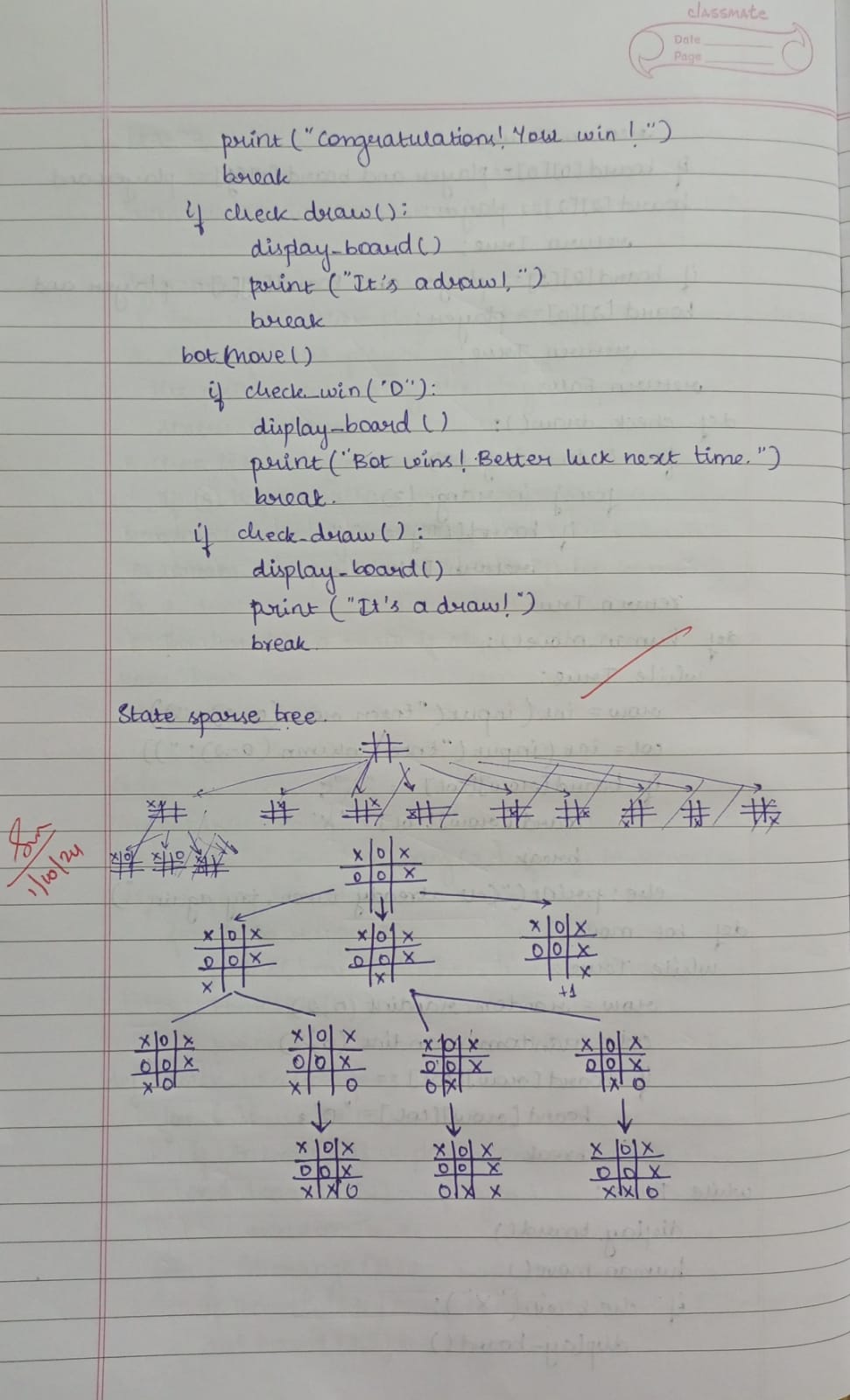
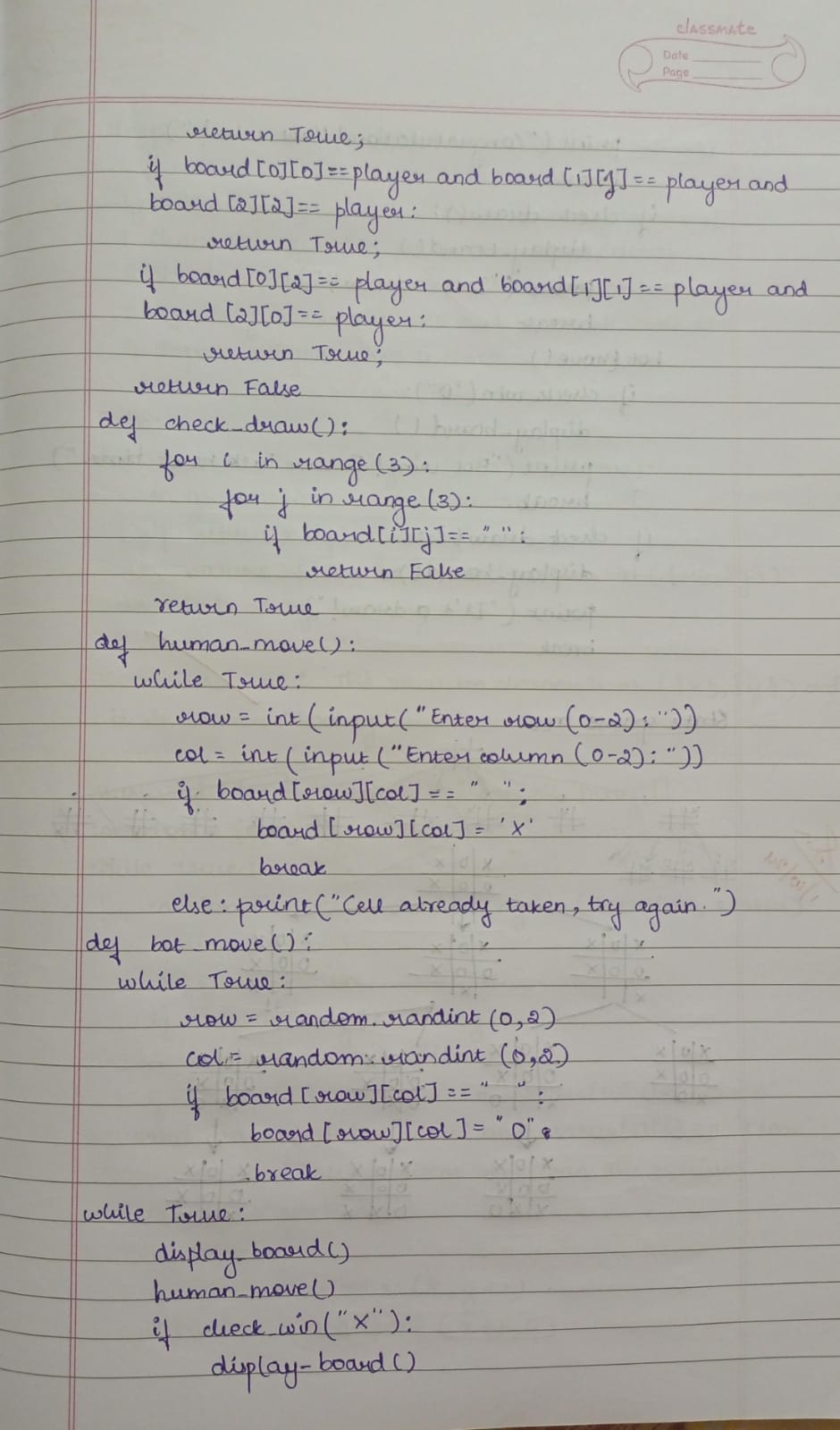
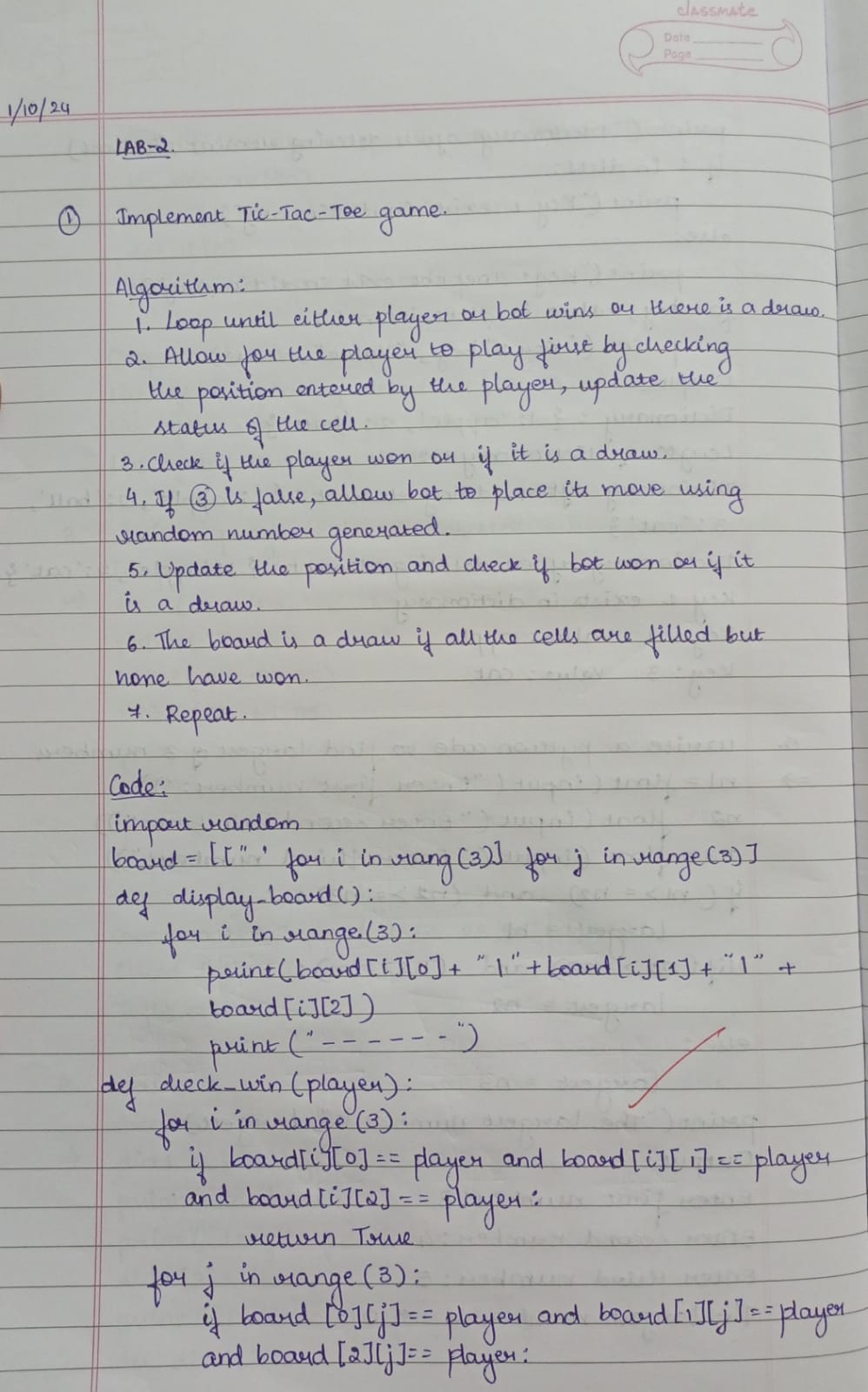
**AI LAB REPORT**

1. **Implement Tic-Tac-Toe game.**

**Algorithm:**

****

**Code:**

import random

board = [[" " for i in range(3)] for j in range(3)]

def display\_board():

for i in range(3):

print(board[i][0] + " | " + board[i][1] + " | " + board[i][2])

print("---------")

def check\_win(player):

for i in range(3):

if board[i][0] == player and board[i][1] == player and board[i][2] == player:

return True

for j in range(3):

if board[0][j] == player and board[1][j] == player and board[2][j] == player:

return True

if board[0][0] == player and board[1][1] == player and board[2][2] == player:

return True

if board[0][2] == player and board[1][1] == player and board[2][0] == player:

return True

return False

def check\_draw():

for i in range(3):

for j in range(3):

if board[i][j] == " ":

return False

return True

def human\_move():

while True:

row = int(input("Enter row (0-2): "))

col = int(input("Enter column (0-2): "))

if board[row][col] == " ":

board[row][col] = "X"

break

else:

print("Cell already taken, try again.")

def bot\_move():

while True:

row = random.randint(0, 2)

col = random.randint(0, 2)

if board[row][col] == " ":

board[row][col] = "O"

break

while True:

display\_board()

human\_move()

if check\_win("X"):

display\_board()

print("Congratulations! You win!")

break

if check\_draw():

display\_board()

print("It's a draw!")

break

bot\_move()

if check\_win("O"):

display\_board()

print("Bot wins! Better luck next time.")

break

if check\_draw():

display\_board()

print("It's a draw!")

break

**Output:**

**| |**

**---------**

**| |**

**---------**

**| |**

**---------**

**Enter row (0-2): 0**

**Enter column (0-2): 0**

**X | |**

**---------**

**| |**

**---------**

**| O |**

**---------**

**Enter row (0-2): 0**

**Enter column (0-2): 2**

**X | O | X**

**---------**

**| |**

**---------**

**| O |**

**---------**

**Enter row (0-2): 2**

**Enter column (0-2): 0**

**X | O | X**

**---------**

**| | O**

**---------**

**X | O |**

**---------**

**Enter row (0-2): 1**

**Enter column (0-2): 2**

**Cell already taken, try again.**

**Enter row (0-2): 1**

**Enter column (0-2): 0**

**X | O | X**

**---------**

**X | | O**

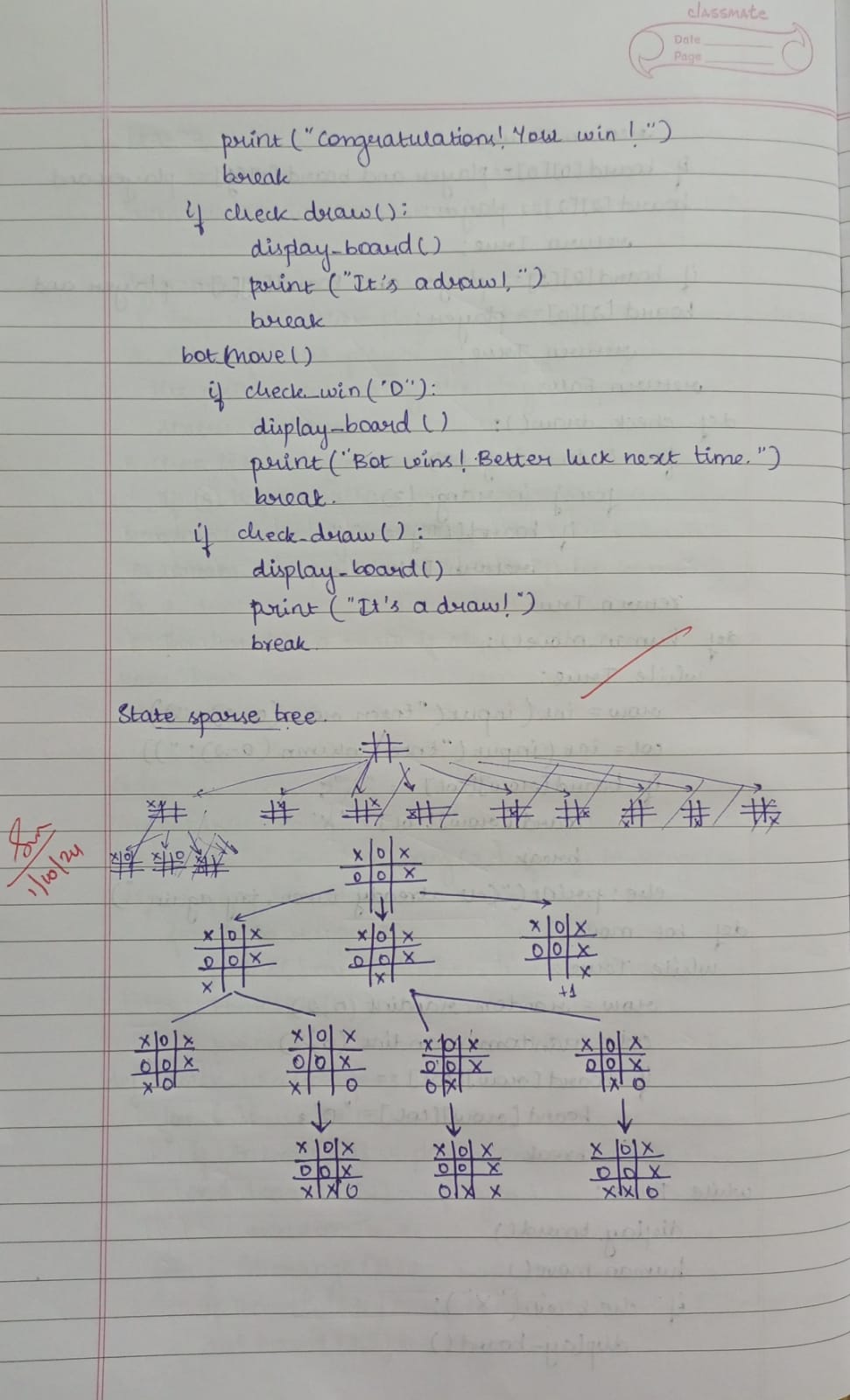
**---------**

**X | O |**

**---------**

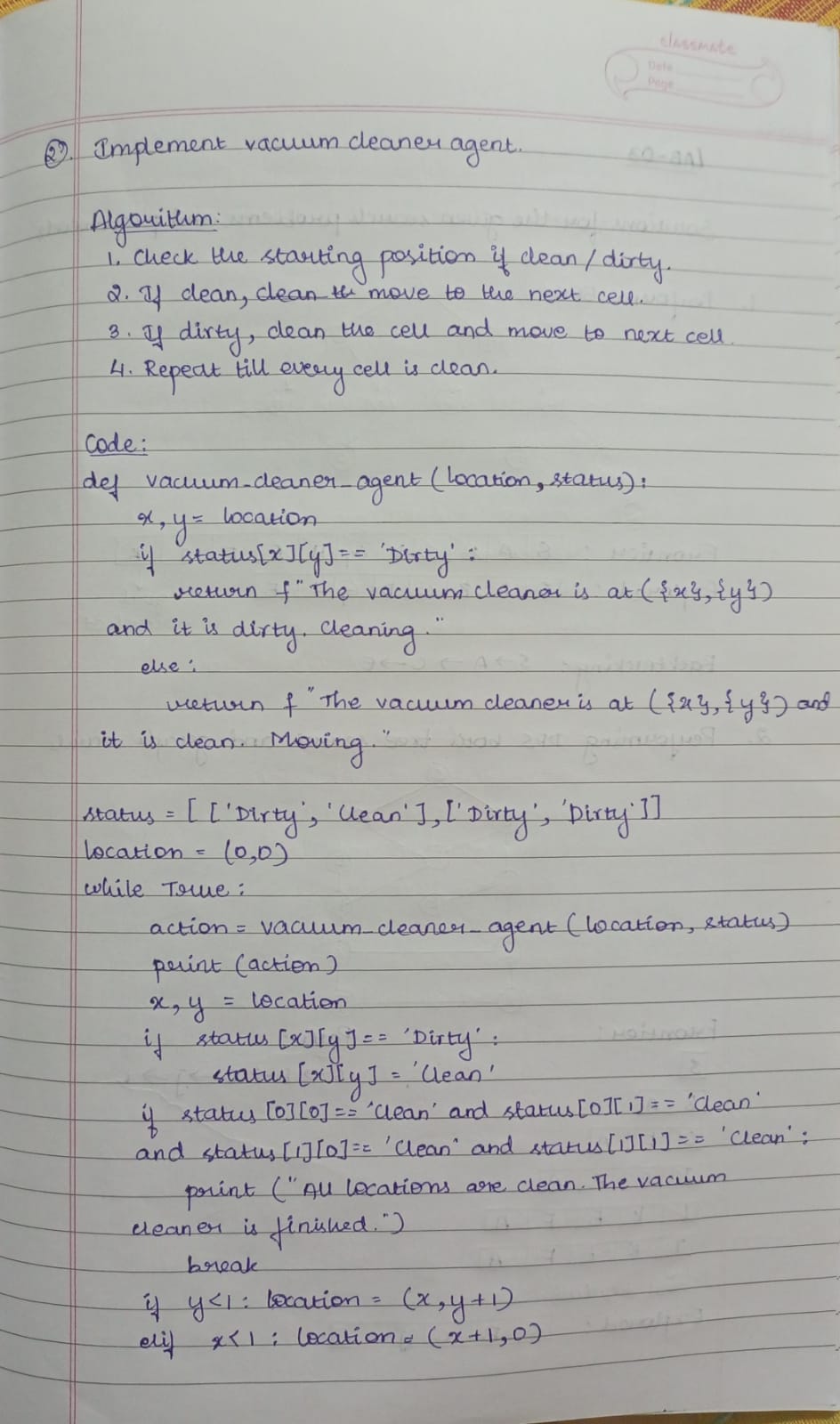
**Congratulations! You win!**

**State Space Tree:**

****

1. **Implement a vacuum cleaner agent..**

**Algorithm:**

****

**Code:**

def vacuum\_cleaner\_agent(location, status):

x, y = location

if status[x][y] == 'Dirty':

return f"The vacuum cleaner is at ({x}, {y}) and it is dirty. Cleaning."

else:

return f"The vacuum cleaner is at ({x}, {y}) and it is clean. Moving."

status = [['Dirty', 'Clean'], ['Dirty', 'Dirty']]

location = (0, 0)

while True:

action = vacuum\_cleaner\_agent(location, status)

print(action)

x, y = location

if status[x][y] == 'Dirty':

status[x][y] = 'Clean'

if status[0][0] == 'Clean' and status[0][1] == 'Clean' and status[1][0] == 'Clean' and status[1][1] == 'Clean':

print("All locations are clean. The vacuum cleaner is finished.")

break

if y < 1:

location = (x, y + 1)

elif x < 1:

location = (x + 1, 0)

**Output:**

**The vacuum cleaner is at (0, 0) and it is dirty. Cleaning.**

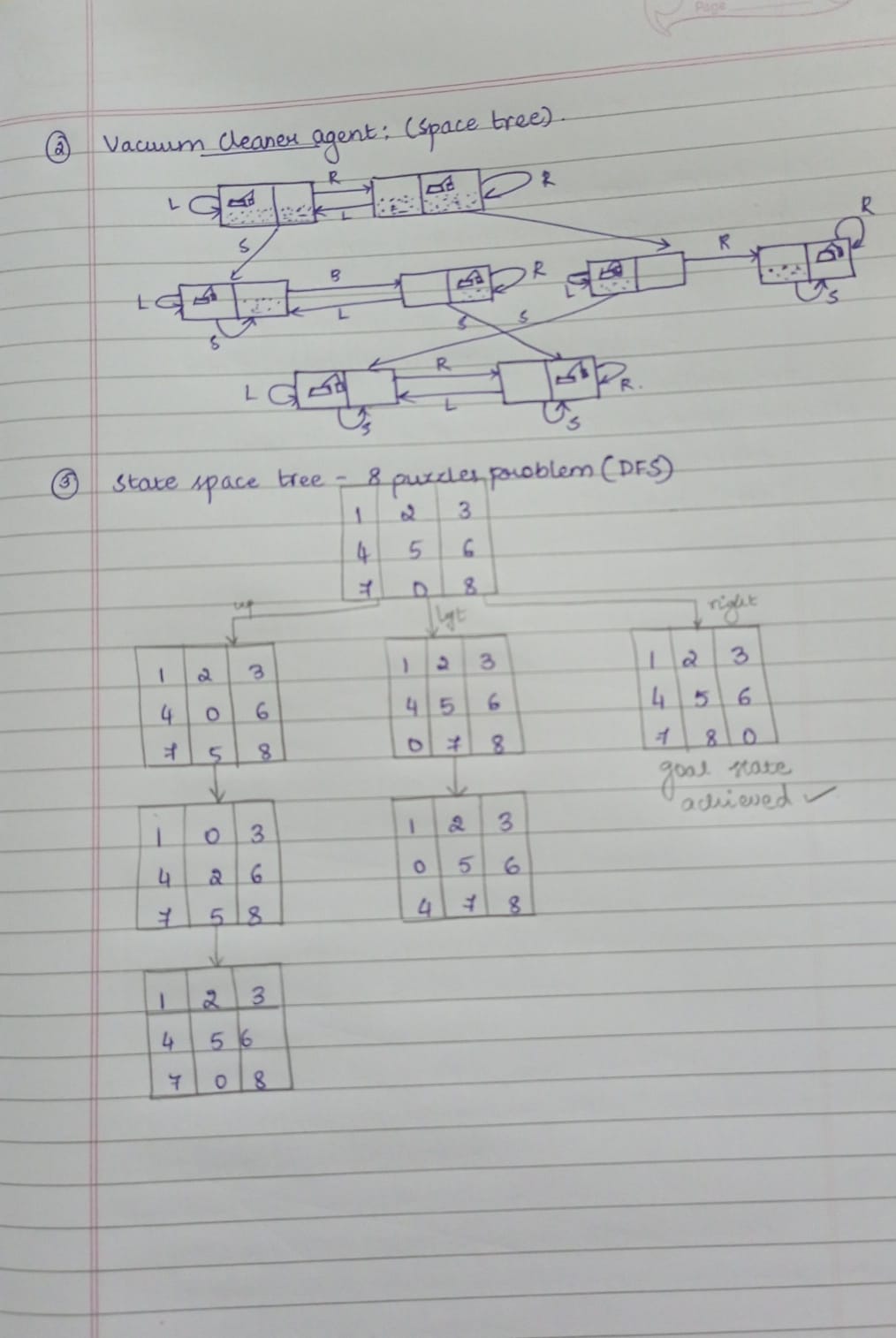
**The vacuum cleaner is at (0, 1) and it is clean. Moving.**

**The vacuum cleaner is at (1, 0) and it is dirty. Cleaning.**

**The vacuum cleaner is at (1, 1) and it is dirty. Cleaning.**

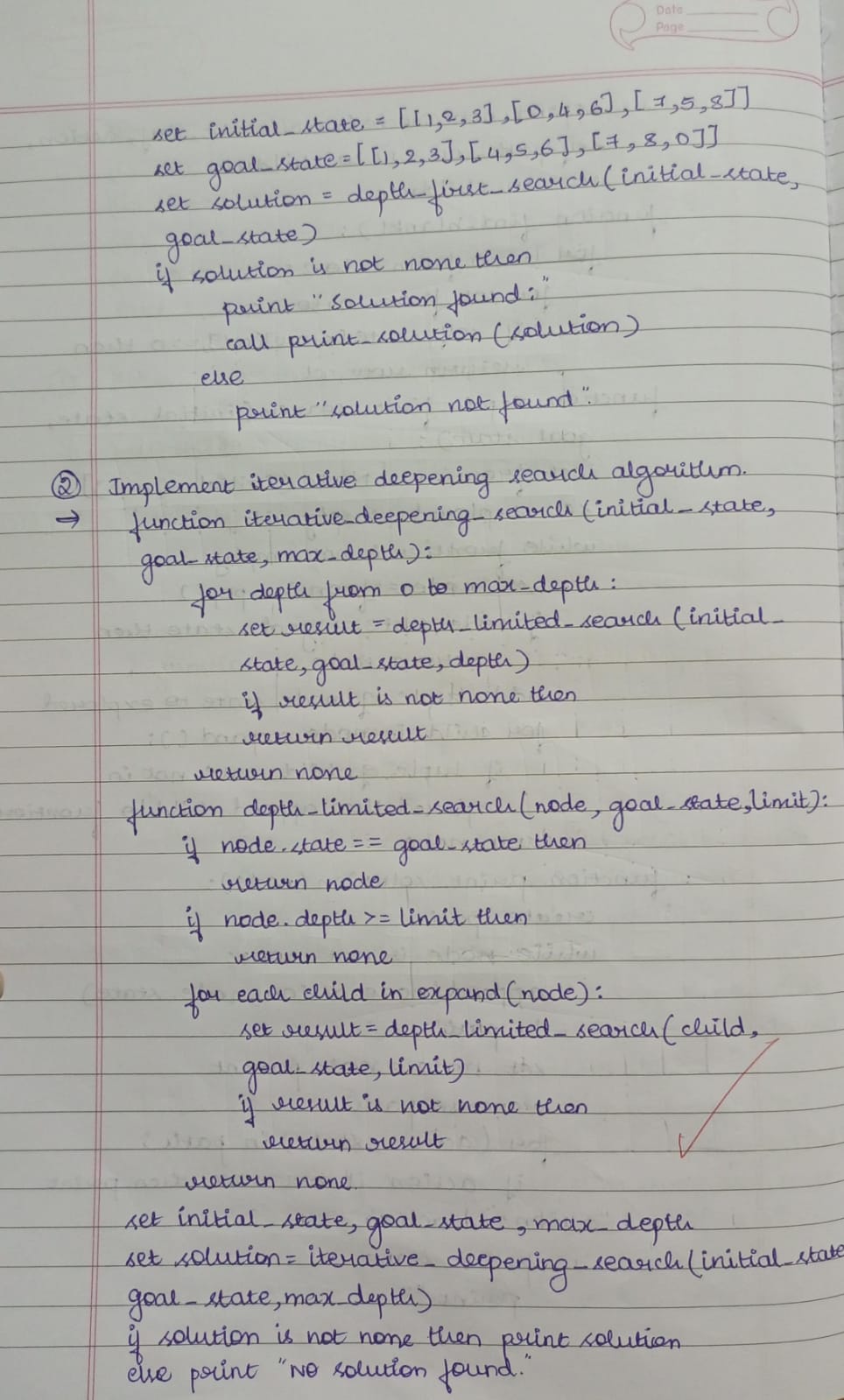
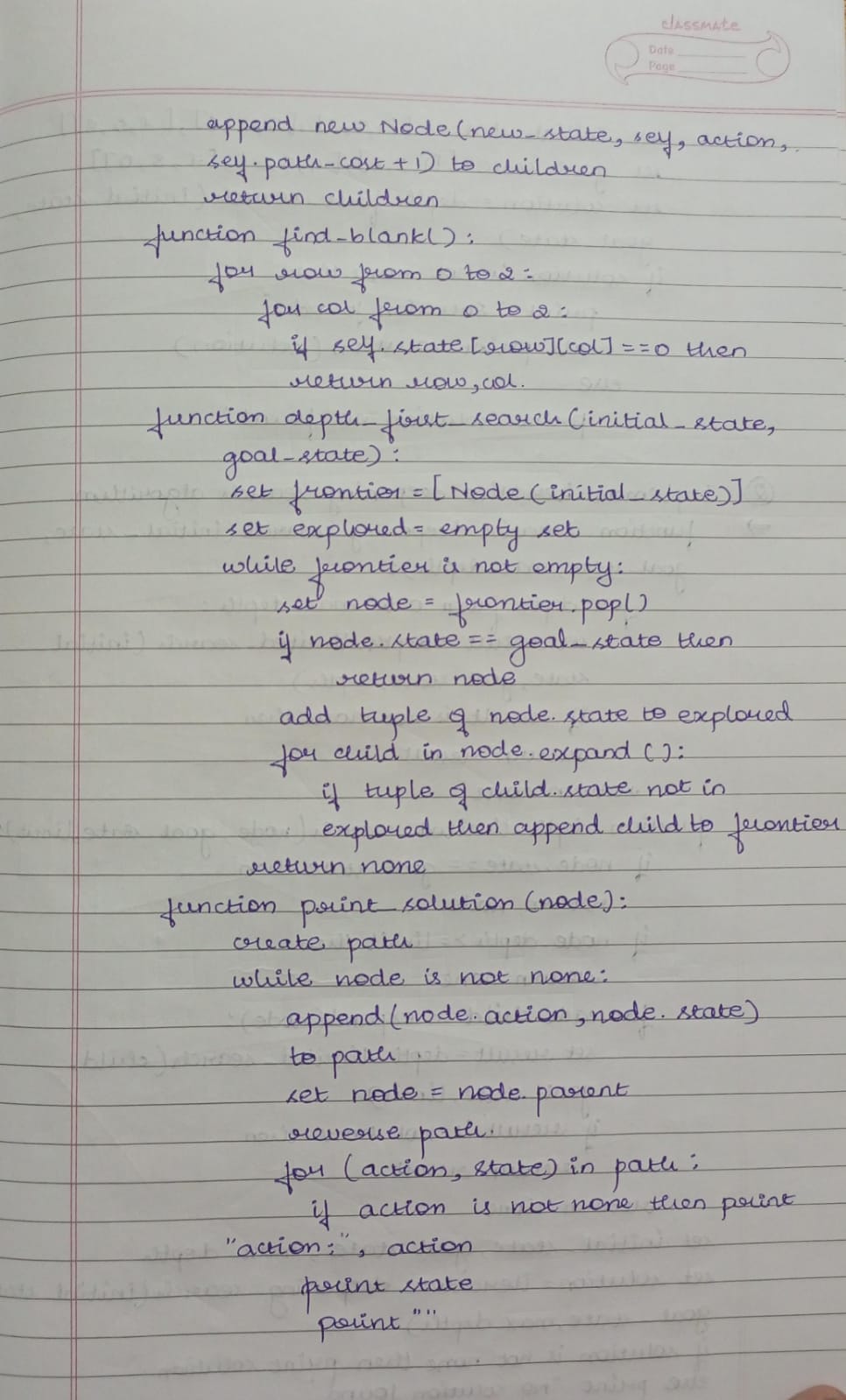
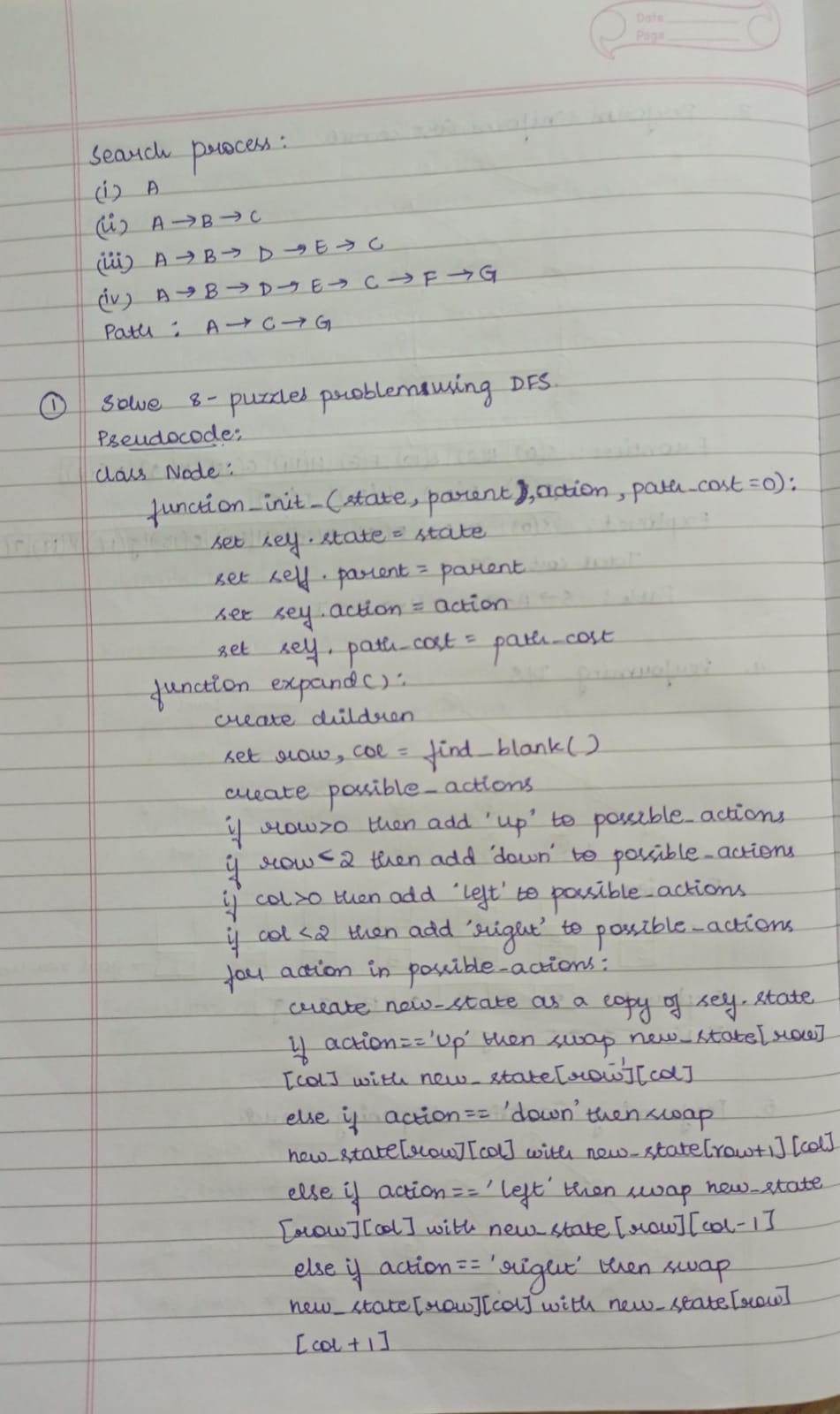
**All locations are clean. The vacuum cleaner is finished.**

**State Space Tree:**

****

1. **Solve 8-puzzle’s problem using DFS.**

**Algorithm:**

****

**Code:**

import copy

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, path\_cost=0):

self.state = state

self.parent = parent

self.action = action

self.path\_cost = path\_cost

def \_\_lt\_\_(self, other):

return self.path\_cost < other.path\_cost

def expand(self):

children = []

row, col = self.find\_blank()

possible\_actions = []

if row > 0:

possible\_actions.append('Up')

if row < 2:

possible\_actions.append('Down')

if col > 0:

possible\_actions.append('Left')

if col < 2:

possible\_actions.append('Right')

for action in possible\_actions:

new\_state = copy.deepcopy(self.state)

if action == 'Up':

new\_state[row][col], new\_state[row - 1][col] = new\_state[row - 1][col], new\_state[row][col]

elif action == 'Down':

new\_state[row][col], new\_state[row + 1][col] = new\_state[row + 1][col], new\_state[row][col]

elif action == 'Left':

new\_state[row][col], new\_state[row][col - 1] = new\_state[row][col - 1], new\_state[row][col]

elif action == 'Right':

new\_state[row][col], new\_state[row][col + 1] = new\_state[row][col + 1], new\_state[row][col]

children.append(Node(new\_state, self, action, self.path\_cost + 1))

return children

def find\_blank(self):

for row in range(3):

for col in range(3):

if self.state[row][col] == 0:

return row, col

def depth\_first\_search(initial\_state, goal\_state):

frontier = [Node(initial\_state)]

explored = set()

while frontier:

node = frontier.pop()

if node.state == goal\_state:

return node

explored.add(tuple(map(tuple, node.state))) # Track visited states

for child in node.expand():

child\_state\_tuple = tuple(map(tuple, child.state))

if child\_state\_tuple not in explored and child not in frontier:

frontier.append(child)

return None

def print\_solution(node):

path = []

while node is not None:

path.append((node.action, node.state))

node = node.parent

path.reverse() # Reverse the path to start from the initial state

for action, state in path:

if action:

print(f"Action: {action}")

for row in state:

print(row)

print()

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

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

solution = depth\_first\_search(initial\_state, goal\_state)

if solution:

print("Solution found:")

print\_solution(solution)

else:

print("Solution not found.")

**Output:**

**Action: Left**

**[1, 2, 3]**

**[7, 4, 5]**

**[8, 0, 6]**

**Action: Left**

**[1, 2, 3]**

**[7, 4, 5]**

**[0, 8, 6]**

**Action: Up**

**[1, 2, 3]**

**[0, 4, 5]**

**[7, 8, 6]**

**Action: Right**

**[1, 2, 3]**

**[4, 0, 5]**

**[7, 8, 6]**

**Action: Right**

**[1, 2, 3]**

**[4, 5, 0]**

**[7, 8, 6]**

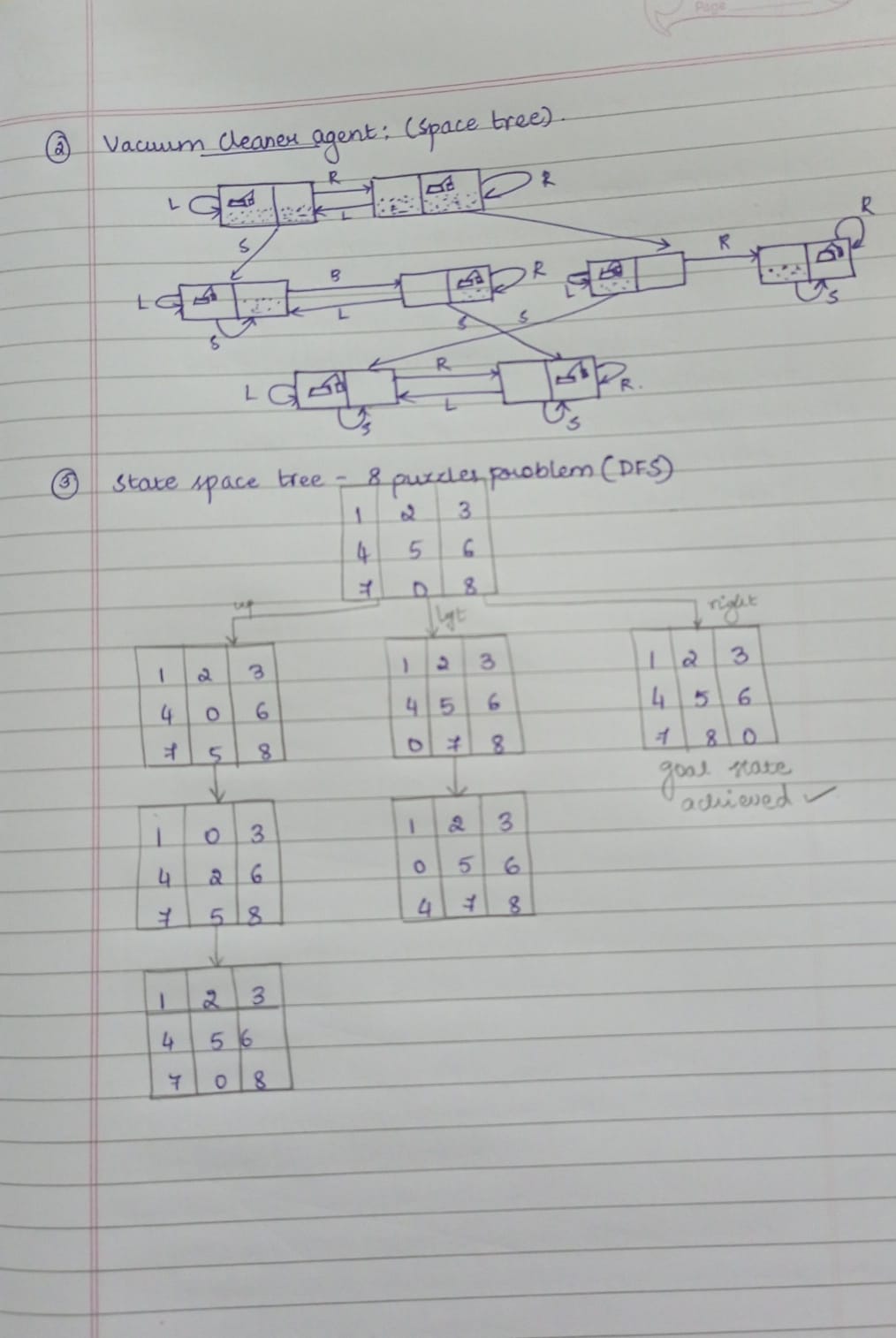
**Action: Down**

**[1, 2, 3]**

**[4, 5, 6]**

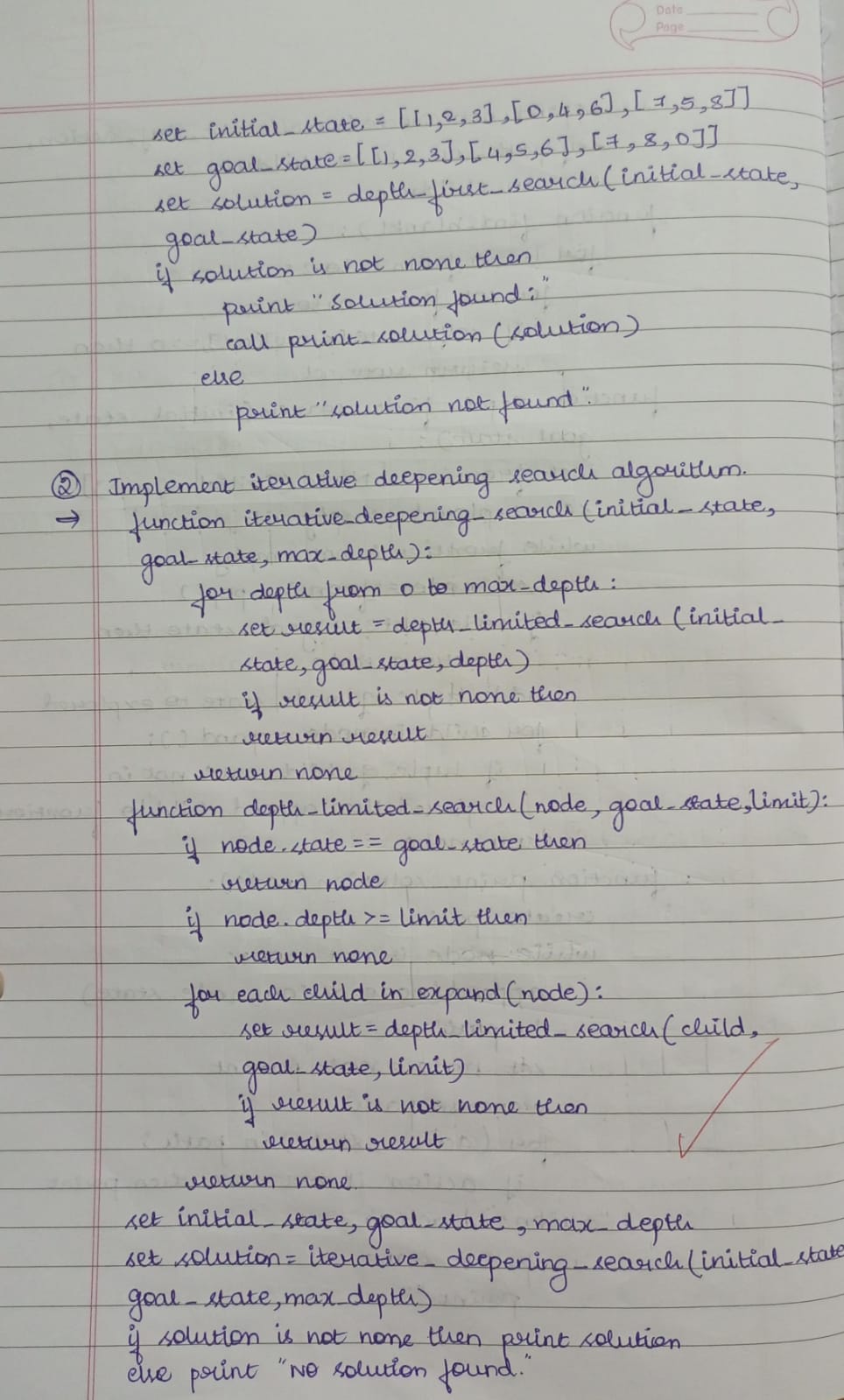
**[7, 8, 0]**

**State Space Tree:**

****

1. **Implement Iterative Deepening Search algorithm.**

**Algorithm:**

****

**Code:**

import copy

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, depth=0):

self.state = state

self.parent = parent

self.action = action

self.depth = depth

def \_\_lt\_\_(self, other):

return self.depth < other.depth

def expand(self):

children = []

row, col = self.find\_blank()

possible\_actions = []

if row > 0: # Can move the blank tile up

possible\_actions.append('Up')

if row < 2: # Can move the blank tile down

possible\_actions.append('Down')

if col > 0: # Can move the blank tile left

possible\_actions.append('Left')

if col < 2: # Can move the blank tile right

possible\_actions.append('Right')

for action in possible\_actions:

new\_state = copy.deepcopy(self.state)

if action == 'Up':

new\_state[row][col], new\_state[row - 1][col] = new\_state[row - 1][col], new\_state[row][col]

elif action == 'Down':

new\_state[row][col], new\_state[row + 1][col] = new\_state[row + 1][col], new\_state[row][col]

elif action == 'Left':

new\_state[row][col], new\_state[row][col - 1] = new\_state[row][col - 1], new\_state[row][col]

elif action == 'Right':

new\_state[row][col], new\_state[row][col + 1] = new\_state[row][col + 1], new\_state[row][col]

children.append(Node(new\_state, self, action, self.depth + 1))

return children

def find\_blank(self):

for row in range(3):

for col in range(3):

if self.state[row][col] == 0:

return row, col

def depth\_limited\_search(node, goal\_state, limit):

if node.state == goal\_state:

return node

if node.depth >= limit:

return None

for child in node.expand():

result = depth\_limited\_search(child, goal\_state, limit)

if result is not None:

return result

return None

def iterative\_deepening\_search(initial\_state, goal\_state, max\_depth):

for depth in range(max\_depth):

result = depth\_limited\_search(Node(initial\_state), goal\_state, depth)

if result is not None:

return result

return None

def print\_solution(node):

path = []

while node is not None:

path.append((node.action, node.state))

node = node.parent

path.reverse()

for action, state in path:

if action:

print(f"Action: {action}")

for row in state:

print(row)

print()

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

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

max\_depth = 20

solution = iterative\_deepening\_search(initial\_state, goal\_state, max\_depth)

if solution:

print("Solution found:")

print\_solution(solution)

else:

print("Solution not found.")

**Output:**

**Solution found:**

**[1, 2, 3]**

**[0, 4, 6]**

**[7, 5, 8]**

**Action: Right**

**[1, 2, 3]**

**[4, 0, 6]**

**[7, 5, 8]**

**Action: Down**

**[1, 2, 3]**

**[4, 5, 6]**

**[7, 0, 8]**

**Action: Right**

**[1, 2, 3]**

**[4, 5, 6]**

**[7, 8, 0]**