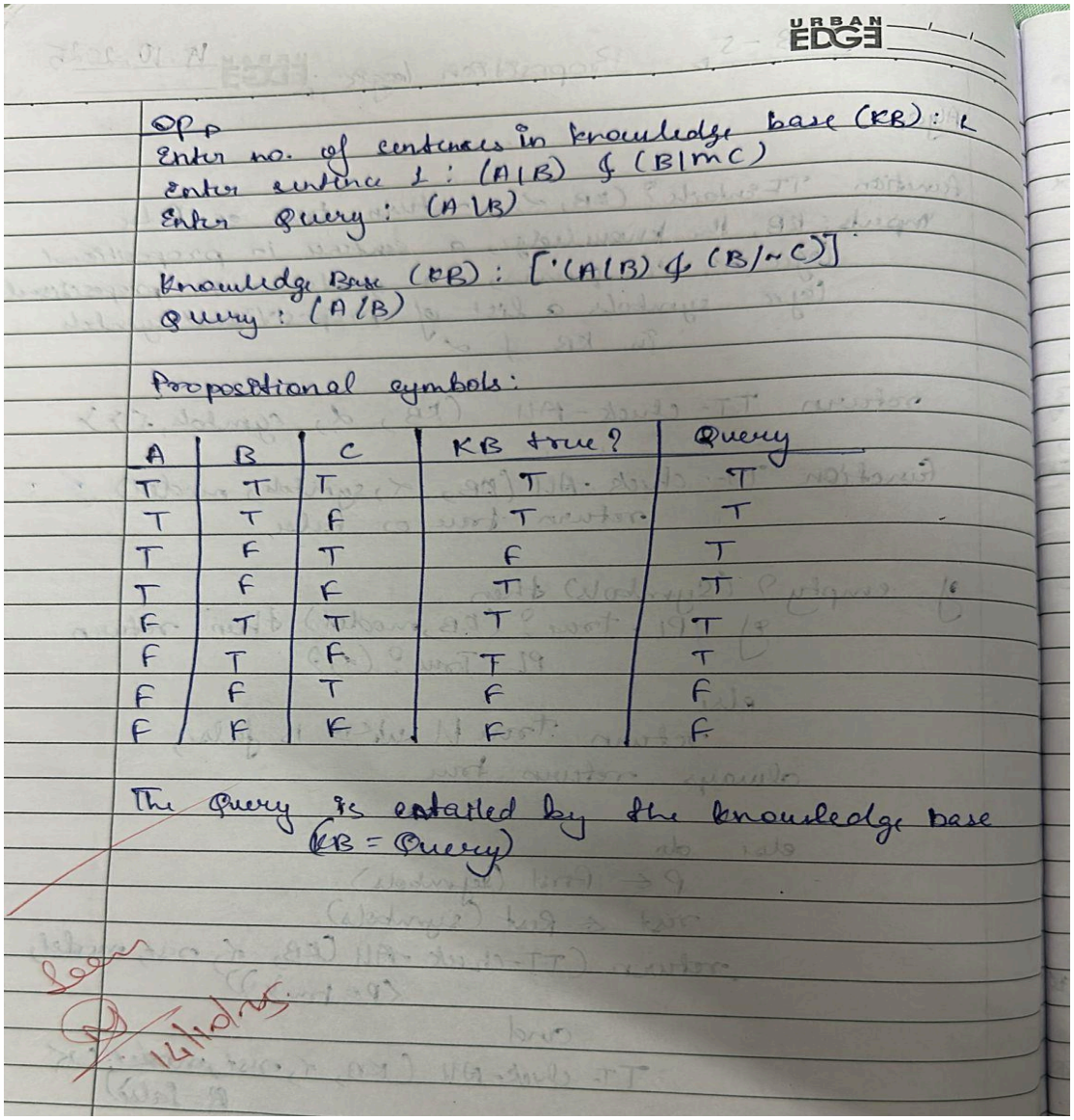
function TT-entails? (KB, α) return true or false
inputs: KB, the knowledge, a sentence in propositional logic α, the query, a sentence in propositional logic symbols a list of propositional symbols in KB & α

return TT-check-All (KB, α, symbols, {})

function TT-check-All (KB, α, symbols, model)
return true or false

if empty? (symbols) then
if PL-true? (KB, model) then return PL-True? (α,)
else
return true // ub<KB is false, always return true

else do
P ← First (symbols)
rest ← Rest (symbols)
return (TT-check-All (KB, α, rest, model, {P = true}))
and
TT-check-All (KB, α, rest, model, {P = false})



Code:

Proposition Logic

```
import itertools
import re

def pl_true(expr, model):
    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("↔", " == ")

    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):
    print(f"\nTruth Table for {KB} ⊨ {alpha}:")
    result = tt_check_all(KB, alpha, symbols, {}, True)
    return result

def tt_check_all(KB, alpha, symbols, model, print_table=False):
    if not symbols:
        kb_val = pl_true(KB, model)
        alpha_val = pl_true(alpha, model)

        if print_table:
            row = [f"{model.get(sym, False)}" for sym in sorted(model.keys())]
            print("\t".join(row) + f"\t| {kb_val} \t| {alpha_val}")

        if kb_val:
            return alpha_val
        else:
            return True
    else:
        P = symbols[0]
        rest = symbols[1:]
        return (tt_check_all(KB, alpha, rest, {**model, P: True}, print_table) and
                tt_check_all(KB, alpha, rest, {**model, P: False}, print_table))


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

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

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

op:

```
Truth Table for (A ∨ B) ∧ (¬C) ⊨ A ∨ (B ∧ ¬C):
True    True    True    | False         | True
True    True    False   | True  | True
True    False   True    | False         | True
True    False   False   | True  | True
False   True    True    | False         | False
False   True    False   | True  | True
False   False   True    | False         | False
False   False   False   | False         | False
Does KB ⊨ α? True
```

Programme 6:

Implement Alpha-Beta Pruning

Algorithm:

LAB-6 ▷ Alpha-Beta Search ▷ URBAN EDGE 11,11,2025

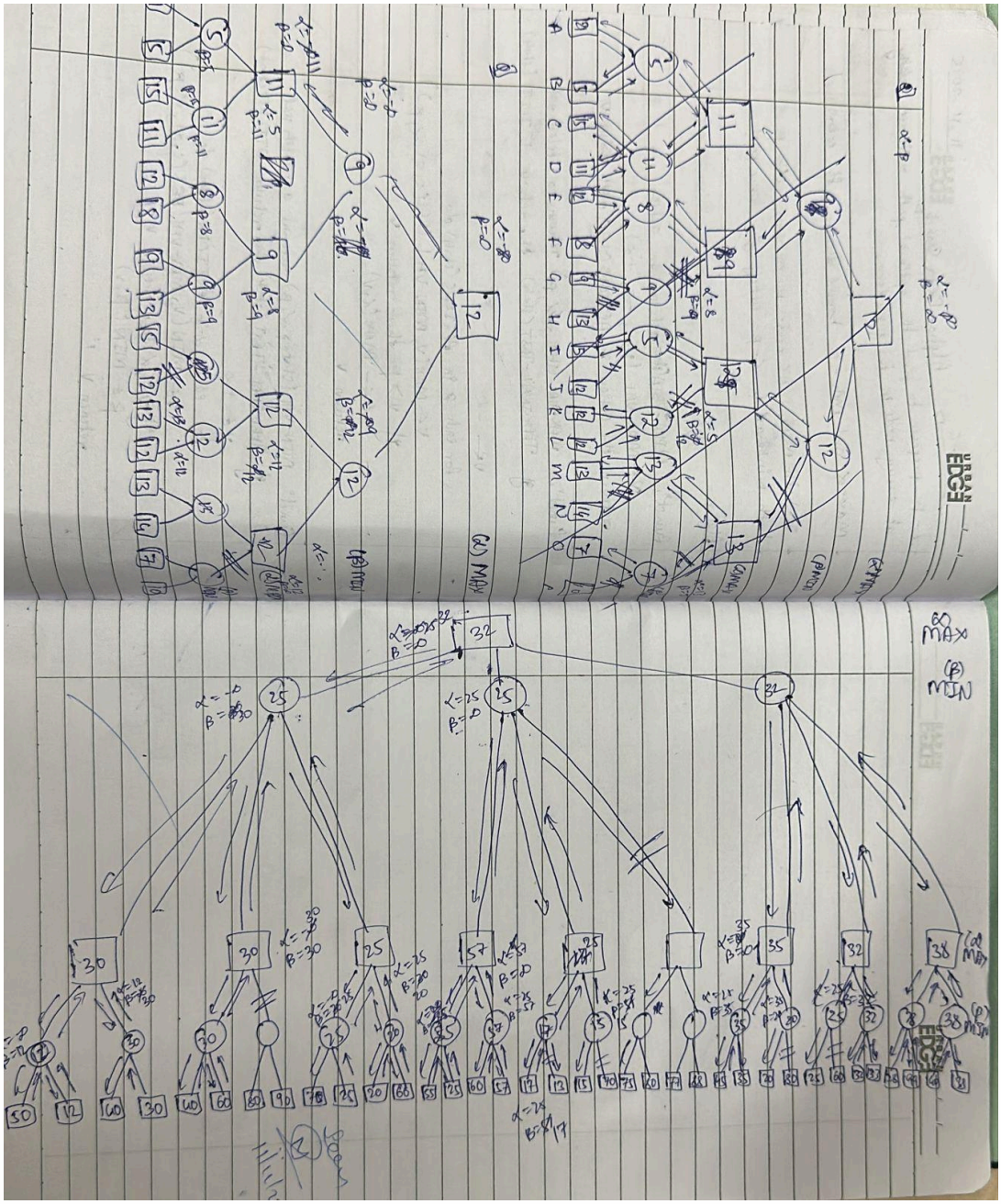
- α-β proposed to find the optimal path without looking at every node in the game tree.
- Max-α and min-β bound during the calc.
- In both MIN, MAX node we return $\alpha \geq \beta$
- both minimax & α-β cutoff give same path.

Algo ▷

funct", ALPHA-BETA(state) returns an action
  $v \leftarrow$ MAX-VAL (state, $-\infty$, $+\infty$)
  return the action in ACTIONS(state) with value v

funct", MAX-VAL (state, α, β) returns a utility val
  if TERMINAL-TEST (state) then return UTILITY (state)
  $v \leftarrow -\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MAX(v, MIN-VAL (RES(s, a), α, β))
    if $v \geq \beta$ then return v
      $\alpha \leftarrow$ MAX(α, v)
  return v

funct", MIN-VAL (state, α, β) returns a utility val
  if TERMINAL-TEST (state) then return UTILITY (state)
  $v \leftarrow +\infty$
  for each a in ACTIONS(state) do
    $v \leftarrow$ MIN (v, MAX-VAL(RES(s, a), α, β))
    if $v \leq \alpha$ then return v
      $\beta \leftarrow$ MIN (β, v)
  return v



Code:

Alpha-beta search

```
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 = [21,5,15,11,12,8,9,13,5,12,13,12,13,14,7,10]
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)
```

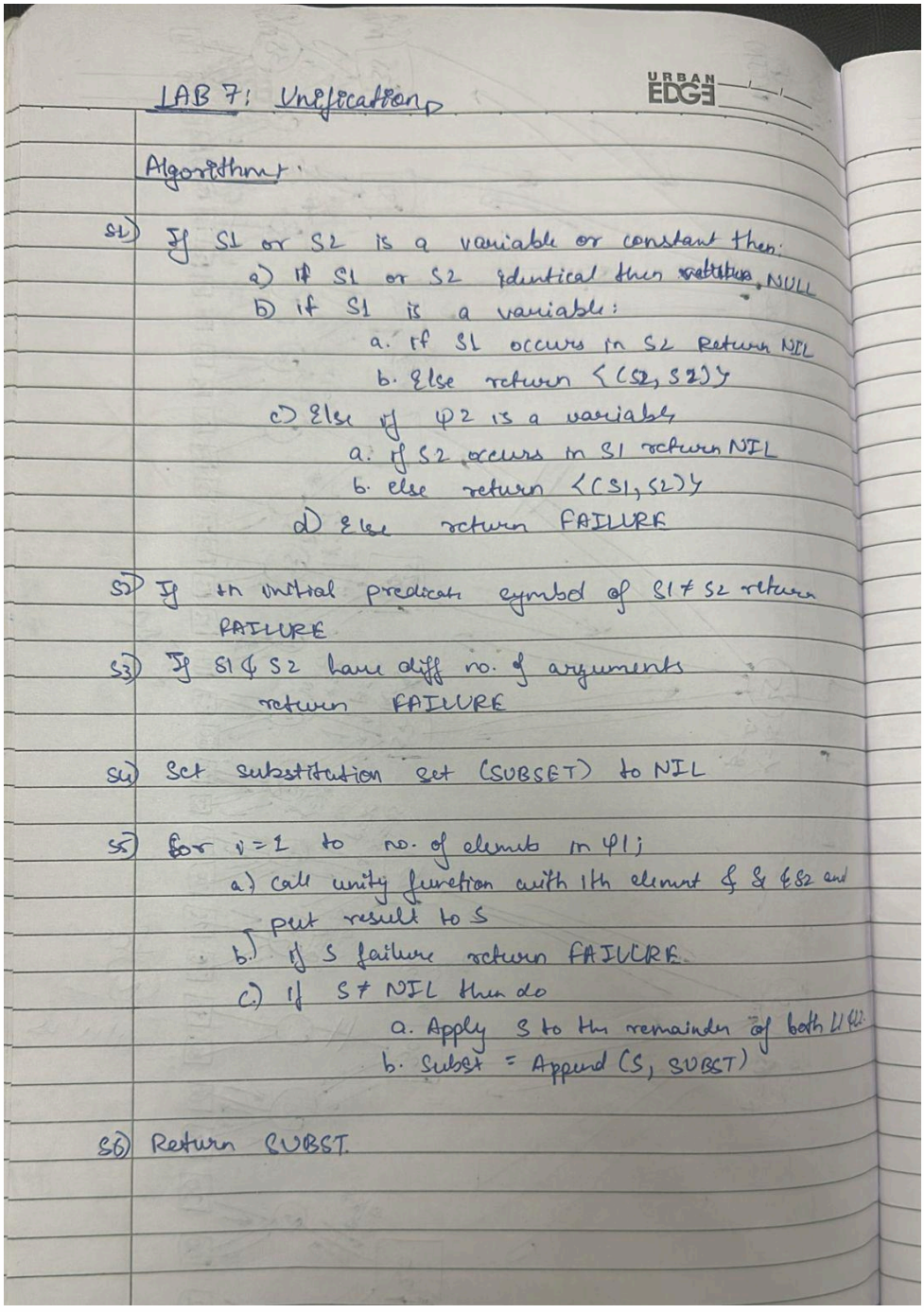
op:

```
            MIN updating β=11
            MAX node (α=-inf, β=11)
               MIN node (α=-inf, β=11)
                  Leaf node[4] = 12
                  MIN updating β=11
                  Leaf node[5] = 8
                  MIN updating β=8
               MAX updating α=8
               MIN node (α=8, β=11)
                  Leaf node[6] = 9
                  MIN updating β=9
                  Leaf node[7] = 13
                  MIN updating β=9
               MAX updating α=9
            MIN updating β=9
         MAX updating α=9
         MIN node (α=9, β=inf)
            MAX node (α=9, β=inf)
               MIN node (α=9, β=inf)
                  Leaf node[8] = 5
                  MIN updating β=5
                  α cut-off (β=5, α=9)
               MAX updating α=9
               MIN node (α=9, β=inf)
                  Leaf node[10] = 13
                  MIN updating β=13
                  Leaf node[11] = 12
                  MIN updating β=12
               MAX updating α=12
            MIN updating β=12
            MAX node (α=9, β=12)
               MIN node (α=9, β=12)
                  Leaf node[12] = 13
                  MIN updating β=12
                  Leaf node[13] = 14
                  MIN updating β=12
               MAX updating α=13
               β cut-off (β=12, α=13)
            MIN updating β=12
         MAX updating α=12

      Optimal value: 12
```

Programme 7:

Implement unification in first order logic

Algorithm:



LAB 7: Unification

Algorithm:

S1) If S1 or S2 is a variable or constant then:
  a) If S1 or S2 identical then return NULL
  b) if S1 is a variable:
    a. If S1 occurs in S2 Return NIL
    b. Else return {(S2, S1)}
  c) Else if ψ2 is a variable
    a. If S2 occurs in S1 return NIL
    b. else return {(S1, S2)}
  d) Else return FAILURE

S2) If the initial predicate symbol of S1 ≠ S2 return FAILURE

S3) If S1 & S2 have diff no. of arguments return FAILURE

S4) Set substitution set (SUBSET) to NIL

S5) for i = 1 to no. of elements in ψ1;
  a) Call unify function with ith element of S & ψ2 and put result to S
  b) If S failure return FAILURE
  c) If S ≠ NIL then do
    a. Apply S to the remainder of both ψ1 & ψ2
    b. Subst = Append (S, SUBST)

S6) Return SUBST

Code:

Unification

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

def occurs_check(var, x, subst):
    if var == x:
        return True
```

```
    if is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    if isinstance(x, list):
        return any(occurs_check(var, xi, subst) for xi in x)
    return False

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    if is_variable(x) and x in subst:
        return unify(var, subst[x], subst)
    if occurs_check(var, x, subst):
        return None
    subst[var] = x
    return subst

def unify(x, y, subst=None):
    if subst is None:
        subst = {}

    if x == y:
        return subst

    if is_variable(x):
        return unify_var(x, y, subst)

    if is_variable(y):
        return unify_var(y, x, subst)
   if isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for xi, yi in zip(x, y):
            subst = unify(xi, yi, subst)
            if subst is None:
                return None
        return subst

    return None

expr1 = ['f','x','y']
expr2 = ['f','apple','ball']

print("MGU:", unify(expr1, expr2))
```

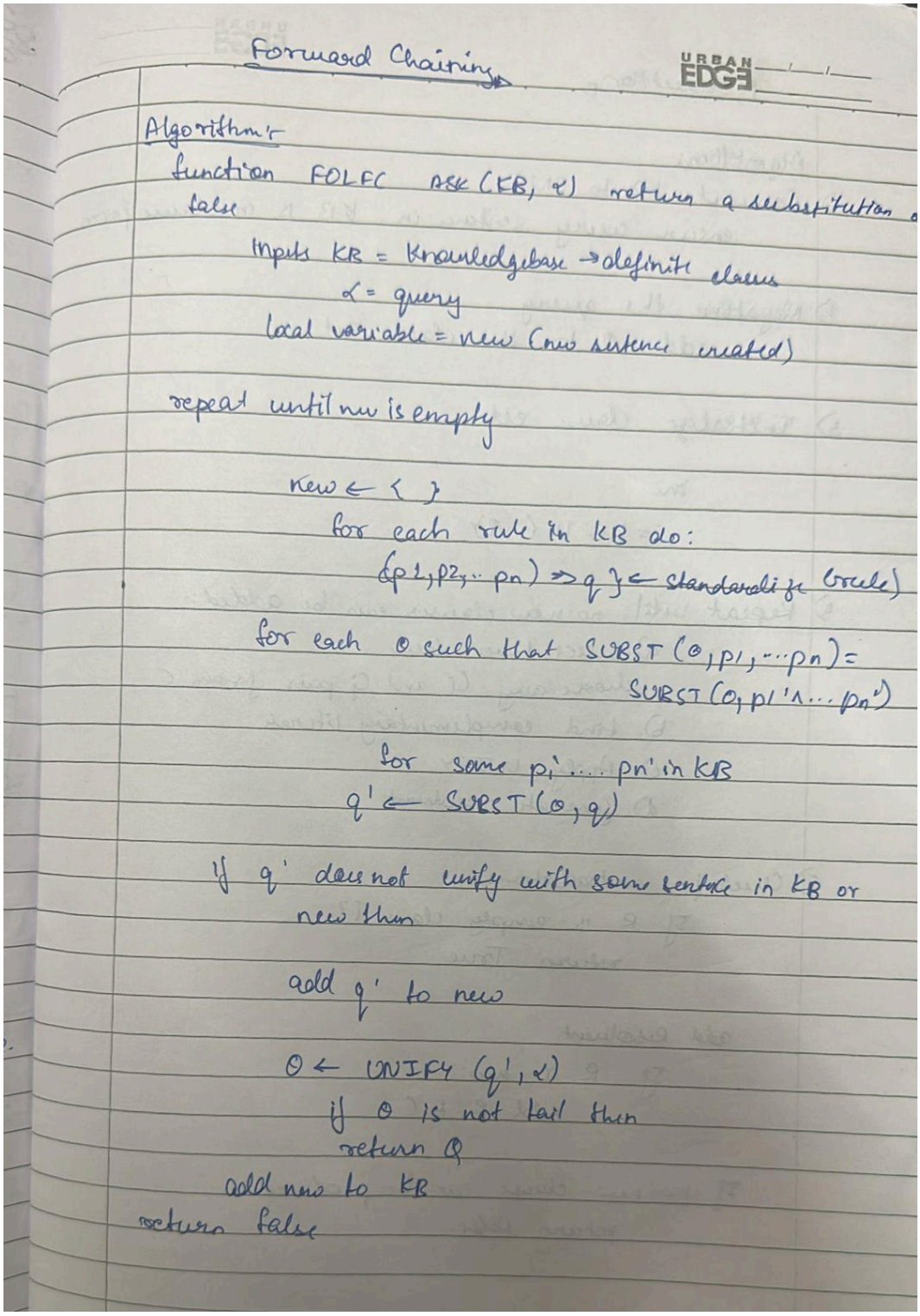
Op:

```
MGU: {'x': 'apple', 'y': 'ball'}
```

Programme 8:

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

Algorithm:



Code:

Forward Chaining

```
def is_variable(x):
    return x[0].islower()
```

```
def unify(x, y, subst):
    if subst is None:
        return None
    if x == y:
        return subst
    if is_variable(x):
        return unify_var(x, y, subst)
    if is_variable(y):
        return unify_var(y, x, subst)
    return None

def unify_var(var, val, subst):
    if var in subst:
        return unify(subst[var], val, subst)
    new = subst.copy()
    new[var] = val
    return new


def parse_predicate(pred):
    name = pred[:pred.index("(")]
    args = pred[pred.index("(")+1 : pred.index(")")].split(", ")
    return name, args


def match_premises(premises, facts):
    subst = {}

    for prem in premises:
        prem_name, prem_args = parse_predicate(prem)
        found = False

        for fact in facts:
            fact_name, fact_args = parse_predicate(fact)

            if prem_name == fact_name and len(prem_args) == len(fact_args):
                temp = subst.copy()
               ok = True
                for p, f in zip(prem_args, fact_args):
                    result = unify(p, f, temp)
                    if result is None:
                        ok = False
                        break
                    temp = result

                if ok:
                    subst = temp
                    found = True
```

```
                    break

        if not found:
            return None

    return subst


def apply_substitution(pred, subst):
    name, args = parse_predicate(pred)
    new_args = [subst.get(a, a) for a in args]
    return f"{name}({', '.join(new_args)})"


def forward_chaining(facts, premises, conclusion, query):
    facts = set(facts)

    while True:
        subst = match_premises(premises, facts)
        if subst is None:
            break

        inferred = apply_substitution(conclusion, subst)

        if inferred not in facts:
            print(f"Inferred: {inferred}")
            facts.add(inferred)
           if inferred == query:
                return True, facts
        else:
            break

    return query in facts, facts

facts = {
    "Parent(Tanmay, Niddhi)",
    "Parent(Niddhi, Tejas)",
    "Male(Tanmay)",
    "Female(Niddhi)"
}

premises = [
    "Parent(x, y)",
    "Parent(y, z)"
]
conclusion = "Grandparent(x, z)"
query = "Grandparent(Tanmay, Tejas)"
result, derived = forward_chaining(facts, premises, conclusion, query)
```

```
print("\nFinal Derived Facts:")
for f in derived:
    print(f)

print("\nQuery:", query)
print("Result:", "TRUE  (Tanmay is Tejas's grandparent)" if result else "FALSE
(Not provable)")
```

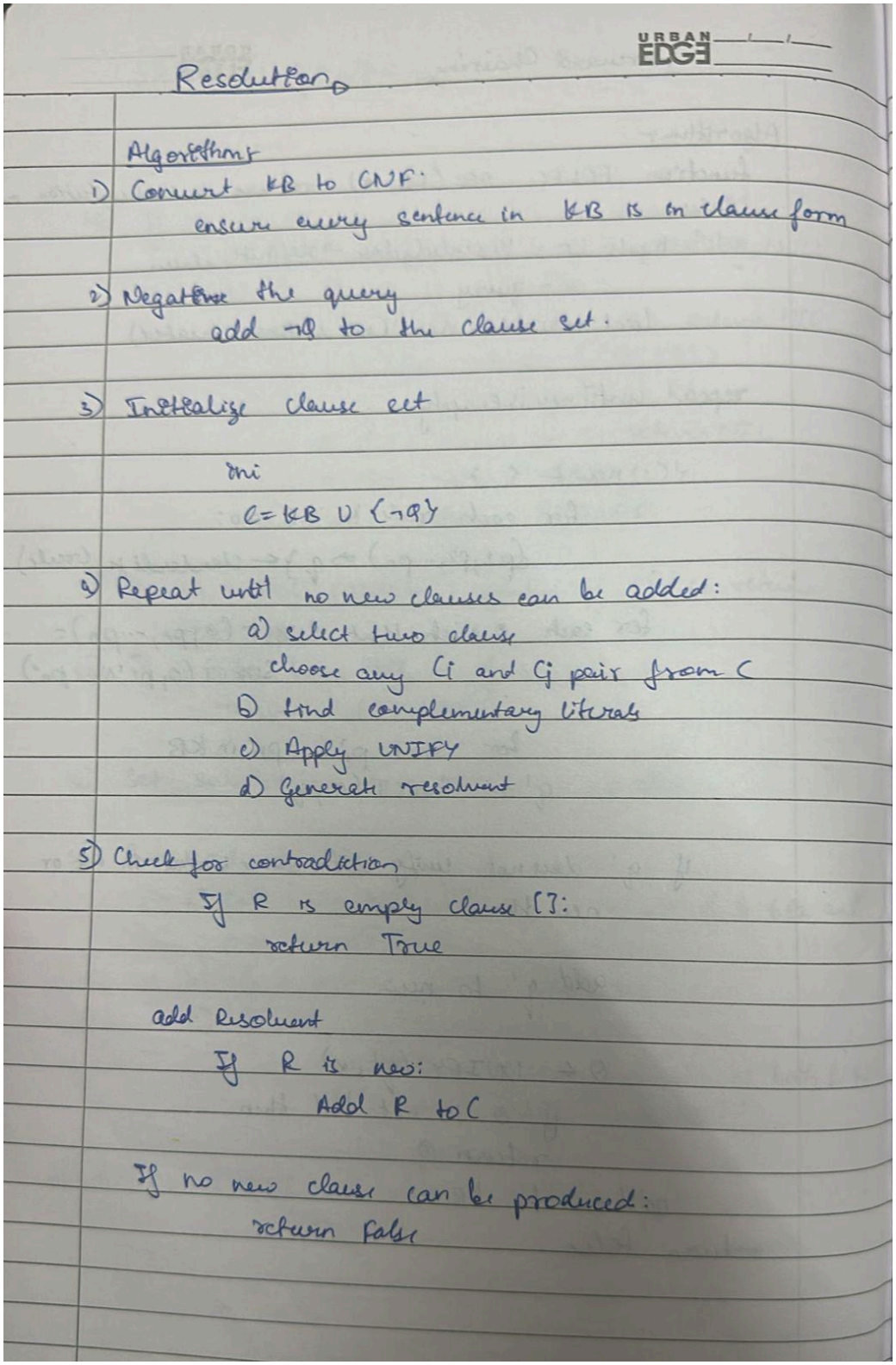
Op:

```
Inferred: Grandparent(Tanmay, Tejas)

Final Derived Facts:
Female(Niddhi)
Grandparent(Tanmay, Tejas)
Male(Tanmay)
Parent(Tanmay, Niddhi)
Parent(Niddhi, Tejas)

Query: Grandparent(Tanmay, Tejas)
Result: TRUE  (Tanmay is Tejas's grandparent)
```

Programme 9:

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

Algorithm:



Code:

Resolution

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

def substitute(theta, clause):
    return [(p, [theta.get(t, t) for t in terms]) for (p, terms) in clause]

def unify(x, y, theta):
    if theta is None: return None
    if x == y: return theta
    if is_variable(x): return unify_var(x, y, theta)
    if is_variable(y): return unify_var(y, x, theta)
    if isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for xi, yi in zip(x, y):
            theta = unify(xi, yi, theta)
            if theta is None: return None
        return theta
    return None

def unify_var(v, x, theta):
    if v in theta: return unify(theta[v], x, theta)
    if x in theta: return unify(v, theta[x], theta)
    new = theta.copy()
    new[v] = x
    return new

def resolve(ci, cj):
    resolvents = []
    for pi in ci:
        for pj in cj:
            if pi[0].lstrip("¬") == pj[0].lstrip("¬") and
(pi[0].startswith("¬") != pj[0].startswith("¬")):
                theta = unify(pi[1], pj[1], {})
                if theta is not None:
                    new_clause = substitute(theta, [x for x in ci if x != pi])
\
                                + substitute(theta, [x for x in cj if x != pj])
                    res = []
                    for lit in new_clause:
                        if lit not in res:
                            res.append(lit)
                    resolvents.append(res)
   return resolvents

def resolution(kb, query):
    clauses = [list(c) for c in kb] + [[("¬" + query[0], query[1])]]
    new = []

    while True:
```

```
        for i in range(len(clauses)):
            for j in range(i + 1, len(clauses)):
                for r in resolve(clauses[i], clauses[j]):
                    if r == []:
                        return True
                    if r not in new and r not in clauses:
                        new.append(r)

        if not new:
            return False

        clauses.extend(new)
        new = []

KB = [
    [("¬Food", ["x"]), ("Likes", ["Tarun", "x"])],
    [("Food", ["Apple"])],
    [("Food", ["Vegetable"])],
    [("¬Eats", ["a", "b"]), ("¬NotKilled", ["a"]), ("Food", ["b"])],
    [("Eats", ["TR", "Peanuts"])],
    [("Alive", ["TR"])],
    [("¬Eats", ["TR", "x"]), ("Eats", ["Harry", "x"])],
    [("¬Alive", ["x"]), ("NotKilled", ["x"])],
    [("¬NotKilled", ["x"]), ("Alive", ["x"])]
]

query = ("Likes", ["Tarun", "Peanuts"])
print("Proved?", resolution(KB, query))
```

op:

```
Proved? True
```