VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

Lab REPORT
on

Artificial Intelligence

Submitted by

Harsh Ghiya (1BM21CS073)

Under the Guidance of
Prof. Swarthi Sridharan
Assistant Professor, BMSCE

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

Nov-2023 to Feb-2024

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**" carried out by **Harsh Ghiya (1BM21CS073)** who are bona fide students 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 Visveswaraiah Technological University, Belgaum during the year 2023-2024. The lab report has been approved as it satisfies the academic requirements in respect of Artificial Intelligence lab **(22CS5PCAIN)** work prescribed for the said degree.

Swathi Sridharan
Assistant Professor
Dept. of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak
Prof.& Head
Dept. of CSE
BMSCE, Bengaluru

17/11/23

1. Lab-3

Write a program for Tic-Tac-Toe game.

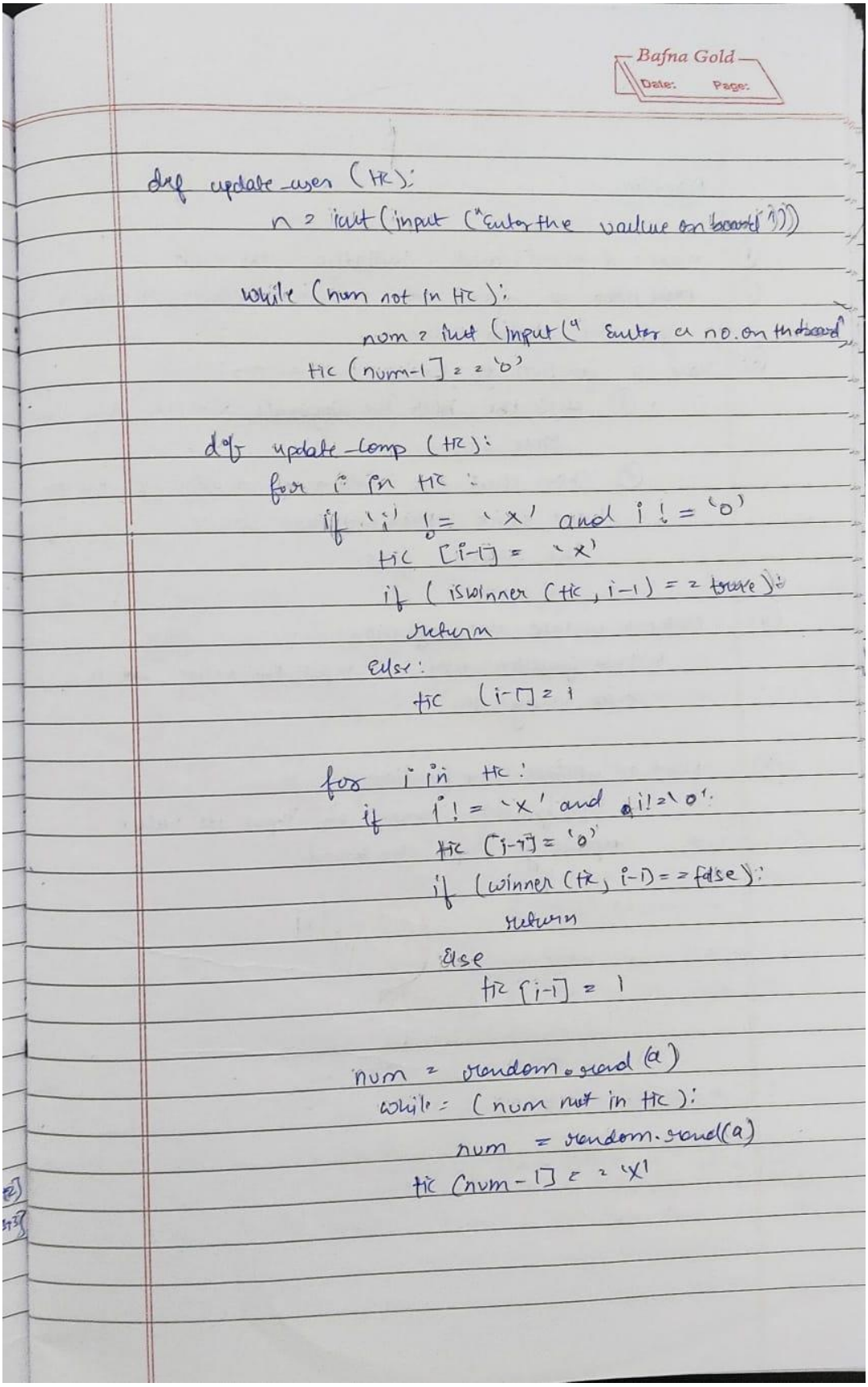
Algo :-

→ Create a empty global 3x3 array.

→ Make the board of 3x3 tic-tac-toe.

→ Then we will

```
import random
tic = [1,2,3,4,5,6,7,8,9]

def printboard (tic)
    print (tic[0] + '|' + tic[1] + '|' + tic[2])
    print ("- - - - - - -")
    print (tic[3] + '|' + tic[4] + '|' + tic[5])
    print ("- - - - - - -")
    print (tic[6] + '|' + tic[7] + '|' + tic[8])

def winner (tic, pos)
    if (tic[0] == tic[4] and tic[4] == tic[8] or
        tic[2] == tic[4] and tic[4] == tic[6]
        return true
    Elif if
        tic[pos-0] == tic[pos-3] & tic[pos-3] ==
                                   tic[pos-6]
        Return true
    Elif if
        tic[pos//3+1] == tic[pos//3+2] and tic[pos//3*3]
                                       == tic[pos//3+3]
        return true.
    return false.
```


Bafna Gold
Date: Page:
def update_user (tic):
n = int (input ("Enter the value on board"))
while (num not in tic):
num = int (input ("Enter a no. on the board"
tic (num-1] = = 'O'
def update_comp (tic):
for i in tic:
if 'i' != 'x' and i != 'O'
tic [i-1] = 'x'
if (iswinner (tic, i-1) = = true):
return
Else:
tic (i-1] = i
for i in tic:
if i != 'x' and i != 'O':
tic [i-1] = 'O'
if (winner (tic, i-1) = = false):
return
Else
tic [i-1] = i
num = random.rand (a)
while = (num not in tic):
num = random.rand(a)
tic (num-1] = = 'x'

Algorithm:-

1. make a board and initialize the value.
2. import the random library in order to get the random values.
3. Make a function for checking the winner:
   1. check both the diagonals whether they have same values.
   2. Then check the columns & row if all are same then return true.
      Else return false.
4. Make a update user function:
   In this function user can input the value. in an array tic
5. Make a update-comp function
   In this function comp can input its value depending upon the board.

output:

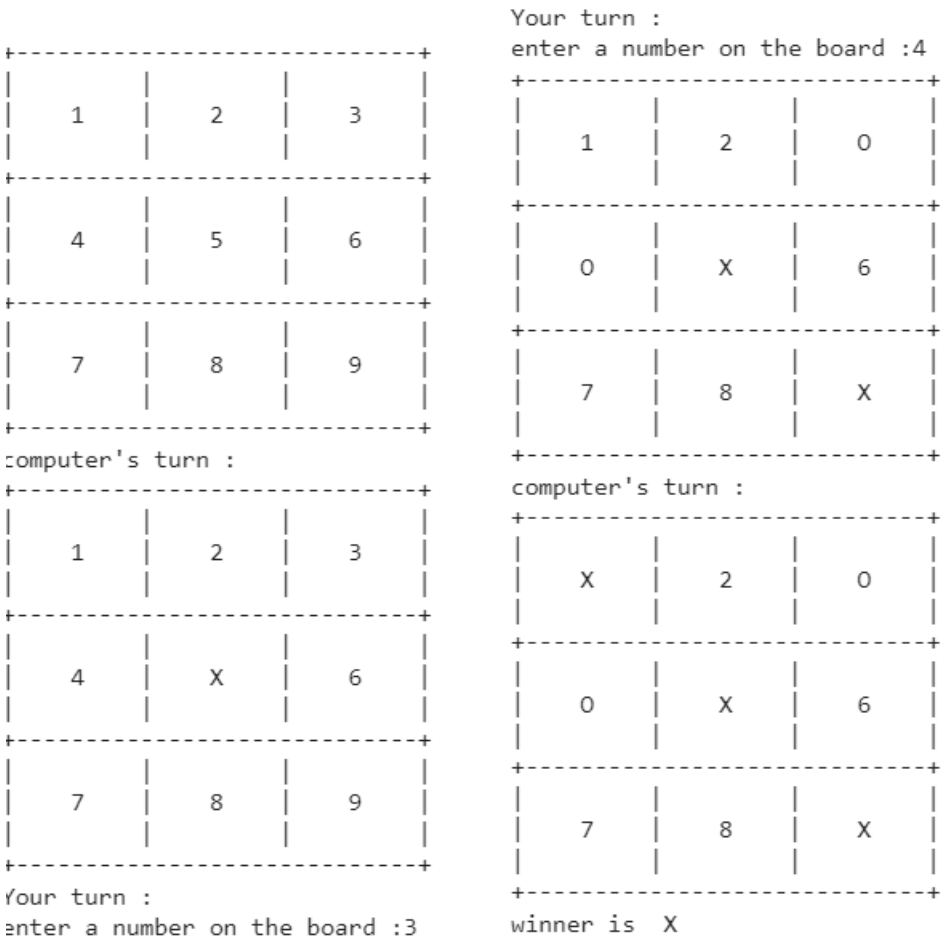
1	2	3
4	5	6
7	8	9

1	2	□
0	X	6
7	8	X

1	2	3
4	X	6
7	8	9

X	2	0
0	X	6
7	8	X

Output:


+-----------------------------+
1	2	3
+-----------------------------+
4	5	6
+-----------------------------+
7	8	9
+-----------------------------+
computer's turn :
+-----------------------------+
1	2	3
+-----------------------------+
4	X	6
+-----------------------------+
7	8	9
+-----------------------------+
Your turn :
enter a number on the board :3
Your turn :
enter a number on the board :4
+-----------------------------+
1	2	0
+-----------------------------+
0	X	6
+-----------------------------+
7	8	X
+-----------------------------+
computer's turn :
+-----------------------------+
X	2	0
+-----------------------------+
0	X	6
+-----------------------------+
7	8	X
+-----------------------------+
winner is X

Bafna Gold
Date: Page:

24/11/23

Lab - 8

Write an algorithm for 8-puzzle problem:-

Algorithm:-

1. Start with an empty queue
   queue ⟶ []

2. Enque with initial state.
   queue.append(src)

3. While the queue is not empty, dequeue the 1st initial. state from the queue
   def solve_puzzle(initial state, goal-state)
   queue = deque([initial state, []])

4. Check if the current state is the target state., if it is return "success".
   def check_current: (initial-state, goal-state)
   if (initial-state == goal-state)
   print("success");

5. Generate all possible moves from current state by determining the direction of the empty spot & considering all possible direction.
   def generate(current-state):

6. moves = []
   blank_row, blank_column = find_blank(board
   possible_moves [(1,0),(-1,0),(0,1),(0,-1)]

6. For each possible moves, calculate the resulting st

7. Check if the resulting state have already been visited. If it has not, enqueue the resulting state.

8. If the target state has not been found, continue with the next iteration with loop.

9. ~~If the target is not found after visiting all possible states return -1.~~

Found

⇒ Code:-

```
from collection import deque

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

```
def generate_moves (board):
    moves = []
    blank_row, blank_col = find_blank (board)

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

    for dr, dc in possible_moves:
        new_row, new_col = blank_row + dr, blank_col + dc

        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_board = [row[:] for row in board]
            new_board [blank_row] [blank_col], new_board [new_row] [new_col] = new_board [new_row] [new_col], new_board [blank_row] [blank_col]
            moves.append (new_board)

    return moves

def sol_puzzle (initial_state, goal_state):
    visited = set()
    queue = deque ([(initial_state, [])])
    while queue:
        current_state, path = queue.popleft()
        visited.add (tuple (map (tuple, current_state)))

        if current_state == goal_state:
            return path

        possible_moves = generate_moves (current_state)
```

```
def print_steps(solution_path):
    if solution_path:
        print("Steps to reach the goal:")
        for step in solution_path:
            print("-----")
            for row in step:
                print("|", end="")
                for val in row:
                    if val == 0
                        print("", end="|")
                    else:
                        print(val, end="|")
                print()
            print("-----")
            print()
    ~~else:~~
        ~~print("No solution exists")~~

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

solution_path = solve_puzzle(initial, goal)
print_steps(solution_path)
```

Bajna Gold
Date: Page:

Output:

Steps to reach the goal:

1 2 3
4 5
6 7 8

1 2 5
3 4
6 7 8

2 3
1 4 5
6 7 8

1 2
3 4 5
6 7 8

2 3
1 4 5
6 7 8

1 2
3 4 5
6 7 8

2 3 5
1 4
6 7 8

2 5
1 3 4
6 7 8

2 5
1 3 4
6 7 8

1 2 5
3 4
6 7 8

Output:

```
1 | 2 | 3
4 | 5 | 6
0 | 7 | 8
-----------
1 | 2 | 3
0 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 0 | 8
-----------
0 | 2 | 3
1 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
5 | 0 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 0 | 6
7 | 5 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 8 | 0
-----------
Success
```

8/12/2023

Lab-5

→ 8 Puzzle problem using Iterative Deepening Search Algorithm:

(1) Node (data, level) : initialize node with puzzle state & level.

(2) def puzzle:
→ accept () ← start & goal state
→ dls() (node, goal, depth) ← perform dls

(3) def DLS (node, goal, depth):
if (current_state == goal):
return solution
else:
generate child node, recursively call with increase level.

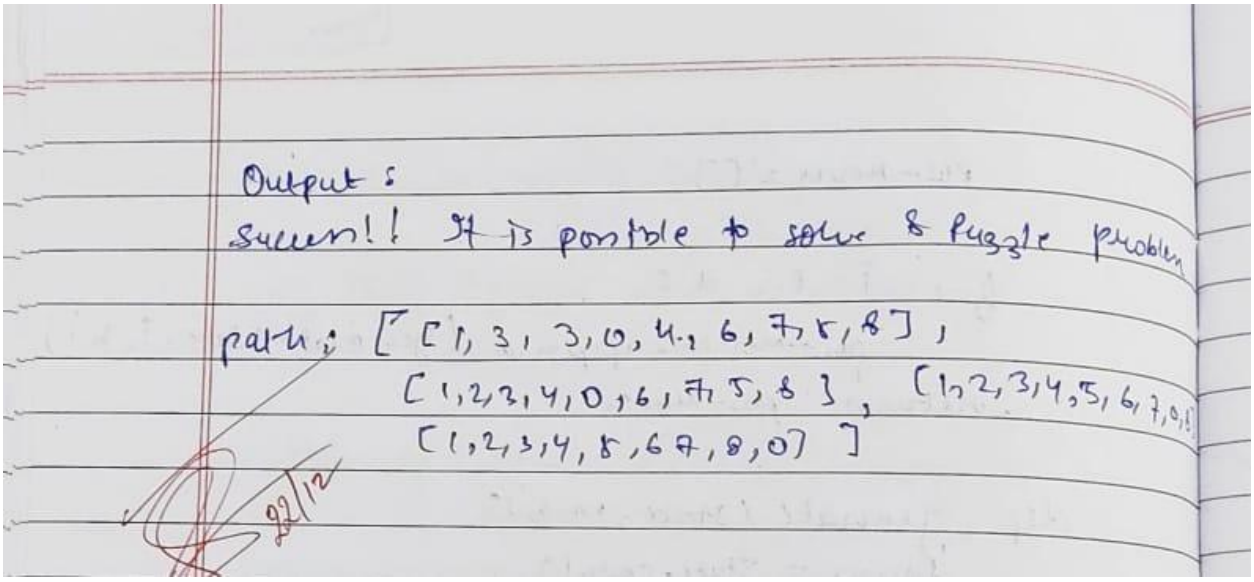
(4) def IDS (start, goal)
→ Start with depth : 0
→ Repeat until goal is found :
→ perform DLS with current depth
→ if sol" found ← exit
increment depth

(5) → Create puzzle instance
→ Call IDS (start, goal)

→ Code Iterative-Deepening-search :-

```
→ def id_dfs(puzzle, goal, get_moves)
      import intertools

    def dfs(route, depth):
        if (depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get_moves(route[-1]):
            if move not in route:
                next_route = dfs(route + [move], dfs
                if next_route:
                    return next_route
    for depth in intertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route

def possible_moves(state):
    b = state.index(0)
    d = []
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')
```

```
    pos_moves = []

    for i in d:
        pos_moves.append(generate(state, i, b))
    return pos_moves

def generate(state, m, b):
    temp = state.copy()

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

    return temp

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

route = id_dfs(initial, goal, possible_moves)

if route:
    print("Success! It is possible to solve 8 puzzle problem")
    print("Path:", route)
else:
    print("Failed to find a solution")
```



Output:

```
Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [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]]
```

```
[ ] Start coding or generate with AI.
```

4 → 8 puzzle problem using A* algorithm:

(1) def Node (data, level) : initialize node with puzzle state & level.

(2) def puzzle ( ) :
- accept( ) : ← Accept start & goal state.
- f(start, goal) ← calculate f. = h + g
- process ( ) ← process A* algorithm.

(3) Heuristic
h(start, goal) : Manhattan dist. blw current & goal state

(4) A* Algorithm:
- initialize start node, & add to open list
- loop until goal state come.
  - select node have lowest f from open list
  - Display state & check goal.
  - generate child node & calculate f
  - And current node to closed list & remove from open list
  - explore list by f
- Display find move.

(5) Execution: - Create puzzle instance
- call process for the execute.

1	2	3
4	5	6
7	8	0

h=0

1	2	3
4	5	6
7	8	

h=4

1	2	3
4	5	6
	7	8

h=3 h=8

1	2	3
4	5	6
7		8

h=2

92/12

⇒ Code of 8 puzzle using A* algorithm:

```
Class node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x,y = self.find(self.data, '_')
        val_list = [[x,y-1], [x,y+1], [x-1,y], [x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x,y, i[0],i[1])
            if child is not None
                child_node = Node(child, self.level+1,0)
                children.append(child_node)
            return children.

def shuffle(self, puz, x1, y1, x2, y2):
    if x2 >=0 x2 < len(self.data) and y2 >=0 and
                                    y2 < len(
```

```
        temp_puz=[]
        temp_puz = self.copy(puz)
        temp = temp_puz[x2][y2]
        temp_puz[x2][y2] = temp_puz[x1][x1]
        temp_puz[x1][y1] = temp
        return temp_puz
    Else
        return None

def find(self, puz, x):
    for i in range(0, len(self.data)):
        for j in range(0, len(self.data)):
            if puz[i][j] == x:
                return i,j

class puzzle:
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

def f(self, start, goal):
    return self.h(start.data, goal) + start.level
```

```
def h(self, start, goal):
    temp = 0
    for i in range (0, self.n):
        for j in range (0, self.n):
            if (start[i][j] != goal[i][j] and
                                    start[i][j] !=
                temp = temp + 1
    return temp.

def process(self):
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix \n")
    goal = self.accept()

    start = Node(start, 0, 0)
    start.fval = self.f(start, goal)
    self.open.append(start)
    print("\n\n")
    while true:
        cur = self.open[0]
        print(" ")
        print("  |  \n  |  \n")
        for i in cur.data:
            for j in i:
                print(j, end = " ")

        self.open.sort(key = lambda x: x.fval,

p43 = puzzle(3)
```

Bafna Gold
Date: Page:

Output

Enter the start Matrix

1 2 3
4 5 6
- 7 8

Enter the goal state Matrix.

1 2 3
4 5 6
7 8 -

1 2 3 → 1 2 3 → 1 2 3
4 5 6 4 5 6 4 5 6
- 7 8 7 - 8 7 8 -

22/12

False)

Output:

```
Enter the start state matrix

1 2 3
4 5 6
_ 7 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _




  |
  |
 \'/

1 2 3
4 5 6
_ 7 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/

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

22/12/2023

5 Lab-6

→ Implement the vacuum cleaner agent.

Algorithm:

⇒ Vacccumn - Cleaning - Agent class :

1 → __init__(self, room1, room2): ⟶ construction
→ initialize the cleaning agent with 2 rooms and set the current room.

2→ def clean_rooms (self): Initiates the cleaning process each rooms, if it calls thes 'clean_room' function to each room & then it calls the 'move_to_next_funtion'.

3→ def clean_room (self, room):
if (room == 'clean'):
return (clean)
elif (room == 'dirty'):
initiates the cleaning process
room == 'clean'
return (clean).

4→ def move_to_next_room:
Count = 0:
→ Switches the current room to adjacent room.
if Count == 2:
end
else:
Move_to_next_room

Bafna Gold
Date: Page:

Code :-

```
class VacuumCleaner:

    def __init__(self, initial_location):
        self.location = initial_location

    def move_left(self):
        print("Moving Left")
        self.location = 'A'

    def move_right(self):
        print("Moving right")
        self.location = 'B'

    def suck(self, room):
        print("Sucking dirt in {room}")
        return 'Clean'

def simulate_cleaning():
    initial_vacuum_location = input("Enter initial location").upper()

    vacuum = VacuumCleaner(initial_vacuum_location)

    room_A_state = input("Enter state for Room A").lower()

    room_A_state = input("Enter state for Room B:").lower()

    rooms = {
        'A': room_A_state,
        'B': room_B_state,
    }
```

```
print("\n Initial state:")
print("Vacuum cleaner is in Room {vacuum.location}")
print("Room A: {rooms['A']}")
print("Room B: {rooms['B']}\n")

if rooms['A'] == 'clean' and rooms['B'] == 'clean':
    print("Both rooms are already clean. No cleaning
needed.")
else:
    print("Starting the cleaning process...")

    current_room = vacuum.location
    cleaned_room = vacuum.suck(current_room)

    if cleaned_room == 'A':
        vacuum.move_right()
        current_room = 'A'

    cleaned_room = vacuum.suck(current_m

    if cleaned_room == 'clean':
        rooms[current_room] = 'clean'

print("\n Cleaning completed.")
print("Final state:")
print("Vacuum cleaner is in room {vacuum.loc
print("Room A: {rooms['A']}")
print("Room B: {room['B']}")

Simulate_Cleaning()
```

Bafna Gold
Date: Page:

Output:

Enter initial location of vacuum cleaner (A/B) : A
Enter State for Room A (clean / dirty) : dirty
Enter State for Room B (clean / dirty) : dirty

Initial State :
Vacuum Cleaner in Room A
Room A : dirty
Room B : dirty

Starting the cleaning process...
Sucking dirt in Room A
Moving right
Sucking dirt in room B

Cleaning completed
Final State :
Vacuum Cleaner is in Room B
Room A : clean
Room B : clean

29/12

Output:

```
0 indicates clean and 1 indicates dirty
Enter Location of VacuumB
Enter status of B0
Enter status of other room1
Vacuum is placed in location B
0
Location B is already clean.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT 1
Cost for SUCK 2
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

```
[ ]  Start coding or generate with AI.
```

29/12/2023

Lab-7

⇒ Q(1) Create a knowledge based using propositional logic & show that the given query entails the knowledge base or not.

Algorithm:

Knowledge Base class:

→ initialization:

```
def __init__(self):
    self.clauses = []
```

→ Create an instance of Knowledgebase class with an empty list

→ Adding a clauses:

Appends a new clause to the list of clauses in the knowledge base.

→ Resolving Clauses:

Combine two clauses by resolving

→ Negate the Query:-

obtain the negation of query.

→ Combine with knowledge Base:

→ Check satisfiability:

```
def satisfiability():
    Check the conjunction is sat
```

Code(1)

```
from sympy import symbols

def create_knowledge_base():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')

    knowledge_base = And(Implies(P,q), Implies(q,
                                        Not(r)))
    return knowledge_base

def query_entails(knowledge_base, query):
    entailment = satisfiable(And(knowledge_base,
                                 Not(query)))
    return not entailment

if __name__ == "__main__":
    Kb = Create_knowledge_base()
    query = symbols('p')
    result = query_entails(Kb, query)

    print("knowledge Base", Kb)
    print("Query", query)
    print("query entail knowledg base", result)
```

Output:

knowledge base : q & r & (implies(P,q)) & (implies(q,r))

Query : P

Query entails knowledge base : false.

Output:

```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

```
[ ]  Start coding or generate with AI.
```

7 ⇒ (2) Knowledged Based ~~entailment~~ resolution:

Algorithm:-

Initialization:

⇒ Create an instance of knowledge Base class with an empty list of clauses:-

self.clauses = []

⇒ Adding a clause:-

append a new clause to the list of clauses in the knowledge Base.

⇒ Resolving clause:-

combine 2 clauses by ~~Resolving literals~~ common literals

(Eliminating complementary literals)

Negate the query & adds it to the knowledge base

⇒ Repeatedly resolve the pairs of clauses in the knowledge base until a contradiction found or No new resolution are possible.

def resolve (self, clause_a, clause_b)

return [literal for literal in clause_a + clause_b if ('not' + literal) not in clause_a or literal not in clause_b]

Bafna Gold
Date: Page:

code (2) def negate-literal (literal)
if literal[0] == '~':
return literal[1:]
else:
return '~' + literal

def resolve (c1, c2):
resolved-clause = set(c1) | set(c2)

for literal in c1:
if negate-literal (literal) in c2:
resolved-clause.remove (literal)
resolved-clause.remove (negate-literal (literal))
return tuple (resolved-clause)

def resolution (knowledge-base):

while True:
new clauses = set()
for i, c1 in enumerate (knowledge-base)
for j, c2 in enumerate (knowledge-base):
if i != j:
new-clause = resolve (c1, c2)
if len (new-clause) > 0 & new-clause not in knowledge-base:
new-clause.add (new-clause)

if not new-clause
break

```
        knowledge_base != new_clause
    return knowledge_base

if __name__ == "__main__":
    KB = set([('P', 'Q'), ('~P', 'R'), ('~Q', '~R')])
    result = resolution(KB)
    print("Original KB", KB)
    print("Resolved KB", result)
```

Output:-

Enter statement = ~P negation.

The statement is not entailed by knowledge-based

```
rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR
goal = 'R'
main(rules, goal)
```

```
Step    |Clause |Derivation
-----------------------------
 1.     | PvQ   | Given.
 2.     | ~PvR  | Given.
 3.     | ~QvR  | Given.
 4.     | ~R    | Negated conclusion.
 5.     | QvR   | Resolved from PvQ and ~PvR.
 6.     | PvR   | Resolved from PvQ and ~QvR.
 7.     | ~P    | Resolved from ~PvR and ~R.
 8.     | ~Q    | Resolved from ~QvR and ~R.
 9.     | Q     | Resolved from ~R and QvR.
 10.    | P     | Resolved from ~R and PvR.
 11.    | R     | Resolved from QvR and ~Q.
 12.    |       | Resolved R and ~R to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

19/01/2024

Bafna Gold
Date: Page:

Lab - 8

(1) Unification :- Algorithm:

Eg: knows (John, x) knows (John, Jane)
{ x / Jane }

Step 1 : If term1 or term2 is a variable or constant then:

(a) term1 or term2 are identical return NIL

(b) Else if term is available
if term1 occurs in term2
return FAIL

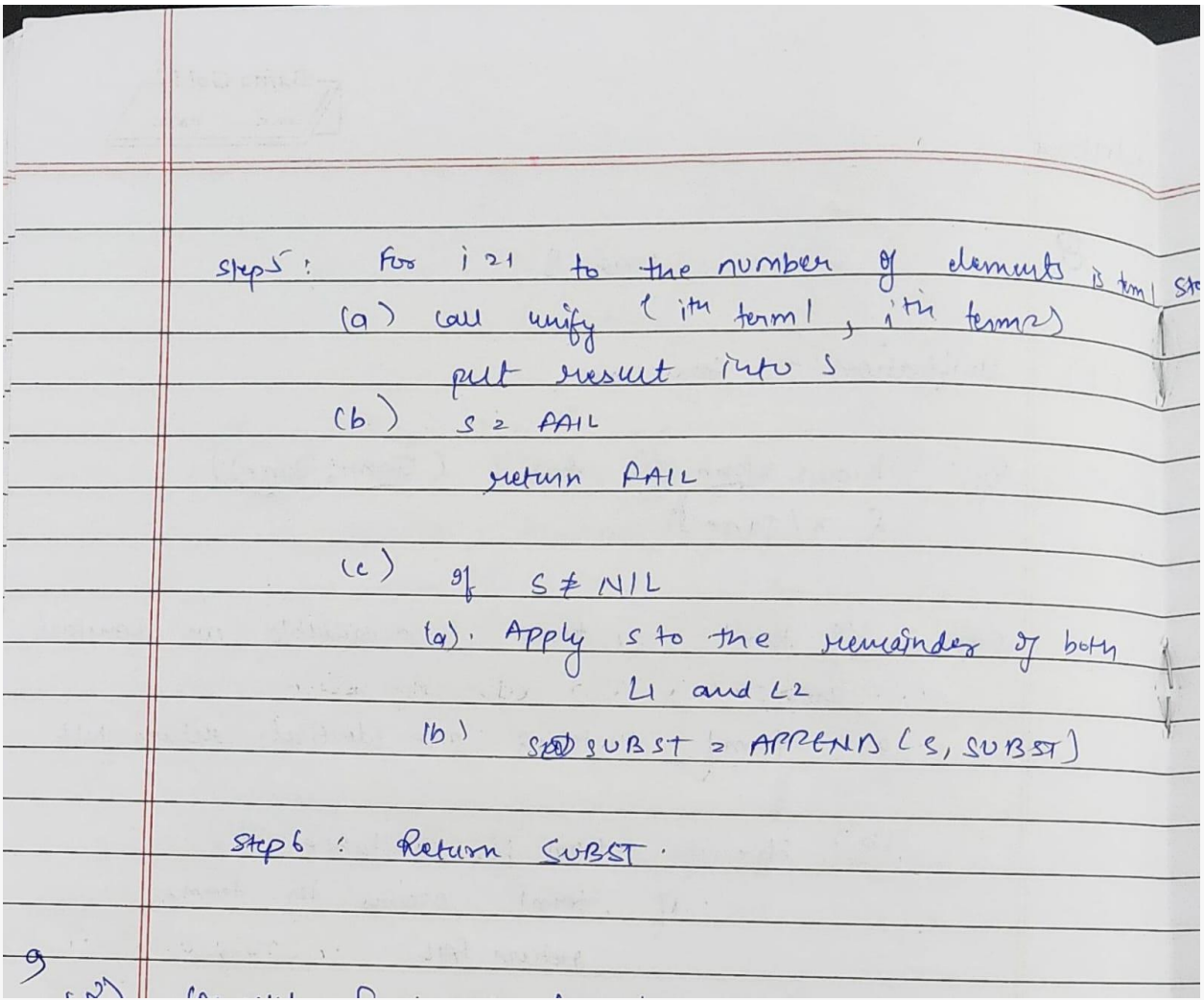
(c) else if term 2 is a variable.
if term 2 occurs in term1
return FAIL
Else
return { (term 1 / term 2) }

(d) else return FAIL

Step 2 : If predicate (term1) ≠ predicate (term2)
return FAIL

Step 3 : number of argument ≠
return FAIL

Step 4 : set (SUBST) to NIL



Step 5 : For i 21 to the number of elements in term1 Ste

(a) call unify (ith term1 , ith term2)
put result into S

(b) S = FAIL
return FAIL

(c) If S ≠ NIL
(a). Apply S to the remainder of both
L1 and L2
(b) SUBST = APPEND (S, SUBST)

Step 6 : Return SUBST.

Code :-

18 ⇒ unification:

```
(1)  import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?!\(.),(?!.\))", expression)
    return expression

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

def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
```

```
predicate = getInitialPredicate(exp)
return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitution
    for substitution in substitution:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp.


def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return false
    return true.

def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]

def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return false

    if isConstant(exp1)
        return [(exp2, exp1)]
```

```
if isConstant(exp2):
    return [(exp2, exp1)]

if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicates do not match")
    return False.

attributesCount1 = len(getAttributes(exp1))
attributesCount2 = len(getAttributes(exp2))

head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2)

if not initialSubstitution:
    return False
if attributeCount1 == 1:
    return initialSubstitutions

tail1 = getRemainingPart(exp1)
tail2 = getRemainingPart(exp2)

exp1 = "Knows(A,x)"
exp2 = "Knows(y,y)"

Substitution = unify(exp1, exp2)
print("Substitutions:")
print(Substitutions)
```

Date: Page:

```
exp1 = "knows (A,x)"
exp2 = "knows ( y, Mother (y))"
substitution = unify ( exp1 , exp2 )
print ("substitution :")
print ( substitution )
```

Output :-

① substitution :
[('A', 'y'), ('y', 'x')]

substitution
[('A', 'y'), ('Mother(y)', 'x')]

Output:

```
exp1 = "knows(X)"
exp2 = "knows(Richard)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

```
Substitutions:
[('X', 'Richard')]
```

```
exp1 = "knows(A,x)"
exp2 = "knows(y,mother(y))"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

```
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

9

(a) Convert first order logic to CNF conversion:
Algorithm:-

Step1: Create a list of SKOLEM CONSTANTS. → A, B, C, D

Step2: find ∀, ∃

- If the attributes are lower case, replace them with a skolem constant.
- remove use skolem constant or function from the list
- If the attributes are both lowercase and uppercase replace the uppercase attribute with a skolem function

Step3: replace ⟺ with '→' $P \Leftrightarrow Q$

transform → as $Q \equiv (P \Rightarrow Q) \wedge (Q \Rightarrow P)$

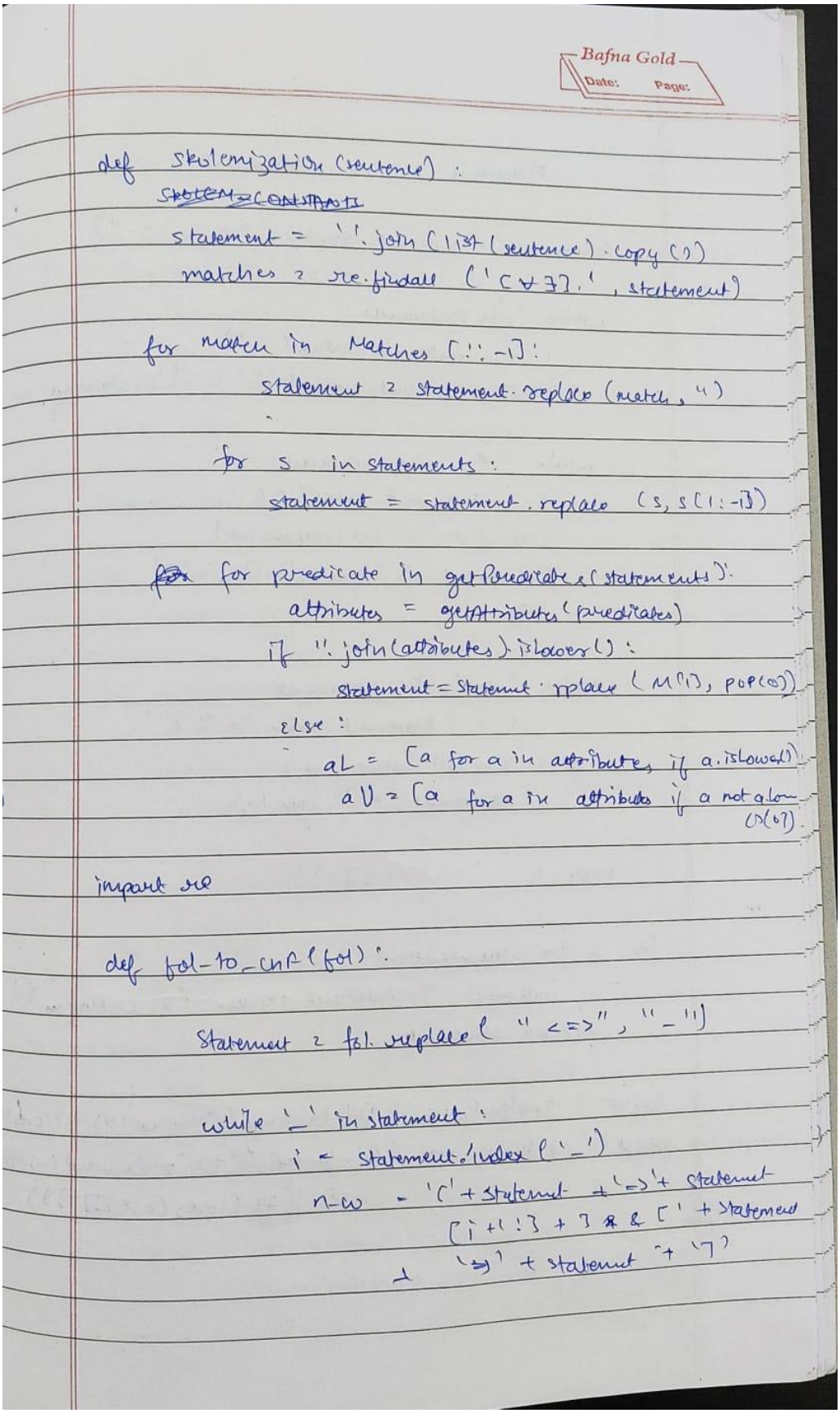
Step 4: replace ⇒ with '→' $P \Rightarrow Q$

transform → as $Q \equiv \sim P \vee Q$

9 Convert FOL to CNF

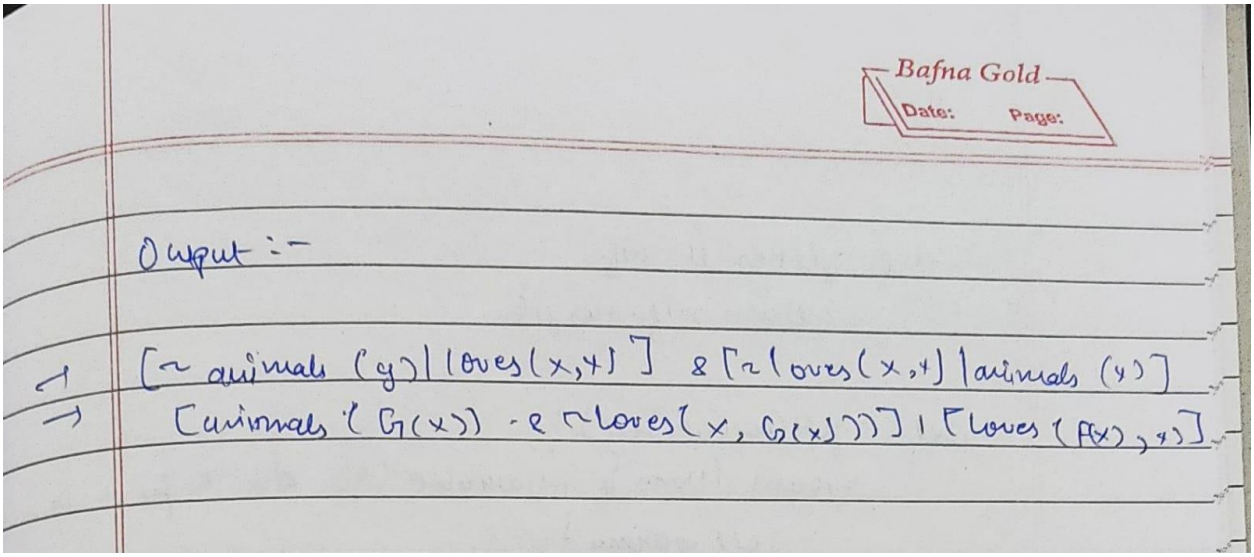
(Q) Code

```
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicate(string)
    expr : '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence)
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string

    for predicate in getPredicate(string)
        string = string.replace(predicate, f'~{predicate}')

    s = list(string)

    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&'
            s[i] = '|'

    return f'[{string}]' if flag else string
```



```
def skolemization(sentence):
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[A-Z].', statement)

    for match in matches[::-1]:
        statement = statement.replace(match, "")

    for s in statements:
        statement = statement.replace(s, s[1:-1])

    for predicate in getPredicates(statements):
        attributes = getAttributes(predicates)
        if ''.join(attributes).islower():
            statement = statement.replace(M[1], pop(0))
        else:
            aL = [a for a in attributes if a.islower()]
            aU = [a for a in attributes if a not a.lower()][0]

import re

def fol_to_cnf(fol):

    Statement = fol.replace("<=>", "_")

    while '_' in statement:
        i = statement.index('_')
        new = '(' + statement + '->' + statement
              [i+1:] + ')&(' + statement
        + '=>' + statement + ')'
```

```
statement = new_statement

Statement = statemnt.replace("=>", "-")

while '-' in statement:
    i = statement.index('-')
    br = statement.index('(') if '(' in statement else 0

while '~A' in statement:
    i = statement.index('~A')
    statement = list(statement)
    statement[i]; statement[i+1] = '∃'


while '~∃' in statement:
    i = statement.index('~∃')
    s = list(statement)
    statement = ''.join(s)

expr = '(x[A|∃].)'

for s in statements:
    statement = statement.replace(s, DeMorgan(s))
return statement

print(skolemization(fol_to_cnf("animal(y) => loves(x,y)")))
print(skolemization(fol_to_cnf("∀x[∀y[animal(y) => loves
                    => ∃z[loves(z,x)]]")))
```



Output:

In [3]:
```
print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))
```

```
[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

(3) Crack a knowledge consisting of FOL statements and prove the give query using forward chaining, memory

Algorithm:-

1) Initialize the Agenda:-
   a) add the query to the agent.

2) while the agent do is not empty:

a) pop a statement from the agenda.

b) if the statement is already known, continues to reject statement

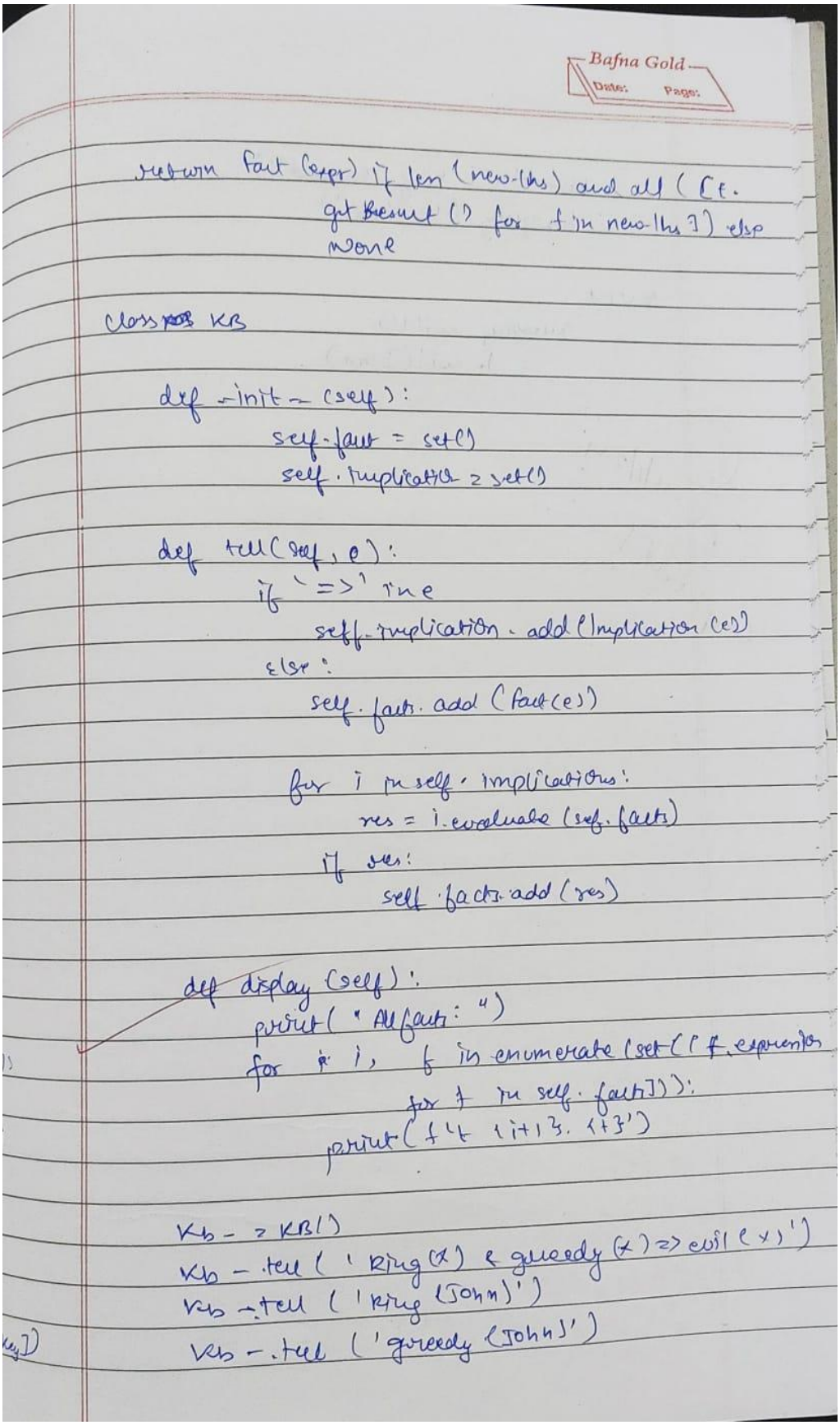
c) if the statement is a law, add it to the set of known fold.

d). If the statement is a rule, apply the rule to the generate new statements & add them to the agenda.

13) Code for KB consisting for + prove the given query using forward reasoning.

```
import re

def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicate(string):
    expr = '([a-zA-Z]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())
```

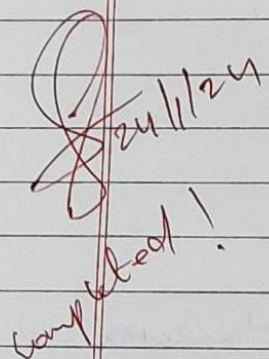
```
def getResult(self):
    return self.result

def getConstant(self):
    return (None if isVariable(c) else c for c in
        self.params)


Class Implication:
    def __init__(self, expression):

        self.expression = expression
        l = expression.split('=>')
        self.lhs = [fact(f) for f in l[0].split('&')]
        self.rhs = fact(l[1])

def evaluate(self, facts):
    constants = {}

    new_lhs = []
    for fact in facts:
        for val in self.lhs:
            if val.predicate == fact.predicate:
                for i, v, in enumerate(val.getVariables()):
                    if v:
                        constants[v] = fact.getConstants()[i]
                new_lhs.append(fact)
    for key in constants:
        if constants[key]:
            attributes = attributes.replace(key, constants
```


Bafna Gold
Date: Page:
return fact(expr) if len(new_lhs) and all([f.
getResult() for f in new_lhs]) else
None
class KB
def __init__(self):
self.facts = set()
self.implications = set()
def tell(self, e):
if '=>' in e
self.implication.add(Implication(e))
else:
self.facts.add(Fact(e))
for i in self.implications:
res = i.evaluate(self.facts)
if res:
self.facts.add(res)
def display(self):
print("All facts:")
for i, f in enumerate(set([f.expression
for f in self.facts])):
print(f'{i+1}. {f}')
kb = KB()
kb.tell('King(x) & greedy(x) => evil(x)')
kb.tell('King(John)')
kb.tell('greedy(John)')

```
kb.tell('king(Richard)')
kb.query('evil(x)')
```

output

```
Querying evil(x)
    1. evil(John)
```



completed!

Output:

In [ ]:

```
kb_ = KB()
kb_.tell('king(x)&greedy(x)=>evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

```
Querying evil(x):
        1. evil(John)
```