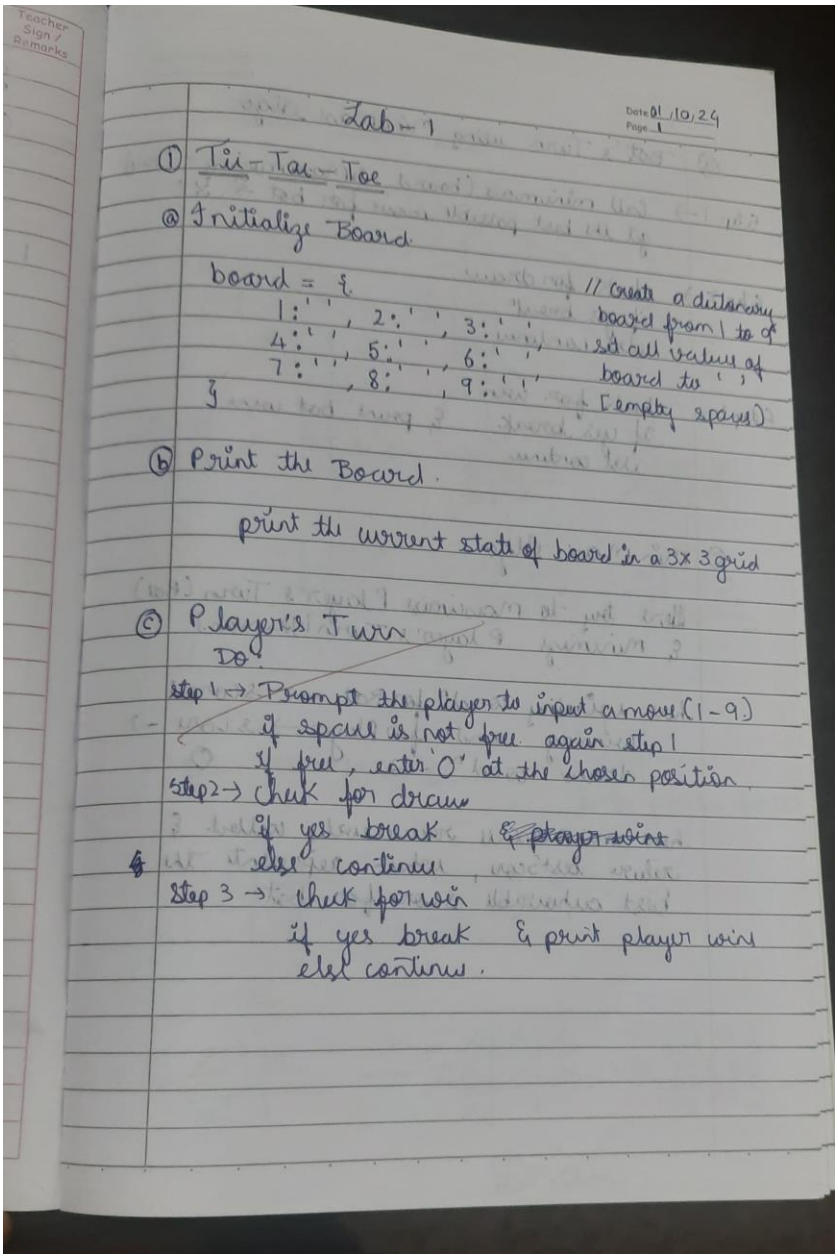
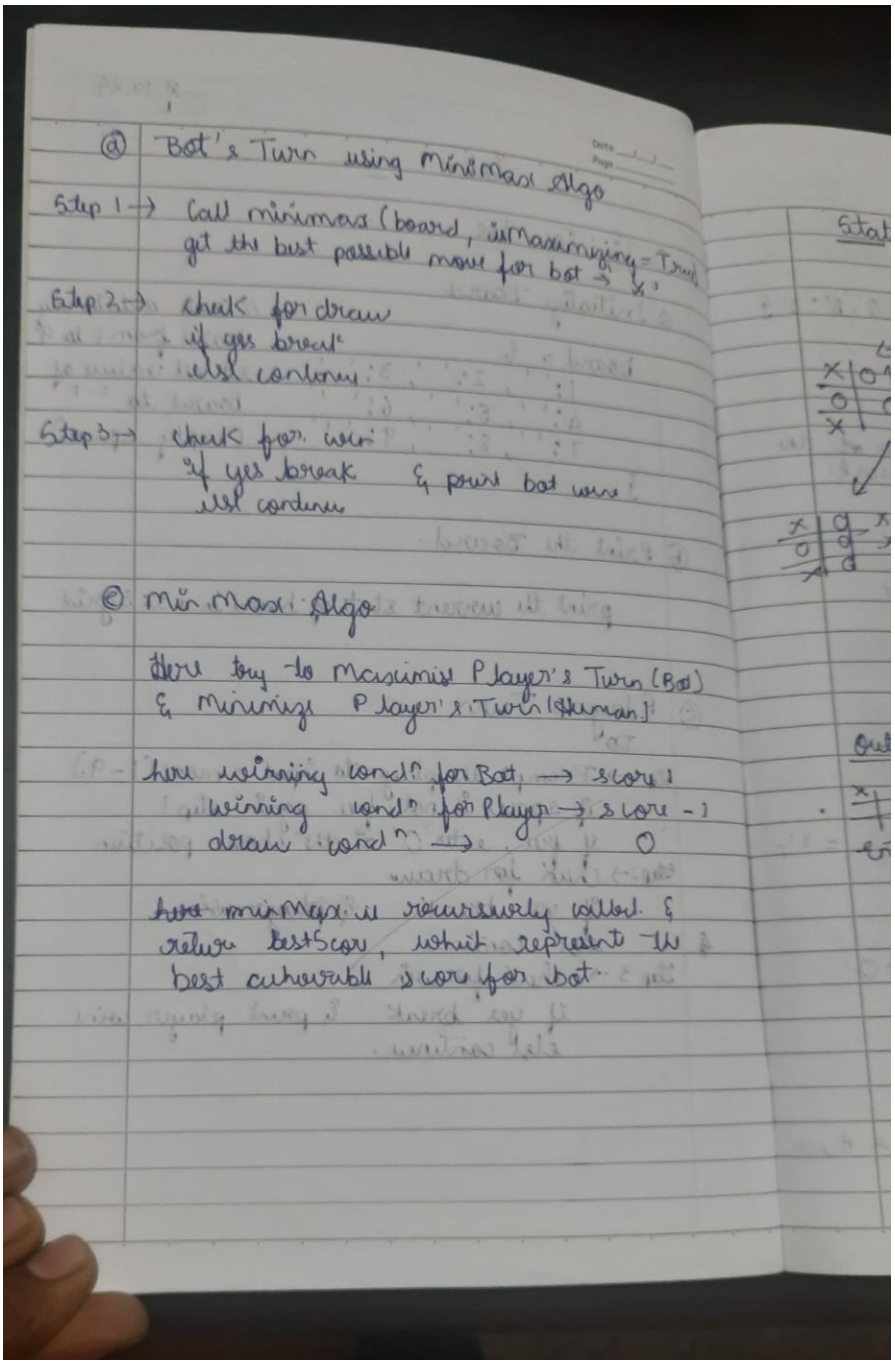
AI LAB REPORT

Deekshith B - 1BM22CS082

1. Implement Tic –Tac –Toe Game.

ALGORITHM





@ Bot's Turn using MiniMax Algo

Step 1 → Call minimax (board, isMaximizing = True)
get the best possible move for bot → 'x'

Step 2 → check for draw
if yes break
else continue

Step 3 → check for win
if yes break & print bot wins
else continue

© MiniMax Algo

Here try to Maximise Player's Turn (Bot)
& Minimize Player's Turn (Human)

here winning cond^n for Bot → score 1
winning cond^n for Player → score -1
draw cond^n → 0

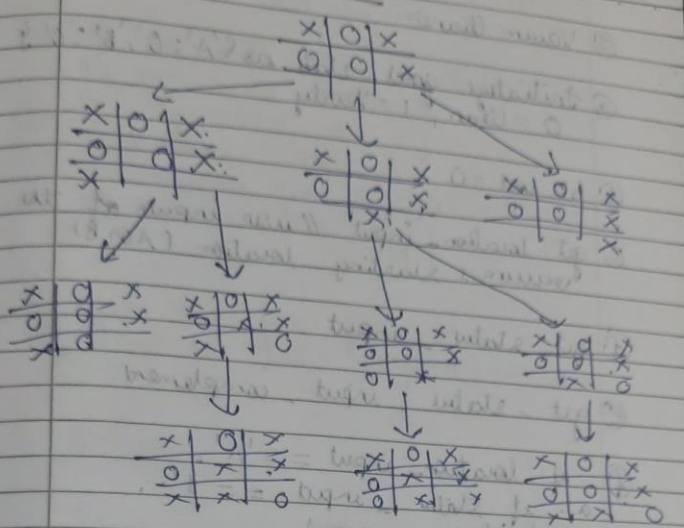
here minMax is recursively called & return bestScore, which represent the best achievable score for bot.

Stat

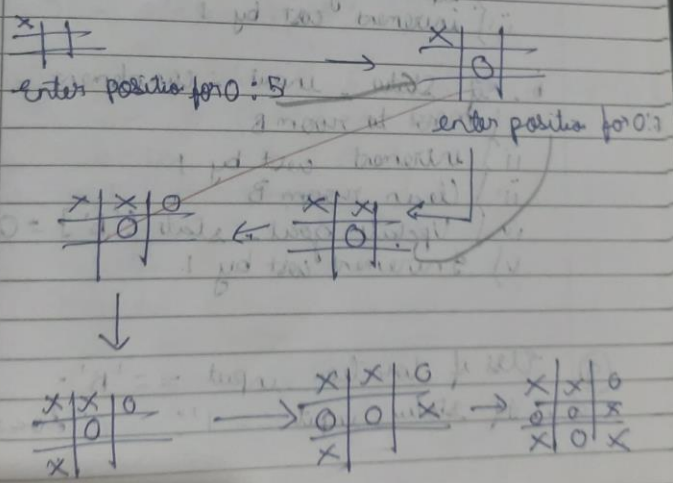
Out

.

State Space Tree



Output



Draw.

CODE:

```
board = {
    1: ' ', 2: ' ', 3: ' ',
    4: ' ', 5: ' ', 6: ' ',
    7: ' ', 8: ' ', 9: ' '
}

gameOver = False

def printBoard(board):
    print(board[1] + '|' + board[2] + '|' + board[3])
    print('-+-+-')
    print(board[4] + '|' + board[5] + '|' + board[6])
    print('-+-+-')
    print(board[7] + '|' + board[8] + '|' + board[9])
    print('\n')

def spaceFree(pos):
    return board[pos] == ' '

def checkWin():
    if (board[1] == board[2] and board[1] == board[3] and board[1] != ' '):
        return True
    elif (board[4] == board[5] and board[4] == board[6] and board[4] != ' '):
        return True
    elif (board[7] == board[8] and board[7] == board[9] and board[7] != ' '):
        return True
    elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
        return True
    elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
        return True
    elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
        return True
    elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
        return True
```

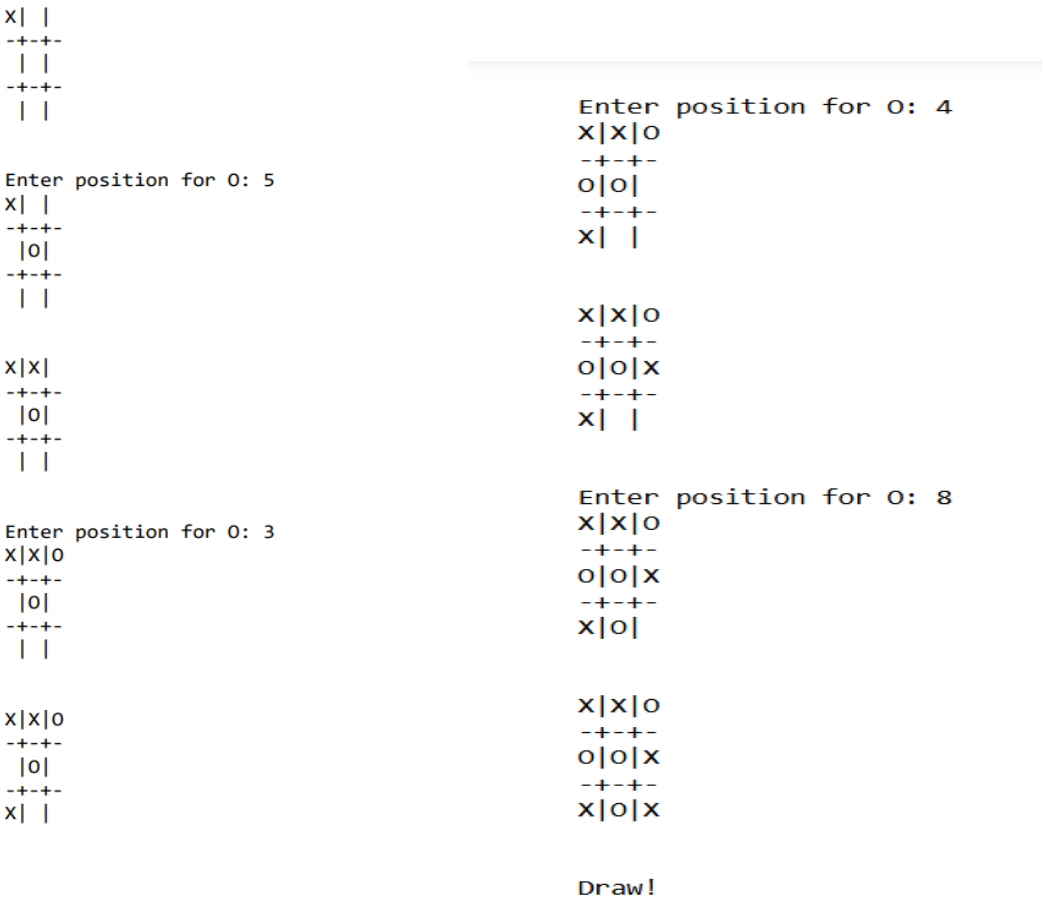
```
    elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
        return True
    else:
        return False

def checkMoveForWin(move):
    if (board[1] == board[2] and board[1] == board[3] and board[1] == move):
        return True
    elif (board[4] == board[5] and board[4] == board[6] and board[4] == move):
        return True
    elif (board[7] == board[8] and board[7] == board[9] and board[7] == move):
        return True
    elif (board[1] == board[5] and board[1] == board[9] and board[1] == move):
        return True
    elif (board[3] == board[5] and board[3] == board[7] and board[3] == move):
        return True
    elif (board[1] == board[4] and board[1] == board[7] and board[1] == move):
        return True
    elif (board[2] == board[5] and board[2] == board[8] and board[2] == move):
        return True
    elif (board[3] == board[6] and board[3] == board[9] and board[3] == move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if board[key] == ' ':
            return False
    return True

def insertLetter(letter, position):
    global gameOver

    if spaceFree(position):
```

```
        board[position] = letter
        printBoard(board)

        if checkWin():
            gameOver = True
            if letter == 'X':
                print('Bot wins!')
            else:
                print('You win!')
        elif checkDraw():
            gameOver = True
            print('Draw!')
        return
    else:
        if not gameOver:
            print('Position taken, please pick a different position.')
            position = int(input('Enter new position: '))
            insertLetter(letter, position)
        return

player = 'O'
bot = 'X'

def playerMove():
    if not gameOver:
        position = int(input('Enter position for O: '))
        insertLetter(player, position)
    return

def compMove():
    if not gameOver:
        bestScore = -1000
        bestMove = 0
        for key in board.keys():
            if board[key] == ' ':
```

```
            board[key] = bot
            score = minimax(board, False)
            board[key] = ' '
            if score > bestScore:
                bestScore = score
                bestMove = key

    insertLetter(bot, bestMove)
    return

def minimax(board, isMaximizing):
    if checkMoveForWin(bot):
        return 1
    elif checkMoveForWin(player):
        return -1
    elif checkDraw():
        return 0

    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                if score > bestScore:
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
```

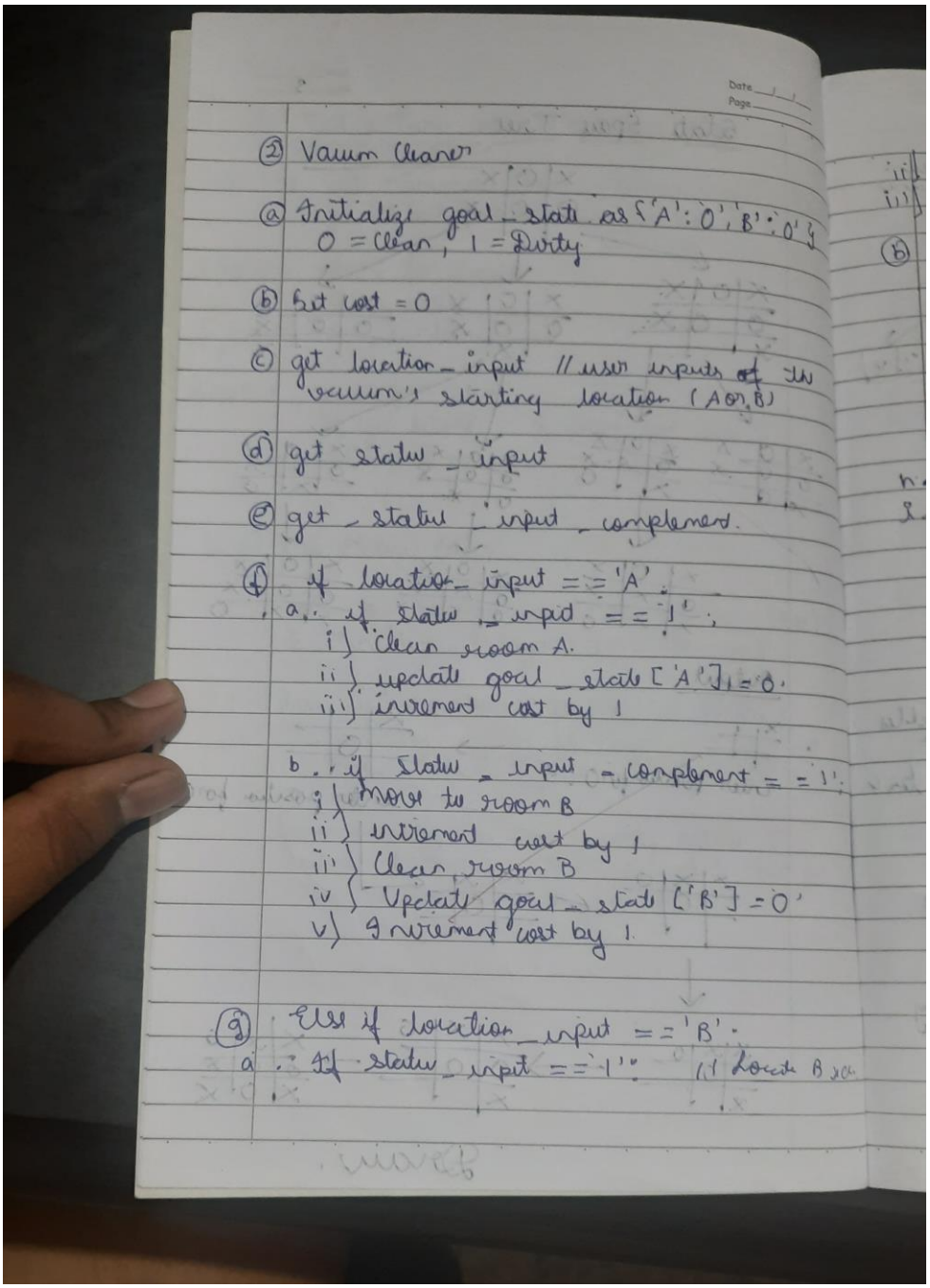
```
            if score < bestScore:
                bestScore = score
        return bestScore


while not gameOver:
    compMove()
    playerMove()
```

OUTPUT:



2.Implement Vacuum Cleaner

ALGORITHM:



ii) Update goal - state['B'] = '0'
iii) Increment cost by 1

(b) if status_input_complement == '1'
i) move to room A
ii) Increment cost by 1
iii) Clean room A
iv) Update goal_state['A'] = '0'
v) Increment cost by 1

n. Print the final goal_state
i. Print the total performance cost

Output:

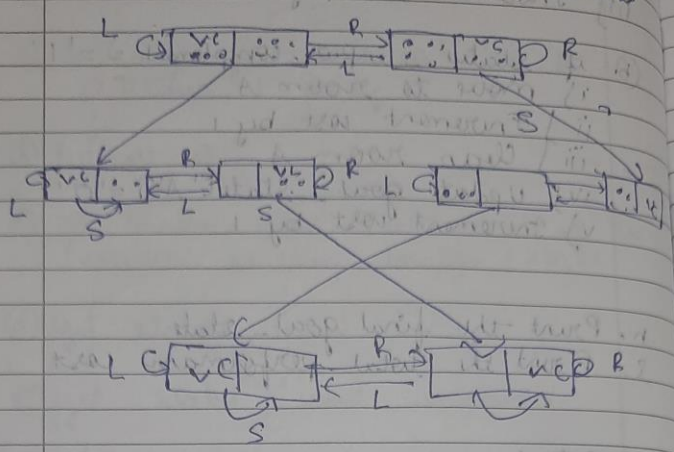
Enter location of Vacuum (A (B)): A
Enter status of A: 1
Enter status of B: 1

Initial condition of A: 1 & A': 0': B': 0'
Vacuum is placed in location A.
Location A is Dirty
cost for cleaning A: 1
Location A has been cleaned.
Location B is Dirty
Moving right to location B
cost for moving right: 2
cost for suck B: 3
location B has been cleaned.
Goal state: {'A': 0': B': 0'}
Performance measurement: 3

Date / /
Page

State space

vc → Vaum cleaner



Evaluation Strategy

① Completeness :- Implementation handles all scenario for cleaning both room based on their & correctly

② Time complexity :- O(1) as op^ are constant regardless of input size

③ Space complexity :- O(1) for storing of room & for & independent i/p size

CODE:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input + " (0 for Clean, 1 for
Dirty): ")
    status_input_complement = input("Enter status of other room: ")

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':

        print("Vacuum is placed in Location A")

        if status_input == '1':
            print("Location A is Dirty.")

            goal_state['A'] = '0'
            cost += 1
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to the Location B.")
                cost += 1
                print("COST for moving RIGHT: " + str(cost))

                goal_state['B'] = '0'
                cost += 1
```

```
            print("COST for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean.")

        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B.")
            cost += 1
            print("COST for moving RIGHT: " + str(cost))

            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location B is already clean.")

else:
    print("Vacuum is placed in Location B")


    if status_input == '1':
        print("Location B is Dirty.")

        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING: " + str(cost))
        print("Location B has been Cleaned.")
```

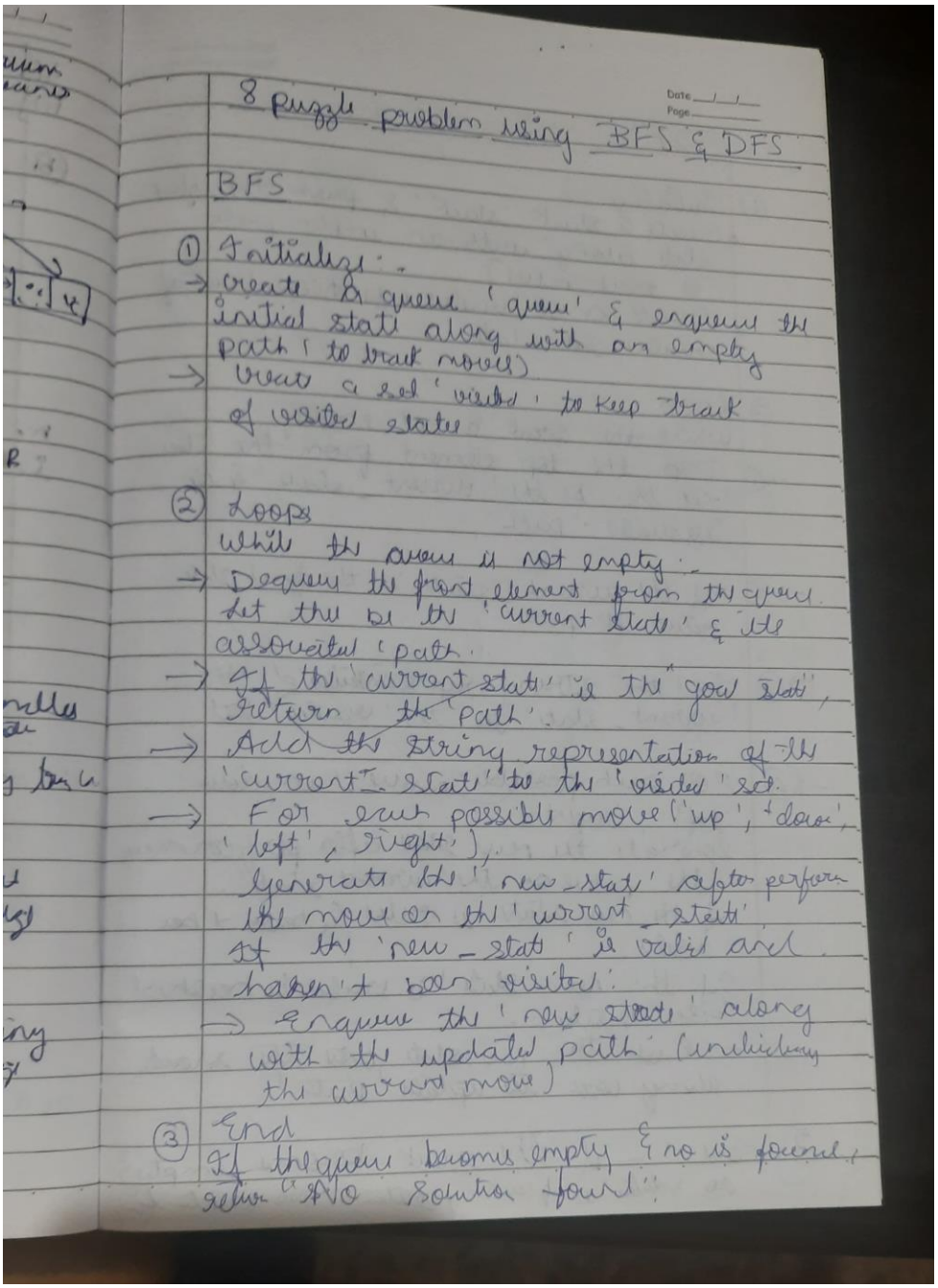
```
            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A.")
                cost += 1
                print("COST for moving LEFT: " + str(cost))

                goal_state['A'] = '0'
                cost += 1
                print("COST for SUCK: " + str(cost))
                print("Location A has been Cleaned.")
            else:
                print("Location A is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A.")
            cost += 1
            print("COST for moving LEFT: " + str(cost))

            goal_state['A'] = '0'
            cost += 1
            print("Cost for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location A is already clean.")


    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))


vacuum_world()
```

OUTPUT:

```
Enter Location of Vacuum: A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of other room: 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A: 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT: 2
COST for SUCK: 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

3.Solve 8 puzzle problems using BFS

ALGORITHM:



8 Puzzle problem using BFS & DFS

BFS

① Initialize :-
→ Create a queue 'queue' & enqueue the initial state along with an empty path (to track moves)
→ Create a set 'visited' to keep track of visited states

② Loops
While the queue is not empty :-
→ Dequeue the front element from the queue. Let the be the 'current state' & its associated 'path'.
→ If the 'current state' is the 'goal state', return the 'path'.
→ Add the string representation of the 'current_state' to the 'visited' set.
→ For each possible move ('up', 'down', 'left', 'right'),
Generate the 'new_state' after performing the move on the 'current_state'
If the 'new_state' is valid and hasn't been visited :
→ Enqueue the 'new state' along with the updated path (including the current move)

③ End
If the queue becomes empty & no is found, return "No Solution found".

CODE:

```
from collections import deque

class Node:
    def __init__(self, puzzle, x, y, parent=None):
        self.puzzle = [row[:] for row in puzzle]
        self.x = x
        self.y = y
        self.parent = parent

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

dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

def is_goal(puzzle):
    return puzzle == goal

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

def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def bfs(root):
    queue = deque([root])
    visited = set()
```

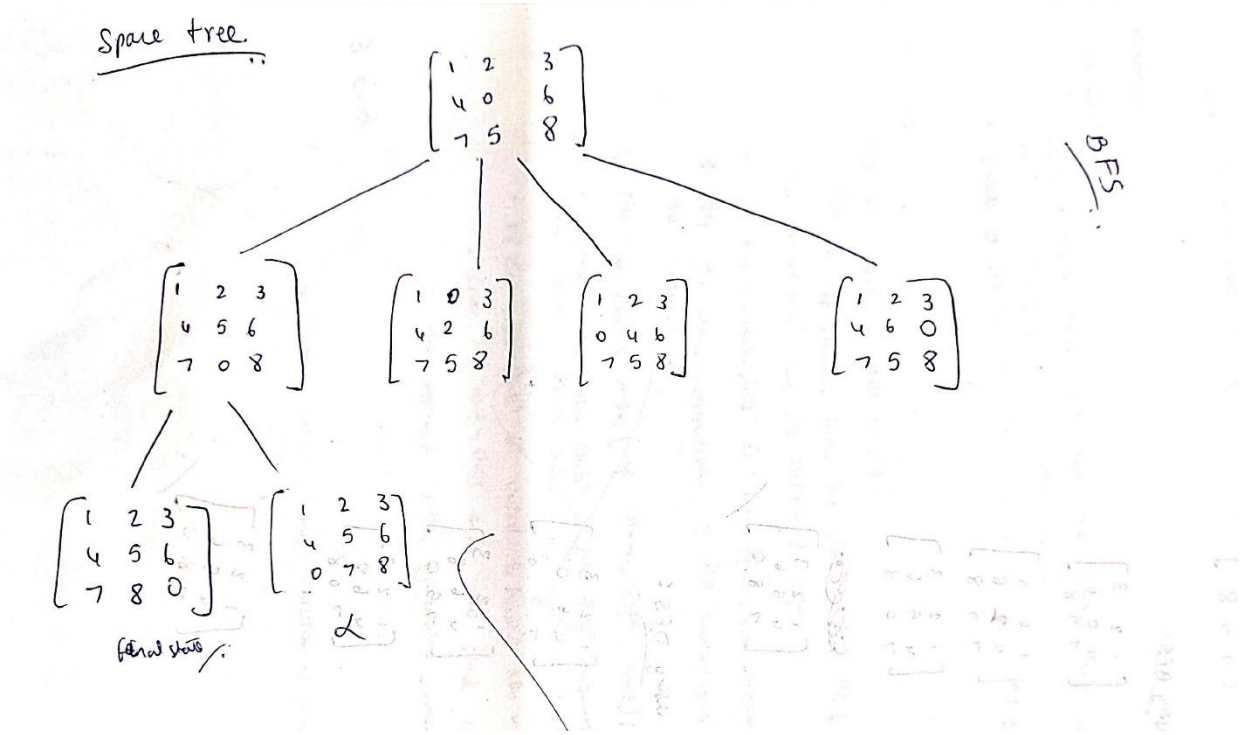
```
    while queue:
        node = queue.popleft()

        if is_goal(node.puzzle):
            print("Solution found:")
            path = []
            while node:
                path.append(node.puzzle)
                node = node.parent
            for state in reversed(path):
                print_puzzle(state)
            return True

        puzzle_tuple = tuple(map(tuple, node.puzzle))
        if puzzle_tuple in visited:
            continue
        visited.add(puzzle_tuple)

        for i in range(4):
            new_x = node.x + dx[i]
            new_y = node.y + dy[i]

            if is_valid(new_x, new_y):
                new_puzzle = [row[:] for row in node.puzzle]
                new_puzzle[node.x][node.y], new_puzzle[new_x][new_y] =
new_puzzle[new_x][new_y], new_puzzle[node.x][node.y]

                new_puzzle_tuple = tuple(map(tuple, new_puzzle))
                if new_puzzle_tuple not in visited:
                    new_node = Node(new_puzzle, new_x, new_y, node)
                    queue.append(new_node)

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

```
def main():
    initial_puzzle = [
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ]

    print("Initial Puzzle:")
    print_puzzle(initial_puzzle)

    x, y = [(i, j) for i in range(3) for j in range(3) if initial_puzzle[i][j] == 0][0]

    root = Node(initial_puzzle, x, y)

    if not bfs(root):
        print("Failed to find a solution.")

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

OUTPUT:

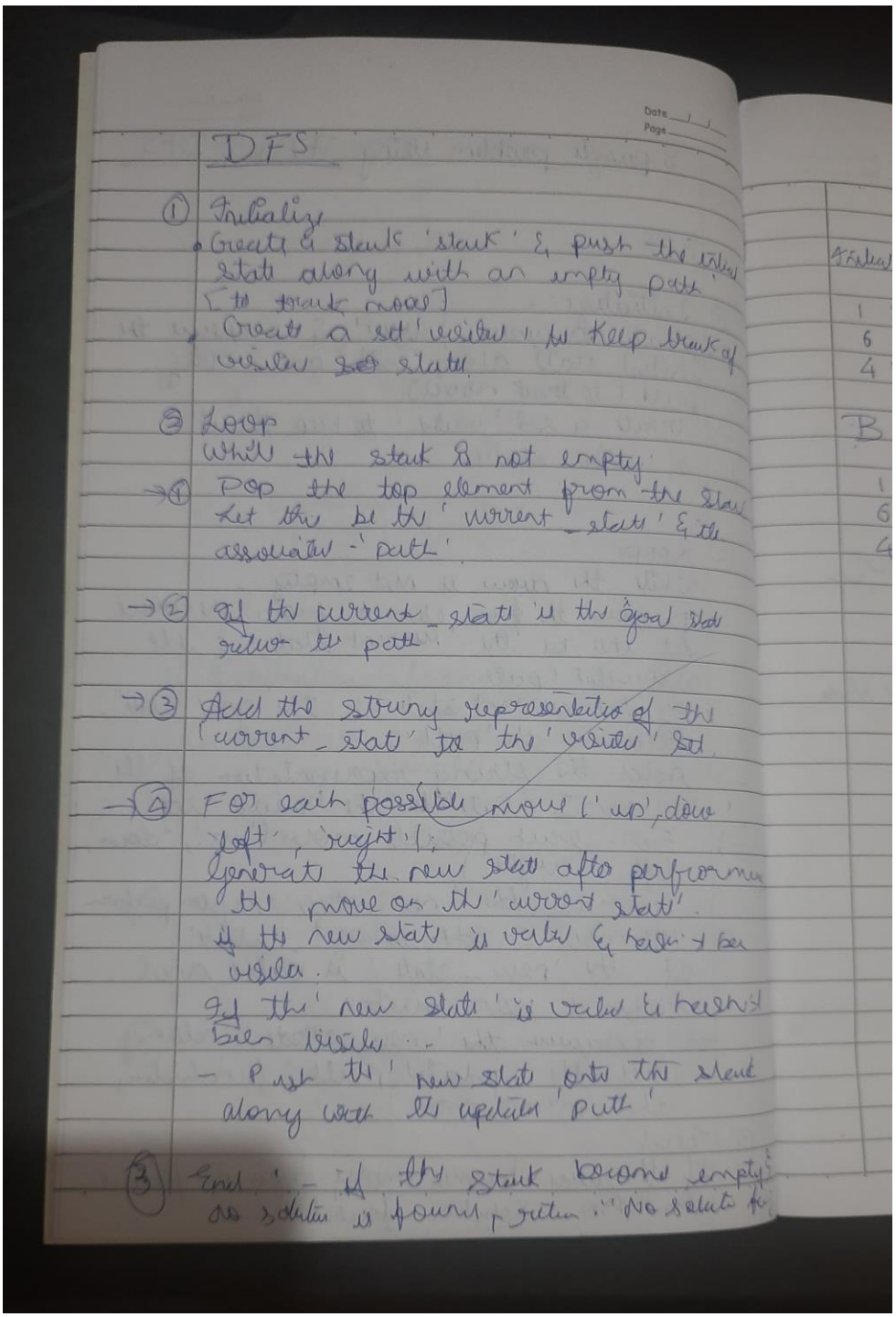
```
Initial Puzzle:
1 2 3
4   5
7 8 6

Solution found:
1 2 3
4   5
7 8 6

1 2 3
4 5
7 8 6

1 2 3
4 5 6
7 8
```

STATE-SPACE TREE:



4.Solve 8 puzzle problems using DFS

ALGORITHM:



CODE:

```
class Node:
    def __init__(self, puzzle, x, y, parent=None):
        self.puzzle = [row[:] for row in puzzle]
        self.x = x
        self.y = y
        self.parent = parent

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

dx = [-1, 1, 0, 0]
dy = [0, 0, -1, 1]

def is_goal(puzzle):
    return puzzle == goal

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

def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def count_inversions(puzzle):
    flat_puzzle = [num for row in puzzle for num in row if num != 0]
    inversions = 0
```

```
    for i in range(len(flat_puzzle)):
        for j in range(i + 1, len(flat_puzzle)):
            if flat_puzzle[i] > flat_puzzle[j]:
                inversions += 1
    return inversions

def is_solvable(puzzle):
    return count_inversions(puzzle) % 2 == 0

def dfs(root):
    stack = [root]
    visited = set()

    while stack:
        node = stack.pop()

        if is_goal(node.puzzle):
            print("Solution found:")
            path = []
            while node:
                path.append(node.puzzle)
                node = node.parent
            for state in reversed(path):
                print_puzzle(state)
            return True

        puzzle_tuple = tuple(map(tuple, node.puzzle))
        if puzzle_tuple in visited:
            continue
        visited.add(puzzle_tuple)

        for i in range(4):
            new_x = node.x + dx[i]
            new_y = node.y + dy[i]
```

```
            if is_valid(new_x, new_y):
                new_puzzle = [row[:] for row in node.puzzle]
                new_puzzle[node.x][node.y], new_puzzle[new_x][new_y] =
new_puzzle[new_x][new_y], new_puzzle[node.x][node.y]

                new_puzzle_tuple = tuple(map(tuple, new_puzzle))
                if new_puzzle_tuple not in visited:
                    new_node = Node(new_puzzle, new_x, new_y, node)
                    stack.append(new_node)

    print("No solution found.")
    return False

def main():
    initial_puzzle = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 0, 8]
    ]

    print("Initial Puzzle:")
    print_puzzle(initial_puzzle)

    if not is_solvable(initial_puzzle):
        print("The provided puzzle is unsolvable.")
        return

    try:
        x, y = [(i, j) for i in range(3) for j in range(3) if initial_puzzle[i][j] == 0][0]
    except IndexError:
        print("Invalid puzzle: No blank space (0) found.")
        return

    root = Node(initial_puzzle, x, y)
```

```
    if not dfs(root):
        print("Failed to find a solution.")

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

OUTPUT:

```
Initial Puzzle:
1 2 3
4 5 6
7   8

Solution found:
1 2 3
4 5 6
7   8

1 2 3
4 5 6
7 8
```