01/10/2024

Lab-01

1. Program Title: Tic Tac Toe game Code:

import random

def check\_win(board, r, c):

ch = 'X' if board[r - 1][c - 1] == 'X' else 'O' # Check the row

if all(cell == ch for cell in board[r - 1]):

return True

# Check the column

if all(board[i][c - 1] == ch for i in range(3)): return True

# Check main diagonal

if r == c and all(board[i][i] == ch for i in range(3)): return True

# Check anti-diagonal

if r + c == 4 and all(board[i][2 - i] == ch for i in range(3)): return True

return False

def display\_board(board): for row in board:

print(" | ".join(row)) print()

def find\_block\_move(board):

# Check rows and columns for blocking opportunity for i in range(3):

# Check rows

if board[i].count('X') == 2 and board[i].count('-') == 1: return i, board[i].index('-')

# Check columns

col = [board[0][i], board[1][i], board[2][i]] if col.count('X') == 2 and col.count('-') == 1:

return col.index('-'), i

# Check diagonals for blocking opportunity

diag1 = [board[0][0], board[1][1], board[2][2]]

if diag1.count('X') == 2 and diag1.count('-') == 1: idx = diag1.index('-')

return idx, idx

diag2 = [board[0][2], board[1][1], board[2][0]]

if diag2.count('X') == 2 and diag2.count('-') == 1: idx = diag2.index('-')

return idx, 2 - idx

return None # No blocking move found

def find\_winning\_move(board):

# Check rows and columns for winning opportunity for i in range(3):

# Check rows

if board[i].count('O') == 2 and board[i].count('-') == 1: return i, board[i].index('-')

# Check columns

col = [board[0][i], board[1][i], board[2][i]] if col.count('O') == 2 and col.count('-') == 1:

return col.index('-'), i

# Check diagonals for winning opportunity

diag1 = [board[0][0], board[1][1], board[2][2]]

if diag1.count('O') == 2 and diag1.count('-') == 1: idx = diag1.index('-')

return idx, idx

diag2 = [board[0][2], board[1][1], board[2][0]]

if diag2.count('O') == 2 and diag2.count('-') == 1:

idx = diag2.index('-') return idx, 2 - idx

return None # No winning move found

def bot\_move(board):

# First, check if there's a move to win winning\_move = find\_winning\_move(board) if winning\_move:

r, c = winning\_move board[r][c] = 'O'

print(f"Bot placed O at winning position: ({r + 1}, {c + 1})") display\_board(board)

return r + 1, c + 1

# Then, check if there's a move to block the human block\_move = find\_block\_move(board)

if block\_move:

r, c = block\_move board[r][c] = 'O'

print(f"Bot blocked X at position: ({r + 1}, {c + 1})") display\_board(board)

return r + 1, c + 1

# Otherwise, make a random move

available\_moves = [(r, c) for r in range(3) for c in range(3) if board[r][c] == '-'] if available\_moves:

move = random.choice(available\_moves) board[move[0]][move[1]] = 'O'

print(f"Bot placed O at position: ({move[0] + 1}, {move[1] + 1})") display\_board(board)

return move[0] + 1, move[1] + 1 # Return the move for win check return None, None

# Initial board setup

board = [['-', '-', '-'], ['-', '-', '-'], ['-', '-', '-']]

display\_board(board)

xo = 1 # 1 for human, 0 for bot

flag = 0 # Flag to check for win or draw

while '-' in board[0] or '-' in board[1] or '-' in board[2]:

if xo == 1: # Human's turn (X)

print("Enter position to place X (row and column between 1-3):") x = int(input())

y = int(input())

if x > 3 or y > 3 or x < 1 or y < 1: print("Invalid position") continue

if board[x - 1][y - 1] == '-':

board[x - 1][y - 1] = 'X'

xo = 0 # Switch to bot's turn display\_board(board)

else:

print("Invalid position") continue

if check\_win(board, x, y): print("X wins!")

flag = 1 break

else: # Bot's turn (O) print("Bot's turn:")

x, y = bot\_move(board)

if x and y: # If bot made a valid move xo = 1 # Switch back to human's turn if check\_win(board, x, y):

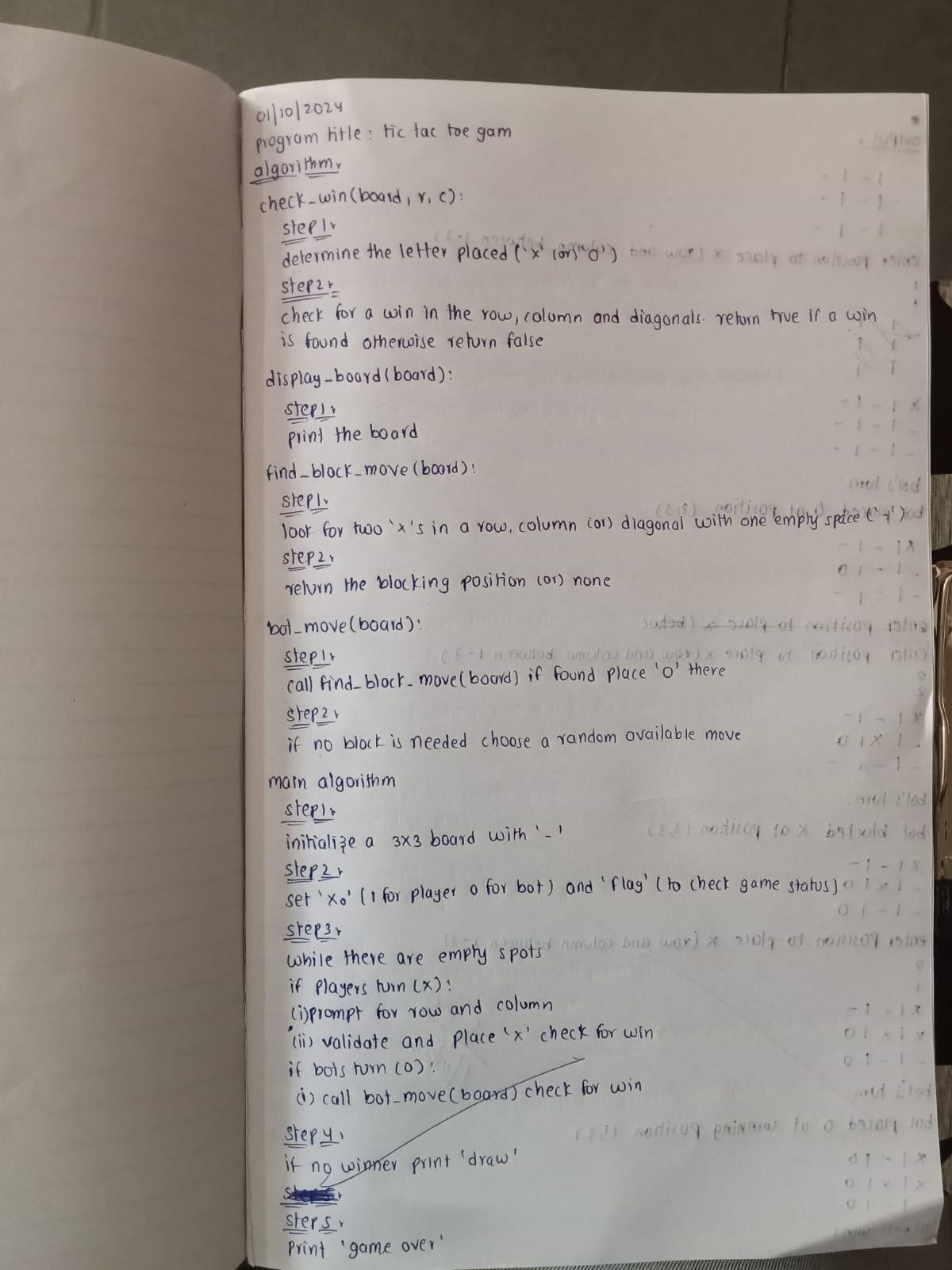
print("O (Bot) wins!") flag = 1

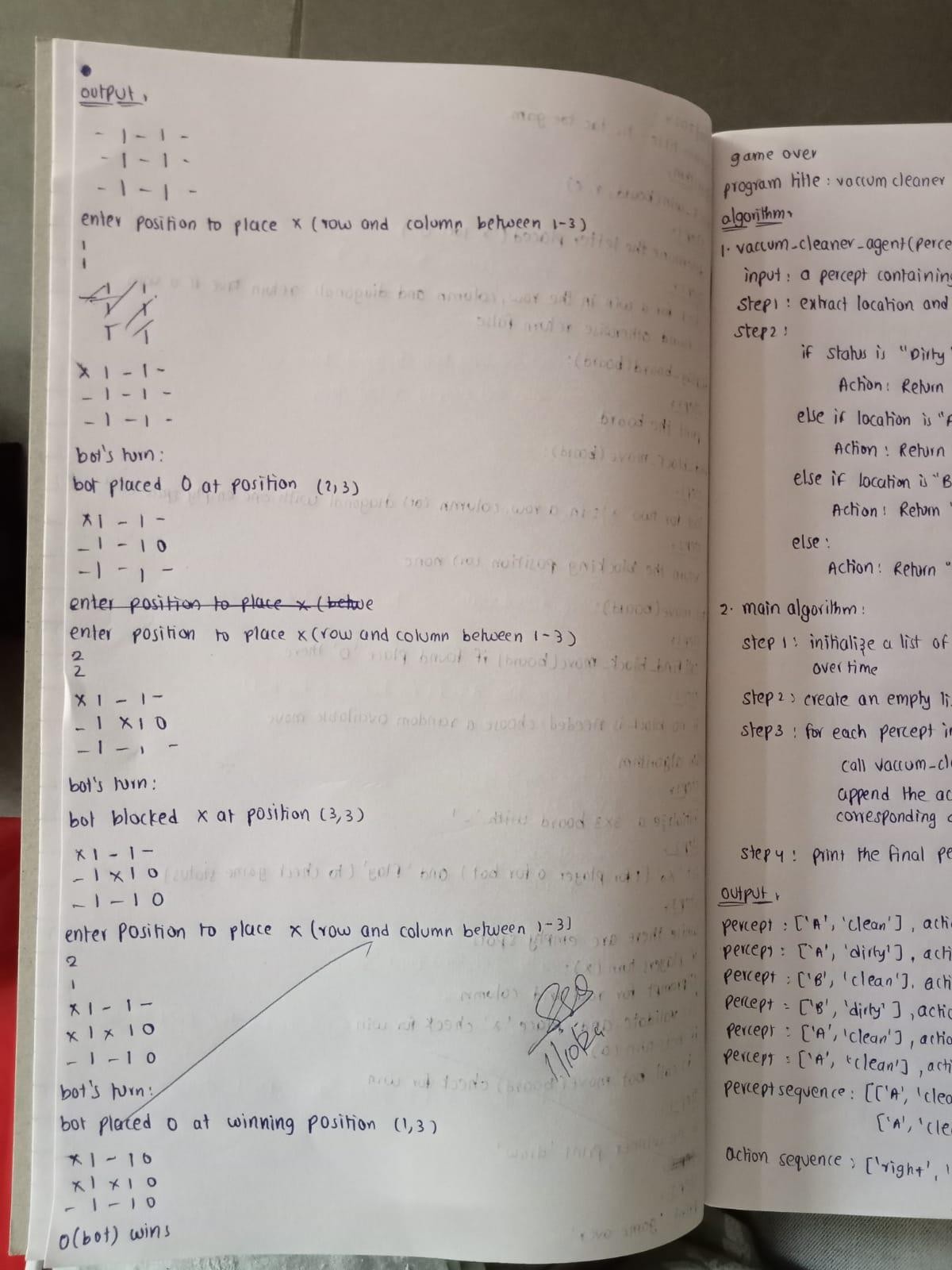
break

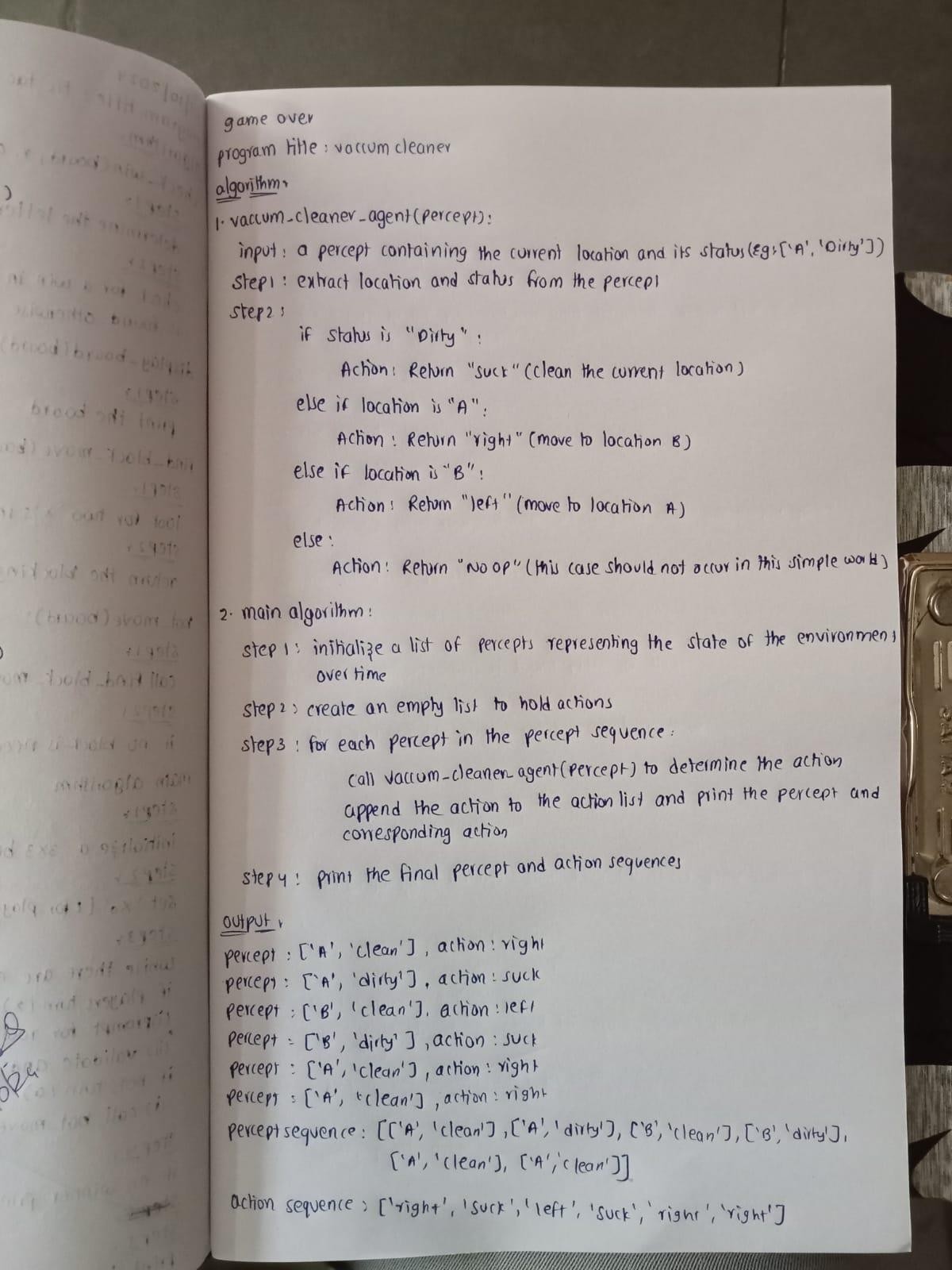
if flag == 0: print("Draw")

print("Game Over")

Algorithm:







Output:

- | - | -

- | - | -

- | - | -

Enter position to place X (row and column between 1-3): 1

1

X | - | -

- | - | -

- | - | -

Bot's turn:

Bot placed O at position: (3, 2) X | - | -

- | - | -

- | O | -

Enter position to place X (row and column between 1-3): 2

2

X | - | -

- | X | -

- | O | -

Bot's turn:

Bot blocked X at position: (3, 3) X | - | -

- | X | -

- | O | O

Enter position to place X (row and column between 1-3): 3

1

X | - | -

- | X | - X | O | O

Bot's turn:

Bot blocked X at position: (2, 1) X | - | -

O | X | - X | O | O

Enter position to place X (row and column between 1-3): 1

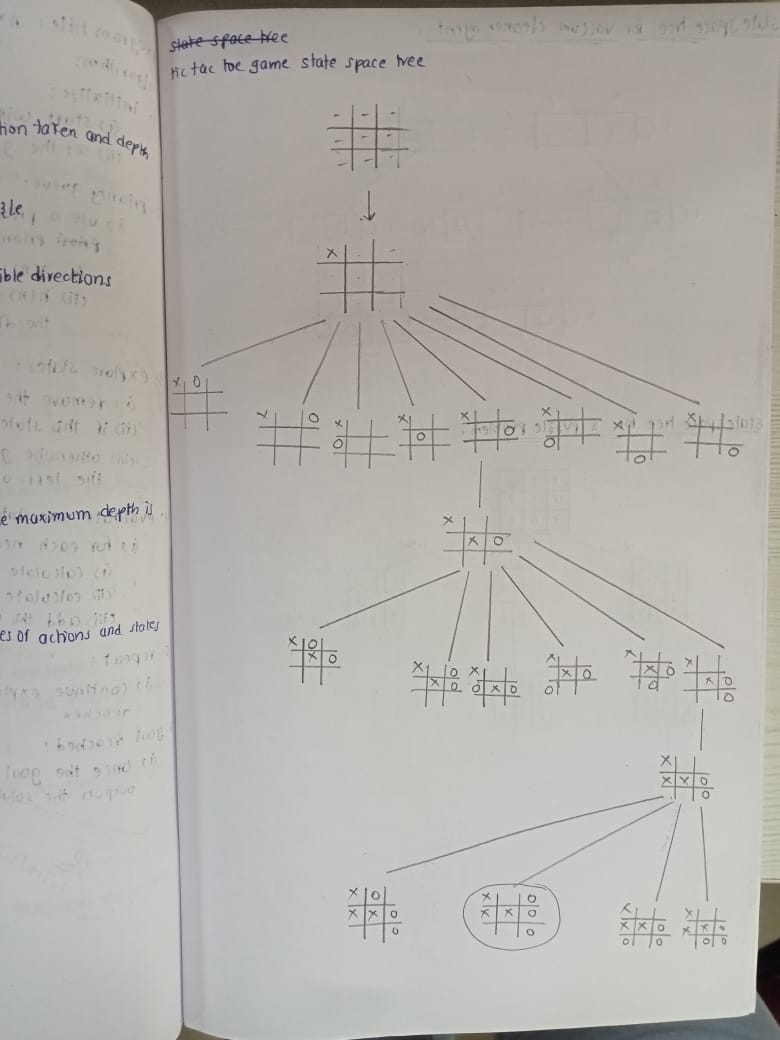
3

X | - | X O | X | - X | O | O

X wins!

Game Over

State space tree:



1. Program Title: vacuum cleaner Code:

def vacuum\_cleaner\_agent(percept):

"""

A simple vacuum cleaner agent that operates in a two-location world.

Args:

percept: A list containing the current location and whether it is dirty. e.g., ['A', 'Dirty']

Returns:

The action to be taken by the agent (Left, Right, Suck, NoOp). """

location, status = percept

if status == 'Dirty':

return 'Suck'

elif location == 'A':

return 'Right'

elif location == 'B':

return 'Left' else:

return 'NoOp' # Should not reach here in this simple world.

# Example percept sequence and action execution

percepts = [['A', 'Clean'], ['A', 'Dirty'], ['B', 'Clean'], ['B', 'Dirty'], ['A', 'Clean'], ['A', 'Clean']] actions = []

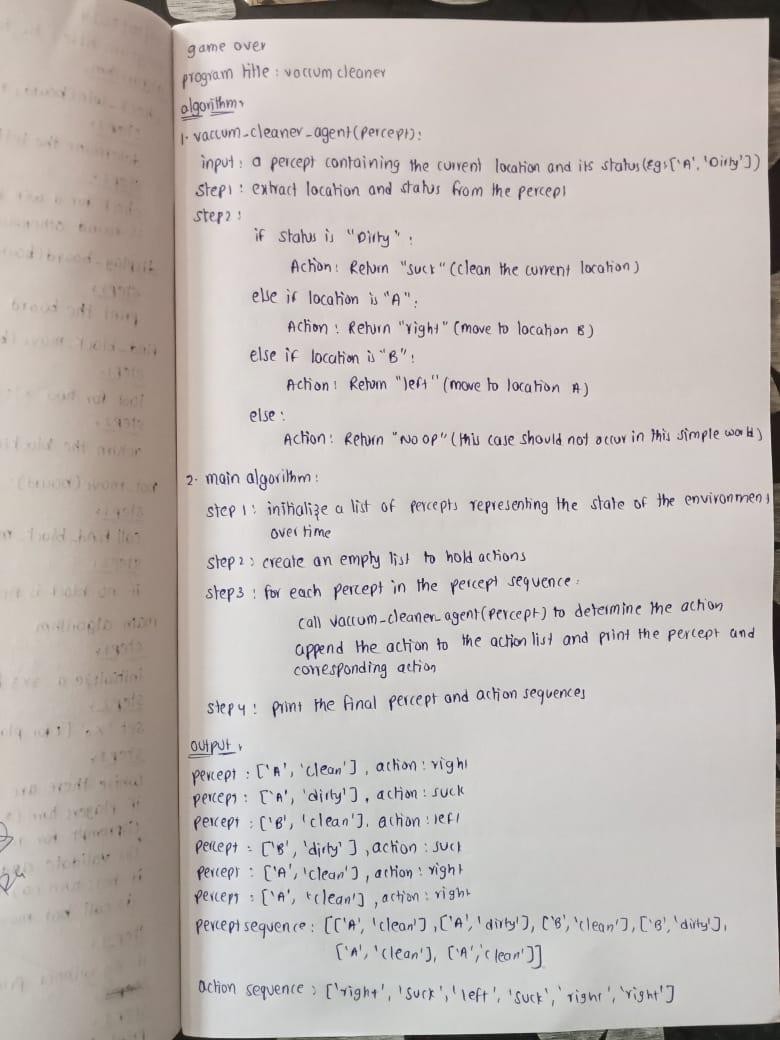
for percept in percepts:

action = vacuum\_cleaner\_agent(percept) actions.append(action)

print(f"Percept: {percept}, Action: {action}")

print("\nPercept Sequence:", percepts) print("Action Sequence:", actions)

Algorithm:



Output:

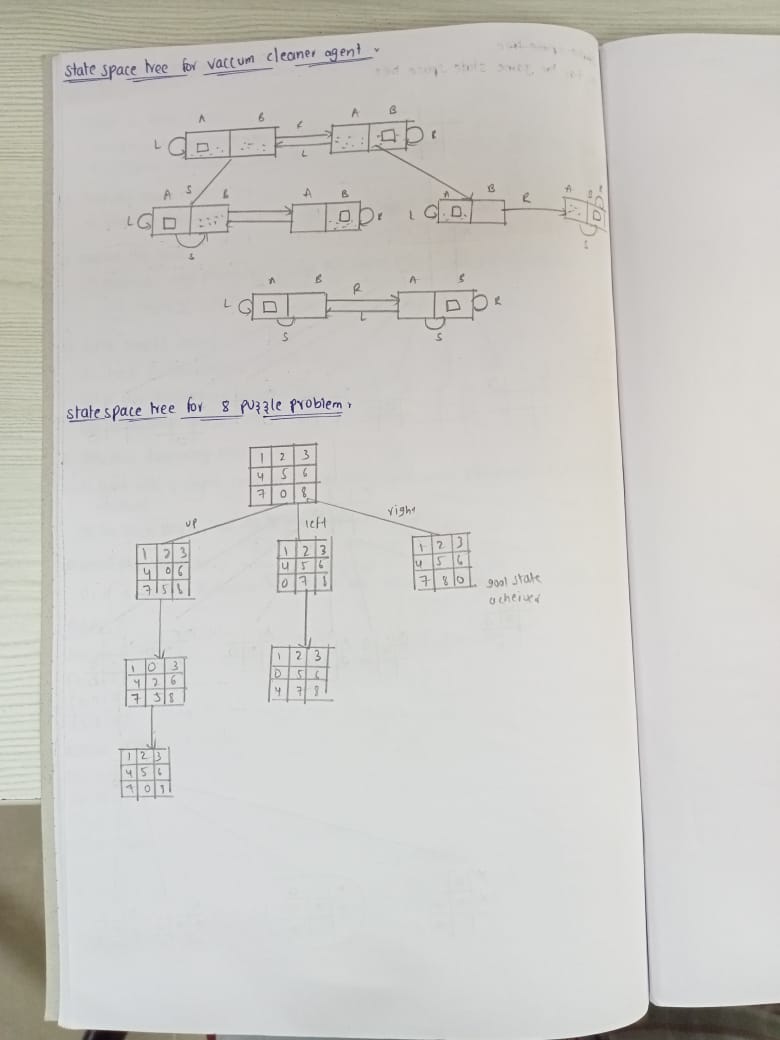
Percept: ['A', 'Clean'], Action: Right Percept: ['A', 'Dirty'], Action: Suck

Percept: ['B', 'Clean'], Action: Left

Percept: ['B', 'Dirty'], Action: Suck Percept: ['A', 'Clean'], Action: Right Percept: ['A', 'Clean'], Action: Right

Percept Sequence: [['A', 'Clean'], ['A', 'Dirty'], ['B', 'Clean'], ['B', 'Dirty'], ['A', 'Clean'], ['A', 'Clean']] Action Sequence: ['Right', 'Suck', 'Left', 'Suck', 'Right', 'Right']

State space tree:



08/10/2024

Lab-02

1. Program title: 8 puzzle problem Code:

import copy

# Directions for movement: up, down, left, right

moves = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}

# Check if a state is the goal state def is\_goal(state, goal\_state):

return state == goal\_state

# Get the position of the empty space (0) def get\_empty\_position(state):

for i in range(3):

for j in range(3):

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

return i, j

# Move the empty space in a specified direction if possible def move\_tile(state, direction):

new\_state = copy.deepcopy(state)

empty\_i, empty\_j = get\_empty\_position(state) di, dj = moves[direction]

new\_i, new\_j = empty\_i + di, empty\_j + dj if 0 <= new\_i < 3 and 0 <= new\_j < 3:

new\_state[empty\_i][empty\_j], new\_state[new\_i][new\_j] = new\_state[new\_i][new\_j], new\_state[empty\_i][empty\_j]

return new\_state return None

# Depth-limited search

def depth\_limited\_search(state, goal\_state, depth\_limit, path): if is\_goal(state, goal\_state):

return state, path

if depth\_limit == 0:

return None, []

empty\_i, empty\_j = get\_empty\_position(state) for direction in moves:

new\_state = move\_tile(state, direction)

if new\_state is not None and new\_state not in path: # Avoid loops

result, new\_path = depth\_limited\_search(new\_state, goal\_state, depth\_limit - 1, path + [new\_state])

if result:

return result, new\_path

return None, []

# Iterative deepening search

def iterative\_deepening\_search(initial\_state, goal\_state): depth = 0

while True:

result, path = depth\_limited\_search(initial\_state, goal\_state, depth, [initial\_state]) if result is not None:

return path, depth depth += 1

# Print the state of the puzzle def print\_state(state):

for row in state:

print(row) print()

# Test the 8-puzzle initial\_state = [

[1, 2, 3],

[4, 0, 5],

[6, 7, 8]

]

goal\_state = [ [1, 2, 3],

[4, 5, 6],

[7, 8, 0]

]

# Solve the puzzle using iterative deepening search

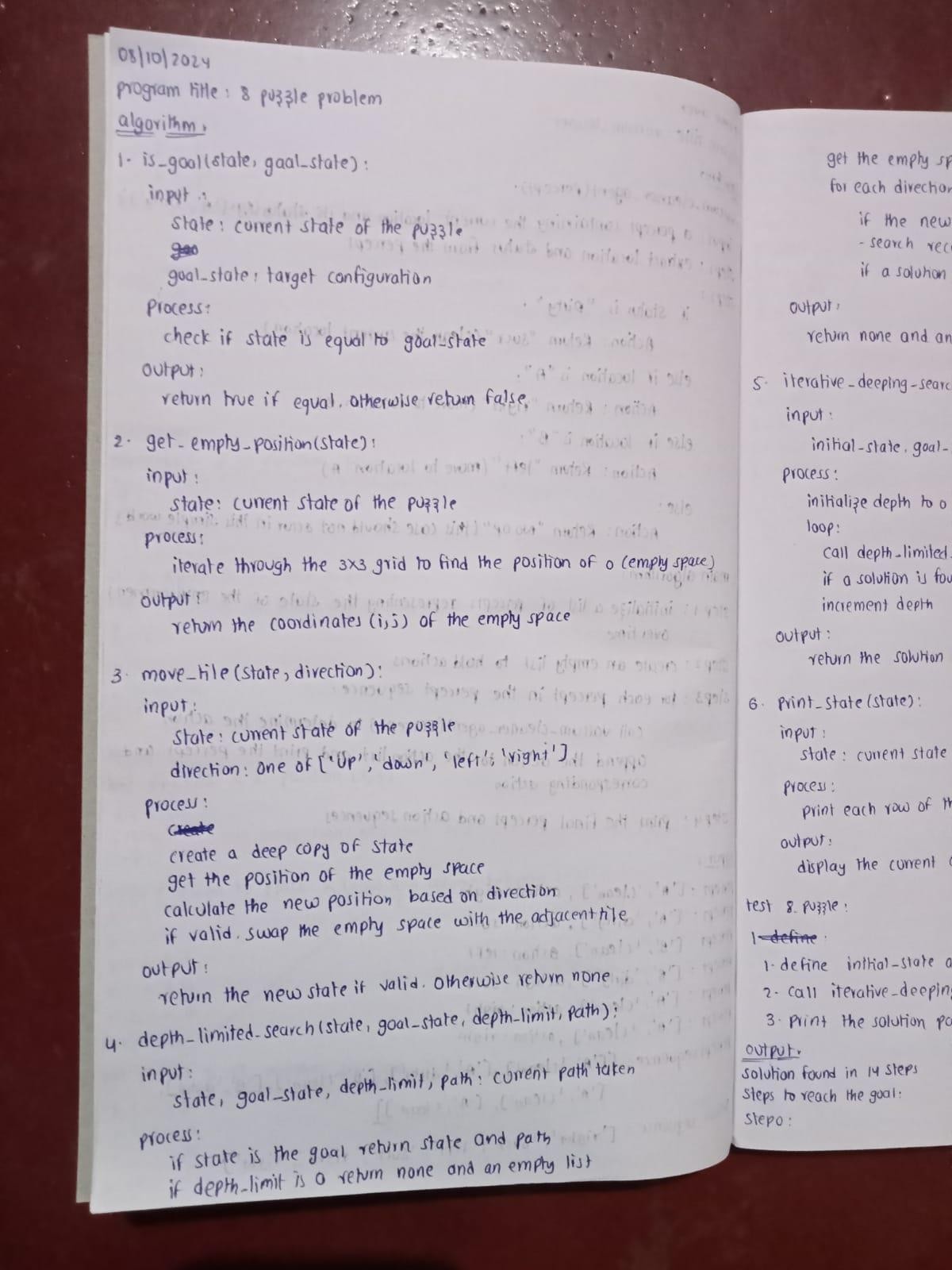
solution\_path, depth = iterative\_deepening\_search(initial\_state, goal\_state)

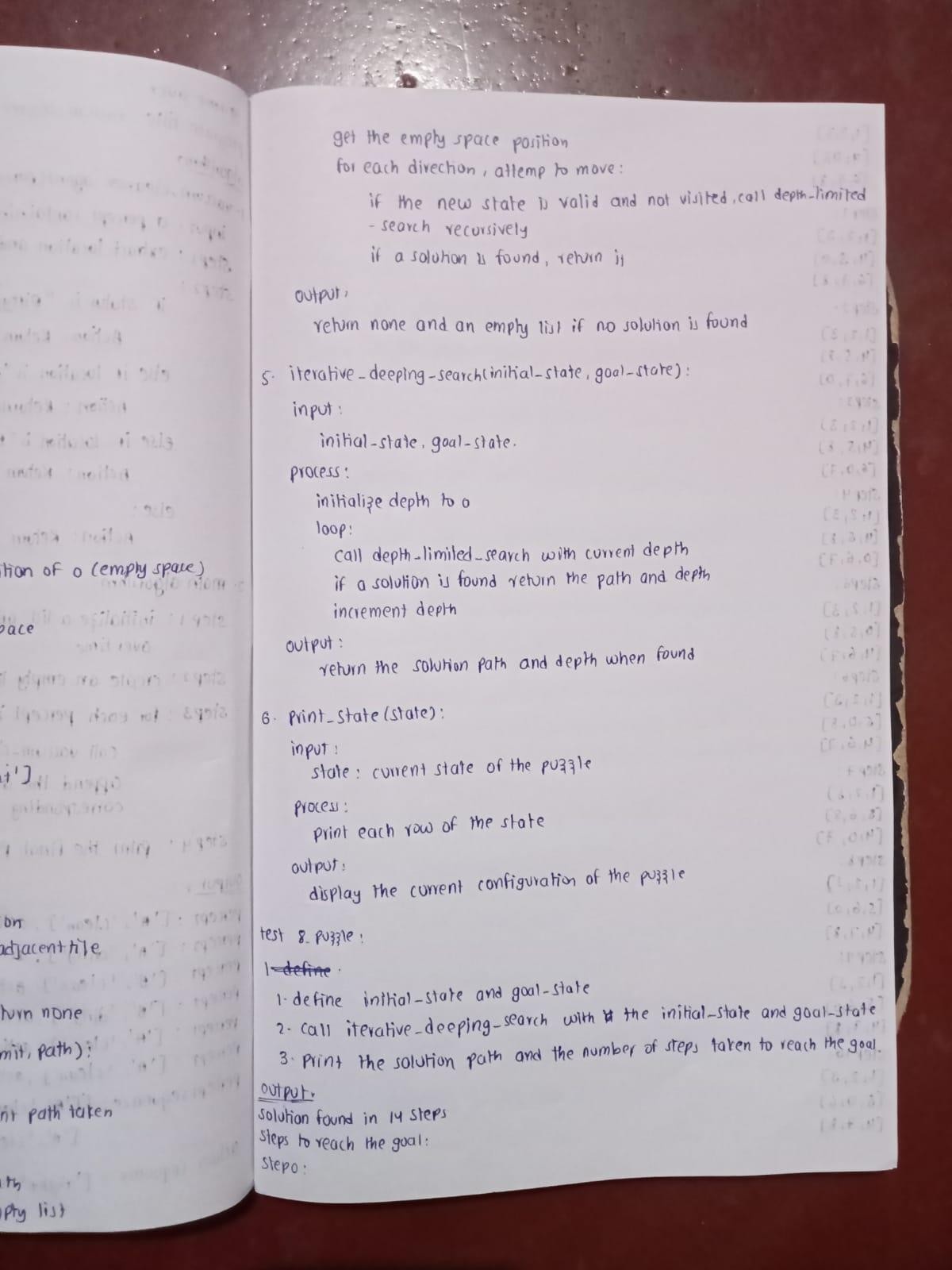
# Output the steps

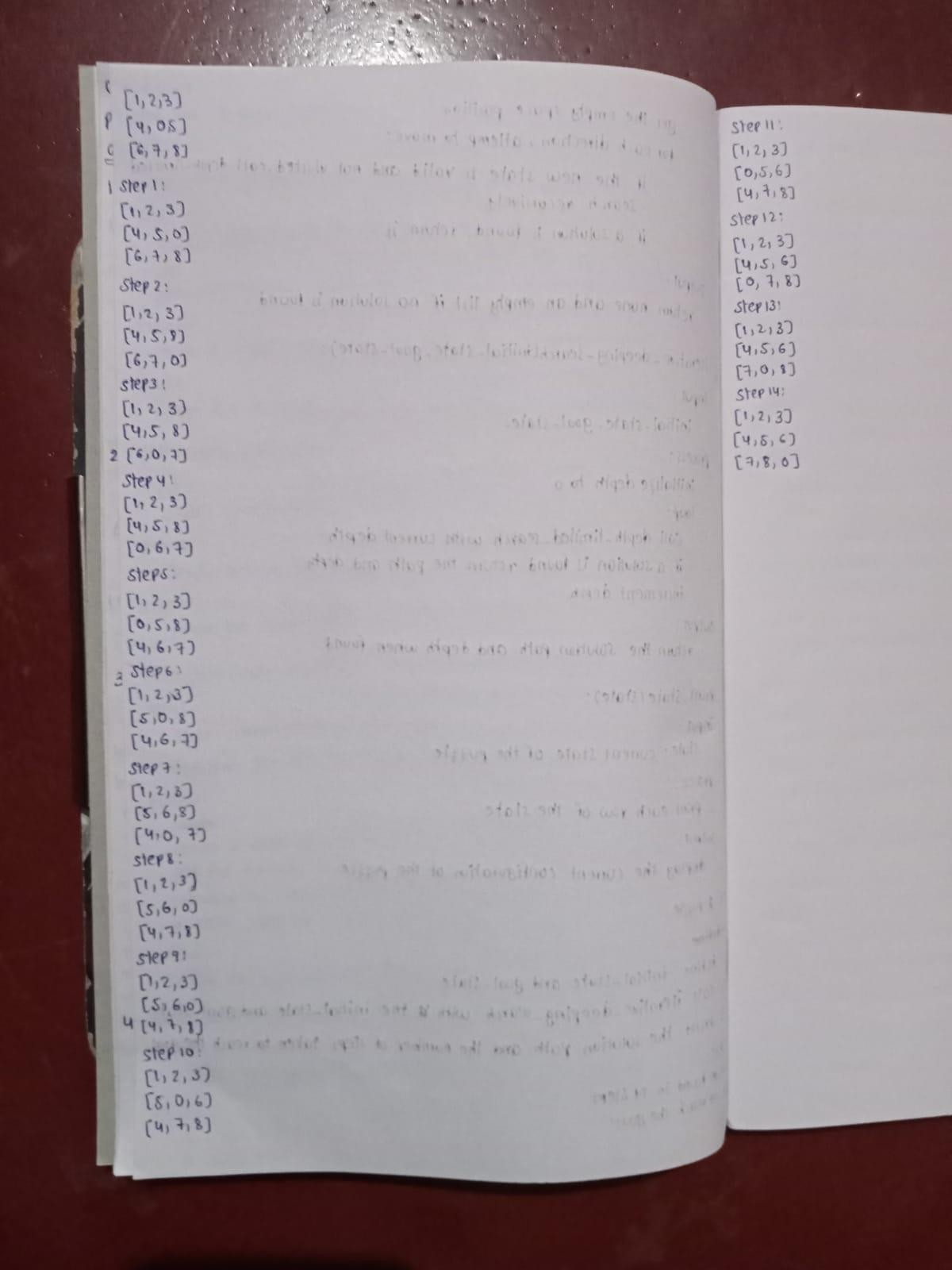
print(f"Solution found in {depth} steps.\n") print("Steps to reach the goal:")

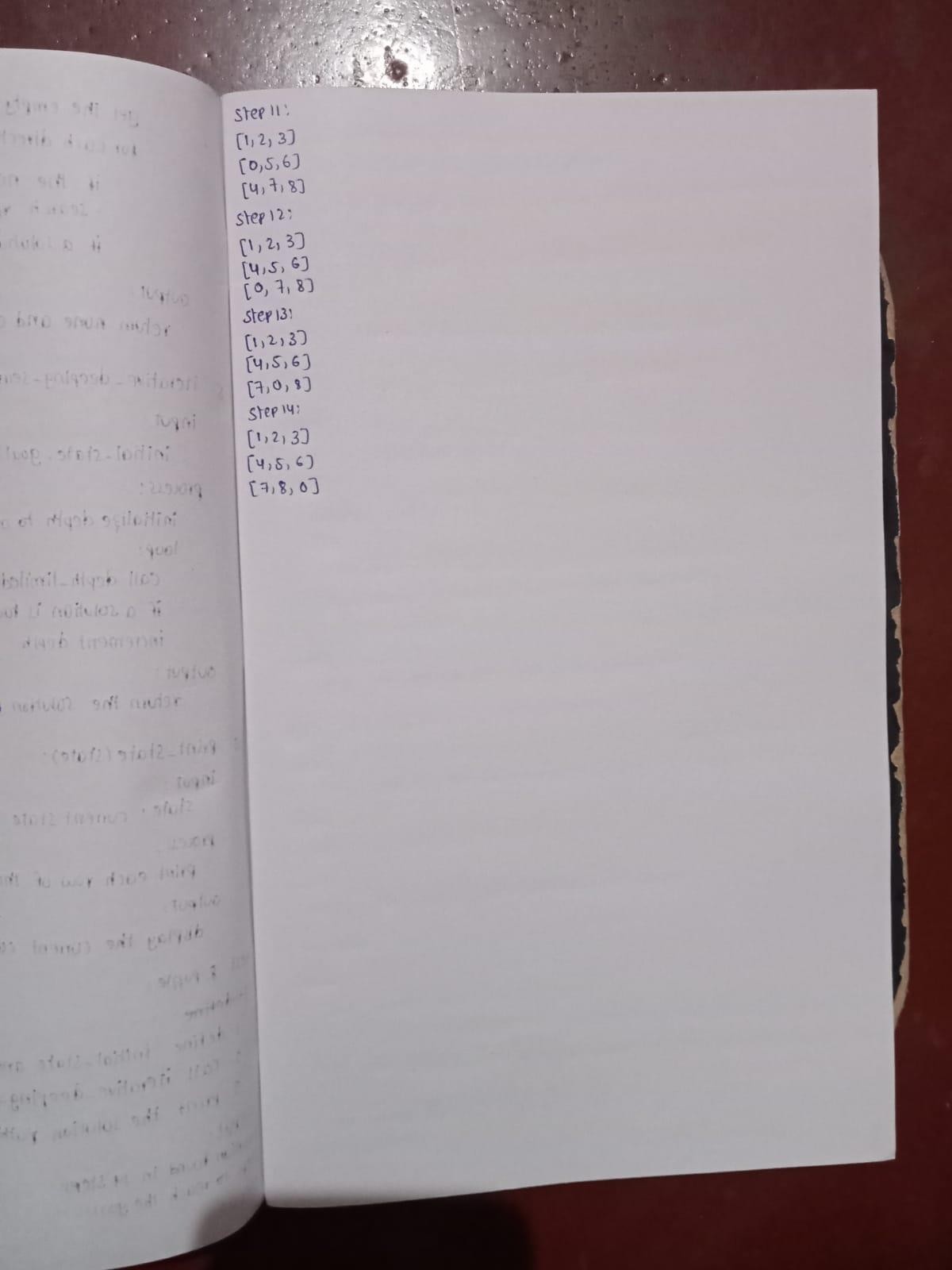
for i, state in enumerate(solution\_path): print(f"Step {i}:")

print\_state(state) Algorithm:









Output:

Solution found in 14 steps.

Steps to reach the goal:

Step 0:

[1, 2, 3]

[4, 0, 5]

[6, 7, 8]

Step 1:

[1, 2, 3]

[4, 5, 0]

[6, 7, 8]

Step 2:

[1, 2, 3]

[4, 5, 8]

[6, 7, 0]

Step 3:

[1, 2, 3]

[4, 5, 8]

[6, 0, 7]

Step 4:

[1, 2, 3]

[4, 5, 8]

[0, 6, 7]

Step 5:

[1, 2, 3]

[0, 5, 8]

[4, 6, 7]

Step 6:

[1, 2, 3]

[5, 0, 8]

[4, 6, 7]

Step 7:

[1, 2, 3]

[5, 6, 8]

[4, 0, 7]

Step 8:

[1, 2, 3]

[5, 6, 8]

[4, 7, 0]

Step 9:

[1, 2, 3]

[5, 6, 0]

[4, 7, 8]

Step 10:

[1, 2, 3]

[5, 0, 6]

[4, 7, 8]

Step 11:

[1, 2, 3]

[0, 5, 6]

[4, 7, 8]

Step 12:

[1, 2, 3]

[4, 5, 6]

[0, 7, 8]

Step 13:

[1, 2, 3]

[4, 5, 6]

[7, 0, 8]

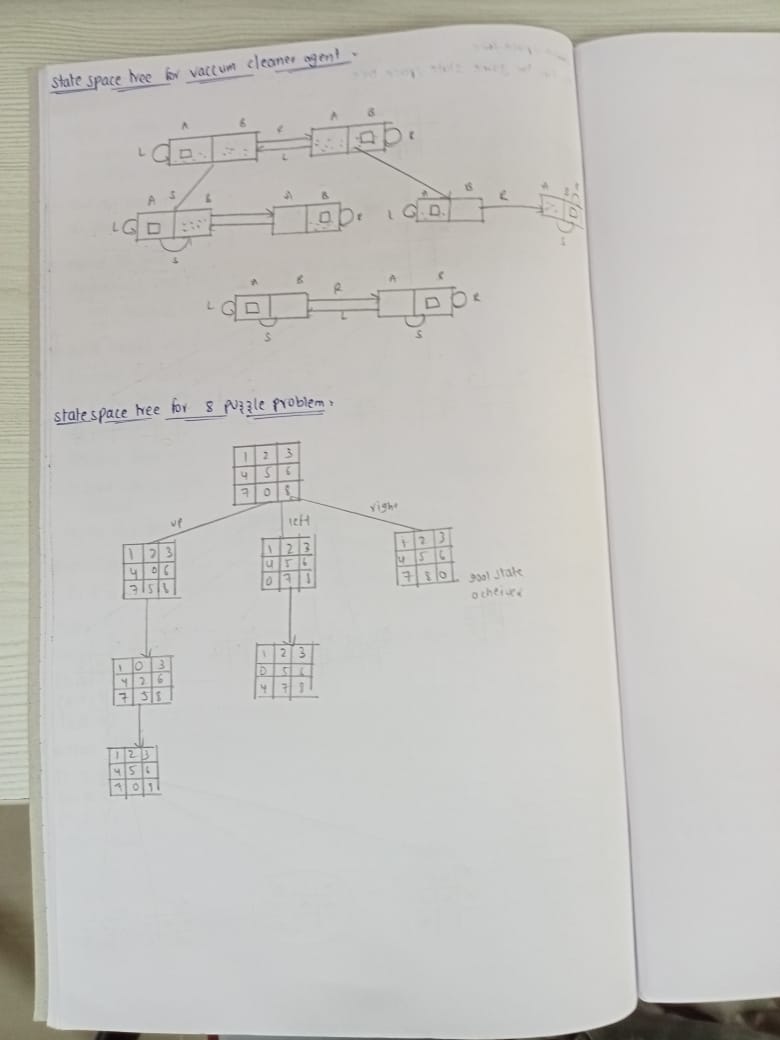
Step 14:

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

State space tree:



1. Program title: Implement Iterative deepening search 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()

# Example usage

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

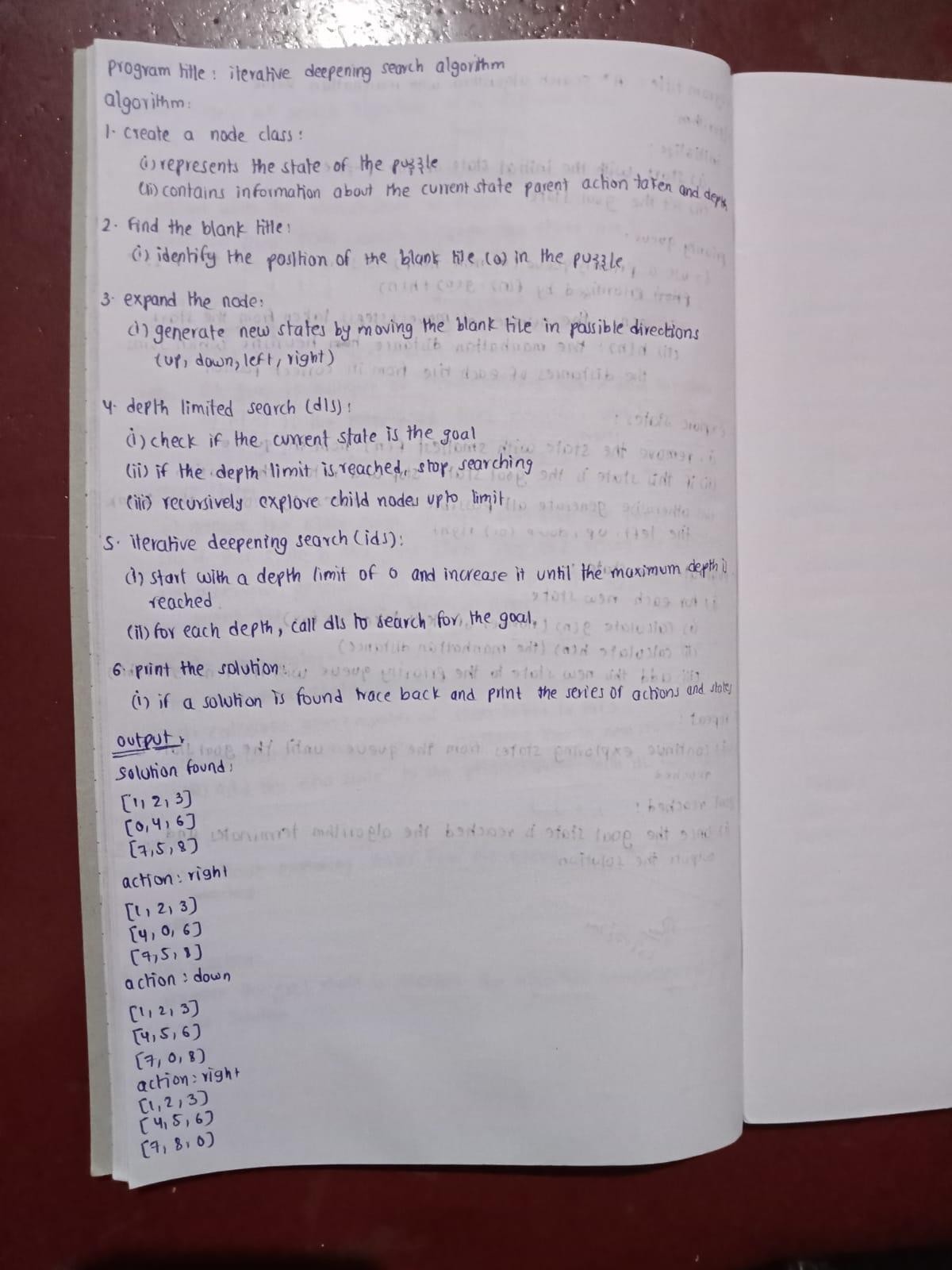
solution = iterative\_deepening\_search(initial\_state, goal\_state, max\_depth)

if solution: print("Solution found:") print\_solution(solution)

else:

print("Solution not found.")

Algorithm:



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]