VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Pooja Gaikwad(1BM22CS194)

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

Dr.Umadevi
Professor
Department of 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 outby Pooja Gaikwad **(1BM22CS194),** 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.

Sneha S Bagalkot Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Githublink: https://github.com/poojagaikwad10/Artificial-intelligence

LAB 01:Tic Tac Toe Problem

DATE: 24/07/24 PAGE:

* Tic tac toe problem:-

* Algorithm :-

Step 1: create a board game
→ create 3x3 grid which has 9 cells
→ each can be empty or x or o

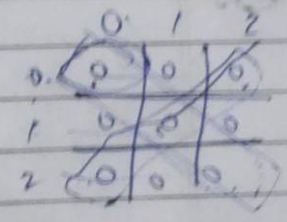
Step 2: Turn loop for the players
→ Alternate turns b/w x and o players

Step 3: Give input (place x or o respective in columns or rows where they want to place)
→ update the board after each players move.
→ After each player move, check if there is any 3 consecutive x or o by this decide winner or draw. (vertical, diagonal, horizontal)
→ If it is winner, end the game or else it is draw when the there are no 3 consecutive x or o

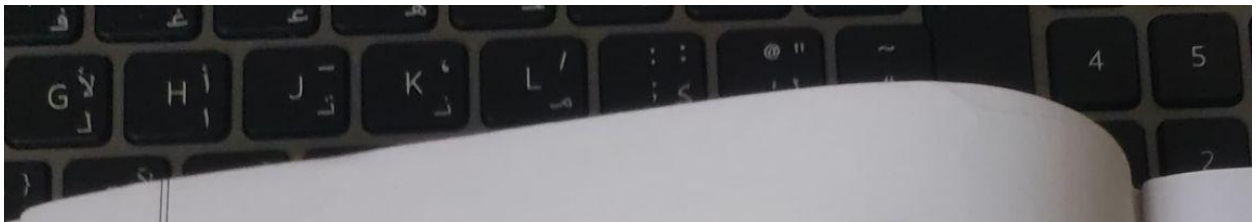
Step 4: End the Game.

* Functions to print the board.
* Function to check for a win condition
* Function to get the player move
* Function to check the board.
* Function to run the game.
* ##

[0][i]



[i][0]



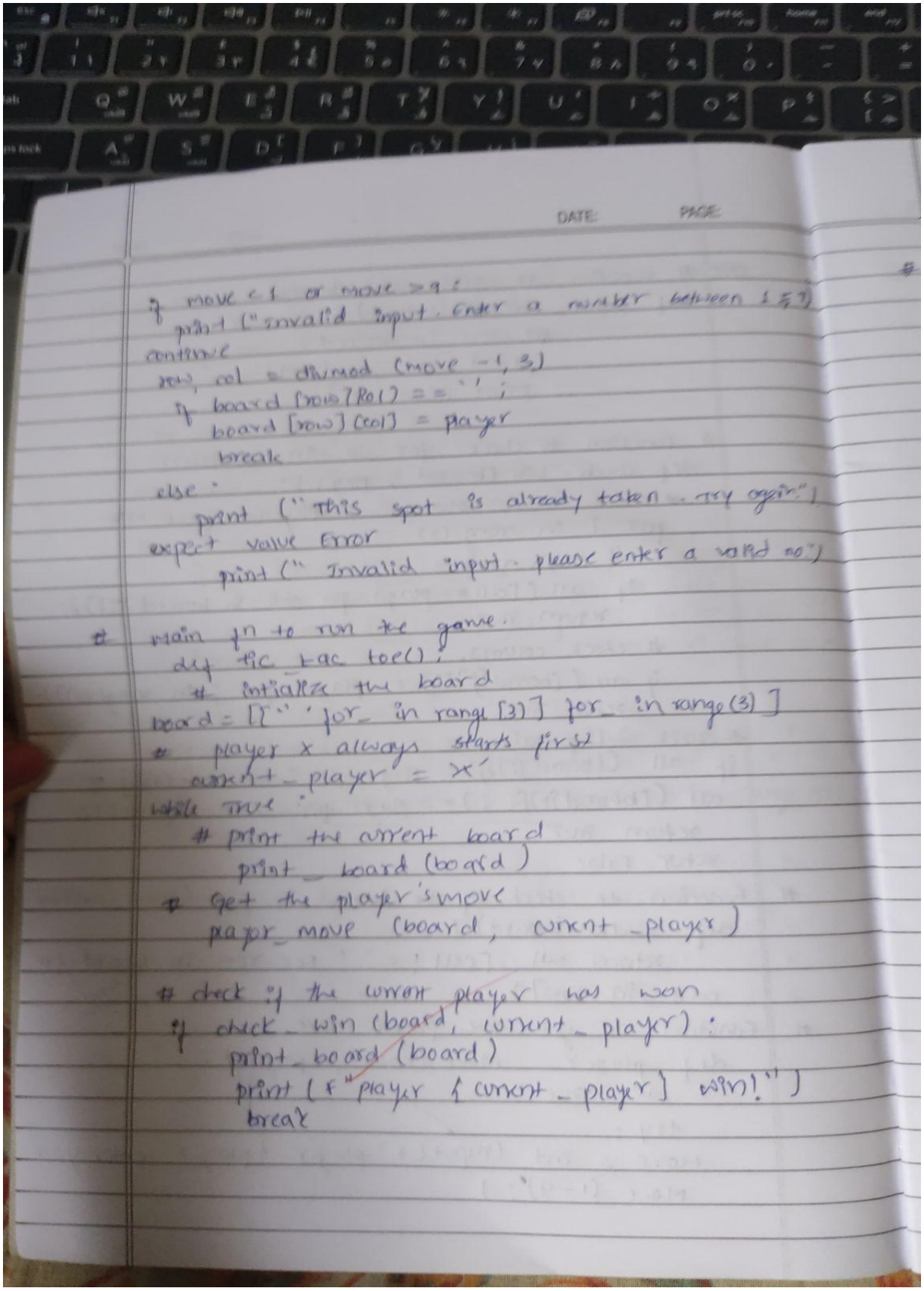
DATE: PAGE:

```
# def check_win(board, player)
    for row in board
        if row == [player, player, player]:
            return true

for col in range(3) ] x

def check_win(board, player)
    # check rows & columns.
for i in range(3):
    if all(board[i][j] = player for j in rang(3)):
        return true                    # check row
    if all(board[j][i) = player for i in range(3)):
        return True                    # check col

# Check diagonals
    if all [board[i][i] == player for i in range(3)):
        return True
    if all [board[i][2-i] == player for i in rang(3)):
        retur True

return flase  # No win found or draw.
```

DATE: PAGE:

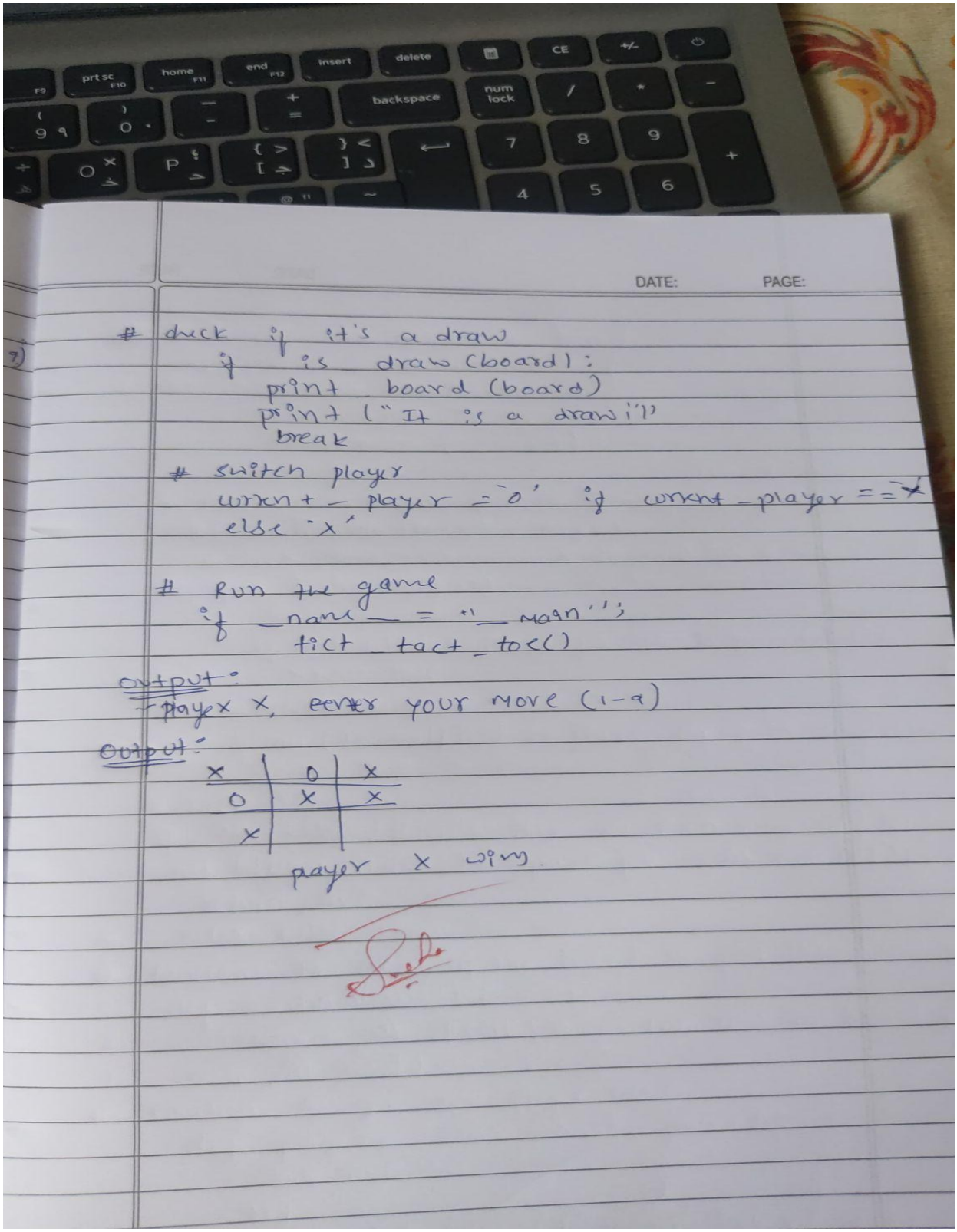
code:

```
# Function to print the board
def print_board(board):
    for row in board:
        print("|".join(row))
        print("-" * 9)

# Function to check for a win condition
def check_win(board, player):
    # check rows, columns, & diagonals for a win
    for i in range(3):
        # check rows
        if all([cell == player for cell in board[i]]):
            return True
        # check columns
        if all([board[j][i] == player for j in range(3)]):
            retur True
    #check diagonals
    if all[[board[i][i] = player for i in rang(3)]] or
    all([board[i][2-i]) == player for i in range(3)]):
        return True
    retur False
# function to check if the board is full (draw condition)
def is_draw(board):
    return all[cell != ' ' For row in board for
    cell in row]).
# Function to get the player move
def player_move(board, player):
    while True:
    try:
    move = int(input(f"player {player}, enter your
    move (1-9)":))
```



DATE: PAGE:

```
if move < 1 or move > 9:
    print ("Invalid input. Enter a number between 1 & 9")
continue
row, col = divmod (move - 1, 3)
if board [row][col] == ' ':
    board [row] [col] = player
    break
else:
    print ("This spot is already taken. Try again.")
except value Error
    print ("Invalid input. please enter a valid no")

# main fn to run the game.
def tic_tac_toe():
    # initialize the board
board = [[' ' for _ in range(3)] for _ in range(3)]
# player x always starts first
current_player = 'X'
while True:
    # print the current board
    print_board (board)
# Get the player's move
    player_move (board, current_player)

# check if the current player has won
if check_win (board, current_player):
    print_board (board)
    print (f"player {current_player} win!")
    break
```



DATE: PAGE:

7)

```
# check if it's a draw
    if is_draw(board):
        print_board(board)
        print("It is a draw!")
        break

# switch player
    current_player = 'O' if current_player == 'X'
    else 'X'

# Run the game
    if __name__ == "__main__":
        tict_tact_toe()
```

Output:

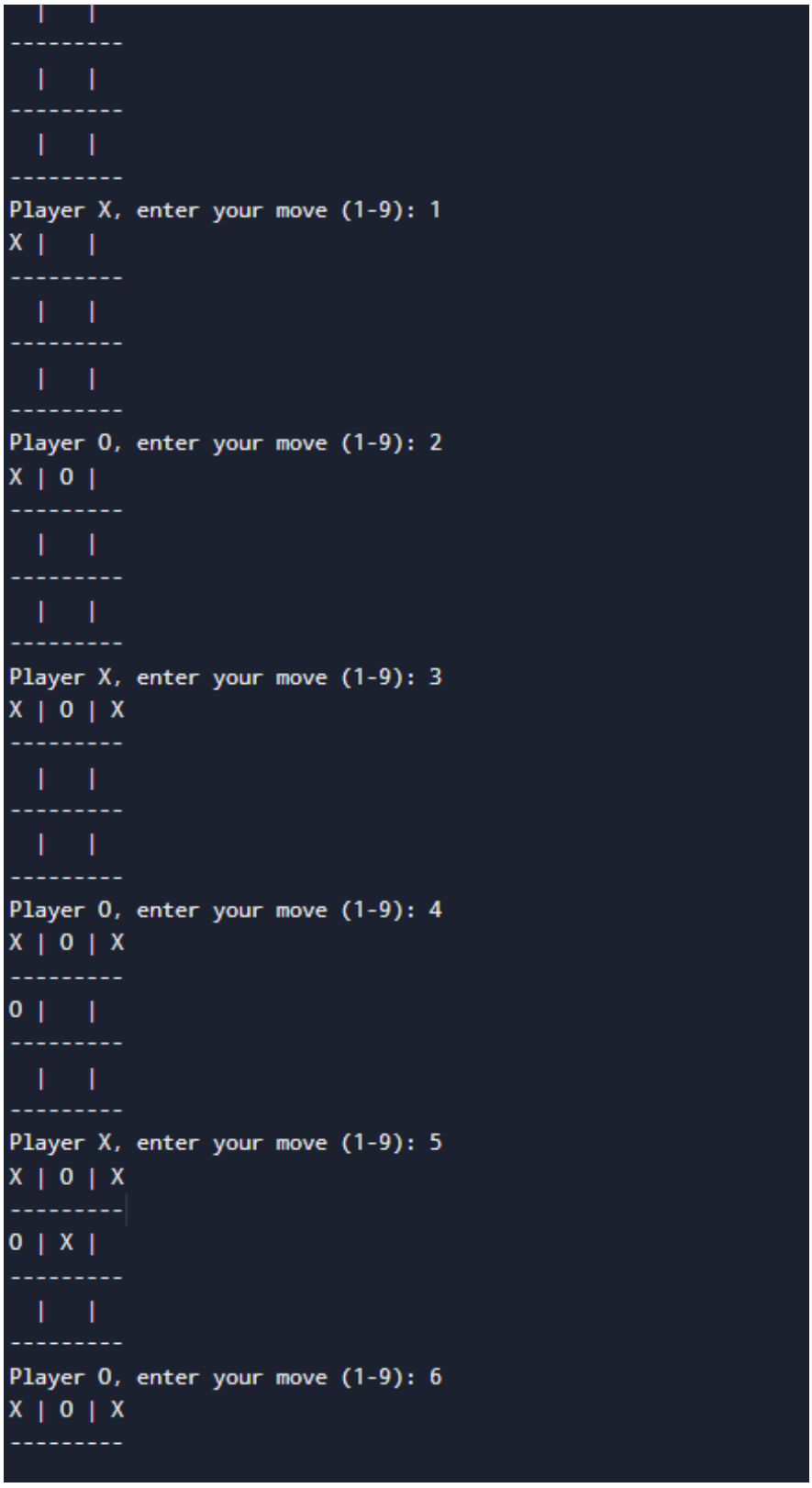
player X, enter your move (1-9)

Output:

X	O	X
O	X	X
X		

player X wins.

Output:


| |

| |

| |

Player X, enter your move (1-9): 1
X | |

| |

| |

Player O, enter your move (1-9): 2
X | O |

| |

| |

Player X, enter your move (1-9): 3
X | O | X

| |

| |

Player O, enter your move (1-9): 4
X | O | X

O | |

| |

Player X, enter your move (1-9): 5
X | O | X

O | X |

| |

Player O, enter your move (1-9): 6
X | O | X

```
Player X, enter your move (1-9): 3
X | O | X
---------
  |   |
---------
  |   |
---------
Player O, enter your move (1-9): 4
X | O | X
---------
O |   |
---------
  |   |
---------
Player X, enter your move (1-9): 5
X | O | X
---------
O | X |
---------
  |   |
---------
Player O, enter your move (1-9): 6
X | O | X
---------
O | X | O
---------
  |   |
---------
Player X, enter your move (1-9): 7
X | O | X
---------
O | X | O
---------
X |   |
---------
Player X wins!

=== Code Execution Successful ===-
```

DATE: 01/10/24 PAGE:

* Lab: 02:

vaccum cleaner :—

* Algorithm:

Step 1: there are two rooms which are room A and room B

* Intalize : start with room A
  * check room A
    → * If ROOM A is dirty clean it
    → * If ROOM A is clean move to next step or room
* Note : each rooms can either be clean or dirty.
* check room B
  → If ROOM B is dirty clean it.
  → If ROOM B is clean move to next ROOM A.
* Repeat : continue until both the rooms are clean.
* Stop: when both rooms are clean.
* Result : display that both the rooms are clean.

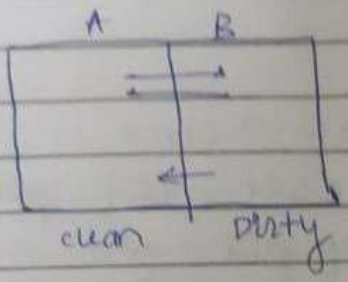
* percept sequence :-

1) ('ROOM A' 'Dirty')
('ROOM B', 'Dirty')
('ROOM A', 'clean')
('ROOM B', 'clean')

('ROOM A' 'clean')
('ROOM B' 'clean')

DATE: PAGE: 07

Rooms



① (A, left)

① (A, left) ② (B, clean)

① (A, left) ② (B, clean) ③ (A, right)

* code:

```
class vacuum cleaner :
    def __init__(self, grid) :
        self.grid = grid
        self.position = (0,0)
    def clean (self) :
        x, y = self.position
        if self.grid [x][y] == 1:
            print (f"cleaning position {self.position}")
            self.grid [x][y] = 0
        else :
            print (f" position {self.position} is already
                    clean")
    def move (self, direction) :
        x, y = self.position
        if direction == 'up' and x > 0 :
            self.position = (x, -1, y)
        elif direction == 'down' and x < len (self.grid)-1:
            self.position = (x+1, y)
        elif direction == 'left' and y > 0 :
            self.position = (x, y-1)
        elif direction == 'right' and y < len (self.grid[0]
                                                -1 :
            self.position = (x, y+1)
        else :
            print ("move not possible")
    def run (self) :
        rows = len (self.grid)
        cols = len (self.grid [0])
    for i in range (rows) :
        for j in range (cols) :
```

```
        self.position = (i, j)
        self.clean()
    print("Final grid state:")
    for row in self.grid:
        print(row)

def get_dirty_coordinates(rows, cols, num_dirty_cells):
    dirty_cells = set()
    while len(dirty_cells) < num_dirty_cells:
        try:
            coords = input(f"Enter coordinates for dirty
                cell {len(dirty_cells)+1}
            (format: row, col): ")
            x, y = map(int, coords.split(','))
            if 0 <= x < rows and 0 <= y < cols:
                dirty_cells.add((x, y))
            else:
                print("Co-ordinates are out of bounds. Try again.")
        except ValueError:
            print("Invalid input. Pls enter co-ordinates in
            the format: row, col")
    return dirty_cells

rows = int(input("Enter the no. of rows"))
cols = int(input("Enter the no. of columns"))
num_dirty_cells = int(input("Enter the number
                    of dirty cells:"))
```

DATE: PAGE:

```
intial_grid = [[0 for _ in range(cols)] for _ in
    range(rows)]
dirty_coordinates = get_dirty_coordinates
    (rows, cols, num_dirty_cells)

vaccum.run().
```

→ output:

```
Enter the no of rows : 2
Enter the no of cols : 2
Enter the no of dirty cells : 1
intial grid state:
    [0, 1]
    [0, 0]
position (0, 0) is already clean
cleaning position (0, 1)
position (1, 0) is already clean
position (1, 1) is already clean
Final grid state
    [0, 0]
    [0, 0]
```

Sudeep 7/10/24

Lab 02:-

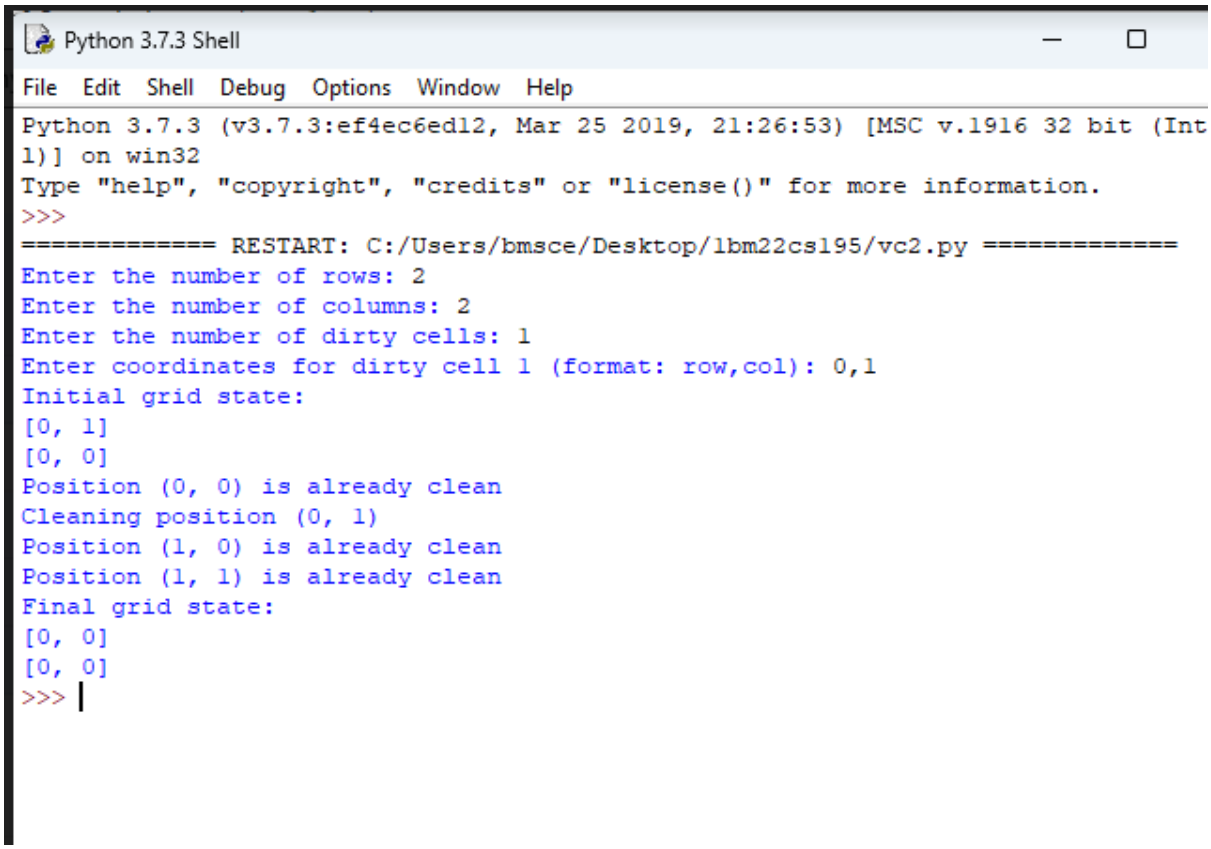
Vacuum Cleaner

Code:-

```
class VacuumCleaner:
    def __init__(self, grid):
        self.grid = grid
        self.position = (0, 0)

    def clean(self):
        x, y = self.position
        if self.grid[x][y] == 1:
            print(f"Cleaning position {self.position}")
            self.grid[x][y] = 0
        else:
            print(f"Position {self.position} is already clean")

    def move(self, direction):
        x, y = self.position
        if direction == 'up' and x > 0:
            self.position = (x - 1, y)
        elif direction == 'down' and x < len(self.grid) - 1:
            self.position = (x + 1, y)
        elif direction == 'left' and y > 0:
            self.position = (x, y - 1)
        elif direction == 'right' and y < len(self.grid[0]) - 1:
            self.position = (x, y + 1)
        else:
            print("Move not possible")

    def run(self):
        rows = len(self.grid)
        cols = len(self.grid[0])


        for i in range(rows):
            for j in range(cols):
                self.position = (i, j)
                self.clean()

        print("Final grid state:")
        for row in self.grid:
            print(row)

def get_dirty_coordinates(rows, cols, num_dirty_cells):
    dirty_cells = set()
    while len(dirty_cells) < num_dirty_cells:
```

```
        try:
            coords = input(f"Enter coordinates for dirty cell {len(dirty_cells) + 1} (format: row,col): ")
            x, y = map(int, coords.split(','))
            if 0 <= x < rows and 0 <= y < cols:
                dirty_cells.add((x, y))
            else:
                print("Coordinates are out of bounds. Try again.")
        except ValueError:
            print("Invalid input. Please enter coordinates in the format: row,col")
    return dirty_cells

rows = int(input("Enter the number of rows: "))
cols = int(input("Enter the number of columns: "))
num_dirty_cells = int(input("Enter the number of dirty cells: "))

if num_dirty_cells > rows * cols:
    print("Number of dirty cells exceeds total cells in the grid. Adjusting to maximum.")
    num_dirty_cells = rows * cols

initial_grid = [[0 for _ in range(cols)] for _ in range(rows)]
dirty_coordinates = get_dirty_coordinates(rows, cols, num_dirty_cells)

for x, y in dirty_coordinates:
    initial_grid[x][y] = 1

vacuum = VacuumCleaner(initial_grid)
print("Initial grid state:")
for row in initial_grid:
    print(row)

vacuum.run()
```

Output:-


Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Int
l)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:/Users/bmsce/Desktop/1bm22cs195/vc2.py ==========
Enter the number of rows: 2
Enter the number of columns: 2
Enter the number of dirty cells: 1
Enter coordinates for dirty cell 1 (format: row,col): 0,1
Initial grid state:
[0, 1]
[0, 0]
Position (0, 0) is already clean
Cleaning position (0, 1)
Position (1, 0) is already clean
Position (1, 1) is already clean
Final grid state:
[0, 0]
[0, 0]
>>>

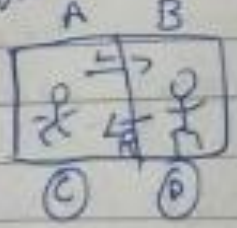
LAB-2

Vacuum cleaner

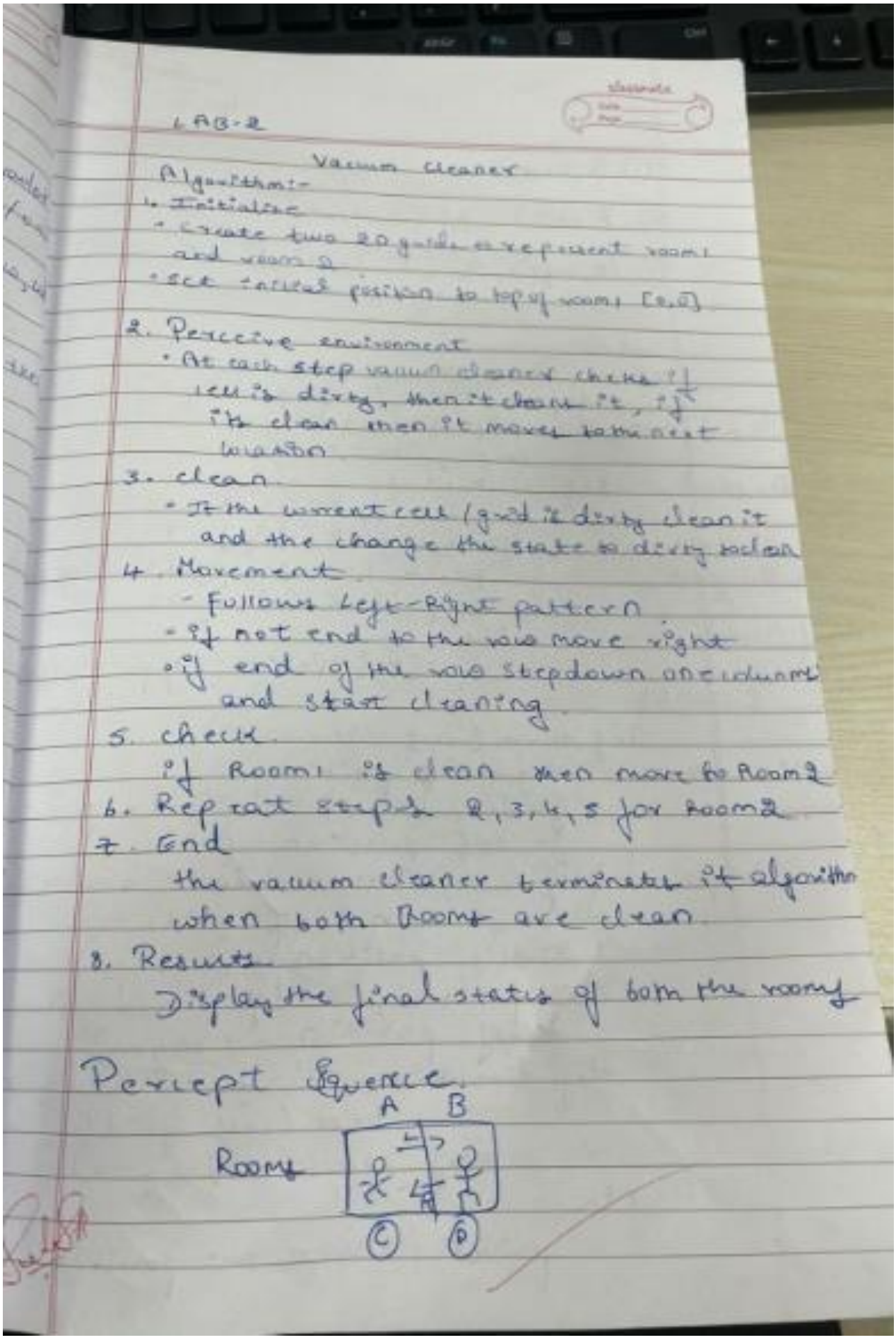
Algorithm:-

1. Initialize
   - Create two 2D grids to represent room1 and room2
   - Set initial position to top of room1 [0,0]
2. Perceive environment
   - At each step vacuum cleaner checks if cell is dirty, then it cleans it, if it's clean then it moves to the next location
3. clean
   - If the current cell / grid is dirty clean it and the change the state to dirty to clean
4. Movement
   - Follows Left-Right pattern
   - if not end of the row move right
   - if end of the row stepdown one room and start cleaning
5. check
   if Room1 is clean then move to Room2
6. Repeat steps 2, 3, 4, 5 for Room2
7. End
   the vacuum cleaner terminates it algorithm when both Rooms are clean
8. Results
   Display the final status of both the rooms

Percept Sequence



Observation Book:-



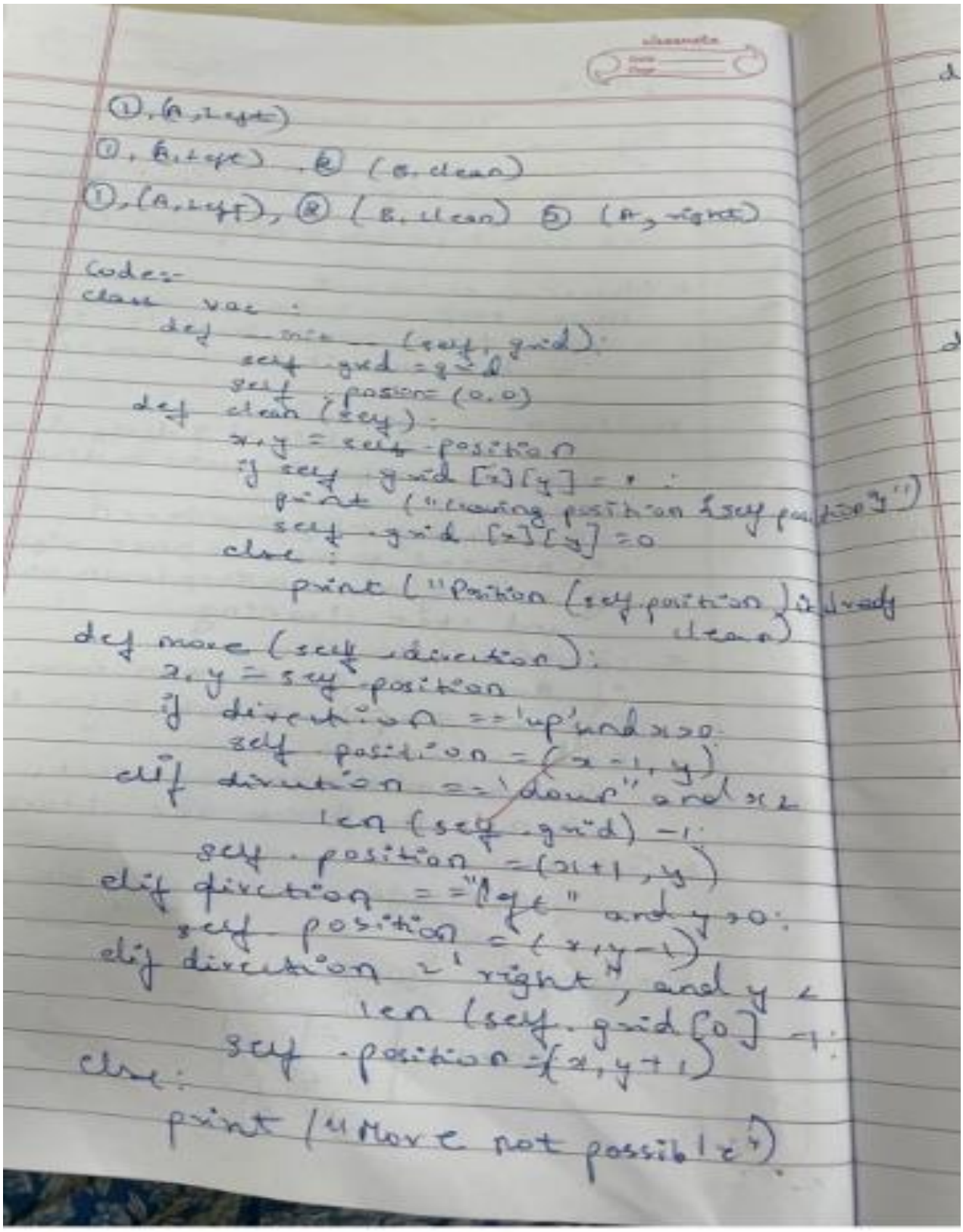
LAB-2

Vacuum cleaner

Algorithm:-

1. Initialize
   - Create two 2D grids to represent room1 and room 2
   - Set initial position to top of room1 [0,0]
2. Perceive environment
   - At each step vacuum cleaner checks if cell is dirty, then it cleans it, if it is clean then it moves to the next location
3. clean
   - If the current cell / grid is dirty clean it and the change the state to dirty to clean
4. Movement
   - Follows Left-Right pattern
   - if not end of the row move right
   - if end of the row stepdown one column and start cleaning
5. check
   if Room1 is clean then move to Room2
6. Repeat steps 2, 3, 4, 5 for Room2
7. End
   the vacuum cleaner terminates it algorithm when both Rooms are clean
8. Results
   Display the final status of both the rooms

Percept Sequence



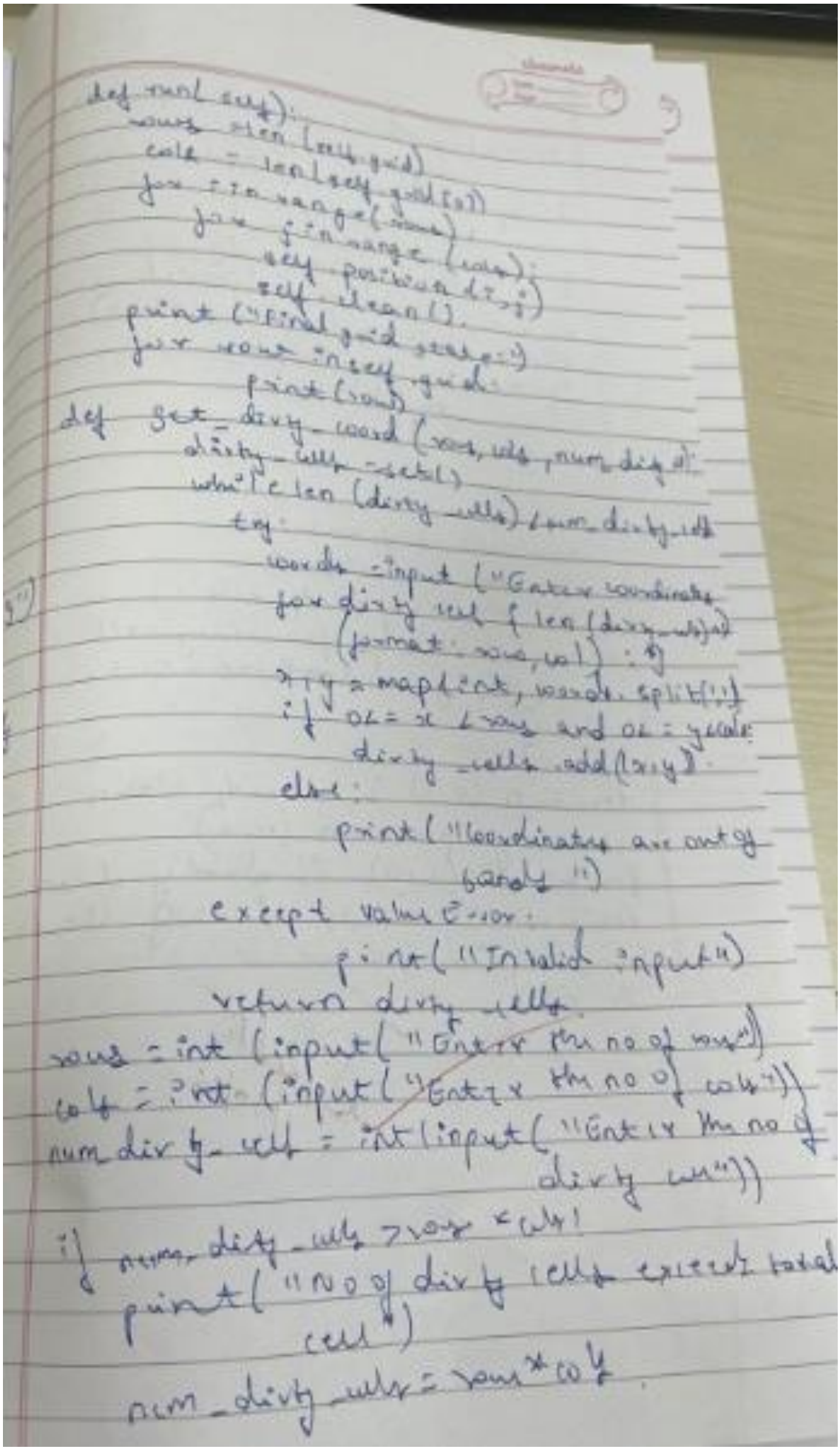
① (A, Left)

① (B, Left) ② (B, clean)

① (A, Left), ② (B, clean) ③ (A, right)

Code:-

```
class vac :
    def __init__(self, grid):
        self.grid = grid
        self.posn = (0,0)
    def clean(self):
        x,y = self.position
        if self.grid[x][y] = 1:
            print("cleaning position {self.position}")
            self.grid[x][y] = 0
        else:
            print("Position (self.position) is already
                                    clean")
    def move(self, direction):
        x,y = self.position
        if direction == 'up' and x > 0:
            self.position = (x-1, y)
        elif direction == 'down' and x <
                len(self.grid) - 1:
            self.position = (x+1, y)
        elif direction == "left" and y > 0:
            self.position = (x, y-1)
        elif direction == 'right' and y <
                len(self.grid[0]) - 1:
            self.position = (x, y+1)
        else:
            print("Move not possible")
```



```
def __init__(self):
    rows = len(self.grid)
    cols = len(self.grid[0])
    for i in range(rows):
        for j in range(cols):
            self.position[i,j]
            self.clean()
    print("Final grid state:")
    for row in self.grid:
        print(row)
def get_dirty_coord(rows, cols, num_dirty):
    dirty_cells = set()
    while len(dirty_cells) < num_dirty:
        try:
            coords = input("Enter coordinates
            for dirty cell {len(dirty_cells)+1}
                (format: row,col): ")
            x,y = map(int, coords.split(','))
            if 0 <= x < rows and 0 <= y < cols:
                dirty_cells.add((x,y))
            else:
                print("Coordinates are out of
                    bounds!")
        except ValueError:
            print("Invalid input")
    return dirty_cells
rows = int(input("Enter the no of rows"))
cols = int(input("Enter the no of cols"))
num_dirty_cell = int(input("Enter the no of
                    dirty cells"))
if num_dirty_cells > rows * cols:
    print("No of dirty cells exceeds total
        cell")
num_dirty_cells = rows * cols
```

```
initial_grid = [[0]*cols for _ in range(rows)]
dirty_coordinates = get_dirty_coordinates(rows, cols, num_dirty_cells)
for x, y in dirty_coords:
    initial_grid[x][y] = 1
vacuum = VacuumCleaner(initial_grid)
print("Initial grid state:")
for row in initial_grid:
    print(row)
vacuum.run()
```

Output:-

```
Enter the no of rows: 2
Enter the no of cols: 2
Enter the no of dirty cells: 1
Enter coordinates for dirty cell: 0,1
Initial grid state:
[0, 1]
[0, 0]
Position (0, 0) is already clean
Cleaning position (0, 1)
Position (1, 0) is already clean
Position (1, 1) is already clean
Final grid state:
[0 0]
[0 0]
```

Sreelal
11/10/24

* Lab-03 :-

DATE: 08/10/24 PAGE: 11

* solve 8 puzzle problems using DFS and Manhattan distance

→ The 8 puzzle problem consisting of a 3x3 grid with tiles which are numbered from 1 to 8 and one empty space. the goal is to rearrange the tiles to reach a specified end state.

1	2	3
4	5	6
7	8	-

→ end state.

* algorithm :-

initialize :

* create a stack to store the states to be explored. (stack data structure used by DFS)
* Add initial state to the stack
* Keep track of states that have been explored as visited set.

while the stack is Not empty :-

- pop the top state from
- check if the current state is the goal state.
- return - if yes
- Mark the current state as visited.
- Generate all moves to reach end state.

* calculate the manhattan distance
prioritize moves using Manhattan distance
* Repeat the same thing until you get a end state.

* Return result : If the goal is reached give it as output or else the puzzle is unsolved.

DATE: PAGE:

* Manhattan distance :-
* Initialization
  - start with the initial state of the puzzle.
  - Define the goal state of the puzzle.
  - check the current state of the puzzle.
  - cost of reaching that state
  - path of moves that led to that state

* calculate the Manhattan distance.
  - compute the manhattan distance blw the current position & the target position of that tile.
  - calcuate the sum of distances of all tiles.
* Begin search :- check if this state matches the goal state.
  if yes, retur the path that led to this state or else continue exploring neighboring states
* Explore the neighboring states.
  For each valid move.
  * compute the total cost for reaching the new state (path length + Manhattan distance)
  if the state has been visited before mark it as visted otherwise continue to explore
* Repeat till you get end state.
* Return the solution path.
  if the goal is reached give it as output
  Otherwise the puzzle is unsolved.

initial state.

1 2 3
4 0 5
6 7 8

1. up : Moves the empty tile up

1 0 3
4 2 5
6 7 8

* Down : Moves the empty tile down
1 2 3
4 7 5
6 0 8

* Left : Moves the empty tile to the left
1 2 3
0 4 5
6 7 8

* DFS:

1. Initialize:
   - create a stack and push the initial state onto it
   - create a set to track visited states.
2. while the stack is not empty:
   a. Pop the top state from the stack.
   b. If this state is the goal return the solution path.
   c. If the state is not visited
      - Mark it as visited.
      - push all possible new states (by moving the empty tile on to the stack)

* If the goal state is not reached return "No solution found"

* Initial state:
1 2 3
4 0 5
6 7 8

DATE: PAGE:

* up : swap with 2, resulting in
  1 0 3
  4 2 5
  6 7 8
* Down : swap with 7, resulting in
  1 2 3
  4 7 5
  6 0 8
* Left : swap with 4, resulting in
  1 2 3
  0 4 5
  6 7 8

code : DFS

```
class Node:
def __init__(self, state, parent = None, move = None
    self.state = state
    self.parent = parent
    self.move = move.
def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
def get_neighbors(node):
    neighbors = []
    state = node.state
    blank_index = state.index(0)
    MOVE = {
        'up': -3
        'down': 3
```

'left' : -1
'right' : 1
}

For move, pos_change in moves.items():
new_blank_index = blank_index + pos_change
if move == 'left' & blank_index % 3 == 0:
continue
if move == 'right' and blank_index % 3 == 2:
continue
if new_blank_index < 0 or new_blank_index >= 9:
continue
new_state = state[:]
new_state[blank_index], new_state[new_blank_index]
= new_state[new_blank_index], new_state[blank_inde

8/10/24

* code:

def manhattan_distance function
distance = 0
for i in range(3):
for j in range(3):
if state[i][j] != 0:
goal_i = (state[i][j] - 1) // 3
goal_j = (state[i][j] - 1) % 3
distance += abs(i - goal_i) + abs(j - goal_j)
return distance.

def get_neighbors(state):
neighbors = []
for i in range(3):
for j in range(3):

Lab 03:

8-Puzzel Game

Code:-

Using Depth First Search (DFS)

```
class SlidingPuzzle:
    def __init__(self, board, empty_pos, path=[]):
        self.board = board
        self.empty_pos = empty_pos
        self.path = path

    def is_solved(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_moves(self):
        x, y = self.empty_pos
        possible_moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = self.board[:]
                new_board[x * 3 + y], new_board[nx * 3 + ny] = new_board[nx * 3 + ny], new_board[x * 3 +
y]
                possible_moves.append((new_board, (nx, ny)))
        return possible_moves

def  depth_first_search(initial_puzzle):
    stack, visited = [initial_puzzle], set()
    while stack:
         current_puzzle = stack.pop()
        if current_puzzle.is_solved():
            return current_puzzle.path
```

```
        visited.add(tuple(current_puzzle.board))
        for new_board, new_empty_pos in current_puzzle.get_moves():
            new_state = SlidingPuzzle(new_board, new_empty_pos, current_puzzle.path + [new_board])
            if tuple(new_board) not in visited:
                stack.append(new_state)
    return None

def display_board(board):
    for i in range(0, 9, 3):
        print(board[i:i + 3])
    print()

def main():
    initial_board = [1, 2, 3, 4, 0, 5, 7, 8, 6]
    empty_pos = initial_board.index(0)
    initial_puzzle = SlidingPuzzle(initial_board, (empty_pos // 3, empty_pos % 3))

    print("Initial state:")
    display_board(initial_board)

    solution = depth_first_search(initial_puzzle)

    if solution:
        print("Solution found:")
        for step in solution:
            display_board(step)
    else:
        print("No solution found.")

if __name___== "__main__":
```

```
main()
```

Output:-

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

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

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

Code:-

Using Manhattan Distance

```
class SlidingPuzzleSolver:
    def __init__(self, initial_state):
        self.initial_state = initial_state
        self.goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def manhattan_distance(self, state):
        distance = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] != 0:
                    goal_i = (state[i][j] - 1) // 3
                    goal_j = (state[i][j] - 1) % 3
                    distance += abs(i - goal_i) + abs(j - goal_j)
        return distance

    def get_neighbors(self, state):
```

```
        i, j = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)
        moves = [(i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1)]
        return [self.swap(state, i, j, x, y) for x, y in moves if 0 <= x < 3 and 0 <= y < 3]

    def swap(self, state, i1, j1, i2, j2):
        new_state = [row[:] for row in state]
        new_state[i1][j1], new_state[i2][j2] = new_state[i2][j2], new_state[i1][j1]
        return new_state

    def dfs_with_manhattan(self, state, visited=set()):
        if state == self.goal_state:
            return [state]
        visited.add(str(state))
        neighbors = sorted(self.get_neighbors(state), key=lambda x: self.manhattan_distance(x))
        for neighbor in neighbors:
            if str(neighbor) not in visited:
                path = self.dfs_with_manhattan(neighbor, visited)
                if path:
                    return [state] + path
        return None

    def solve(self):
        solution = self.dfs_with_manhattan(self.initial_state)
        return solution


initial_state = [[int(x) for x in input(f"Enter row {i + 1}: ").split()] for i in range(3)]
solver = SlidingPuzzleSolver(initial_state)
solution = solver.solve()

if solution:
```

```
    print("Solution found:")
    for state in solution:
        print(*state, sep='\n', end='\n\n')
else:
    print("No solution found.")
```

Output:-

```
>>
===== RESTART: C:/Users/User/AppData/Local/Programs/
Enter row 1: 1 0 3
Enter row 2: 4 2 6
Enter row 3: 7 5 8
Solution found:
[1, 0, 3]
[4, 2, 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]
```

* Lab-04:

DATE: 15/10/24 PAGE: 18

* Iterative developing search algorithm
A* Algorithm for 8 puzzle.

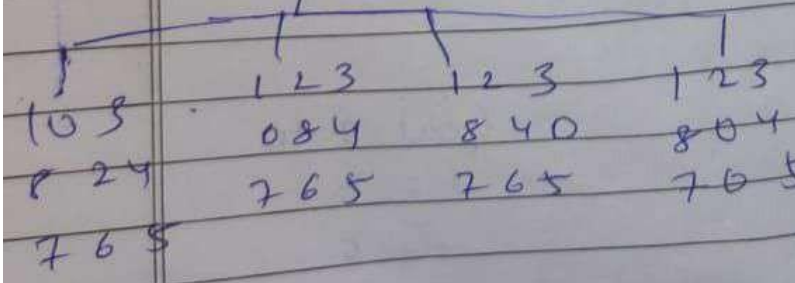
1. Initalize
   * start with the initial state of puzzle
   * set the goal state
2. Queue :-
   To store the state of puzzle
   $f(n) = g(n) + h(n)$.
   g(n) :- is the no. of moves taken from start
   h(n) - misplaced title which count how many titles are not in their goal positions
3. States :
   - If the goal state is achieved then return goal state achieved
   - If the goal state not achieved then given all the possible move
4. continue / Repeat until the goal state is reached

Initial state

1	2	3
8		4
7	6	5

goal state

2	8	1
	4	3
7	6	5



pseudo code :- A* 8puzzle

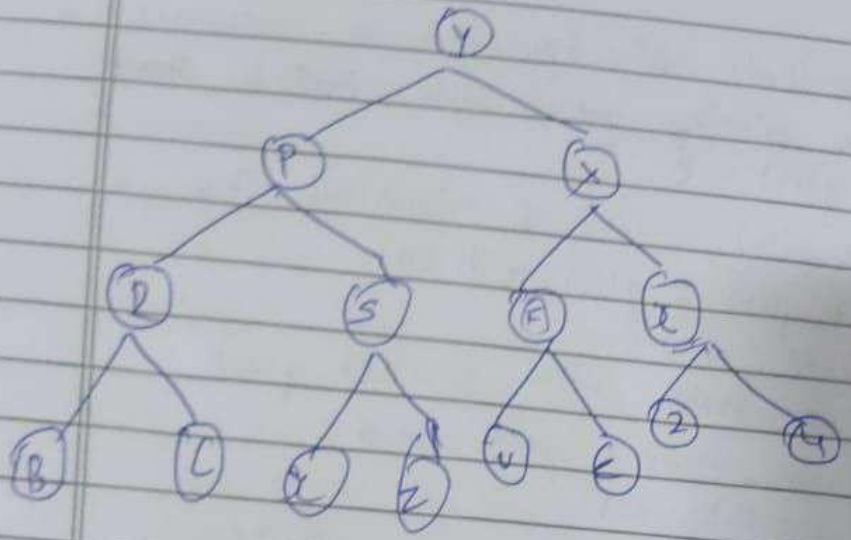
1. Initialize priority queue with initial start
   set g(n) = 0
   sets f(n) = no of misplaced tits
   set f(n) = g(n) + h(n)
2. while ques ! = 0:
   * Remove state with smaller f(n)
   * If cur - state = goal - state
     - Rets sol
   * Generate all the new states.
     - calculate g(n), h(n), f(n)
3. If goal reached then return the soln

Pseudocode :- A* IDFS

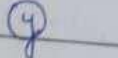
1. Function IDFS (root, goal)
   for d = 0 to ∞ :
   res = DLS (root, goal, d)
   if res ≠ NULL :
   return res
   return NULL
   function DLS (node, goal, d)
   if d == 0 and node == goal
   return node
   if dept > 0
   for child in node.children
   res = DLS (child, goal)
   if res ≠ NULL
   return result
   return NULL

DATE:

PAGE: 24

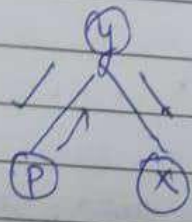


d = 0



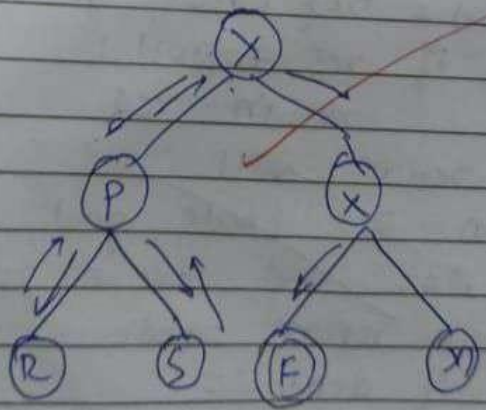
return null

d = 1



return null

d = 2



return 5

LAB-04 - 8 Puzzle with A* and IDDFS on a Graph

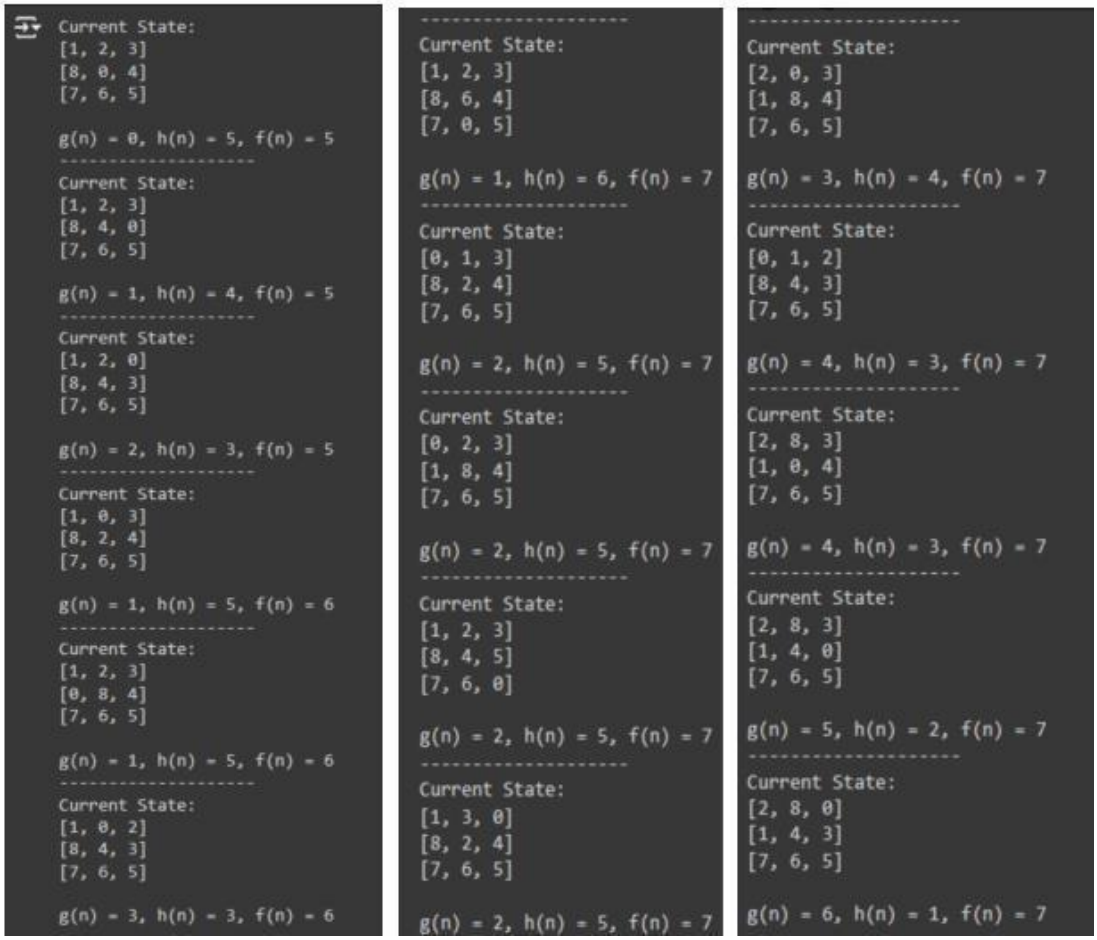
Code: (8 Puzzle with A*)

```
import heapq

# Goal state where blank (0) is the first tile

goal_state = [

[0, 1, 2],

[3, 4, 5],

[6, 7, 8]

]

# Helper functions

def flatten(puzzle):

return [item for row in puzzle for item in row]

def find_blank(puzzle):

for i in range(3):

for j in range(3):

if puzzle[i][j] == 0:

return i, j

def misplaced_tiles(puzzle):

flat_puzzle = flatten(puzzle)

flat_goal = flatten(goal_state)

return sum([1 for i in range(9) if flat_puzzle[i] != flat_goal[i] and flat_puzzle[i] != 0])

def generate_neighbors(puzzle):

x, y = find_blank(puzzle)

neighbors = []

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

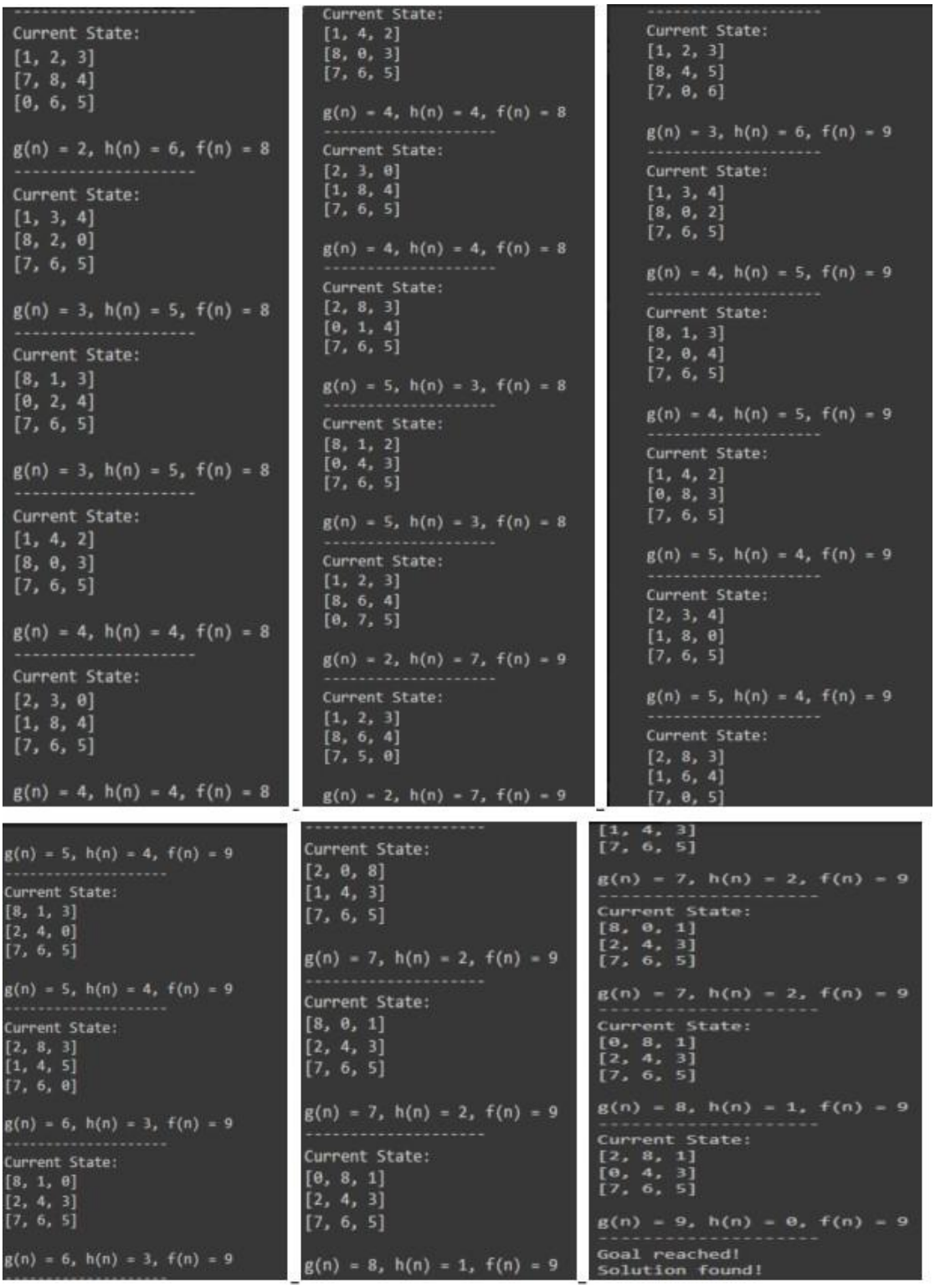
```
for dx, dy in moves:
nx, ny = x + dx, y + dy
if 0 <= nx < 3 and 0 <= ny < 3:
new_puzzle = [row[:] for row in puzzle]
new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny], new_puzzle[x][y]
neighbors.append(new_puzzle)
return neighbors
def is_goal(puzzle):
return puzzle == goal_state
def print_puzzle(puzzle):
for row in puzzle:
print(row)
print()
def a_star_misplaced_tiles(initial_state):
# Priority queue (min-heap) and visited states
frontier = []
heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
visited = set()
while frontier:
f, g, current_state, path = heapq.heappop(frontier)
# Print the current state
print("Current State:")
print_puzzle(current_state)
h = misplaced_tiles(current_state)
print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")
```

```
print("-" * 20)

if is_goal(current_state):

print("Goal reached!")

return path

visited.add(tuple(flatten(current_state)))

for neighbor in generate_neighbors(current_state):

if tuple(flatten(neighbor)) not in visited:

h = misplaced_tiles(neighbor)

heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path + [neighbor]))

return None # No solution found

# Initial puzzle state

initial_state = [

[1, 2, 0],

[3, 4, 5],

[6, 7, 8]

]

solution = a_star_misplaced_tiles(initial_state)

if solution:

print("Solution found!")

else:

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

Ouput:-



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

g(n) = 0, h(n) = 5, f(n) = 5
--------------------
Current State:
[1, 2, 3]
[8, 4, 0]
[7, 6, 5]

g(n) = 1, h(n) = 4, f(n) = 5
--------------------
Current State:
[1, 2, 0]
[8, 4, 3]
[7, 6, 5]

g(n) = 2, h(n) = 3, f(n) = 5
--------------------
Current State:
[1, 0, 3]
[8, 2, 4]
[7, 6, 5]

g(n) = 1, h(n) = 5, f(n) = 6
--------------------
Current State:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

g(n) = 1, h(n) = 5, f(n) = 6
--------------------
Current State:
[1, 0, 2]
[8, 4, 3]
[7, 6, 5]

g(n) = 3, h(n) = 3, f(n) = 6
--------------------
Current State:
[1, 2, 3]
[8, 6, 4]
[7, 0, 5]

g(n) = 1, h(n) = 6, f(n) = 7
--------------------
Current State:
[0, 1, 3]
[8, 2, 4]
[7, 6, 5]

g(n) = 2, h(n) = 5, f(n) = 7
--------------------
Current State:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 2, h(n) = 5, f(n) = 7
--------------------
Current State:
[1, 2, 3]
[8, 4, 5]
[7, 6, 0]

g(n) = 2, h(n) = 5, f(n) = 7
--------------------
Current State:
[1, 3, 0]
[8, 2, 4]
[7, 6, 5]

g(n) = 2, h(n) = 5, f(n) = 7
--------------------
Current State:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

g(n) = 3, h(n) = 4, f(n) = 7
--------------------
Current State:
[0, 1, 2]
[8, 4, 3]
[7, 6, 5]

g(n) = 4, h(n) = 3, f(n) = 7
--------------------
Current State:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

g(n) = 4, h(n) = 3, f(n) = 7
--------------------
Current State:
[2, 8, 3]
[1, 4, 0]
[7, 6, 5]

g(n) = 5, h(n) = 2, f(n) = 7
--------------------
Current State:
[2, 8, 0]
[1, 4, 3]
[7, 6, 5]

g(n) = 6, h(n) = 1, f(n) = 7
```



Code: (IDDFS on a Graph)

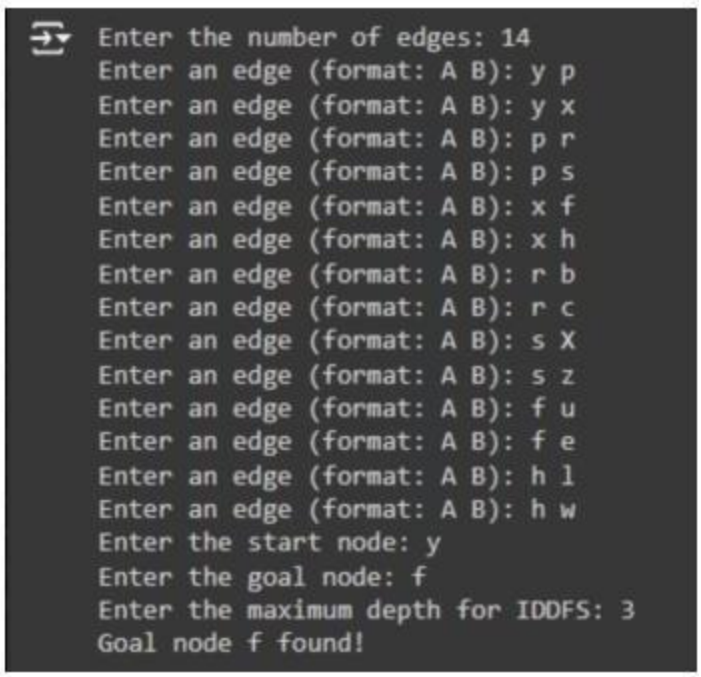
class Graph:

def __init__(self):

```
self.adjacency_list = {}
def add_edge(self, u, v):
if u not in self.adjacency_list:
self.adjacency_list[u] = []
self.adjacency_list[u].append(v)
def depth_limited_dfs(self, node, goal, limit, visited):
if limit < 0:
return False
if node == goal:
return True
visited.add(node)
for neighbor in self.adjacency_list.get(node, []):
if neighbor not in visited:
if self.depth_limited_dfs(neighbor, goal, limit - 1, visited):
return True
visited.remove(node) # Allow revisiting for the next iteration
return False
def iddfs(self, start, goal, max_depth):
for depth in range(max_depth + 1):
visited = set()
if self.depth_limited_dfs(start, goal, depth, visited):
return True
return False
def main():
graph = Graph()
# Input number of edges
num_edges = int(input("Enter the number of edges: "))
# Input edges
```

```
for _ in range(num_edges):
edge = input("Enter an edge (format: A B): ").split()
graph.add_edge(edge[0], edge[1])
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")
max_depth = int(input("Enter the maximum depth for IDDFS: "))
if graph.iddfs(start_node, goal_node, max_depth):
print(f"Goal node {goal_node} found!")
else:
print(f"Goal node {goal_node} not found within depth {max_depth}.")
if __name___== "__main__":
main()
```

Output:



* Simulated Annealing algorithm ;-

- Algorithm

1. Initialize the temperature and a random solution
2. Evaluate the objective function
   - This is to either minize or maxmize
3. Generate a new solution in the neighborhood of the current solution
   - Modification can be done by adding or substracting a small random value from the current solution. This ensure that the new solution is closer to the current one.
4. compare the new solution to current one
   a. if new solution is better, accept it
   b. if the new solution is worse, accept it with certain probability.

the acceptance probability is given by $p = e^{-\Delta x / T}$

Δf = (new sol) - (cur sol)

T = Current temp.

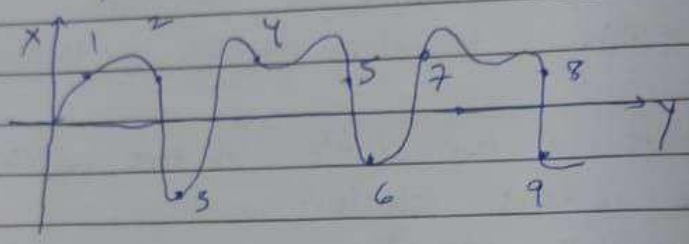
5. Gradually 'cooler the temperature - This process is known as cooling and it ensure that over time, the algorithm becomes more selective about which solutions to accept.
6. Repeat until the system reaches a stopping criterion.

- if the temperature has reached a predefined the - shold.
- Maximum iterations has been reached.

DATE: PAGE: 22

- change in the objective function is too small or almost negligible

* Example:



Pseudocode

```
func obj(x)
  return x^2
  END FUNC
FUNC sa (sit, e, m)
  SET cs = s
  SET bs = cs
  SET bs = obj(cs)
  FOR i FROM 1 to MDO
    SET ns = cs + Random (-1, 1)
    SET nc = obj(ns)
    SET cd = nc - bc
IF cdc0 OR Random (0,1) <
          Exp(-cd/t)
  SETcs = ns
  SET bs = cs IF nc < bc
  END IF
  SET t = t* e
PRINT i, cs, nc, t
  END FOR
  RETURN bs, bc
  END FUNC
```

```
BEGIN
    READ S, t, C, M
If = C<=0 OR C>=, THEN
    PRINT "Invalid, wcling realn"
    EXIT
END If
    bS, bC = sa (s, t, c, m)
    PRINT bS, bC
END
```

Output:

Enter the initial state (starting point) = 6.0
Enter the initial temperature = 12
Enter the cooling rate (blw 0 to 1) = 0.2
Enter the number of iterations : 2.5
i=1 : CS = 9.27, CC = 85.99, Temp = 2.400
i=2 : CS = 9.25, CC = 88.61, Temp = 0.480 0

6.5 = CS = 3.5 CC = 12.83, Temp 0.000
But state : 3.58 best energy :- 12.83

26/11/24

LAB 05:

Stimulated Annealing Algorithm

code:-

```
import numpy as np
import math
import random

def objective_function(x):
    """Objective function to minimize: f(x) = x^2"""
    return x ** 2

def simulated_annealing(initial_state, initial_temp, cooling_rate, max_iterations):
    """Simulated Annealing algorithm to find the minimum of the objective function."""

    current_state = initial_state
    current_energy = objective_function(current_state)
    best_state = current_state
    best_energy = current_energy

    temp = initial_temp

    for iteration in range(max_iterations):
        # Generate a new candidate state by perturbing the current state
        candidate_state = current_state + random.uniform(-1, 1)
        candidate_energy = objective_function(candidate_state)
```

```
        # Calculate energy difference
        energy_diff = candidate_energy - current_energy

        # If the candidate state is better, or accepted with a certain probability
        if energy_diff < 0 or random.uniform(0, 1) < math.exp(-energy_diff / temp):
            current_state = candidate_state
            current_energy = candidate_energy

            # Update best state found
            if current_energy < best_energy:
                best_state = current_state
                best_energy = current_energy

        # Cool down the temperature
        temp *= cooling_rate

        # Print the current state and temperature for debugging
        print(f"Iteration {iteration + 1}: Current State = {current_state:.4f}, Current Energy =
{current_energy:.4f}, Temperature = {temp:.4f}")

    return best_state, best_energy

# Get user input for parameters
try:
    initial_state = float(input("Enter the initial state (starting point): "))
```

```
    initial_temp = float(input("Enter the initial temperature: "))
    cooling_rate = float(input("Enter the cooling rate (between 0 and 1): "))
    max_iterations = int(input("Enter the number of iterations: "))

    # Validate cooling rate
    if cooling_rate <= 0 or cooling_rate >= 1:
        raise ValueError("Cooling rate must be between 0 and 1.")

    # Execute the simulated annealing algorithm
    best_state, best_energy = simulated_annealing(initial_state, initial_temp, cooling_rate,
max_iterations)

    # Output the best state and energy found
    print(f"Best State: {best_state:.4f}, Best Energy: {best_energy:.4f}")

except ValueError as e:
    print(f"Invalid input: {e}")
```

output:-

```
Enter the initial state (starting point): 10
Enter the initial temperature: 12
Enter the cooling rate (between 0 and 1): 0.2
Enter the number of iterations: 25
Iteration 1: Current State = 9.2736, Current Energy = 85.9995, Temperature = 2.4000
Iteration 2: Current State = 9.2528, Current Energy = 85.6140, Temperature = 0.4800
Iteration 3: Current State = 8.4448, Current Energy = 71.3150, Temperature = 0.0960
Iteration 4: Current State = 8.0267, Current Energy = 64.4277, Temperature = 0.0192
Iteration 5: Current State = 8.0267, Current Energy = 64.4277, Temperature = 0.0038
Iteration 6: Current State = 7.1132, Current Energy = 50.5978, Temperature = 0.0008
Iteration 7: Current State = 7.0877, Current Energy = 50.2356, Temperature = 0.0002
Iteration 8: Current State = 7.0877, Current Energy = 50.2356, Temperature = 0.0000
Iteration 9: Current State = 6.8309, Current Energy = 46.6618, Temperature = 0.0000
Iteration 10: Current State = 6.8309, Current Energy = 46.6618, Temperature = 0.0000
Iteration 11: Current State = 6.8309, Current Energy = 46.6618, Temperature = 0.0000
Iteration 12: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 13: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 14: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 15: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 16: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 17: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 18: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 19: Current State = 6.1567, Current Energy = 37.9046, Temperature = 0.0000
Iteration 20: Current State = 5.2467, Current Energy = 27.5274, Temperature = 0.0000
Iteration 21: Current State = 5.2467, Current Energy = 27.5274, Temperature = 0.0000
Iteration 22: Current State = 5.2467, Current Energy = 27.5274, Temperature = 0.0000
Iteration 23: Current State = 4.5909, Current Energy = 21.0761, Temperature = 0.0000
Iteration 24: Current State = 4.3835, Current Energy = 19.2152, Temperature = 0.0000
Iteration 25: Current State = 3.5823, Current Energy = 12.8326, Temperature = 0.0000
Best State: 3.5823, Best Energy: 12.8326
```

DATE: 29/10/24 PAGE: 24

* LAB-06 :-

* Implementation of A* search algorithm for 8 queens

→ * Algorithm :-

1. create an empty board where no queens are placed which start state.
2. Heuristic (h) : h is the number of attacking queens pairs. where lower values are better.
3. priority fn : f(n) = g(n) + h(n)
   where g(n) is the number of queens placed so far
   where as f prioritizes states with fewer attacking queens.
4. Add start state to the priority queue :
   * Add the initial empty board with f = 0
   * while queue not empty
     - select states with lowest attacking queens
     - Now check for the sol^n :- if all the 8 queens are placed without any difficulties ther return the board.
     - place a queen in the next row in valid columns & add each state to the queue.
5. Return solution ~~if the queue is empty~~.

DATE: PAGE: 25

* Implementation of Hill climbing algorithm for 8 queens.

→ initialize :-
- start with a random configuration of 8 queens on a board.
where one queen per colums is placed.

→ Evaluate Heuristic :-
Now calculate the number of pairs of queens that are attacking each other (lower values are better h).

→ For each queen, move it within it's column to each row & calculate new heuristic for each board -ing state.

→ select the neighboring state with the lowest heuristic value h',
if h' is lower than the current h, update the current state to this neighbor.

→ Repeat :- continue evaluating neighbors until h=0

→ If stuck in any state, restart with new state or random state, repeat some

Proceed

DATE: 26 PAGE:

* code:- Pseudocode:

```
Function A* (start)
    open_set = priority queen()
    open_set.enqueue(start, priority = heuristic(start))

while open_set is not empty
    current = open_set.dequeue()
    if heuristic(current) == 0:
        return current
    for neighbor in neighbors(current):
        cost = heuristic(neighbor)
        open_set.enqueue(neighbor, priority = cost)

Function heuristic(state):
    return no of conflicts queen pairs in state.
```

* output:

A* solution : [0, 4, 7, 5, 2, 6, 1, 3]

DATE: 27 PAGE:

* Hill climbing Pseudo code :-

```
function Hill (start)
    current = start
    while true :
        next = best_neighbor (current)
    if heuristic (next) >= heuristic (current)
return current if heuristic (current) == 0
    else failure
function best_no: gh(state)
    return neighbour of state with the lowest
        heuristic
funtion heuristic (state)
    return no of conflicting queen pairs in
        state.
```

* output :

Hill climbing solution : [1, 3, 0, 6, 4, 2, 5, 7]

other possible outputs:

sol : [0, 2, 4, 6, 1, 3, 5, 7]

sol : [3, 0, 6, 2, 5, 1, 7, 4]

LAB 06

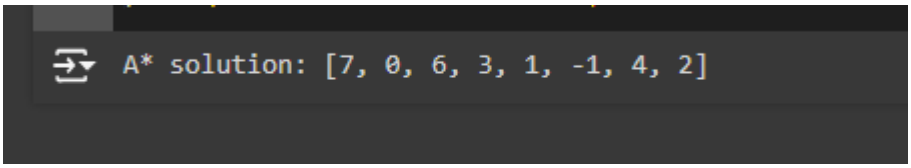
A* Algorithm code for 8 Queens:

Code :

```
import numpy as np
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state  # current state of the board
        self.g = g          # cost to reach this state
        self.h = h          # heuristic cost to reach goal
        self.f = g + h      # total cost
    def __lt__(self, other):
        return self.f < other.f

def heuristic(state):
    # Count pairs of queens that can attack each other
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def a_star_8_queens():
    initial_state = [-1] * 8  # -1 means no queen placed
    open_list = []
    closed_set = set()
```

```
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            for row in range(8):
                if current_state[col] == -1:  # Only place a queen if none is present in this column
                    new_state = current_state.copy()
                    new_state[col] = row
                    if tuple(new_state) not in closed_set:
                        g_cost = current_node.g + 1
                        h_cost = heuristic(new_state)
                        heapq.heappush(open_list, Node(new_state, g_cost, h_cost))

    return None

solution = a_star_8_queens()
print("A* solution:", solution)
```

qoutput:



Hill Climbing for 8 queens

```
import random

def heuristic(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def hill_climbing_8_queens():
    state = [random.randint(0, 7) for _ in range(8)]  # Random initial state
    while True:
        current_h = heuristic(state)
        if current_h == 0:  # Found a solution
            return state

        next_state = None
        next_h = float('inf')

        for col in range(8):
            for row in range(8):
                if state[col] != row:  # Only consider moving the queen
                    new_state = state.copy()
```

```
            new_state[col] = row
            h = heuristic(new_state)
            if h < next_h:
                next_h = h
                next_state = new_state

    if next_h >= current_h:  # No better neighbor found
        return None  # Stuck at local maximum
    state = next_state

solution = hill_climbing_8_queens()
print("Hill Climbing solution:", solution)
```

Output:

```
Hill Climbing solution: [4, 2, 7, 3, 6, 0, 5, 1]
```

LAB 06

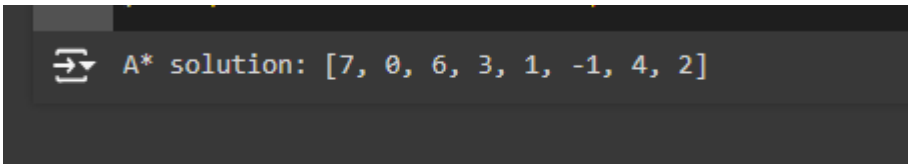
A* Algorithm code for 8 Queens:

Code :

```
import numpy as np
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state  # current state of the board
        self.g = g          # cost to reach this state
        self.h = h          # heuristic cost to reach goal
        self.f = g + h      # total cost
    def __lt__(self, other):
        return self.f < other.f

def heuristic(state):
    # Count pairs of queens that can attack each other
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def a_star_8_queens():
    initial_state = [-1] * 8  # -1 means no queen placed
    open_list = []
    closed_set = set()
```

```
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            for row in range(8):
                if current_state[col] == -1:  # Only place a queen if none is present in this column
                    new_state = current_state.copy()
                    new_state[col] = row
                    if tuple(new_state) not in closed_set:
                        g_cost = current_node.g + 1
                        h_cost = heuristic(new_state)
                        heapq.heappush(open_list, Node(new_state, g_cost, h_cost))

    return None

solution = a_star_8_queens()
print("A* solution:", solution)
```

qoutput:



Hill Climbing for 8 queens

```
import random

def heuristic(state):
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def hill_climbing_8_queens():
    state = [random.randint(0, 7) for _ in range(8)]  # Random initial state
    while True:
        current_h = heuristic(state)
        if current_h == 0:  # Found a solution
            return state

        next_state = None
        next_h = float('inf')

        for col in range(8):
            for row in range(8):
                if state[col] != row:  # Only consider moving the queen
                    new_state = state.copy()
```

```
                new_state[col] = row
                h = heuristic(new_state)
                if h < next_h:
                    next_h = h
                    next_state = new_state

        if next_h >= current_h:  # No better neighbor found
            return None  # Stuck at local maximum
        state = next_state

solution = hill_climbing_8_queens()
print("Hill Climbing solution:", solution)
```

Output:

```
Hill Climbing solution: [4, 2, 7, 3, 6, 0, 5, 1]
```

LAB - 07 :-

DATE: 28 PAGE: 12/11/24

* Solve :

1. Alice is the mother of bob
2. Bob is the father of charlie
3. A father is a parent
4. A mother is a parent
5. All parents have children
6. If someone is a parent, their children are siblings.
7. Alice is married to david.

Hypothesis :

Charlie is sibling of bob

→ Step by step

* Identify Relationships :

From (1) Alice is bob's mother

From (2) Bob is charlie's father

From (3) & (4) both mothers & fathers are parents

From (5) as parents, both Alice & bob have children

& from (6) if someone is a parent, their children are considered siblings.

Since bob is charlie's father, Charlie would be considered bob's child in a sibling.

conclusion :-

(c) Bob as a parent, implies that his children Charlie would be siblings if there were other children.

DATE: PAGE:

A : Alice is the mother of bob
C : charlie is a child of bob
S : children of parent are siblings
H : Hypothesis - charlie is a sibling of Bob

A	C	S	H	conclusion
T	T	T	T	True
T	T	F	F	Falx
T	F	T	F	Falx
F	T	T	F	Falx
F	F	F	F	Falx

LAB 07 :

check entailment :

CODE:

```
# Function to check entailment based on user input
def check_entailment():
    print("Welcome to the Entailment Checker!")

    # Step 1: Gather user input for facts (Premises)
    alice_is_mother_of_bob = input("Enter the fact: Alice is the mother
of Bob. (e.g., 'Alice is the mother of Bob')\n")
    bob_is_father_of_charlie = input("Enter the fact: Bob is the father
of Charlie. (e.g., 'Bob is the father of Charlie')\n")
    father_is_parent = input("Enter the fact: A father is a parent.
(e.g., 'A father is a parent')\n")
    mother_is_parent = input("Enter the fact: A mother is a parent.
(e.g., 'A mother is a parent')\n")
    all_parents_have_children = input("Enter the fact: All parents have
children. (e.g., 'All parents have children')\n")
    parents_children_are_siblings = input("Enter the fact: Parents'
children are siblings. (e.g., 'Parents' children are siblings')\n")
    alice_is_married_to_david = input("Enter the fact: Alice is married
to David. (e.g., 'Alice is married to David')\n")

    # Step 2: Entailment reasoning process
    if ('Alice is the mother of Bob' in alice_is_mother_of_bob and
        'Bob is the father of Charlie' in bob_is_father_of_charlie and
        'A father is a parent' in father_is_parent and
        'A mother is a parent' in mother_is_parent and
        'All parents have children' in all_parents_have_children and
        "Parents' children are siblings" in
parents_children_are_siblings and
        'Alice is married to David' in alice_is_married_to_david):

        # Conclusion: Check if Charlie is a sibling of Bob
        print("\nSince Alice is Bob's mother and Bob is Charlie's
father, Charlie and Bob are siblings.")
        print("Conclusion: Charlie is a sibling of Bob. The hypothesis
is entailed by the knowledge base.")
    else:
        print("\nThe information provided does not fully support the
conclusion.")

# Run the function
check_entailment()
```

output:

```
Welcome to the Entailment Checker!
Enter the fact: Alice is the mother of Bob. (e.g., 'Alice is the mother of Bob')
 Alice is the mother of Bob
Enter the fact: Bob is the father of Charlie. (e.g., 'Bob is the father of Charlie')
Bob is the father of Charlie
Enter the fact: A father is a parent. (e.g., 'A father is a parent')
A father is a parent
Enter the fact: A mother is a parent. (e.g., 'A mother is a parent')
 A mother is a parent
Enter the fact: All parents have children. (e.g., 'All parents have children')
All parents have children
Enter the fact: Parents' children are siblings. (e.g., 'Parents' children are siblings')
Parents' children are siblings
Enter the fact: Alice is married to David. (e.g., 'Alice is married to David')
Alice is married to David

Since Alice is Bob's mother and Bob is Charlie's father, Charlie and Bob are siblings.
Conclusion: Charlie is a sibling of Bob. The hypothesis is entailed by the knowledge base.
```

* First order logic :—

DATE: 19/11/22 PAGE: 30

* statement :- student have passed the exam, if every student who pased the exam receives the certificate, if John is studying & passed exam, will receive a certificate.

$\forall x$ (student(x) → passed (x, Exam))
$\forall x$ (passed (x, Exam) → Receive certificate
student (John)
passed (John, Exam)
Receives certificate (John)

Proof $\forall x$ (student(x) → passed (x, Exam))
tells us that if someone is student they passed the exam.

From student (John) → passed (John, Exam) so John has passed the exam

$\forall x$ (passed (x, Exam) → Recives certificate (x)
tells that if student has passed the exam they receive certificate.

From passed (John, Exam) $\forall$ → Recieves certificate)

Otp

=> Receives certificate (John) is true.

26/11/24

Lab 08

Fol

code:-

```
import re

# Define a simple function for extracting predicates from sentences
def extract_predicate(sentence):
    # Regular expression to find patterns like Predicate(Argument)
    pattern = r"([A-Za-z]+)\((\w+)\)"
    match = re.search(pattern, sentence)

    if match:
        predicate = match.group(1)
        subject = match.group(2)
        return predicate, subject
    return None, None

# Function for unification
def unify(fact, query):
    # Check if the fact and query are the same
    if fact == query:
        return True
    # Extract predicate and subject from fact and query
    fact_predicate, fact_subject = extract_predicate(fact)
    query_predicate, query_subject = extract_predicate(query)
```

```
    # If predicates match, unify the subjects
    if fact_predicate == query_predicate:
        if fact_subject == query_subject:
            return True
        else:
            # Here, we could handle variable substitution (unification)
            return False
    return False

# Function to deduce the goal using given rules
def deduct(rules, goal):
    # Try to find unification for the goal from the rules
    for rule in rules:
        if unify(rule, goal):
            print(f"Unification successful: {rule} matches with {goal}.")
            return True
    return False

# Main function to handle user input
def main():
    # Step 1: Get the rules (facts/implications) from the user
    print("Enter the rules (facts/implications). Type 'done' to finish entering rules.")
    rules = []
```

```
    while True:
        rule_input = input("Enter rule: ")
        if rule_input.lower() == 'done':
            break
        else:
            rules.append(rule_input.strip())

    # Step 2: Get the goal (query) from the user
    goal_input = input("Enter the goal (query) to prove: ").strip()

    # Step 3: Try to deduce the goal using the given rules
    print("\nAttempting to deduce the goal...")
    if deduct(rules, goal_input):
        print(f"Conclusion: The goal '{goal_input}' is true based on the rules.")
    else:
        print(f"Conclusion: The goal '{goal_input}' cannot be proven with the
provided rules.")

# Run the program
main()
```

output:

```
Enter the rules (facts/implications). Type 'done' to finish entering rules.
Enter rule:  Loves(Sam, Everyone)
Enter rule: done
Enter the goal (query) to prove: Loves(Everyone, Sam)

Attempting to deduce the goal...
Unification successful: Loves(Sam, Everyone) matches with Loves(Everyone, Sam).
Conclusion: The goal 'Loves(Everyone, Sam)' is true based on the rules.
```

LAB 09:

Implementation of unification in FLO

code:-

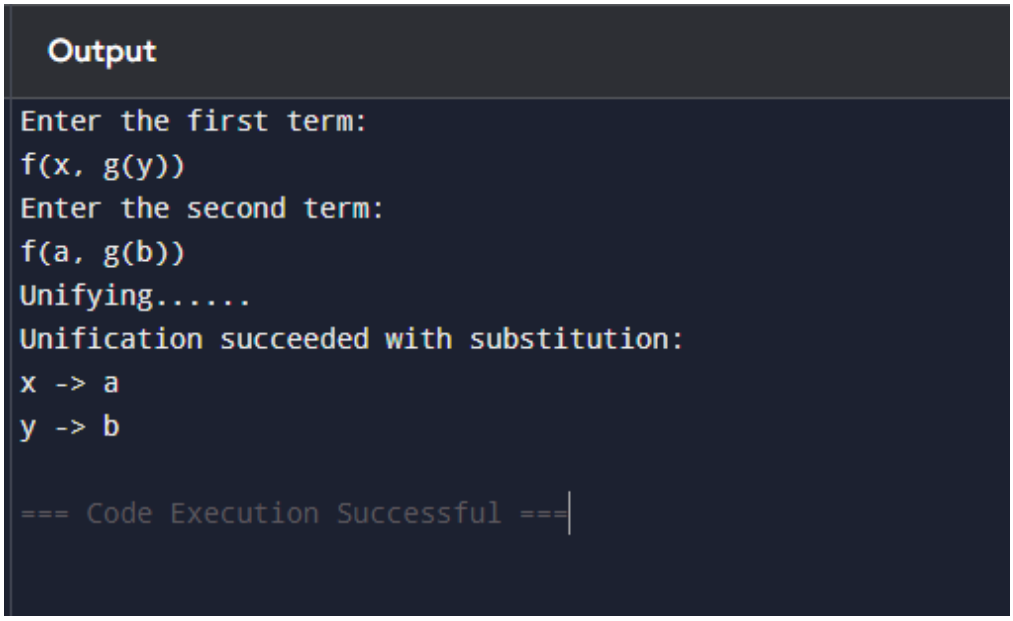
```
def is_variable(term):
    """
    Check if a term is a variable.
    Variables are typically single lowercase letters.
    """
    return isinstance(term, str) and term.islower()


def unify(expr1, expr2, subst={}):
    """
    Unify two expressions expr1 and expr2 under the given substitution subst.
    """
    if subst is None:
        return None  # Failure case
    if expr1 == expr2:
        return subst  # Expressions are identical
    if is_variable(expr1):
        return unify_variable(expr1, expr2, subst)
    if is_variable(expr2):
        return unify_variable(expr2, expr1, subst)
    if isinstance(expr1, tuple) and isinstance(expr2, tuple):
        if len(expr1) != len(expr2):
            return None  # Different arity
        # Recursively unify each component
        for arg1, arg2 in zip(expr1, expr2):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None  # Failure
        return subst
```

```
    return None  # No unification possible

def unify_variable(var, term, subst):
    """
    Unify a variable with a term, updating the substitution.
    """
    if var in subst:
        return unify(subst[var], term, subst)  # Apply substitution to var
    if term in subst:
        return unify(var, subst[term], subst)  # Apply substitution to term
    if occurs_check(var, term, subst):
        return None  # Circular substitution detected
    # Add var -> term to the substitution
    subst = subst.copy()
    subst[var] = term
    return subst

def occurs_check(var, term, subst):
    """
    Check if var occurs in term (directly or indirectly) to prevent circular substitutions.
    """
    if var == term:
        return True
    if isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    if term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def parse_input(expr):
    """
```

```
    Parse user input into a structured format (nested tuples for functions and terms).
    Example: "f(X, g(y))" -> ('f', 'X', ('g', 'y'))
    """
    expr = expr.strip()
    if '(' not in expr:
        return expr  # Simple variable or constant
    func_name = expr[:expr.index('(')].strip()
    args = expr[expr.index('(') + 1:expr.rindex(')')].split(',')
    args = [parse_input(arg.strip()) for arg in args]
    return (func_name, *args)

def format_output(expr):
    """
    Convert the nested tuple representation back into a string for output.
    Example: ('f', 'X', ('g', 'y')) -> "f(X, g(y))"
    """
    if isinstance(expr, str):
        return expr
    return f"{expr[0]}({', '.join(format_output(arg) for arg in expr[1:])})"

# Main Program
if __name___== "__main__":
    print("Enter the first term:")
    expr1 = parse_input(input().strip())
    print("Enter the second term:")
    expr2 = parse_input(input().strip())

    print("Unifying ..... ")
    result = unify(expr1, expr2)

    if result is None:
```

```
        print("Unification failed")
    else:
        print("Unification succeeded with substitution:")
        for var, term in result.items():
            print(f"{var} -> {format_output(term)}")
```

Output:-



Lab10

Code for tic tac toe

```
import math
from copy import deepcopy

# Define the Tic-Tac-Toe board size and players
EMPTY = "-"
PLAYER_X = "X"  # Maximizing player (Computer)
PLAYER_O = "O"  # Minimizing player (User)

# Helper functions
def is_terminal(board):
    """Checks if the game has ended."""
    winner = get_winner(board)
    if winner or not any(EMPTY in row for row in board):
        return True
    return False

def get_winner(board):
    """Checks for a winner on the board."""
    # Check rows and columns
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != EMPTY:
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != EMPTY:
            return board[0][i]
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != EMPTY:
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != EMPTY:
        return board[0][2]
    return None

def utility(board):
    """Returns the utility of a terminal state."""
    winner = get_winner(board)
    if winner == PLAYER_X:
        return 1
    elif winner == PLAYER_O:
        return -1
    return 0

def get_actions(board):
    """Returns a list of possible moves."""
    actions = []
    for i in range(3):
        for j in range(3):
```

```
            if board[i][j] == EMPTY:
                actions.append((i, j))
    return actions

def result(board, action, player):
    """Returns the board resulting from applying an action."""
    new_board = deepcopy(board)
    new_board[action[0]][action[1]] = player
    return new_board

# Alpha-Beta Search
def alpha_beta_search(board):
    """Performs Alpha-Beta Pruning to find the best action."""
    alpha = -math.inf
    beta = math.inf
    best_action = None

    def max_value(state, alpha, beta):
        if is_terminal(state):
            return utility(state)
        v = -math.inf
        for action in get_actions(state):
            v = max(v, min_value(result(state, action, PLAYER_X),
alpha, beta))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta):
        if is_terminal(state):
            return utility(state)
        v = math.inf
        for action in get_actions(state):
            v = min(v, max_value(result(state, action, PLAYER_O),
alpha, beta))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    for action in get_actions(board):
        value = min_value(result(board, action, PLAYER_X), alpha, beta)
        if value > alpha:
            alpha = value
            best_action = action

    return best_action
```

```
# Game loop
def print_board(board):
    """Displays the board."""
    for row in board:
        print(" | ".join(row))
    print()

def play_game():
    """Runs the Tic-Tac-Toe game with user input."""
    board = [[EMPTY for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic-Tac-Toe!")
    print("You are 'O', and the computer is 'X'.")
    print_board(board)

    while not is_terminal(board):
        # User's turn
        user_move = None
        while user_move not in get_actions(board):
            try:
                print("Your turn! Enter your move as 'row col' (e.g.,
'1 2'):")
                row, col = map(int, input().split())
                user_move = (row - 1, col - 1)  # Convert to 0-based
index
                if user_move not in get_actions(board):
                    print("Invalid move! Try again.")
            except ValueError:
                print("Invalid input! Please enter two numbers
separated by a space.")

        board = result(board, user_move, PLAYER_O)
        print("You played:")
        print_board(board)

        if is_terminal(board):
            break

        # Computer's turn
        print("Computer's turn...")
        computer_move = alpha_beta_search(board)
        board = result(board, computer_move, PLAYER_X)
        print("Computer played:")
        print_board(board)

    # Game over
    winner = get_winner(board)
    if winner == PLAYER_X:
```

```
        print("Computer wins!")
    elif winner == PLAYER_O:
        print("Congratulations! You win!")
    else:
        print("It's a draw!")

# Run the game
if __name__ == "__main__":
    play_game()
```

ouput:-

```
Welcome to Tic-Tac-Toe!
You are 'O', and the computer is 'X'.
- | - | -
- | - | -
- | - | -

Your turn! Enter your move as 'row col' (e.g., '1 2'):
2 2
You played:
- | - | -
- | O | -
- | - | -

Computer's turn...
Computer played:
X | - | -
- | O | -
- | - | -

Your turn! Enter your move as 'row col' (e.g., '1 2'):
33
Invalid input! Please enter two numbers separated by a space.
Your turn! Enter your move as 'row col' (e.g., '1 2'):
3 3
You played:
X | - | -
- | O | -
- | - | O

Computer's turn...
Computer played:
X | - | X
- | O | -
- | - | O
```

```
Your turn! Enter your move as 'row col' (e.g., '1 2'):
1 2
You played:
X | O | X
- | O | -
- | - | O

Computer's turn...
Computer played:
X | O | X
- | O | -
- | X | O

Your turn! Enter your move as 'row col' (e.g., '1 2'):
2 1
You played:
X | O | X
O | O | -
- | X | O

Computer's turn...
Computer played:
X | O | X
O | O | X
- | X | O

Your turn! Enter your move as 'row col' (e.g., '1 2'):
3 1
You played:
X | O | X
O | O | X
O | X | O

It's a draw!
```

code for 8 queens

```
import math

def is_terminal(state, n):
    """Check if the board is a valid solution (no conflicts, all queens
placed)."""
    return len(state) == n

def count_conflicts(state):
    """Calculate the number of conflicts between queens."""
    conflicts = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            # Same column or diagonal conflicts
            if state[i] == state[j] or abs(state[i] - state[j]) ==
abs(i - j):
```

```
                conflicts += 1
    return conflicts

def utility(state):
    """Return a utility score based on the number of conflicts (higher
is better)."""
    return -count_conflicts(state)

def actions(state, n):
    """Generate possible next moves."""
    next_row = len(state)
    if next_row >= n:
        return []
    return [state + [col] for col in range(n)]

def max_value(state, alpha, beta, n):
    """Maximizing function."""
    if is_terminal(state, n):
        return utility(state)
    v = -math.inf
    for action in actions(state, n):
        v = max(v, min_value(action, alpha, beta, n))
        if v >= beta:
            return v
        alpha = max(alpha, v)
    return v

def min_value(state, alpha, beta, n):
    """Minimizing function."""
    if is_terminal(state, n):
        return utility(state)
    v = math.inf
    for action in actions(state, n):
        v = min(v, max_value(action, alpha, beta, n))
        if v <= alpha:
            return v
        beta = min(beta, v)
    return v

def alpha_beta_search(n):
    """Perform Alpha-Beta pruning to solve the N-Queens problem."""
    alpha = -math.inf
    beta = math.inf
    best_action = None
    initial_state = []

    for action in actions(initial_state, n):
        value = min_value(action, alpha, beta, n)
```

```
        if value > alpha:
            alpha = value
            best_action = action

    return best_action

# Example usage
if __name__ == "__main__":
    n = 8  # Number of queens
    solution = alpha_beta_search(n)
    if solution:
        print("Solution:", solution)
    else:
        print("No solution found.")
```

output:-

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

Observation book :-

DATE: 03/12/24 PAGE: 32

LAB-10 :

* Solve using Forward chaining :-

Consider the following problem : As per the law, it is a crime for an American to sell weapons to hostile nations country A, an enemy of America, has some missiles and all the missiles were sold to it by Robert, who is an American citizen."
Prove that "Robert is criminal".

→ let's say a, b and x variables

American(a) ∧ weapon(b) ∧ sells(a, b, c) ∧ Hostile(x) → criminal(p)

∃ x (own(A, x) ∧ Missile(x))

∀ x (Missile(x) ∧ owns(A, x)) ⇒ sells(Robert, x, A)

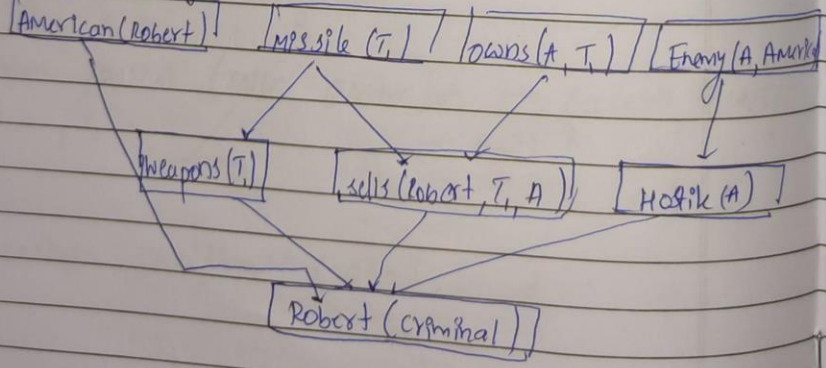
Missel(b) → weapon(b)

∀ x (Enemy(x, America) ⇒ Hostile(x))

American(Robert) (Robert is an American)

Enemy(A, America).

* Forword chaining proof :

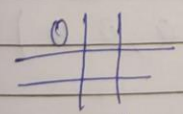


DATE: 33 PAGE:

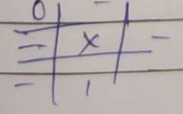
2. Alpha - beta search algorithm :-

function Alpha - beta - search (state) returns an action
  $v \leftarrow$ Max. value (state, $-\infty, +\infty$)
  return the action in Actions (state) with value v
function Max_value (state, α, β) returns a utility value
  if Terminal - Test (state) then return Utility (state)
  $v \leftarrow -\infty$
  for each α in Actions (state) do
    $v \leftarrow$ MAX (v, Min_value (Result (s, a), α, β))
    if $v \geq \beta$ then return v
    $\alpha \leftarrow$ Max (α, v)
    return v
function Min_value (state, α, β) returns a utility value
  if Terminal - Test (state) then return Utility (state)
  $v \leftarrow +\infty$
  for each α in Actions (state) do
    $v \leftarrow$ Min (v, MAX_value (Result (s, a), α, β))
    if $v \leq \alpha$ then return v
    $\beta \leftarrow$ Min (β, v)
    return v

* Output :
user move (1,1)

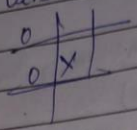


Computer's move

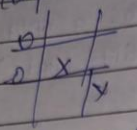


DATE: 24 PAGE:

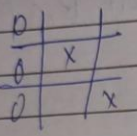
* users move (2,1)



* computer's move (3,3)



* user's move (3,1)



computer wins.

* Alpha beta pruning for 8 queens :

* main function :—

function Alpha - Beta - search (n) returns a solution

$\alpha \leftarrow -\infty$

$\beta \leftarrow +\infty$

best_action ← none

initial_state ← [ ]

for each action in Actions (initial_state)

value ← min_value (action, α, β, n)

if value > α then

$\alpha \leftarrow$ value

best_action ← action

return best_action

MAX value Function :—

Function MAX val (state, α, β, n) returns utility

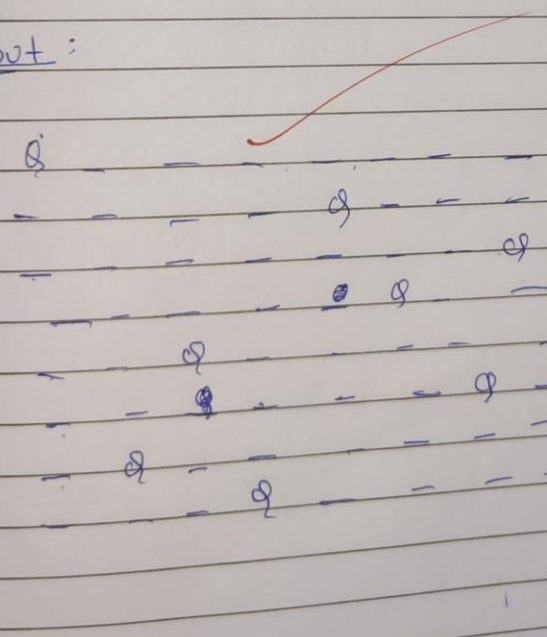
DATE: 3/5 PAGE:

if terminal test (state, n) then
return utility (state)
v ← −∞
for each action in Actions (state, n) do
v ← MAX (v, Min_value (action, α, β, n))
if v ≥ β then
return v
α ← MAX (α, v)
return n

MIN - value

function MIN value (state, α, β, n) returns a value.
if terminal test (state, n) do
v ← Min (v, MXMVS, action) α, β, n)
if v ≤ α then
return v
β > min (β, v)
return v

* output :



3/12/24