VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sagar Bangari(1BM22CS231)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Sagar Bangari(1BM22CS231)**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.

Prof. Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

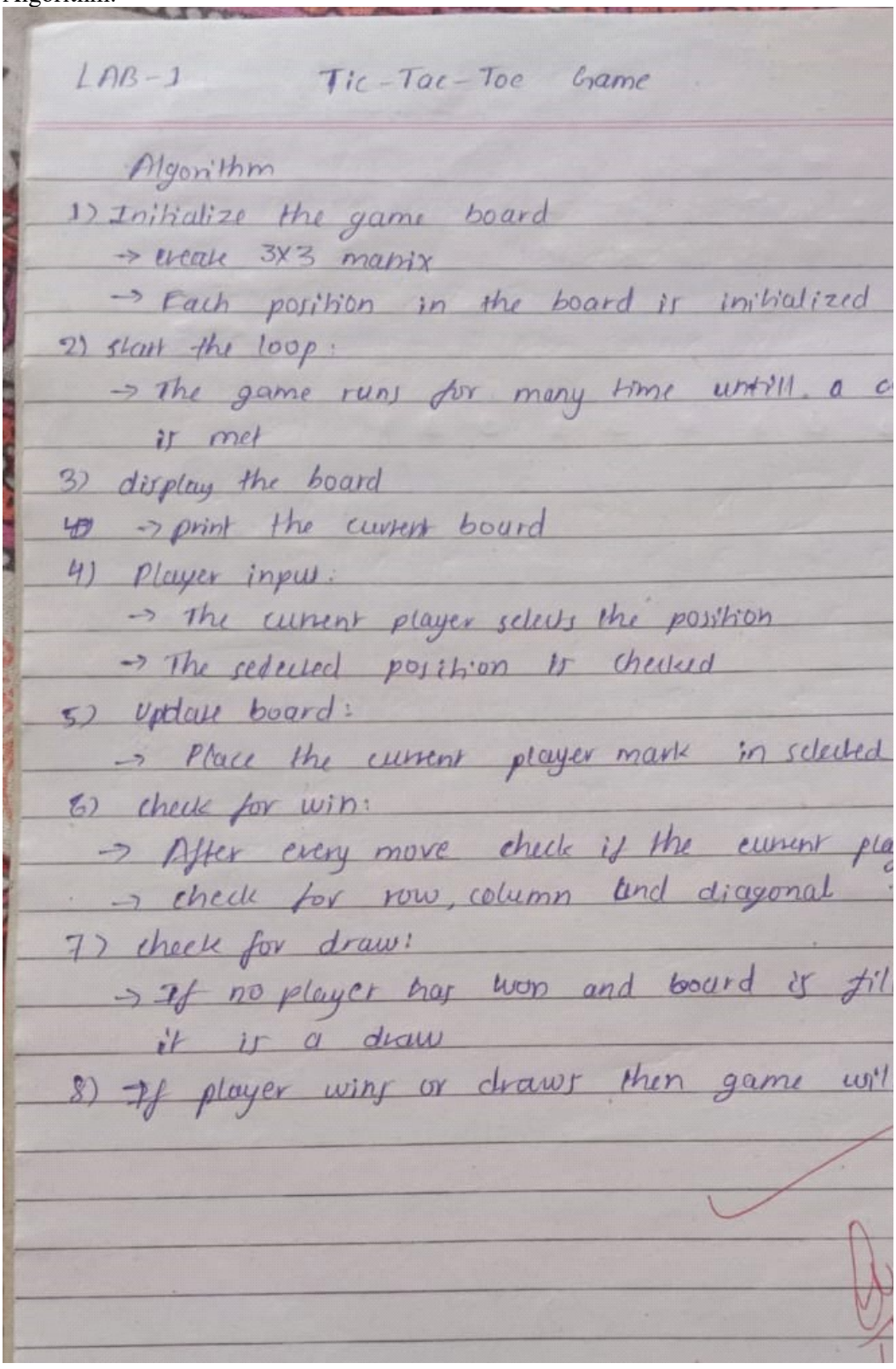
Github Link:
https://github.com/Sagar-bangari/AI

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

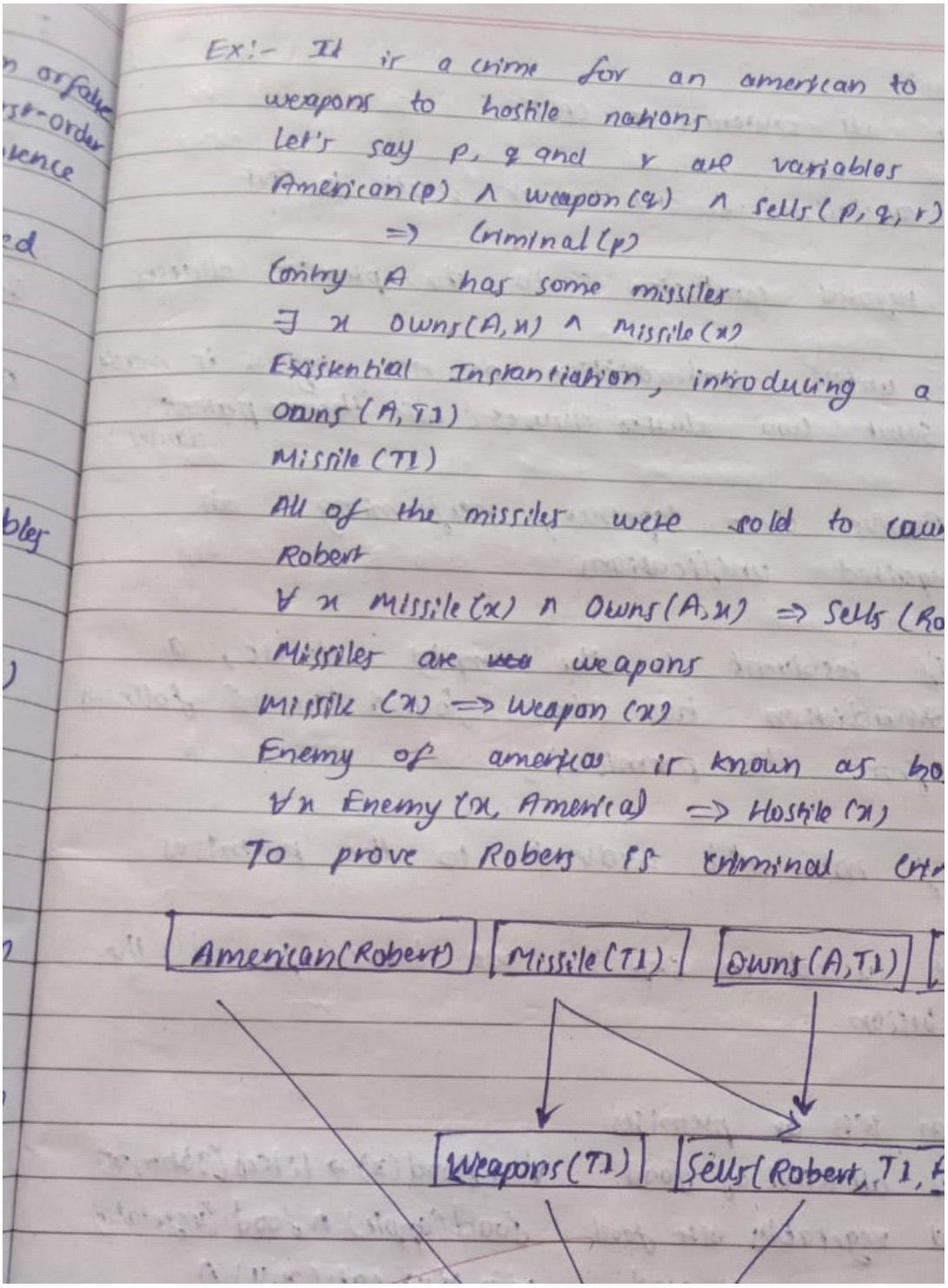
Algorithm:



LAB-1 Tic-Tac-Toe Game

Algorithm

1) Initialize the game board
   → create 3X3 matrix
   → Each position in the board is initialized
2) start the loop:
   → The game runs for many time until a c
   is met
3) display the board
4) → print the current board
4) Player input:
   → The current player selects the position
   → The selected position is checked
5) Update board:
   → Place the current player mark in selected
6) check for win:
   → After every move check if the current pla
   → check for row, column and diagonal
7) check for draw:
   → If no player has won and board is fil
   it is a draw
8) If player wins or draws then game will



Code: 1: Tic - Tac - Toe

```
import numpy as np

board=np.array([['-','-','-'],['-','-','-'],['-','-','-']])
```

```
current_player='X'
flag=0

def check_win():
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != '-':
            return True
    for i in range(3):
        if board[0][i] == board[1][i] == board[2][i] != '-':
            return True
    if board[0][0] == board[1][1] == board[2][2] != '-':
        return True
    if board[0][2] == board[1][1] == board[2][0] != '-':
        return True
    return False

def tic_tac_toe():
    n=0
    print(board)
    while n<9:
        if n%2==0:
            current_player='X'
        else :
            current_player='O'

        row = int(input("Enter row: "))
        col = int(input("Enter column: "))

        if(board[row][col]=='-'):
            board[row][col]=current_player
            print(board)
            flag=check_win();
            if flag==1:
                print(current_player+' wins')
                break
            else:
                n=n+1
        else :
            print("Invalid Position")

    if n==9:
        print("Draw")

tic_tac_toe()
```

Output:

Win :

```
[['-' '-' '-'] ['-' '-' '-'] ['-' '-' '-']]
```

```
Enter row: 0
Enter column: 1
[['-' 'X' '-'] ['-' '-' '-'] ['-' '-' '-']]
Enter row: 0
Enter column: 0
[['O' 'X' '-'] ['-' '-' '-'] ['-' '-' '-']]
Enter row: 1
Enter column: 0
[['O' 'X' '-'] ['X' '-' '-'] ['-' '-' '-']]
Enter row: 1
Enter column: 1
[['O' 'X' '-'] ['X' 'O' '-'] ['-' '-' '-']]
Enter row: 1
Enter column: 2
[['O' 'X' '-'] ['X' 'O' 'X'] ['-' '-' '-']]
Enter row: 2
Enter column: 2
[['O' 'X' '-'] ['X' 'O' 'X'] ['-' '-' 'O']]
O wins

Draw :
[['-' '-' '-'] ['-' '-' '-'] ['-' '-' '-']]
Enter row: 1
Enter column: 1
[['-' '-' '-'] ['-' 'X' '-'] ['-' '-' '-']]
Enter row: 0
Enter column: 1
[['-' 'O' '-'] ['-' 'X' '-'] ['-' '-' '-']]
Enter row: 1
Enter column: 0
[['-' 'O' '-'] ['X' 'X' '-'] ['-' '-' '-']]
Enter row: 1
Enter column: 2
[['-' 'O' '-'] ['X' 'X' 'O'] ['-' '-' '-']]
Enter row: 2
Enter column: 2
[['-' 'O' '-'] ['X' 'X' 'O'] ['-' '-' 'X']]
Enter row: 0
Enter column: 0
[['O' 'O' '-'] ['X' 'X' 'O'] ['-' '-' 'X']]
Enter row: 0
Enter column: 2
[['O' 'O' 'X']['X' 'X' 'O'] ['-' '-' 'X']]
Enter row: 2
Enter column: 0
[['O' 'O' 'X'] ['X' 'X' 'O'] ['O' '-' 'X']]
Enter row: 2
Enter column: 1
[['O' 'O' 'X'] ['X' 'X' 'O'] ['O' 'X' 'X']]
Draw
```

2. Vacuum Cleaner :

```
cost =0
def vacuum_world(state, location):
  global cost
  if(state['A']==0 and state['B']==0):
    print('All rooms are clean')
    return

  if state[location]==1:
    state[location]=0
    cost+=1
    state[location]=(int(input('Is room '+ str(location) +' still dirty : ')))

    if state[location]==1:
      return vacuum_world(state, location)
    else:
      print('Room ' + str(location) + ' cleaned')

  next_location='B' if location=='A' else 'A'
  if state[next_location]==0:
    state[next_location]=(int(input('Is room '+ str(next_location) +' dirty : ')))
  print('Moving to room '+str(next_location))
  return vacuum_world(state, next_location)

state={}
state['A']=int(input('Enter status of room A : '))
state['B']=int(input('Enter status of room B : '))
location=input('Enter initial location of vacuum (A/B) : ')
vacuum_world(state,location)
print("Status = "+str(state))
print('Total cost: ' + str(cost))
```

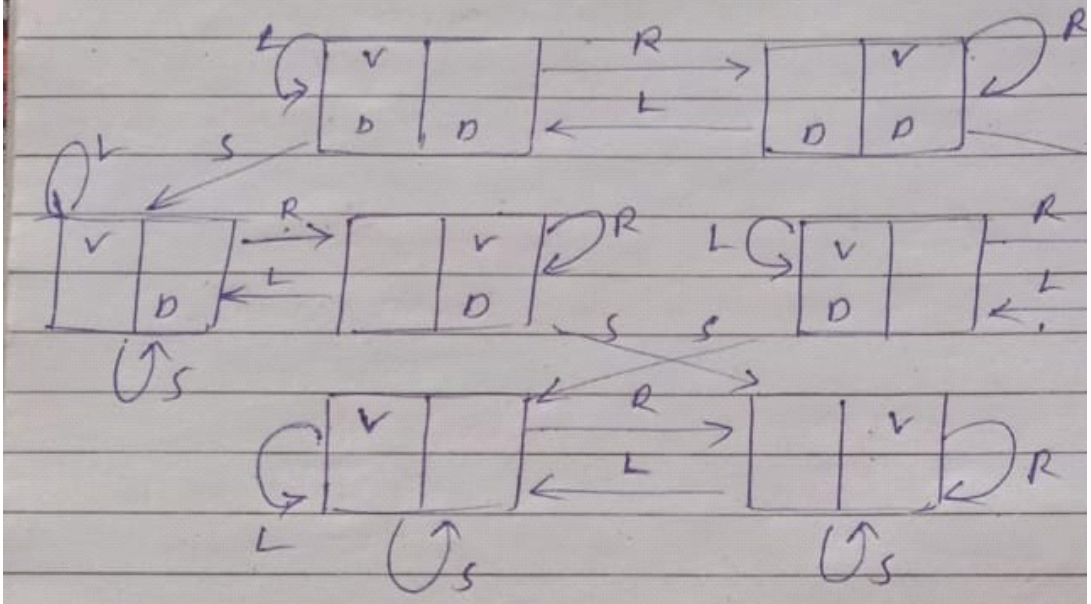
Output :

```
Enter status of room A : 1
Enter status of room B : 1
Enter initial location of vacuum (A/B) : A
Is room A still dirty : 0
Room A cleaned
Moving to room B
Is room B still dirty : 0
Room B cleaned
Is room A dirty : 0
Moving to room A
All rooms are clean
```

Vaccum world cleaner

Pseudocode :-

function REFLEX-VACCUM-AGENT ([location, status])
    if status = Dirty then return suck
    else if location = A then return Right
    else if location = B then return left



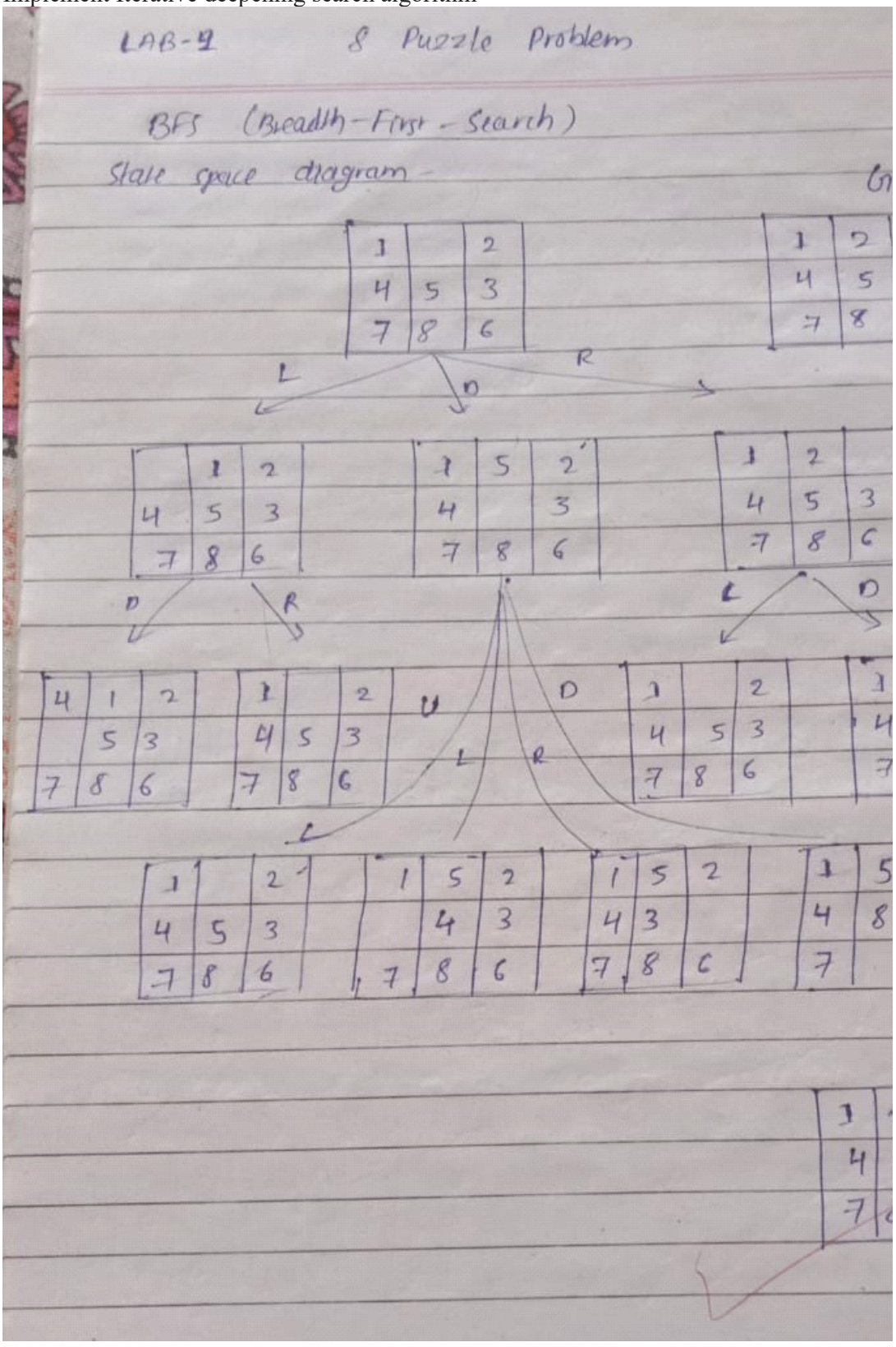
Algorithm :-

1) Make an array to hold data of dirt world
2) Take input from user about where th is and where the dust is
3) Initialize cost of path to 0
4) ~~Create~~ algorithms suck, left and right
5)
4) Create a function suck that will make clean and increases cost
5) Create functions left and right which move robot to respective locations.
6) Create check function that will check boxes are clean
7) create vaccum_cleaner function that will location and status and returns an among suck, right and left
8) Create an infinite loop that will be when all positions are clean 0
   It calculates robot position and cos vaccum cleaner function.
   status is obtained from this fo for next iteration

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

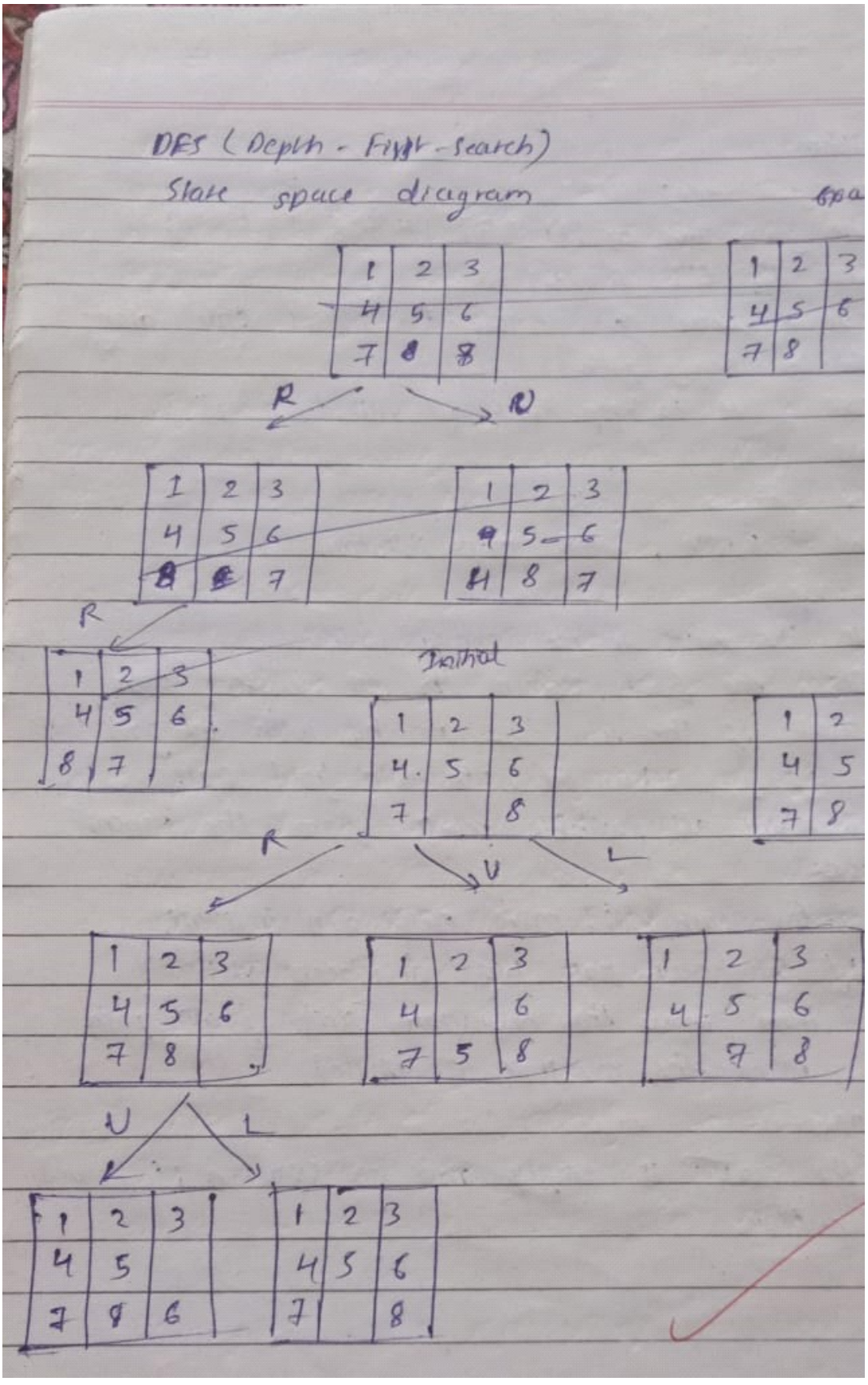


Algorithm:-

1) Define goal-state:
   Set the goal state as [[1, 2, 3], [4, 5, 6], [7, 8
2) Initialize the BFS structure
   → Create a queue and enque the initial s
   with an empty path
   → create a set to keep track of visited state
   avoid cycle
3) Define helper functions
   → find_blank_tile, and get_possible_move
4) BFS Loop
   → While the queue is not empty
      1) Deque the front element to get curre
      2) check if the curr state matches with
         If, yes return the path with goal s
      3) Generate all possibble moves from th
      state
         for each valid move, create a new s
      If this new state has not been vi
         then enque the new state and upd
         mark the new state as visited
5) Return:
   If the queue is empty and no solution
   return None

DFS (Depth - First - Search)

State space diagram



Algorithm:-

1) Define the goal state
   Set the goal state as [[1,2,3],[4,5,0],[
2) Initialize the DFS structure
   → create a stack and push initial state
   with an empty path
   → create a set to keep track of visi
   states to avoid cycle
3) Define helper functions
   → find_blank_tile and get_possible moves
4) DFS loop
   → While the stack is not empty
   1) pop the top element to get current
   2) check if current state matches w
   state if yes return the path of
   3) Generate all possible moves from
   For each valid move create a new
   If this new state has not been vi
   push the new state and update
   mark the new state as visited
5) Return:-
   If the stack is empty and no solutio
   then return None

Wa 10/24

Code:
1: DFS

```
cnt = 0;
def print_state(in_array):
 global cnt
 cnt += 1
 for row in in_array:
  print(' '.join(str(num) for num in row))
 print()

def helper(goal, in_array, row, col, vis):

 vis[row][col] = 1
 drow = [-1, 0, 1, 0]
 dcol = [0, 1, 0, -1]
 dchange = ['U', 'R', 'D', 'L']

 print("Current state:")
 print_state(in_array)

 if in_array == goal:
  print_state(in_array)
  print(f"Number of states : {cnt}")
  return True


 for i in range(4):
  nrow = row + drow[i]
  ncol = col + dcol[i]

  if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not vis[nrow][ncol]:

    print(f"Took a {dchange[i]} move")
    in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]

    if helper(goal, in_array, nrow, ncol, vis):
      return True

    in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]


 vis[row][col] = 0
 return False


iniϴal_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)]
empty_row, empty_col = 1, 0
```

found_soluΘon = helper(goal_state, iniΘal_state, empty_row, empty_col, visited)
print("SoluΘon found:", found_soluΘon)

Output :

```
Current state:
1 2 3
0 4 6
7 5 8

Took a U move
Current state:
0 2 3
1 4 6
7 5 8

Took a R move
Current state:
2 0 3
1 4 6
7 5 8

Took a R move
Current state:
2 3 0
1 4 6
7 5 8

Took a D move
Current state:
2 3 6
1 4 0
7 5 8

Took a D move
Current state:
2 3 6
1 4 8
7 5 0

Took a L move
Current state:
```

```
Took a L move
Current state:
2 3 6
1 4 8
0 7 5

Took a L move
Current state:
2 3 6
1 0 4
7 5 8

Took a D move
Current state:
2 3 6
1 5 4
7 0 8

Took a R move
Current state:
2 3 6
1 5 4
7 8 0

Took a L move
Current state:
2 3 6
1 5 4
0 7 8

Took a D move
Current state:
2 4 3
1 0 6
7 5 8

Took a R move
Current state:
```

```
Took a L move
Current state:
1 0 2
4 6 3
7 5 8

Took a L move
Current state:
0 1 2
4 6 3
7 5 8

Took a D move
Current state:
1 2 3
4 6 8
7 5 0

Took a L move
Current state:
1 2 3
4 6 8
7 0 5

Took a L move
Current state:
1 2 3
4 6 8
0 7 5

Took a D move
Current state:
1 2 3
4 5 6
7 0 8

Took a R move
Current state:
1 2 3
```

2 : Iterative deepening search

```
class PuzzleState:
    def __init__(self, board, empty_tile_pos, depth=0, path=[]):
        self.board = board
        self.empty_tile_pos = empty_tile_pos  # (row, col)
        self.depth = depth
        self.path = path  # Keep track of the path taken to reach this state

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

    def generate_moves(self):
        row, col = self.empty_tile_pos
        moves = []
        directions = [(-1, 0, 'Up'), (1, 0, 'Down'), (0, -1, 'Left'), (0, 1, 'Right')]  # up, down, left,
right
        for dr, dc, move_name in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] =
new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                new_path = self.path + [move_name]  # Update the path with the new move
                moves.append(PuzzleState(new_board, (new_row, new_col), self.depth + 1,
new_path))
        return moves

    def display(self):
        # Display the board in a matrix form
        for i in range(0, 9, 3):
            print(self.board[i:i + 3])
        print(f"Moves: {self.path}")  # Display the moves taken to reach this state
        print()  # Newline for better readability

def iddfs(initial_state, goal, max_depth):
    for depth in range(max_depth + 1):
        print(f"Searching at depth: {depth}")
        found = dls(initial_state, goal, depth)
        if found:
            print(f"Goal found at depth: {found.depth}")
            found.display()
            return found
    print("Goal not found within max depth.")
    return None

def dls(state, goal, depth):
    if state.is_goal(goal):
        return state
```

```
    if depth <= 0:
        return None

    for move in state.generate_moves():
        print("Current state:")
        move.display()  # Display the current state
        result = dls(move, goal, depth - 1)
        if result is not None:
            return result
    return None

def main():
    # User input for initial state, goal state, and maximum depth
    initial_state_input = input("Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3
4 5 6 7 8 0'): ")
    goal_state_input = input("Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5
6 7 8 0'): ")
    max_depth = int(input("Enter maximum depth: "))

    initial_board = list(map(int, initial_state_input.split()))
    goal_board = list(map(int, goal_state_input.split()))
    empty_tile_pos = initial_board.index(0) // 3, initial_board.index(0) % 3  # Calculate the
position of the empty tile

    initial_state = PuzzleState(initial_board, empty_tile_pos)

    solution = iddfs(initial_state, goal_board, max_depth)

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

Output :

```
Enter initial state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0')
Enter goal state (0 for empty tile, space-separated, e.g. '1 2 3 4 5 6 7 8 0'): 1
Enter maximum depth: 2
Searching at depth: 0
```

```
Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Searching at depth: 2
Current state:
[0, 2, 3]
[1, 4, 6]
[7, 5, 8]
Moves: ['Up']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Up', 'Down']

Current state:
[2, 0, 3]
[1, 4, 6]
```

Iterative deepning depth first search for 8 puzzle problem

Algorithm :-

1) For each child of the current node
2) If it is target node, then return
3) If the current maximum depth is reached ret
4) set the current node to this node and go
5) After having gone through all children, go next child of parent (the next sibling)
6) After having gone through all children of S increase the maximum depth and go back t
7) If we have reached ~~out~~ at leaf nodes, goal nodes does not exist

function Iterative-deepning-search (problem) returns
    for depth = 0 to ∞ do
        res ← depth_limited_search (problem, d
    if (res ≠ cutoff) then return result.

depth limit = 1

Initial state

1	2	3
4	5	6
7	8	

Go

1		
4		
7		

R

1	2	3
4	5	6
7	8	

U

1	2	3
4		6
7	5	8

L

1	2	3
4	5	6
	7	8

depth limit = 2

1	2	3
4	5	6
7		8

R

1	2	3
4	5	6
7	8	

U

1	2	3
4		6
7	5	8

L

1	2	
4	5	
	7	

U

1	2	3
4	5	
7	8	6

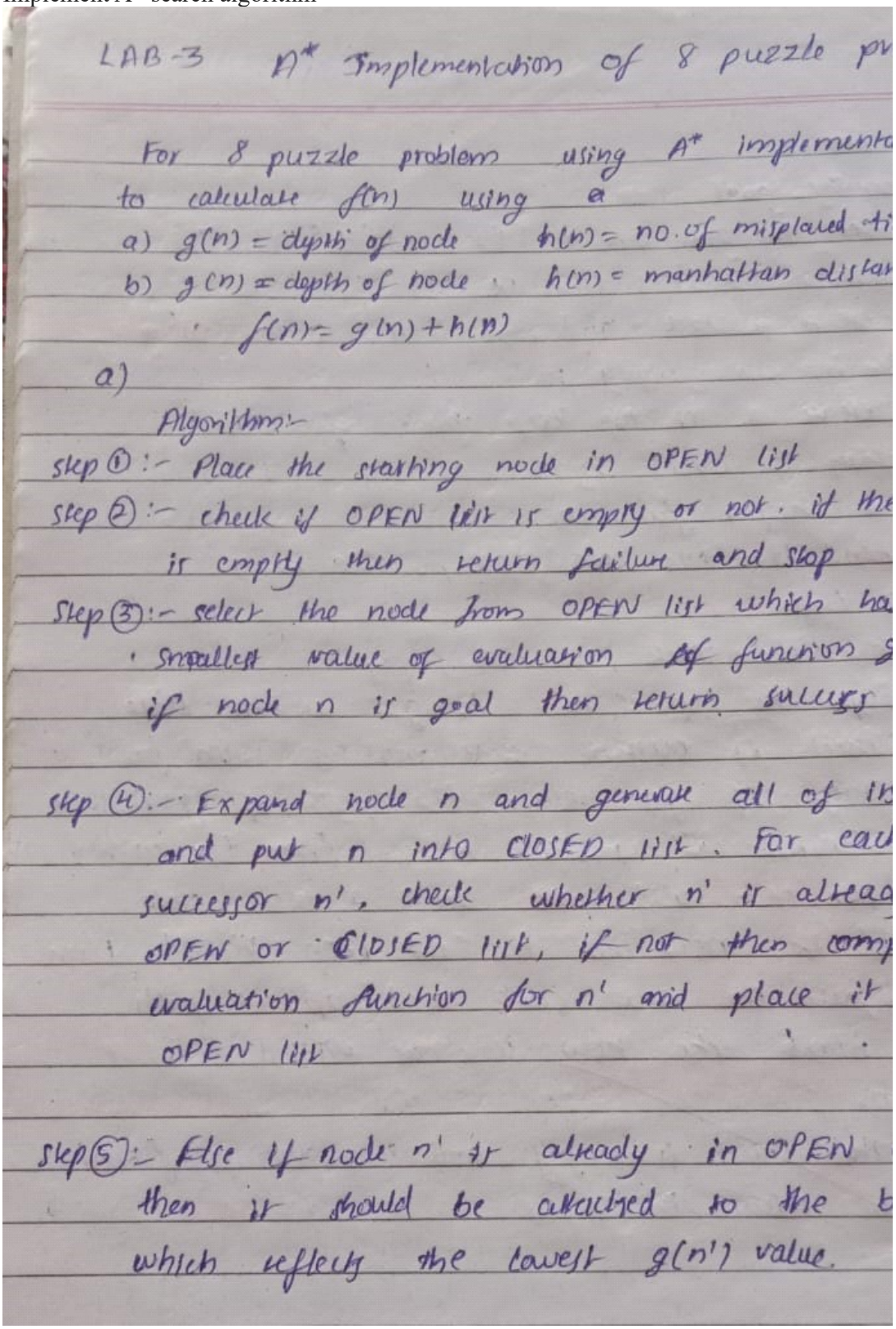
L

1	2	3
4	5	6
7		8

```
Current state:
[1, 2, 3]
[7, 4, 6]
[0, 5, 8]
Moves: ['Down']

Current state:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Moves: ['Down', 'Up']

Current state:
[1, 2, 3]
[7, 4, 6]
[5, 0, 8]
Moves: ['Down', 'Right']

Current state:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]
Moves: ['Right']

Current state:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
Moves: ['Right', 'Up']

Current state:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]
Moves: ['Right', 'Down']

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

Program 3

Implement A* search algorithm



LAB-3 A* Implementation of 8 puzzle pr

For 8 puzzle problem using A* implementa
to calculate f(n) using a
a) g(n) = depth of node h(n) = no. of misplaced ti
b) g(n) = depth of node . h(n) = manhattan distan
. f(n) = g(n) + h(n)

a)

Algorithm:-

step ① :- Place the starting node in OPEN list

step ② :- check if OPEN list is empty or not. if the
is empty then return failure and stop

Step ③ :- select the node from OPEN list which ha
. smallest value of evaluation ~~of~~ function
if node n is goal then return success

step ④ :- Expand node n and generate all of it
and put n into CLOSED list. For eac
successor n', check whether n' is alread
OPEN or CLOSED list, if not then comp
evaluation function for n' and place it
OPEN list

step ⑤ :- Else if node n' is already in OPEN
then it should be attached to the b
which reflects the lowest g(n') value.

b) Algorithm:-

Step ①:- place the starting node in OPEN list.

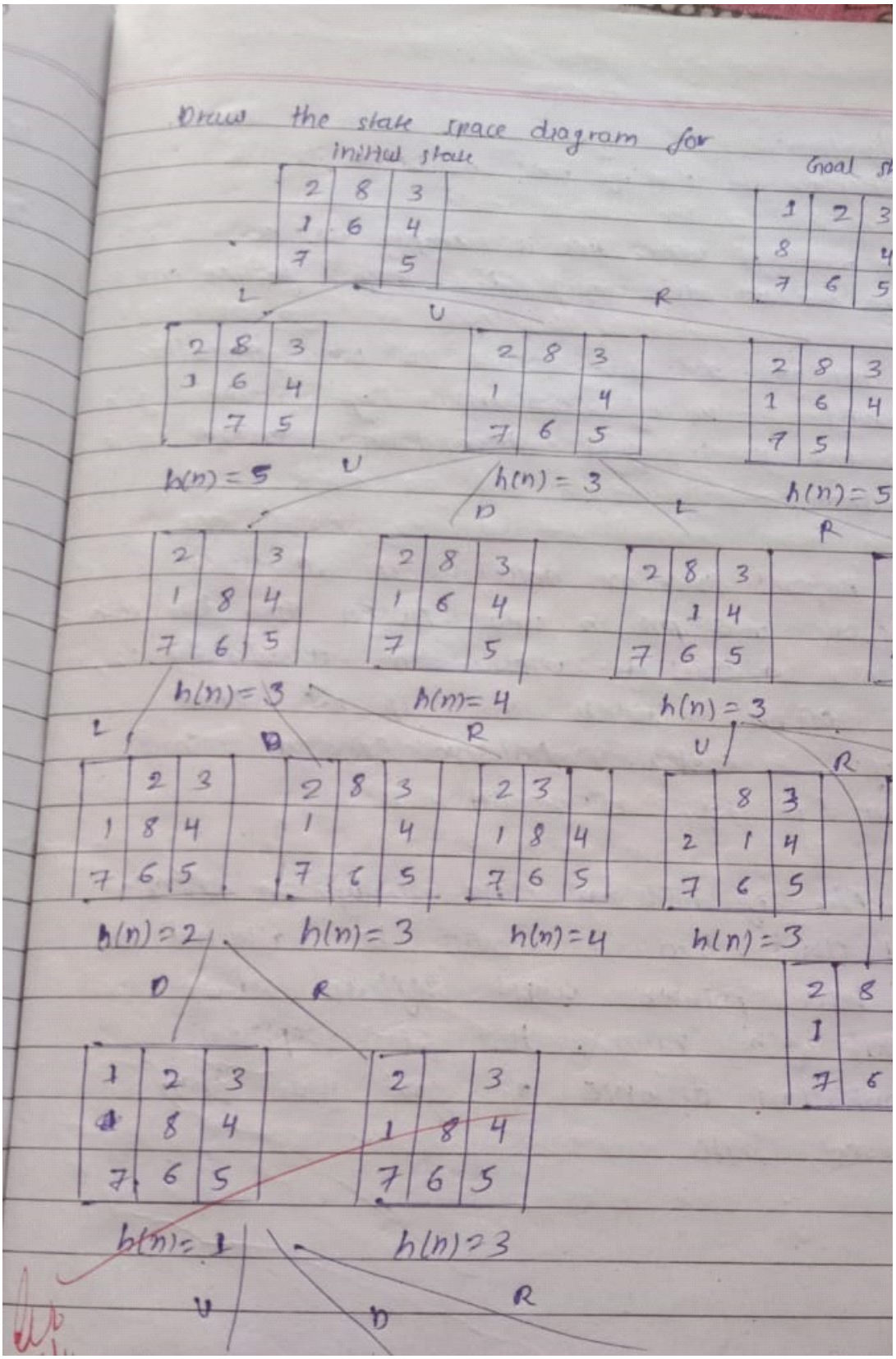
Step ②:- check if OPEN list is empty or no
if the list is empty then return fai

Step ③:- select the node from OPEN list which h
smallest value of evaluation of func
if node n is goal then return suc
and stop

Step ④:- Expand node n and generate all i
successors and put n into CLOSED list.
successor n' check whether n' is al
the OPEN or CLOSED list. If not
compute evaluation function for n'
place it into OPEN list

Step ⑤:- Else if node n' is already in
and closed then it should be att
the back pointer which reflects
lowest g(n') value. Here g(n) repres
manhattan distance of goal state
current state

Draw the state space diagram for



Draw state space diagram for

initial state

2	8	3
1	6	4
7		5

goal

1	2	
8		
7	6	

L U R

2	8	3
1	6	4
	7	5

h(n) = 6

2	8	3
1		4
7	6	5

h(n) = 4

2	8	3
1	6	4
7	5	

h(n) = 6

L R U D

2	8	3
	1	4
7	6	5

h(n) = 5

2	8	3
1	4	
7	6	5

h(n) = 5

2		3
1	8	4
7	6	5

h(n) = 3

L D R

	2	3
1	8	4
7	6	5

h(n) = 2

2	8	3
1		4
7	6	5

h(n) = 4

D R

1	2	3
	8	4
7	6	5

h(n) = 1

2		3
1	8	4
7	6	5

h(n) = 3

D R U

1	2	3
7	8	4

1	2	3
8		4

	2	3
1	8	4

Code:

Misplaced Tiles :

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost

    def heuristic(self):
        goal_state = [[1,2,3], [8,0,4], [7,6,5]]
        count = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                if self.state[i][j] != 0 and self.state[i][j] != goal_state[i][j]:
                    count += 1
        return count

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

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]], new_state[new_blank_pos[0]][new_blank_pos[1]]
= \
        new_state[new_blank_pos[0]][new_blank_pos[1]],
new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)
```

```
    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else "Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (misplaced tiles): {current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state, blank_pos, (new_blank_pos[0],
new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node, move=move_direction,
cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
solution_path = a_star_search(initial_state)

if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
```

```
else:
    print("No solution found.")
```

Output :

```
Current state:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Start
Heuristic value (misplaced tiles): 4
Cost to reach this node: 0

Current state:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Down
Heuristic value (misplaced tiles): 3
Cost to reach this node: 1

Current state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
Heuristic value (misplaced tiles): 3
Cost to reach this node: 2

Current state:
[2, 8, 3]
[0, 1, 4]
[7, 6, 5]
Move: Right
Heuristic value (misplaced tiles): 3
Cost to reach this node: 2

Current state:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Move: Right
Heuristic value (misplaced tiles): 2
Current state:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Move: Left
Heuristic value (misplaced tiles): 0
Cost to reach this node: 5

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

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

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

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

[1, 2, 3]
[0, 8, 4]
```

Code :

Manhattan distance approach

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost

    def heuristic(self):
        goal_positions = {
            1: (0, 0), 2: (0, 1), 3: (0, 2),
            8: (1, 0), 0: (1, 1), 4: (1, 2),
            7: (2, 0), 6: (2, 1), 5: (2, 2)
        }
        manhattan_distance = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                value = self.state[i][j]
                if value != 0:
                    goal_i, goal_j = goal_positions[value]
                    manhattan_distance += abs(i - goal_i) + abs(j - goal_j)
        return manhattan_distance

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

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]], new_state[new_blank_pos[0]][new_blank_pos[1]]
= \
        new_state[new_blank_pos[0]][new_blank_pos[1]],
new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
```

```
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)

    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else "Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (Manhattan distance): {current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state, blank_pos, (new_blank_pos[0],
new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node, move=move_direction,
cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
solution_path = a_star_search(initial_state)
```

```
if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
    print("No solution found.")
```

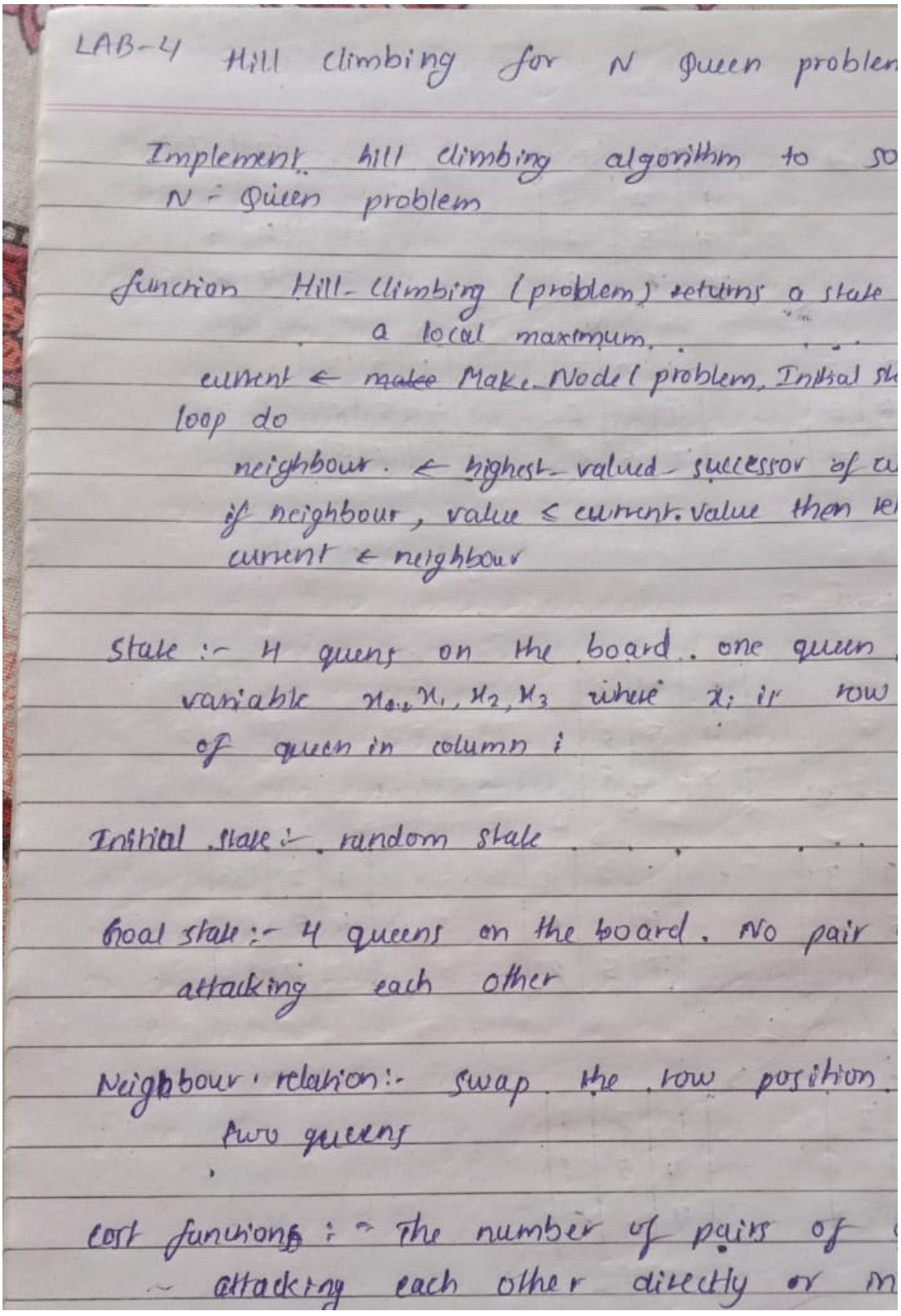
Output :

```
Current state:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Start
Heuristic value (Manhattan distance
Cost to reach this node: 0

Current state:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Down
Heuristic value (Manhattan distance
Cost to reach this node: 1

Current state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
Heuristic value (Manhattan distance
Cost to reach this node: 2

Current state:
```

```
Current state:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
Move: Left
Heuristic value (Manhattan distance)
Cost to reach this node: 5

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

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

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

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

[1, 2, 3]
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem



LAB-4 Hill climbing for N Queen problem

Implement hill climbing algorithm to so
N - Queen problem

function Hill-Climbing (problem) returns a state
a local maximum.

current ← make Make_Node (problem, Initial St
loop do
neighbour. ← highest_valued_successor of cu
if neighbour, value ≤ current.value then re
current ← neighbour

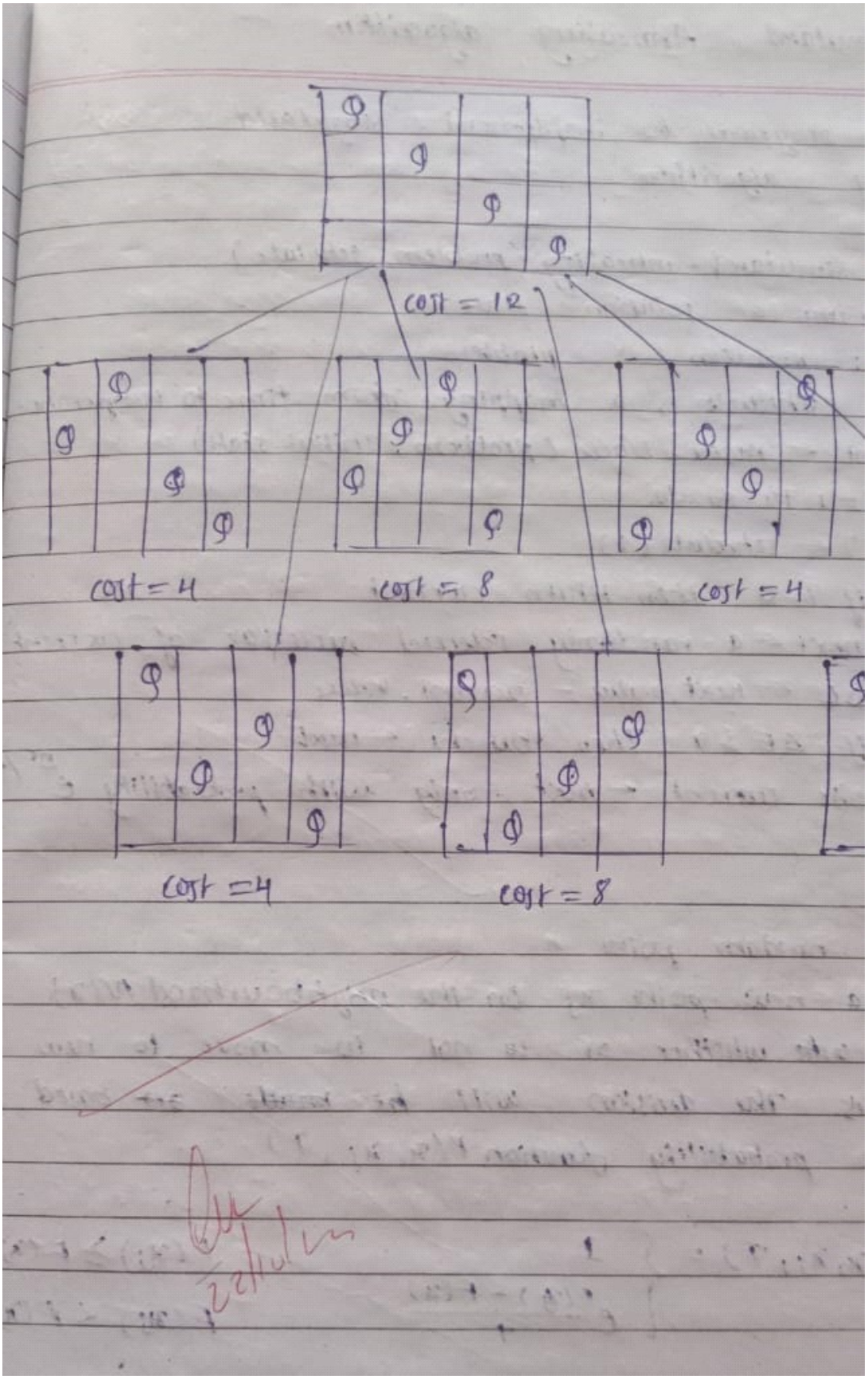
State :- 4 queens on the board. one queen
variable x_0, x_1, x_2, x_3 where x_i is row
of queen in column i

Initial state :- random state

Goal state :- 4 queens on the board. No pair
attacking each other

Neighbour relation :- Swap the row position
two queens

cost functions :- The number of pairs of
attacking each other directly or in


cost = 12
cost = 4
cost = 8
cost = 4
cost = 4
cost = 8

Code:

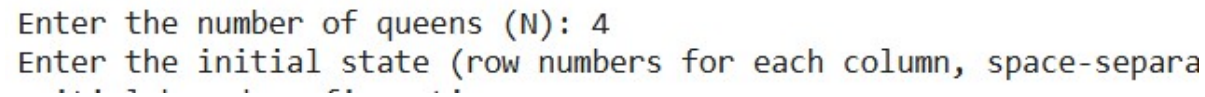
```
import random
def calculate_cost(board):
    n = len(board)
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j]:  # Same column
                attacks += 1
            if abs(board[i] - board[j]) == abs(i - j):  # Same diagonal
                attacks += 1
    return attacks


def get_neighbors(board):
    neighbors = []
    n = len(board)
    for col in range(n):
        for row in range(n):
            if row != board[col]:  # Only change the row of the queen
                new_board = board[:]
                new_board[col] = row
                neighbors.append(new_board)
    return neighbors


def hill_climb(board, max_restarts=100):
    current_cost = calculate_cost(board)
```

```
print("Initial board configuration:")
print_board(board, current_cost)
iteration = 0
restarts = 0

while restarts < max_restarts:  # Add limit to the number of restarts
    while current_cost != 0:  # Continue until cost is zero
        neighbors = get_neighbors(board)
        best_neighbor = None
        best_cost = current_cost

        for neighbor in neighbors:
            cost = calculate_cost(neighbor)
            if cost < best_cost:  # Looking for a lower cost
                best_cost = cost
                best_neighbor = neighbor

        if best_neighbor is None:  # No better neighbor found
            break  # Break the loop if we are stuck at a local minimum

        board = best_neighbor
        current_cost = best_cost
        iteration += 1
        print(f"Iteration {iteration}:")
        print_board(board, current_cost)
```

```
        if current_cost == 0:
            break  # We found the solution, no need for further restarts
        else:
            # Restart with a new random configuration
            board = [random.randint(0, len(board)-1) for _ in range(len(board))]
            current_cost = calculate_cost(board)
            restarts += 1
            print(f"Restart {restarts}:")
            print_board(board, current_cost)

    return board, current_cost

def print_board(board, cost):
    n = len(board)
    display_board = [['.'] * n for _ in range(n)]  # Create an empty board
    for col in range(n):
        display_board[board[col]][col] = 'Q'  # Place queens on the board
    for row in range(n):
        print(' '.join(display_board[row]))  # Print the board
    print(f"Cost: {cost}\n")

if __name__ == "__main__":
    n = int(input("Enter the number of queens (N): "))  # User input for N
    initial_state = list(map(int, input(f"Enter the initial state (row numbers for each column,
space-separated): ").split()))
```

```
    if len(initial_state) != n or any(r < 0 or r >= n for r in initial_state):
        print("Invalid initial state. Please ensure it has N elements with values from 0 to N-1.")
    else:
        solution, cost = hill_climb(initial_state)
        if cost == 0:
            print(f"Solution found with no conflicts:")
        else:
            print(f"No solution found within the restart limit:")
        print_board(solution, cost)
```

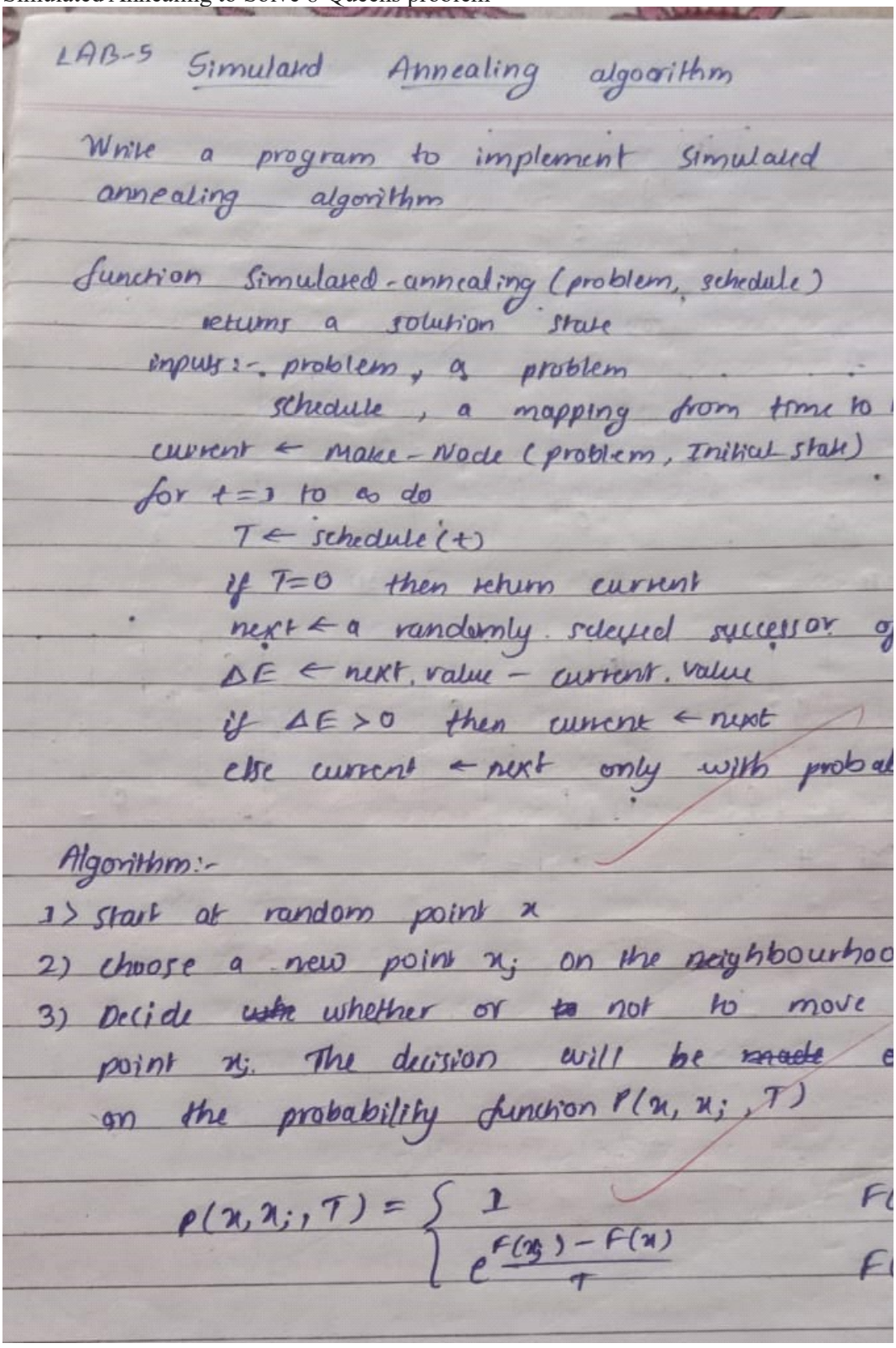
Output :



```
Q . . .
. Q . .
. . Q .
. . . Q
Cost: 6

Iteration 1:
. . . .
Q Q . .
. . Q .
. . . Q
Cost: 4

Iteration 2:
. Q . .
Q . . .
. . Q .
. . . Q
Cost: 2

Restart 1:
. Q Q Q
. . . .
. . . .
Q . . .
Cost: 4

Iteration 3:
. Q . Q
. . . .
. . Q .
Q . . .
Cost: 2

Iteration 4:
. Q . .
. . . Q
. . Q .

Iteration 6:
. . . .
. Q . .
. . Q Q
Q . . .
Cost: 2

Iteration 7:
. . Q .
. Q . .
. . . Q
Q . . .
Cost: 1

Restart 4:
Q . . .
. Q . Q
. . Q .
. . . .
Cost: 5

Iteration 8:
Q . . .
. Q . Q
. . . .
. . Q .
Cost: 2

Iteration 9:
Q Q . .
. . . Q
. . . .
. . Q .
Cost: 1

Iteration 10:
. Q . .
. . . Q
Q . . .
    Q
```

Program 5

Simulated Annealing to Solve 8-Queens problem



LAB-5 Simulated Annealing algorithm

Write a program to implement Simulated annealing algorithm

function Simulated-annealing (problem, schedule)
returns a solution state
inputs :- problem, a problem
schedule, a mapping from time to
current ← Make-Node (problem, Initial-state)
for t = 1 to ∞ do
T ← schedule (t)
if T=0 then return current
next ← a randomly selected successor of
ΔE ← next.value − current.value
if ΔE > 0 then current ← next
else current ← next only with probability

Algorithm:-

1) Start at random point x
2) choose a new point x_i on the neighbourhood
3) Decide ~~whether~~ whether or ~~to~~ not to move point x_i. The decision will be ~~made~~ based on the probability function $P(x, x_i, T)$

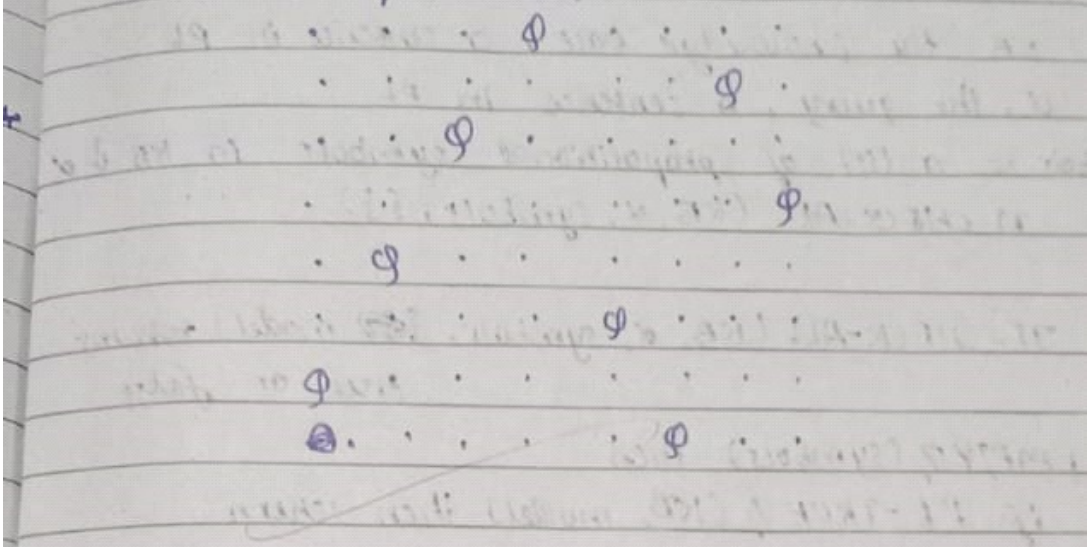
$$p(x, x_i, T) = \begin{cases} 1 & \\ e^{\frac{F(x_i) - F(x)}{T}} & \end{cases}$$

output:-

the best solution found is: [3 6 2

The number of queens that are not a
each other is: 8

Board representation



Tower of hanoi application output

Best state (final configuration): [2 2 2]

Number of correct disks on destination

Tower of hanoi

Peg 0: []

Peg 1: []

Peg 2: [0, 1, 2]

Lab
29/10/24

Code:

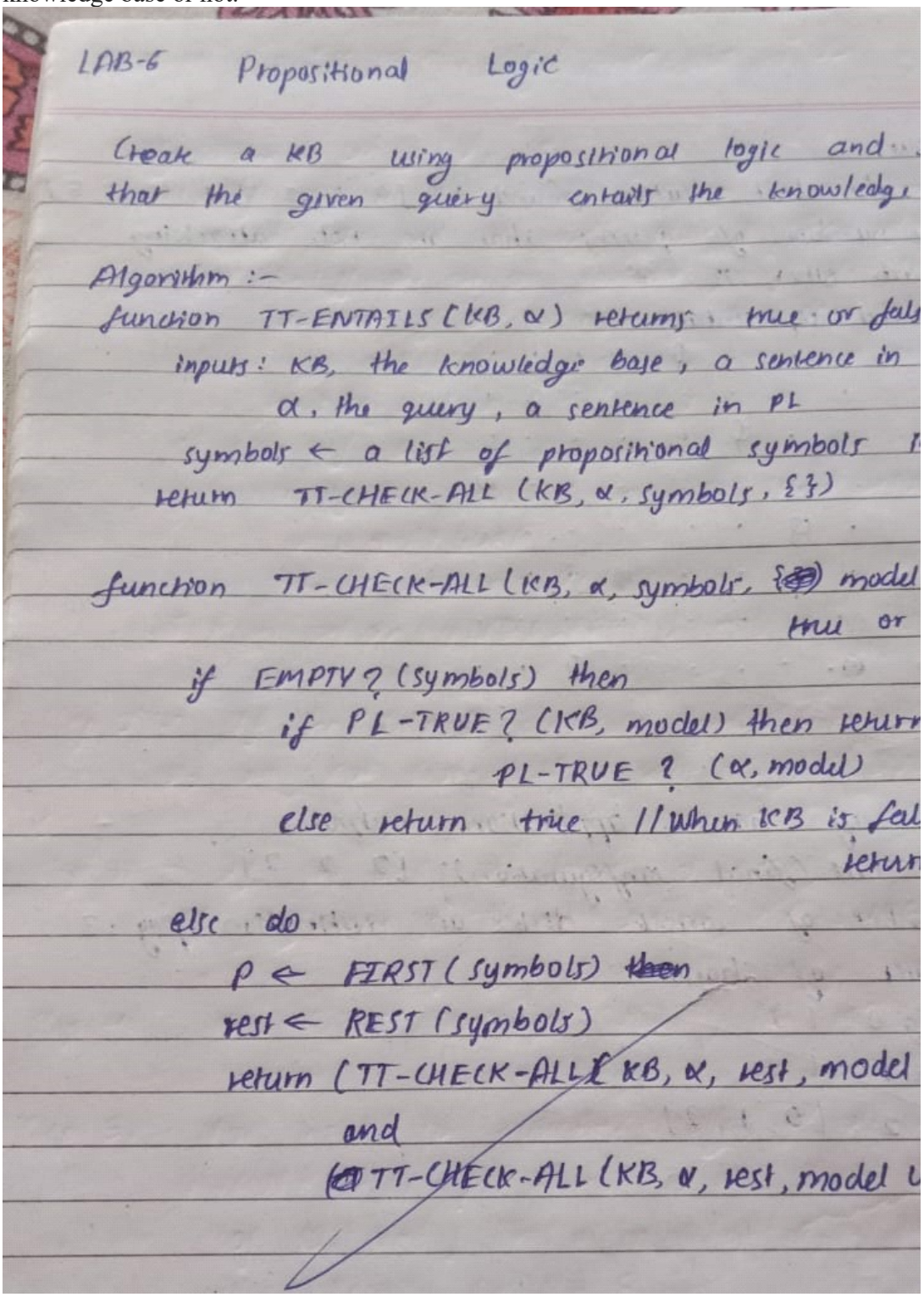
```
#!pip install mlrose-hiive joblib
#!pip install --upgrade joblib
#!pip install joblib==1.1.0
import mlrose_hiive as mlrose
import numpy as np

def queens_max(position):
    no_attack_on_j = 0
    queen_not_attacking = 0
    for i in range(len(position) - 1):
        no_attack_on_j = 0
        for j in range(i + 1, len(position)):
            if (position[j] != position[i]) and (position[j] != position[i] + (j - i)) and (position[j] !=
            position[i] - (j - i)):
                no_attack_on_j += 1
        if (no_attack_on_j == len(position) - 1 - i):
            queen_not_attacking += 1
    if (queen_not_attacking == 7):
        queen_not_attacking += 1
    return queen_not_attacking

objective = mlrose.CustomFitness(queens_max)

problem = mlrose.DiscreteOpt(length=8, fitness_fn=objective, maximize=True, max_val=8)
T = mlrose.ExpDecay()

initial_position = np.array([4, 6, 1, 5, 2, 0, 3, 7])

#The simulated_annealing function returns 3 values, we need to capture all 3
best_position, best_objective, fitness_curve = mlrose.simulated_annealing(problem=problem,
schedule=T, max_attempts=500,
                                                  init_state=initial_position)

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

Output :

```
The best position found is: [4 0 7 5 2 6 1 3]
The number of queens that are not attacking each other is: 8.0
```

Program 6

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



LAB-6 Propositional Logic

Creak a KB using propositional logic and ...
that the given query entails the knowledge

Algorithm :-

function TT-ENTAILS (KB, α) returns true or fals
inputs: KB, the knowledge base, a sentence in
α, the query, a sentence in PL
symbols ← a list of propositional symbols
return TT-CHECK-ALL (KB, α, symbols, {})

function TT-CHECK-ALL (KB, α, symbols, {}) model
true or

if EMPTY? (symbols) then
if PL-TRUE? (KB, model) then return
PL-TRUE? (α, model)
else return true // when KB is fal
return

else do:
P ← FIRST (symbols) then
rest ← REST (symbols)
return (TT-CHECK-ALL (KB, α, rest, model
and
TT-CHECK-ALL (KB, α, rest, model

Propositional Inference : Enumeration meth

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

checking that $KB \models \alpha$

A	B	C	A∨C	B∨C	KB.
F	F	F	F	T	F
F	F	T	T	F	F
F	T	F	F	T	F
F	T	T	T	T	T
T	F	F	T	T	T
T	F	T	T	F	F
T	T	F	T	T	T
T	T	T	T	T	T

O/P seen

SS
12/11/24

A∨B
F
F
T
T
T
T
T
T

Code:

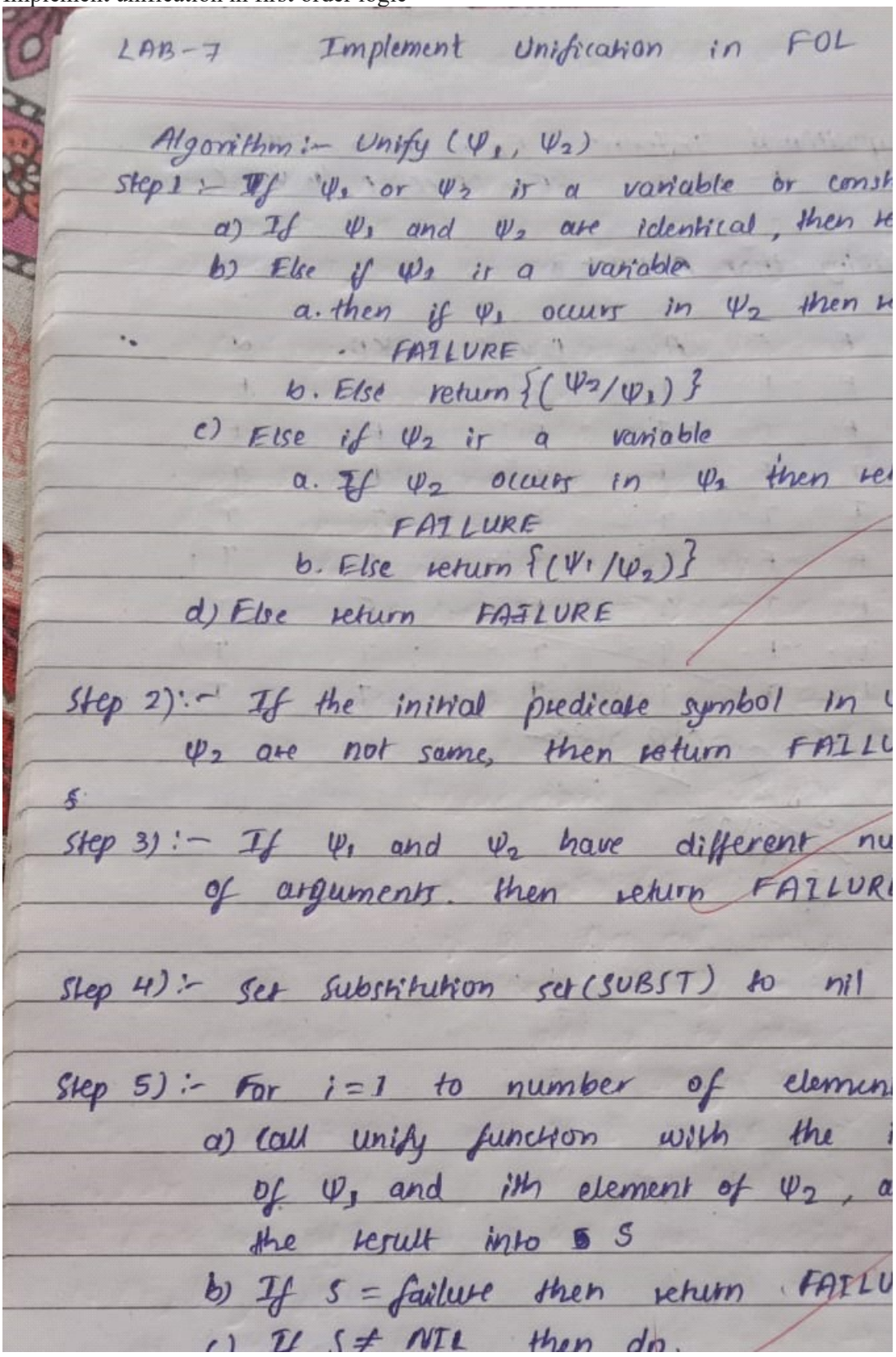
```
import pandas as pd

# Define the truth table for all combinations of A, B, C
truth_values = [(False, False, False),
                (False, False, True),
                (False, True, False),
                (False, True, True),
                (True, False, False),
                (True, False, True),
                (True, True, False),
                (True, True, True)]

# Columns: A, B, C
table = pd.DataFrame(truth_values, columns=["A", "B", "C"])

# Calculate intermediate columns
table["A or C"] = table["A"] | table["C"]       # A ∨ C
table["B or not C"] = table["B"] | ~table["C"]   # B ∨ ¬C

# Knowledge Base (KB): (A ∨ C) ∧ (B ∨ ¬C)
table["KB"] = table["A or C"] & table["B or not C"]

# Alpha (α): A ∨ B
table["Alpha (α)"] = table["A"] | table["B"]

# Define a highlighting function
def highlight_rows(row):
    if row["KB"] and row["Alpha (α)"]:
        return ['font-weight: bold; color: black'] * len(row)
    else:
        return [''] * len(row)

# Apply the highlighting function
styled_table = table.style.apply(highlight_rows, axis=1)

# Display the styled table
styled_table
```

Output :

	A	B	C	A or C	B or not C	KB
0	False	False	False	False	True	False
1	False	False	True	True	False	False
2	False	True	False	False	True	False
3	**False**	**True**	**True**	**True**	**True**	**True**
4	**True**	**False**	**False**	**True**	**True**	**True**
5	True	False	True	True	False	False

Program 7

Implement unification in first order logic



LAB-7 Implement Unification in FOL

Algorithm :- Unify (Ψ_1, Ψ_2)

Step 1 :- If Ψ_1 or Ψ_2 is a variable or consh

a) If Ψ_1 and Ψ_2 are identical, then re

b) Else if Ψ_1 is a variable

a. then if Ψ_1 occurs in Ψ_2 then re

FAILURE

b. Else return $\{(\Psi_2/\Psi_1)\}$

c) Else if Ψ_2 is a variable

a. If Ψ_2 occurs in Ψ_1 then re

FAILURE

b. Else return $\{(\Psi_1/\Psi_2)\}$

d) Else return FAILURE

Step 2) :- If the initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILU

Step 3) :- If Ψ_1 and Ψ_2 have different nu of arguments. then return FAILURE

Step 4) :- Set substitution set (SUBST) to nil

Step 5) :- For i = 1 to number of elemen

a) Call Unify function with the of Ψ_1 and ith element of Ψ_2, a the result into S

b) If S = failure then return FAILU

c) If S ≠ NIL then do.

Q) $\psi_1 = P(x, F(y))$ ——— ①

$\psi_2 = P(a, F(g(x)))$ ——— ②

① and ② are identical if x is rep in ① and y is replaced with $g(x)$

Substitution : $\{x \to a, y \to g(x)\}$

Q)

Q) $\psi_1 = f(g(x), h(y))$ $\psi_2 = f(g(a), h(b))$

Substitution ⇒ $\{x \to a, y \to b\}$

Q3) $\psi_1 = P(f(a), g(Y))$

$\psi_2 = P(X, X)$

We can't convert becau

SUBST ⇒ $\{f(a)$ cant be converted to

initial predicate symbol is not su

Q4) $\psi_1 = P(b, X, f(g(z)))$

$\psi_2 = P(z, f(y), f(y))$

~~b is replaced with~~ z

~~X is replaced with~~ $f(y)$

we can't convert because. In ~~parameter inside function~~ of ψ_1 symbol g but ψ_2 has no

Code:

```
import re

def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list):  # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False


def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst:  # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:  # Handle compound expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):  # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst


def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {}  # Initialize an empty substitution set

    # Step 1: Handle cases where x or y is a variable or constant
    if x == y:  # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():  # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"

        # Step 2: Check if the predicate symbols (the first element) match
        if x[0] != y[0]:  # If the predicates/functions are different
            return "FAILURE"

        # Step 5: Recursively unify each argument
```

```
        for xi, yi in zip(x[1:], y[1:]):  # Skip the predicate (first element)
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:  # If x and y are different constants or non-unifiable structures
        return "FAILURE"


def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
    (is_unified: bool, substitutions: dict or None)
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result


def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})


def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification
algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")

    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)\((.*)\)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term

    return parse_term(input_str)
```

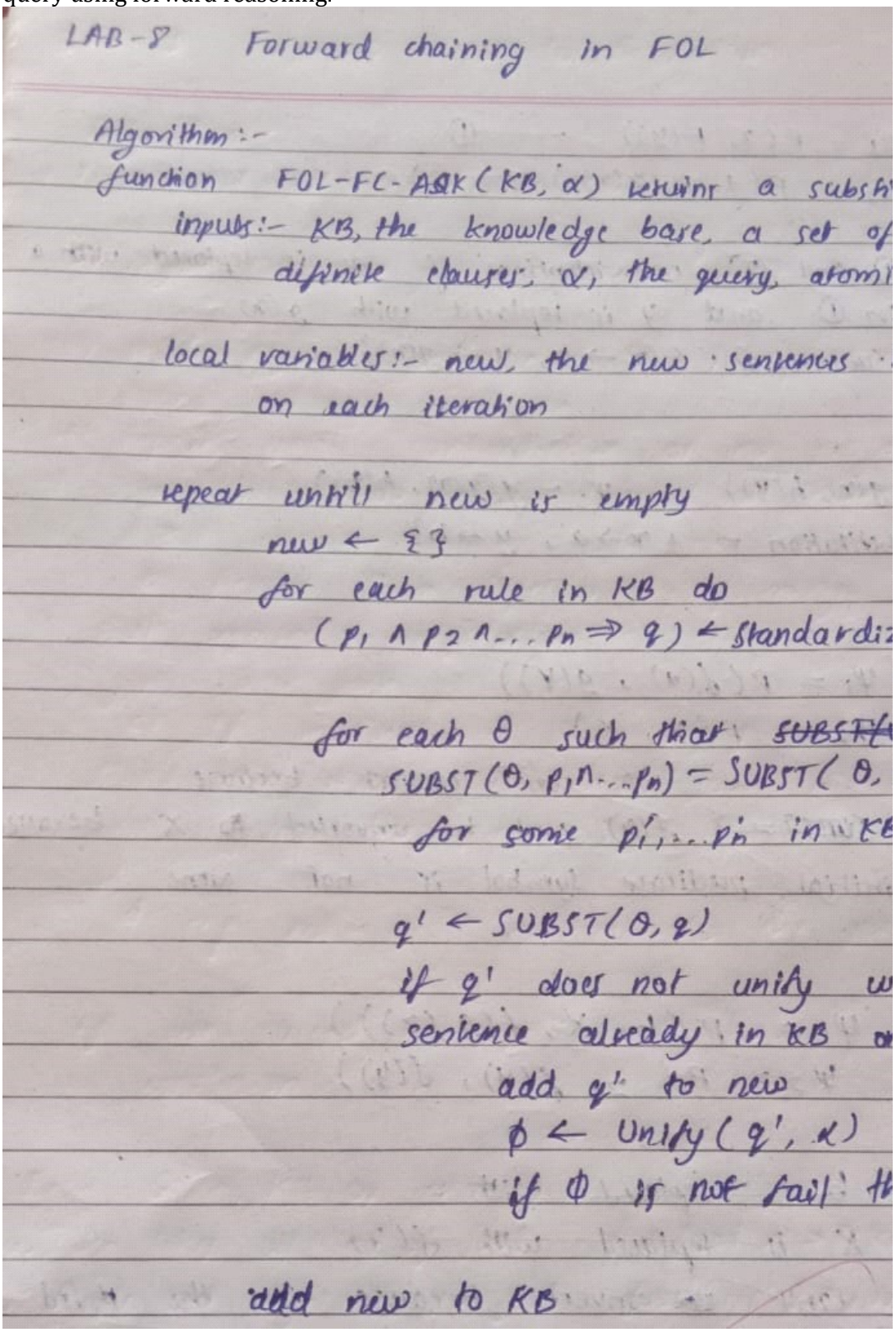
```
# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)

        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)

        # Display the results
        display_result(expr1, expr2, is_unified, result)

        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no):
").strip().lower()
        if another_test != 'yes':
            break


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

Output :

```
Enter the first expression (e.g., p(x, f(y))): p(b,
Enter the second expression (e.g., p(a, f(z))): p(z
Expression 1: ['p', 'b', 'x', ['f', ['g', 'z']]]
Expression 2: ['p', 'z', ['f', 'y'], ['f', 'y']]
Result: Unification Successful
Substitutions: {'b': 'z', 'x': ['f', 'y'], 'y': ['g
Do you want to test another pair of expressions? (y
Enter the first expression (e.g., p(x, f(y))): p(x,
Enter the second expression (e.g., p(a, f(z))): p(a
Expression 1: ['p', 'x', ['h', 'y']]
Expression 2: ['p', 'a', ['f', 'z']]
Result: Unification Failed
Do you want to test another pair of expressions? (y
Enter the first expression (e.g., p(x, f(y))): p(f(
Enter the second expression (e.g., p(a, f(z))): p(x
Expression 1: ['p', ['f', 'a'], ['g', 'y']]
```

Program 8

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



LAB-8 Forward chaining in FOL

Algorithm :-

function FOL-FC-ASK (KB, α) returns a subst

inputs :- KB, the knowledge base, a set of
definite clauses; α, the query, atomi

local variables :- new, the new sentences
on each iteration

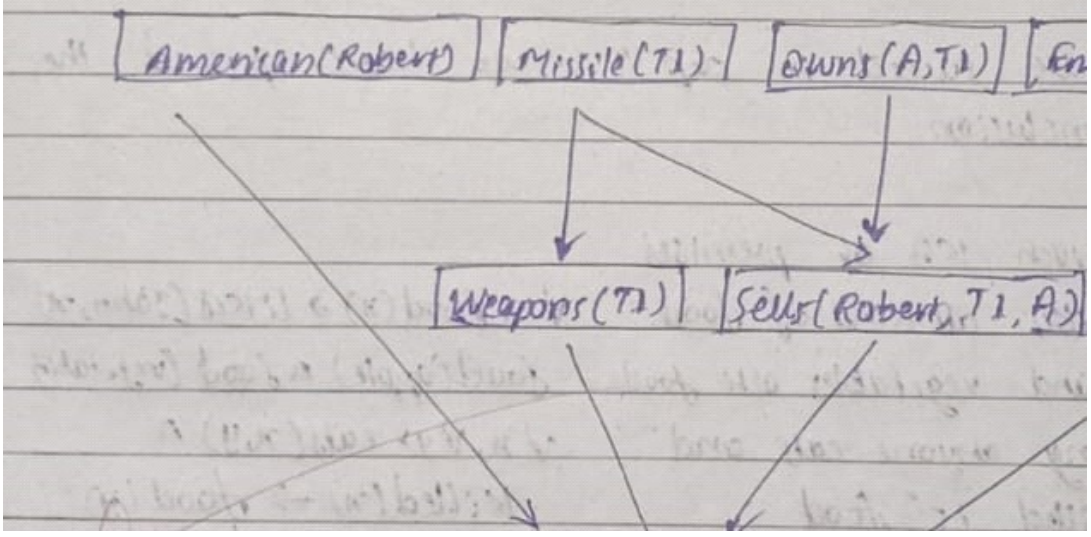
repeat until new is empty
new ← { }
for each rule in KB do
(p1 ∧ p2 ∧ ... pn ⇒ q) ← Standardi

for each θ such that ~~SUBST(~~
SUBST(θ, p1 ∧ ... pn) = SUBST(θ,
for some p1', ... pn' in KB

q' ← SUBST(θ, q)
if q' does not unify w
sentence already in KB o
add q' to new
φ ← Unify(q', α)
if φ is not fail th

add new to KB

Ex:- It is a crime for an american to se
weapons to hostile nations
let's say p, q and r are variables
American(p) ∧ weapon(q) ∧ sells(p, q, r) ∧
⇒ Criminal(p)
Contry A has some missiles
∃ x Owns(A, x) ∧ Missile(x)
Existential Instantiation, introducing a
Owns(A, T1)
Missile(T1)
All of the missiles were sold to country
Robert
∀ x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robe
Missiles are ~~was~~ weapons
Missile(x) ⇒ Weapon(x)
Enemy of america is known as hosti
∀x Enemy(x, America) ⇒ Hostile(x)
To prove Robert is criminal Crimi



Code:

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()   # Set of known facts
        self.rules = []      # List of rules

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

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

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))
```

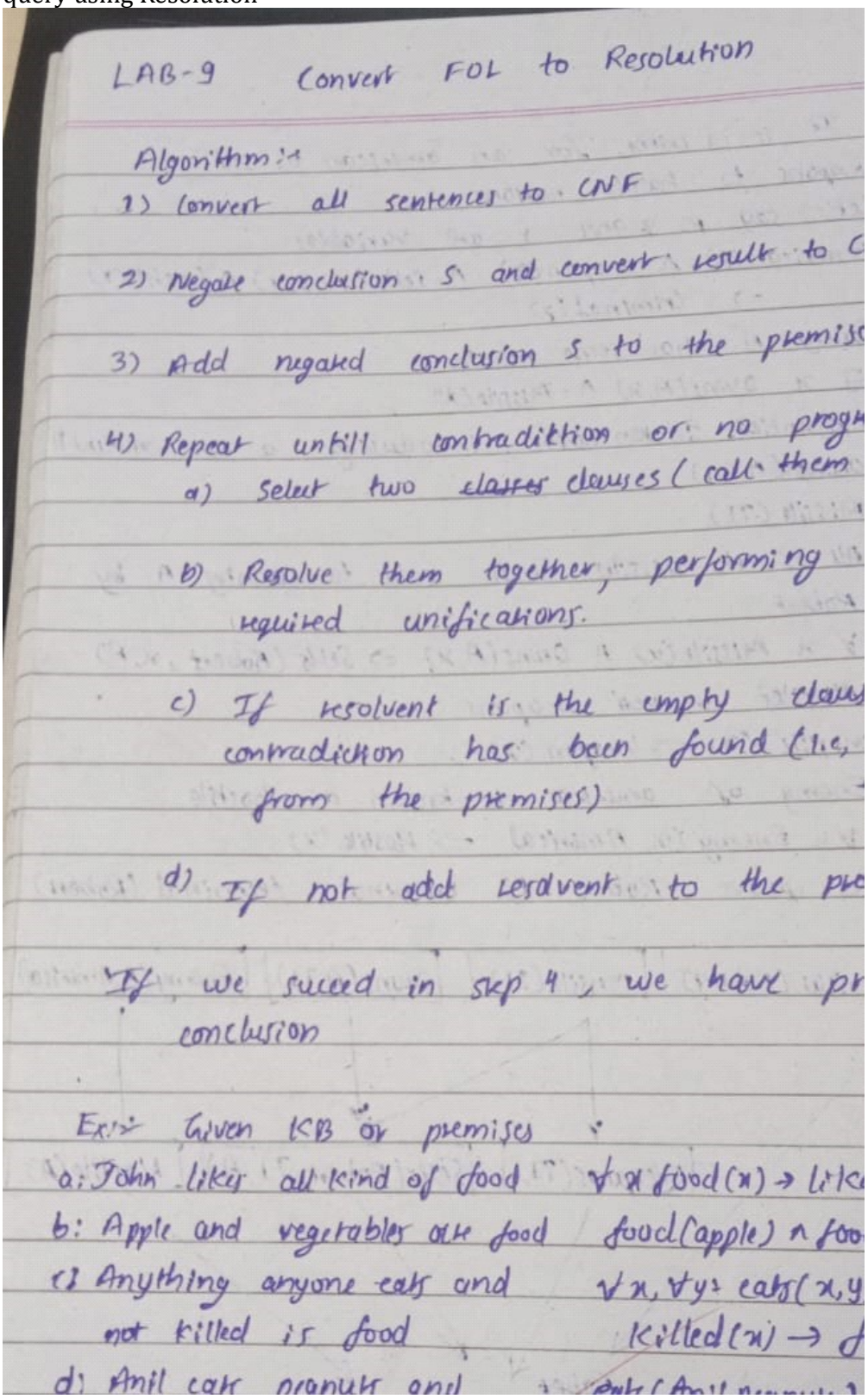
```
# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)",
"Hostile(A)"], "Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

Output :

```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
```

Program 9

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



LAB-9 Convert FOL to Resolution

Algorithm:-

1) Convert all sentences to CNF

2) Negate conclusion S and convert result to C

3) Add negated conclusion S to the premise

4) Repeat untill contradiction or no progr
   a) Select two ~~clause~~ clauses (call them

   b) Resolve them together, performing required unifications.

   c) If resolvent is the empty clau contradiction has been found (i.e, from the premises)

   d) If not add resolvent to the pre

If we succeed in step 4, we have pr conclusion

Ex:- Given KB or premises :

a: John likes all kind of food ∀x food(x) → like

b: Apple and vegetables are food food(apple) ∧ foo

c) Anything anyone eats and not killed is food ∀x, ∀y: eats(x,y Killed(x) → f

d) Anil eats peanuts and

1) Elimination Implication

a. $\forall x \neg food(x) \vee likes(John, x)$

b. food(Apple) $\wedge$ food(vegetables)

c. $\forall x \forall y \neg [eats(x,y) \wedge \neg killed(x)] \vee food(y)$

d. eats(Anil, peanuts) $\wedge$ alive(Anil)

e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$

f. $\forall x \neg [\neg killed(x)] \vee alive(x)$

g. $\forall x \neg alive(x) \vee \neg killed(x)$

h. likes(John, peanuts)

2) Move nega... inward

$\forall x \forall y \neg e$ ... killed(

$\forall x [killed($

3) Rename variables or standardize var...

a. $\forall x \neg food(x) \vee likes(John, x)$

b. food(Apple) $\wedge$ food(vegetables)

c. $\forall y \forall z \neg eats(y,z) \vee killed(y) \vee food(z)$

d. eats(Anil, peanuts) $\wedge$ alive(Anil)

e. $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$

f. $\forall g [killed(g)] \vee alive(g)$

g. $\forall k \neg alive(k) \vee \neg killed(k)$

h. likes(John, Peanuts)

4) Drop un...

$\neg food(x) \vee likes($

$\neg eats(y,z) \vee kil$

$\neg eats(Anil, w) \vee$

killed(g) $\vee$ a...

$\neg alive(k)$ ...

~~likes~~

$\neg likes(John, Peanuts)$ — $\neg food(x) \vee likes[Jo$ ... $\{Peanu$

$\neg food(Peanuts)$ — $\neg eats(y,z) \vee killed(y$ ... $\{Peanuts$

$\neg eats(y, Peanuts) \vee killed(y)$ — $eats(Anil,$ ... $\{Anil$

$killed(Anil)$ — $\neg alive(k) \vee$ ...

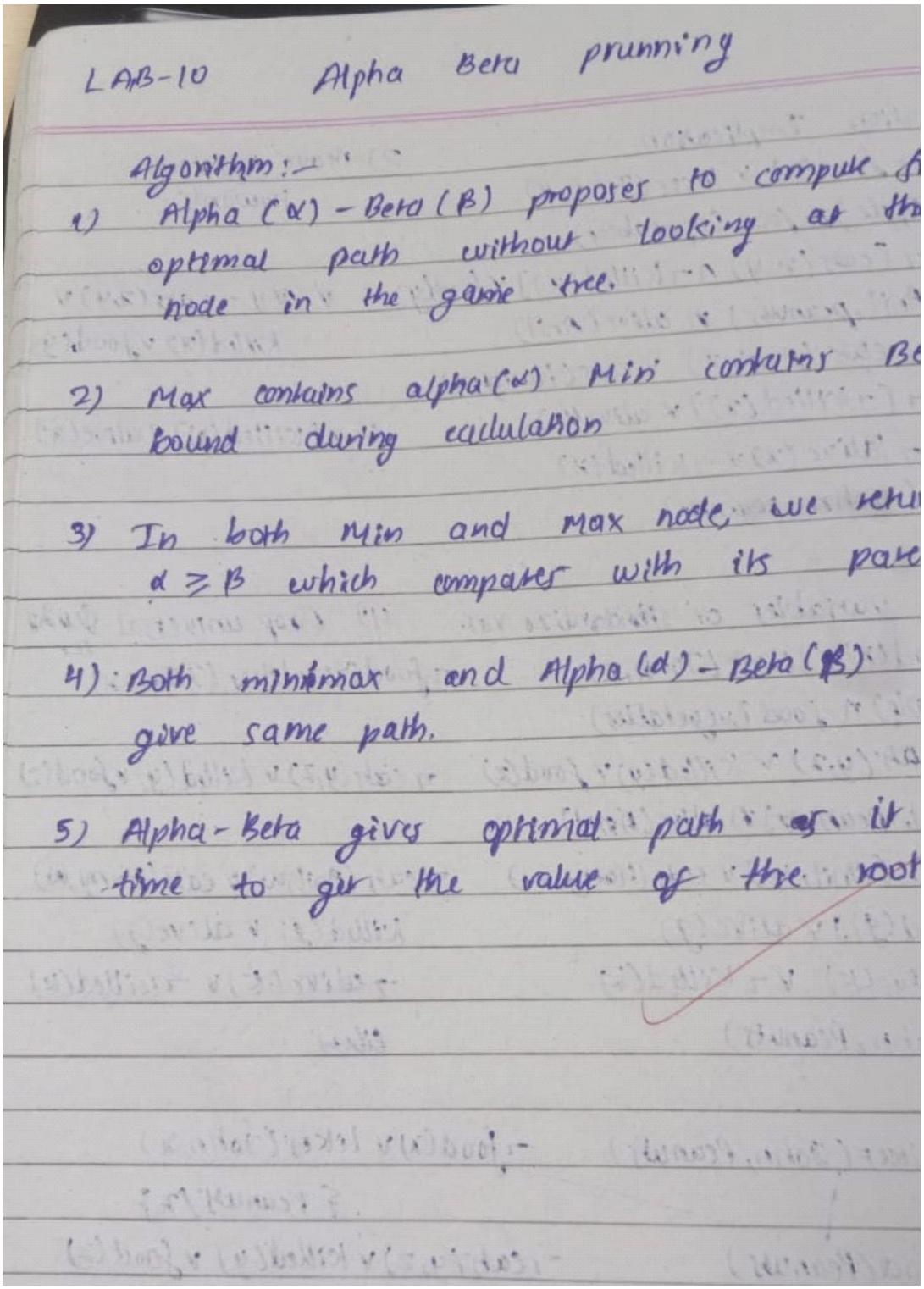
Code:

```
from sympy import symbols, And, Or, Not, Implies, to_cnf

# Define constants (entities in the problem)
John, Anil, Harry, Apple, Vegetables, Peanuts, x, y = symbols('John Anil Harry Apple
Vegetables Peanuts x y')

# Define predicates as symbols (this works as a workaround)
Food = symbols('Food')
Eats = symbols('Eats')
Likes = symbols('Likes')
Alive = symbols('Alive')
Killed = symbols('Killed')

# Knowledge Base (Premises) in First-Order Logic
premises = [
    # 1. John likes all kinds of food: Food(x) → Likes(John, x)
    Implies(Food, Likes),

    # 2. Apples and vegetables are food: Food(Apple) ∧ Food(Vegetables)
    And(Food, Food),

    # 3. Anything anyone eats and is not killed is food: (Eats(y, x) ∧ ¬Killed(y)) → Food(x)
    Implies(And(Eats, Not(Killed)), Food),

    # 4. Anil eats peanuts and is still alive: Eats(Anil, Peanuts) ∧ Alive(Anil)
    And(Eats, Alive),

    # 5. Harry eats everything that Anil eats: Eats(Anil, x) → Eats(Harry, x)
    Implies(Eats, Eats),

    # 6. Anyone who is alive implies not killed: Alive(x) → ¬Killed(x)
    Implies(Alive, Not(Killed)),

    # 7. Anyone who is not killed implies alive: ¬Killed(x) → Alive(x)
    Implies(Not(Killed), Alive),
]

# Negated conclusion to prove: ¬Likes(John, Peanuts)
negated_conclusion = Not(Likes)

# Convert all premises and the negated conclusion to Conjunctive Normal Form (CNF)
cnf_clauses = [to_cnf(premise, simplify=True) for premise in premises]
cnf_clauses.append(to_cnf(negated_conclusion, simplify=True))

# Function to resolve two clauses
def resolve(clause1, clause2):
    """
    Resolve two CNF clauses to produce resolvents.
```

```
    """
    clause1_literals = clause1.args if isinstance(clause1, Or) else [clause1]
    clause2_literals = clause2.args if isinstance(clause2, Or) else [clause2]
    resolvents = []

    for literal in clause1_literals:
        if Not(literal) in clause2_literals:
            # Remove the literal and its negation and combine the rest
            new_clause = Or(
                *[l for l in clause1_literals if l != literal],
                *[l for l in clause2_literals if l != Not(literal)]
            ).simplify()
            resolvents.append(new_clause)

    return resolvents

# Function to perform resolution on the set of CNF clauses
def resolution(cnf_clauses):
    """
    Perform resolution on CNF clauses to check for a contradiction.
    """
    clauses = set(cnf_clauses)
    new_clauses = set()

    while True:
        clause_list = list(clauses)
        for i in range(len(clause_list)):
            for j in range(i + 1, len(clause_list)):
                resolvents = resolve(clause_list[i], clause_list[j])
                if False in resolvents:  # Empty clause found
                    return True  # Contradiction found; proof succeeded
                new_clauses.update(resolvents)

        if new_clauses.issubset(clauses):  # No new information
            return False  # No contradiction; proof failed

        clauses.update(new_clauses)

# Perform resolution to check if the conclusion follows
result = resolution(cnf_clauses)
print("Does John like peanuts? ", "Yes, proven by resolution." if result else "No, cannot be
proven.")
```

Program 10

Implement Alpha-Beta Pruning.



LAB-10 Alpha Beta prunning

Algorithm:-

1) Alpha (α) - Beta (β) proposes to compute
optimal path without looking at
node in the game tree.

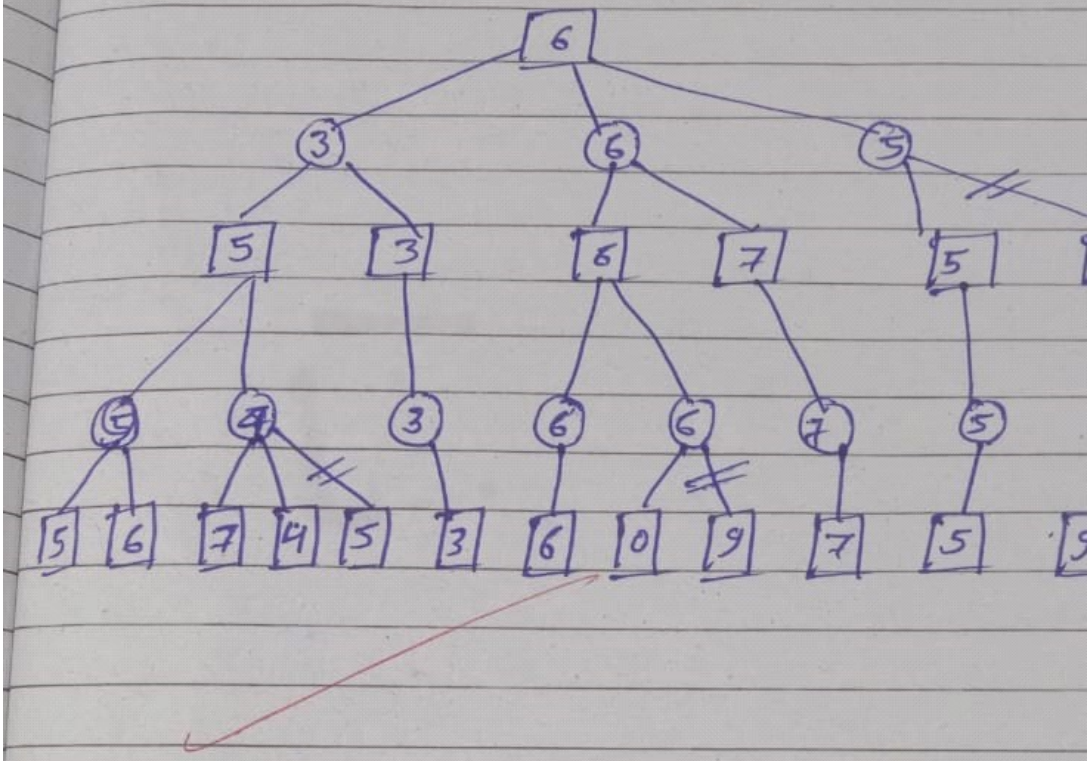
2) Max contains alpha(α) Min contains Be
bound during calculation

3) In both Min and Max node, we retu
α ≥ β which compares with its pare

4) Both minimax and Alpha (α) - Beta (β)
give same path.

5) Alpha - Beta gives optimal path in
time to get the value of the root

Solve using min-max algorithm



7-12-24

Code:

```
import math
def minimax(node, depth, is_maximizing):
  """
  Implement the Minimax algorithm to solve the decision tree.

  Parameters:
  node (dict): The current node in the decision tree, with the following structure:
    {
      'value': int,
      'left': dict or None,
      'right': dict or None
    }
  depth (int): The current depth in the decision tree.
  is_maximizing (bool): Flag to indicate whether the current player is the maximizing
player.

  Returns:
  int: The utility value of the current node.
  """
  # Base case: Leaf node
  if node['left'] is None and node['right'] is None:
    return node['value']

  # Recursive case
  if is_maximizing:
    best_value = -math.inf
    if node['left']:
      best_value = max(best_value, minimax(node['left'], depth + 1, False))
    if node['right']:
      best_value = max(best_value, minimax(node['right'], depth + 1, False))
    return best_value
  else:
    best_value = math.inf
    if node['left']:
      best_value = min(best_value, minimax(node['left'], depth + 1, True))
    if node['right']:
      best_value = min(best_value, minimax(node['right'], depth + 1, True))
    return best_value

# Example usage
decision_tree = {
  'value': 5,
  'left': {
    'value': 6,
    'left': {
      'value': 7,
      'left': {
        'value': 4,
```

```
            'left': None,
            'right': None
        },
        'right': {
            'value': 5,
            'left': None,
            'right': None
        }
    },
    'right': {
        'value': 3,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': {
            'value': 9,
            'left': None,
            'right': None
        }
    }
},
'right': {
    'value': 8,
    'left': {
        'value': 7,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': {
            'value': 9,
            'left': None,
            'right': None
        }
    },
    'right': {
        'value': 8,
        'left': {
            'value': 6,
            'left': None,
            'right': None
        },
        'right': None
    }
}
}
```

```
# Find the best move for the maximizing player
best_value = minimax(decision_tree, 0, True)
print(f"The best value for the maximizing player is: {best_value}")
```