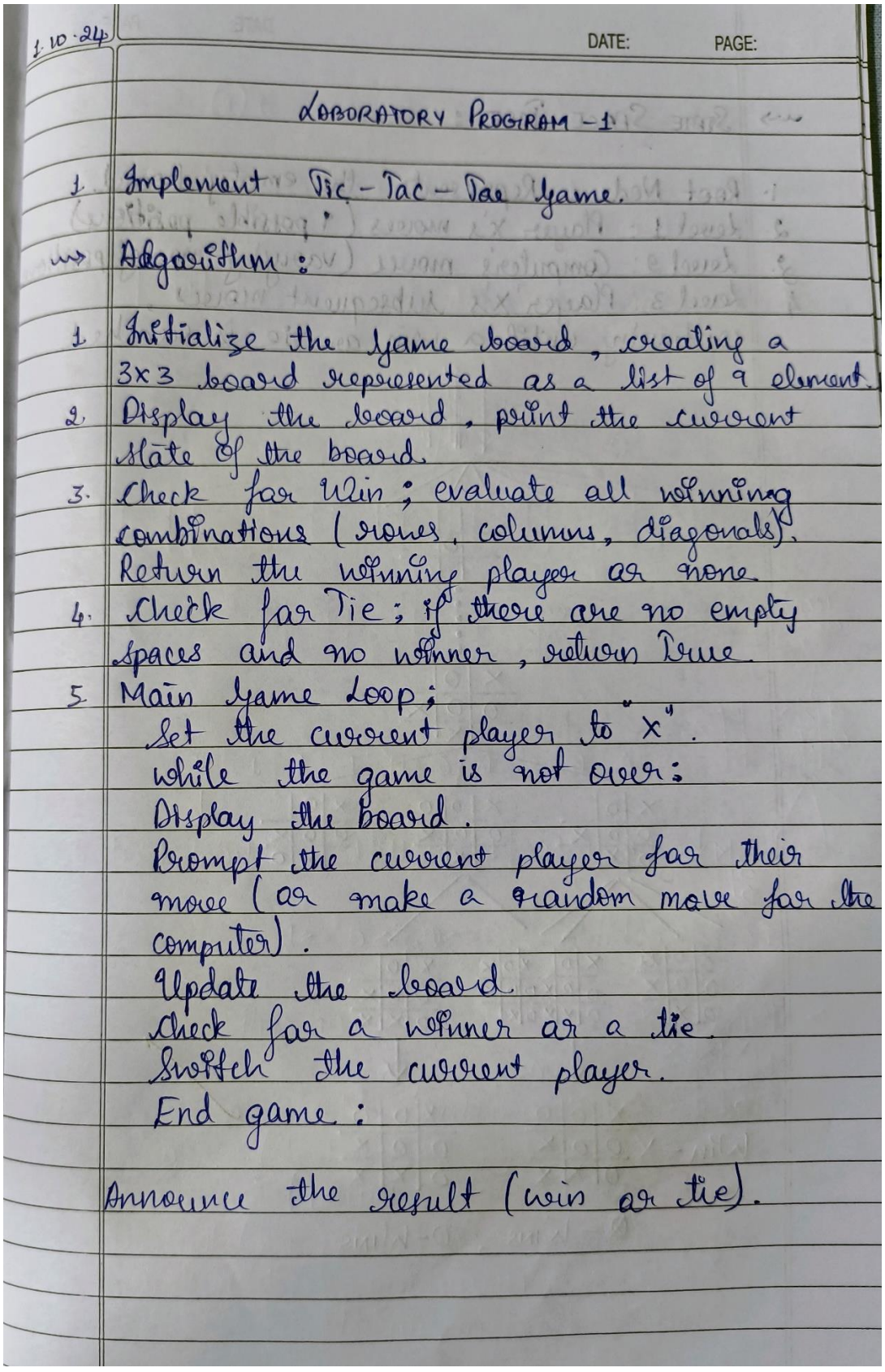
LABORATORY PROGRAM – 1

IMPLEMENT TIC –TAC –TOE GAME

PSEUDOCODE OR ALGORITHM



1.10.24 DATE: PAGE:

LABORATORY PROGRAM - 1

1. Implement Tic - Tac - Toe game.

=> Algorithm:

1. Initialize the game board, creating a 3x3 board represented as a list of 9 element.
2. Display the board, print the current state of the board.
3. Check for Win; evaluate all winning combinations (rows, columns, diagonals). Return the winning player or none.
4. Check for Tie; if there are no empty spaces and no winner, return True.
5. Main game loop;
   Set the current player to "X".
   while the game is not over:
   Display the board.
   Prompt the current player for their move (or make a random move for the computer).
   Update the board.
   Check for a winner or a tie.
   Switch the current player.
   End game:

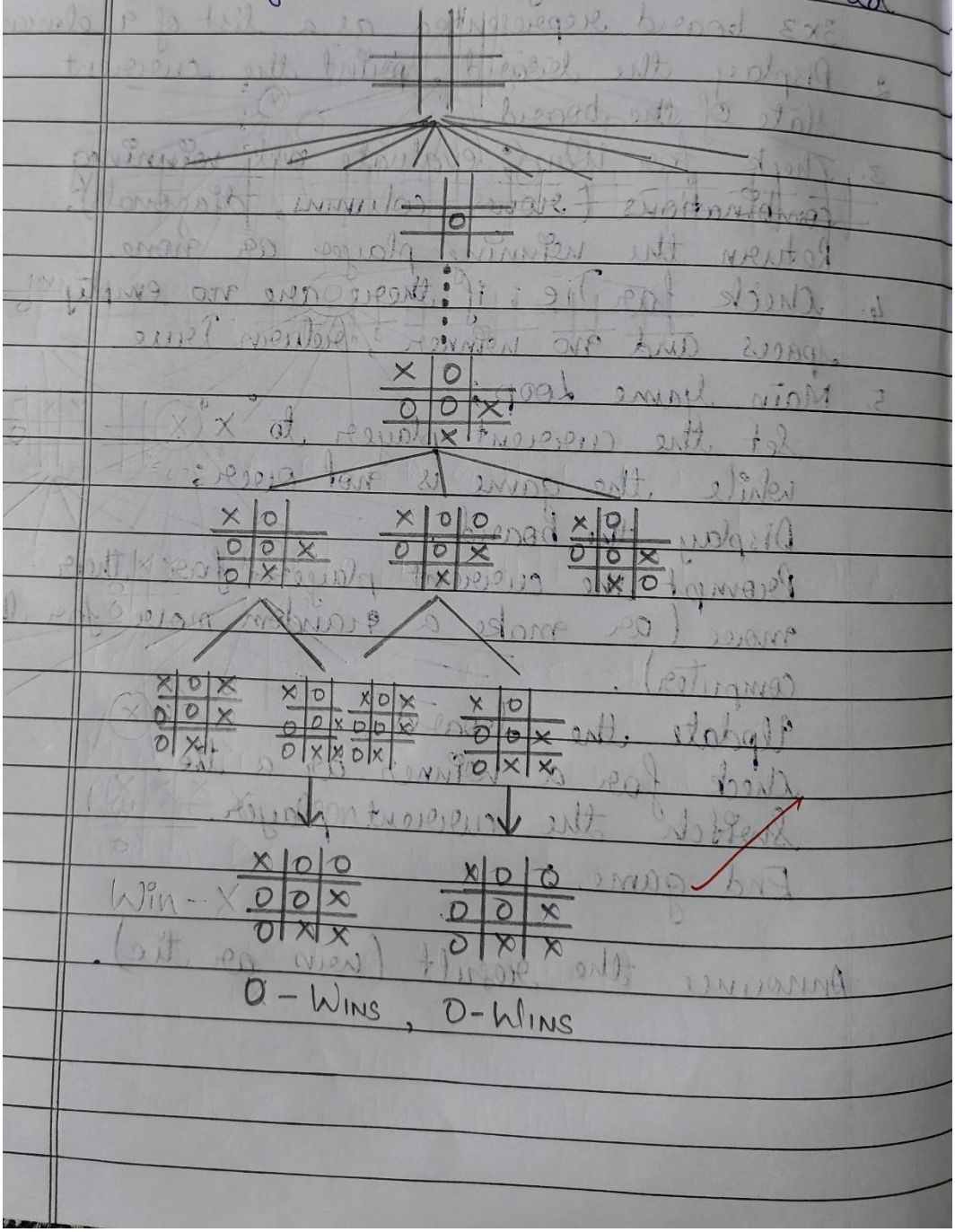
Announce the result (win or tie).

STATE SPACE TREE

DATE: PAGE:

⇒ STATE SPACE TREE:

1. Root Node : Represents the empty board.
2. Level 1 : Player X's moves (9 possible positions)
3. Level 2: Computer's moves (varying no. of positions)
4. Level 3 : Player X's subsequent moves, continuing until a win or tie is reached



CODE

```
import random

board = ["-", "-", "-",
         "-", "-", "-",
         "-", "-", "-"]

def p_board(player):
    print(board[0] + "|" + board[1] + "|" + board[2])
    print(board[3] + "|" + board[4] + "|" + board[5])
    print(board[6] + "|" + board[7] + "|" + board[8])

def check_win():

  for i in range(0, 7, 3):
    if board[i] == board[i + 1] == board[i + 2] and
board[i] != '-':
      return board[i]

  for i in range(3):
    if board[i] == board[i + 3] == board[i + 6] and
board[i] != '-':
      return board[i]

  if board[0] == board[4] == board[8] and board[0] !=
'-':
    return board[0]
  if board[2] == board[4] == board[6] and board[2] !=
'-':
    return board[2]

  return None

def check_tie():
  if '-' not in board:
    return True
  return False

def play_game():
  current_player = "X"
  game_over = False

  while not game_over:
    p_board(current_player)

    if current_player == "X":
      print("It's your turn (X).")
      try:
        position = int(input("Choose a position from 1-9:
")) - 1
        if position < 0 or position > 8 or board[position]
!= '-':
          print("Invalid move. Try again.")
          continue
      except ValueError:
        print("Invalid input. Please enter a number
between 1 and 9.")
        continue
    else:
      print("Computer's turn (O).")
      available_moves = [i for i, spot in
enumerate(board) if spot == '-']
      position = random.choice(available_moves)

    board[position] = current_player
    winner = check_win()

    if winner:
      p_board(current_player)
      print(winner + " won!")
      game_over = True
    elif check_tie():
      p_board(current_player)
      print("It's a tie!")
      game_over = True
    else:
      current_player = "O" if current_player == "X" else
"X"

play_game()
```

OUTPUT

```
-|-|-
-|-|-
-|-|-
It's your turn (X).
Choose a position from 1-9: 5
-|-|-
-|X|-
-|-|-
Computer's turn (O).
-|O|-
-|X|-
-|-|-
It's your turn (X).
Choose a position from 1-9: 7
-|O|-
-|X|-
X|-|-
Computer's turn (O).
-|O|O
-|X|-
X|-|-
It's your turn (X).
Choose a position from 1-9: 1
X|O|O
-|X|-
X|-|-
Computer's turn (O).
X|O|O
-|X|O
X|-|-
It's your turn (X).
Choose a position from 1-9: 4
X|O|O
X|X|O
X|-|-
X won!
```

LABORATORY PROGRAM – 2

SOLVE 8 PUZZLE PROBLEMS

PSEUDOCODE OR ALGORITHM

8.10.24 DATE: PAGE:

LABORATORY PROGRAM :- 2(B)

1. Pseudocode for 8 puzzle problem:

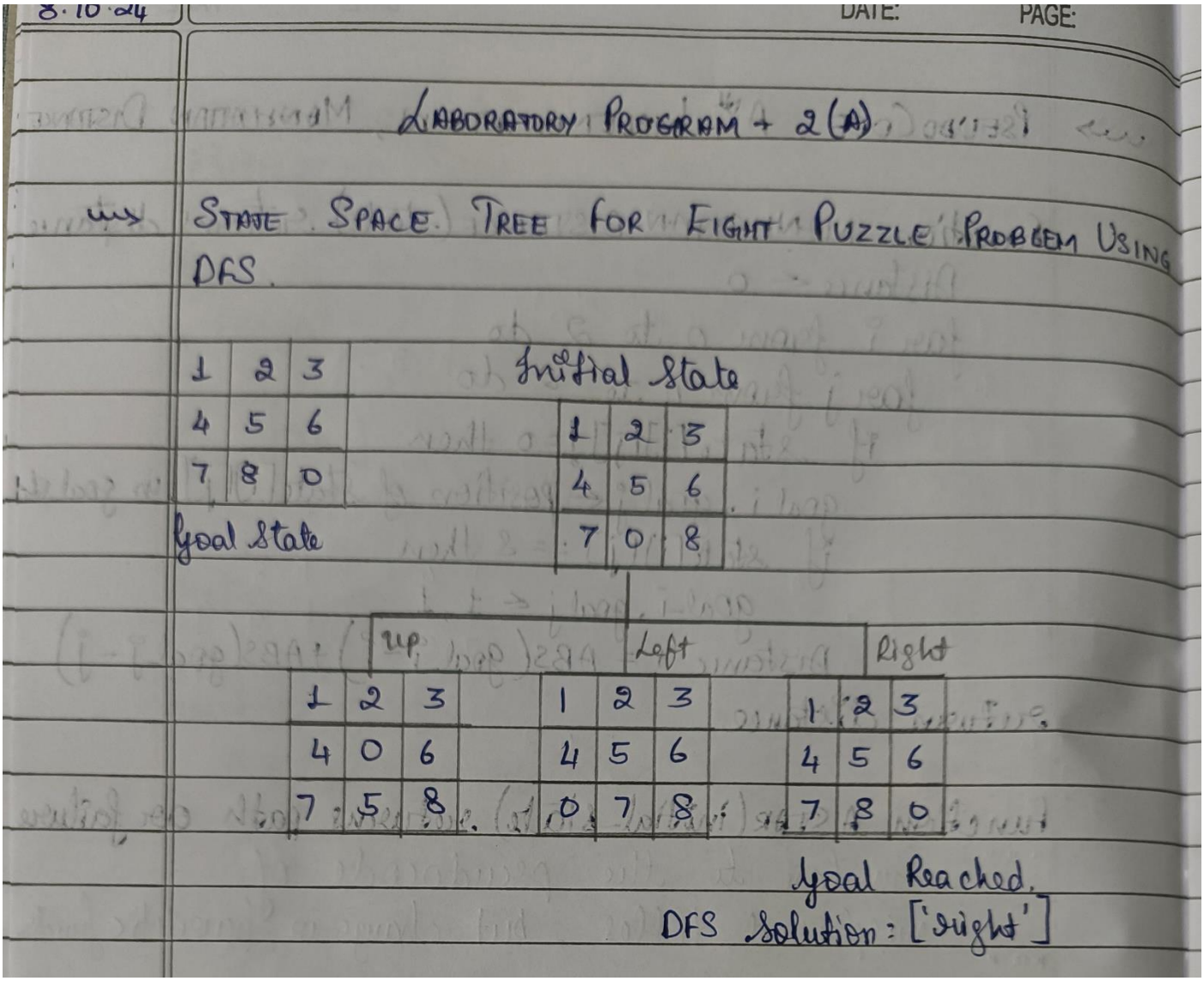
```
function FIND-ZERO(state) returns (row, col)
    for i from 0 to 2 do
        for j from 0 to 2 do
            if state[i][j] == 0 then
                return (i, j)

function MOVE (state, direction) return new_state
    new_state ← copy of state
    (i,j) ← FIND-ZERO(state)
    if direction == "up" and i>0 then
        swap new_state[i][j] with new_state[i-1][j]
    else if direction == "down" and i<2 then
        swap new_state[i][j] with new_state[i+1][j]
    else if direction == "left" and j>0 then
        swap new_state[i][j] with new_state[i][j-1]
    else if direction == "right" and j<2 then
        swap new_state[i][j] with new_state[i][j+1]
return new_state

function IS-GOAL(state) returns boolean
    return state == goal_state

function PRINT-STATE(state)
    for each row in state do
        print row
```

```
function DFS(initial_state) returns: path or failure
    stack ← [(initial_state, [])]
    visited ← empty set
    while not IS-EMPTY(stack) do
        (state, path) ← POP(stack)
        PRINT-STATE(state)

        if IS-GOAL(state) then
            return path.

        visited.add(str(state))
        for direction in ["up", "down", "left", "right"] do
            new_state ← MOVE(state, direction)
            if new_state is not null and str(new_state) not
            in visited then
                PUSH(stack, (new_state, path + [direction]))
    return failure

function GET-INITIAL-STATE() returns: initial_state
    print("Enter the initial state of the 8-puzzle
    (0 for empty space):")
    initial_state ← empty list
    for i from 0 to 2 do
        row ← input row as list of integers
        initial_state.append(row)
    return initial_state.

main
    initial_state ← GET-INITIAL-STATE()
    PRINT-STATE(initial_state)
    start_time ← current time.
    print "Solving using DFS:"
```

DATE: PAGE:

```
    dfs_solution ← DFS(initial_state)
    end_time ← current time
    if dfs_solution is not null then
        print "DFS Solution:", dfs_solution.
    else
        print "No solution found."
    print "Time taken by DFS:", end_time - start_ti
```

STATE SPACE TREE



CODE

```
from collections import deque
import time

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    zero_pos = find_zero(state)
    i, j = zero_pos

    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-
1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] =
new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-
1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] =
new_state[i][j+1], new_state[i][j]
    else:
        return None

    return new_state

def is_goal(state):
    return state == goal_state

def print_state(state):
    for row in state:
        print(row)
    print("\n")

def dfs(initial_state):
    stack = [(initial_state, [])]
    visited = set()

    while stack:
        state, path = stack.pop()
        print("Exploring state in DFS:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(str(state))
        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and str(new_state) not in visited:
                stack.append((new_state, path + [direction]))

    return None

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for
empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1}
(space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain
exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()

    print("Initial State:")
    print_state(initial_state)

    start_time_dfs = time.time()
    print("Solving using DFS:")
    dfs_solution = dfs(initial_state)
    end_time_dfs = time.time()
    if dfs_solution:
        print("DFS Solution:", dfs_solution)
    else:
        print("No solution found with DFS.")
    print(f"Time taken by DFS: {end_time_dfs -
start_time_dfs:.6f} seconds")
```

OUTPUT

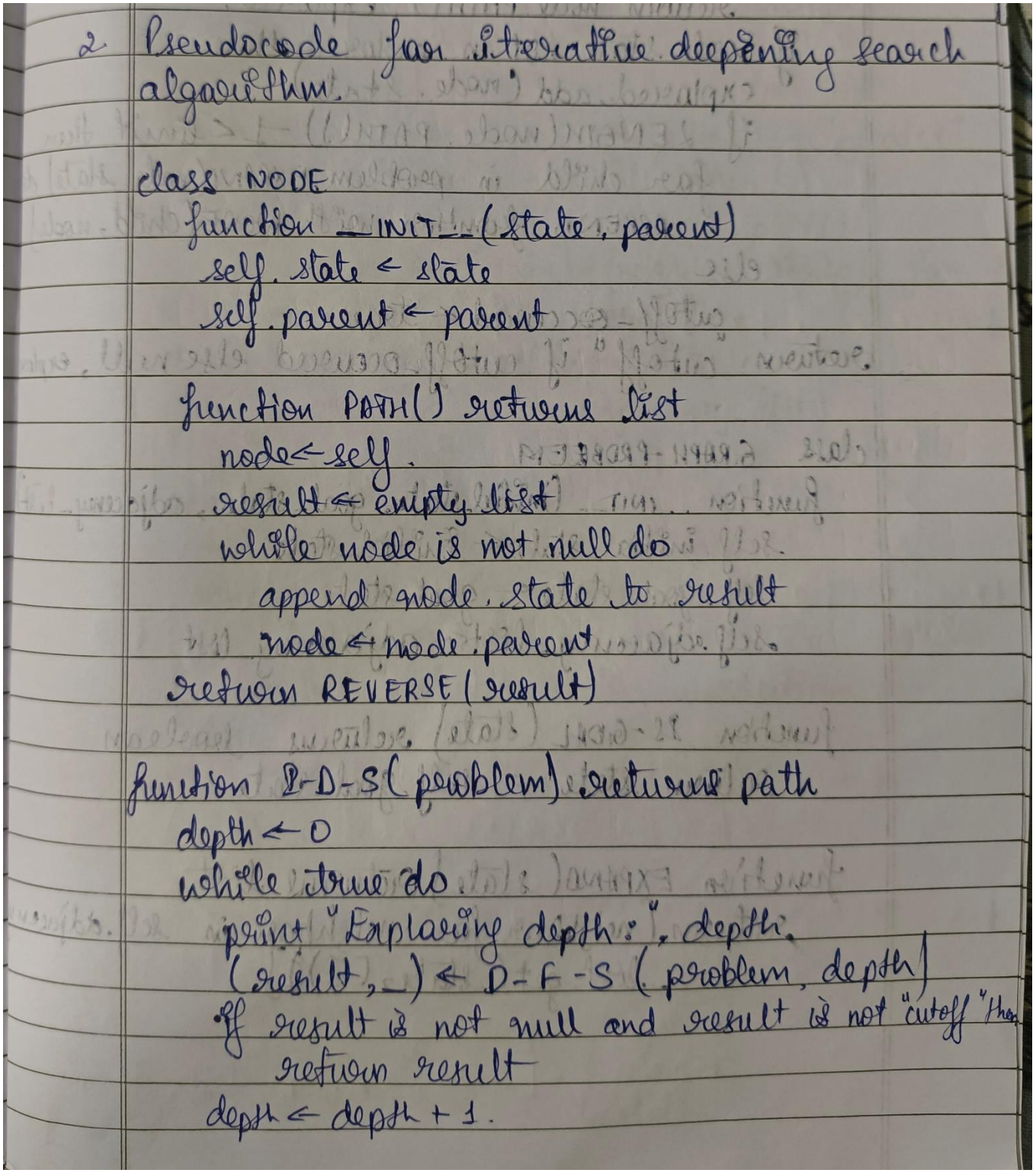
```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 1 2 3
Enter row 2 (space-separated): 4  6 5
Enter row 3 (space-separated): 8 7 0
Initial State:
[1, 2, 3]
[4, 6, 5]
[8, 7, 0]


Solving using DFS:
Exploring state in DFS:
[1, 2, 3]
[4, 6, 5]
[8, 7, 0]


Exploring state in DFS:
[1, 2, 3]
[4, 6, 5]
[8, 0, 7]


Exploring state in DFS:
[1, 2, 3]
[4, 6, 5]
[0, 8, 7]


Exploring state in DFS:
[1, 2, 3]
[0, 6, 5]
[4, 8, 7]
```

```
 Exploring state in DFS:
 [1, 2, 3]
 [4, 8, 5]
 [7, 0, 6]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 8, 5]
 [7, 6, 0]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 0, 5]
 [7, 8, 6]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 5, 0]
 [7, 8, 6]


 Exploring state in DFS:
 [1, 2, 3]
 [4, 5, 6]
 [7, 8, 0]


 DFS Solution: ['left', 'left', 'up', 'right', 'right', 'down', 'left', 'le
 Time taken by DFS: 0.183756 seconds
```

LABORATORY PROGRAM – 3

IMPLEMENT ITERATIVE DEEPENING SEARCH ALGORITHM

PSEUDOCODE OR ALGORITHM



2. Pseudocode for iterative deepening search algorithm.

```
class NODE
    function __INIT__(state, parent)
        self.state ← state
        self.parent ← parent

    function PATH() returns list
        node ← self.
        result ← empty list
        while node is not null do
            append node.state to result
            node ← node.parent
        return REVERSE(result)

function I-D-S(problem) returns path
    depth ← 0
    while true do
        print "Exploring depth:", depth.
        (result, _) ← D-F-S(problem, depth)
        if result is not null and result is not "cutoff" then
            return result
        depth ← depth + 1.
```

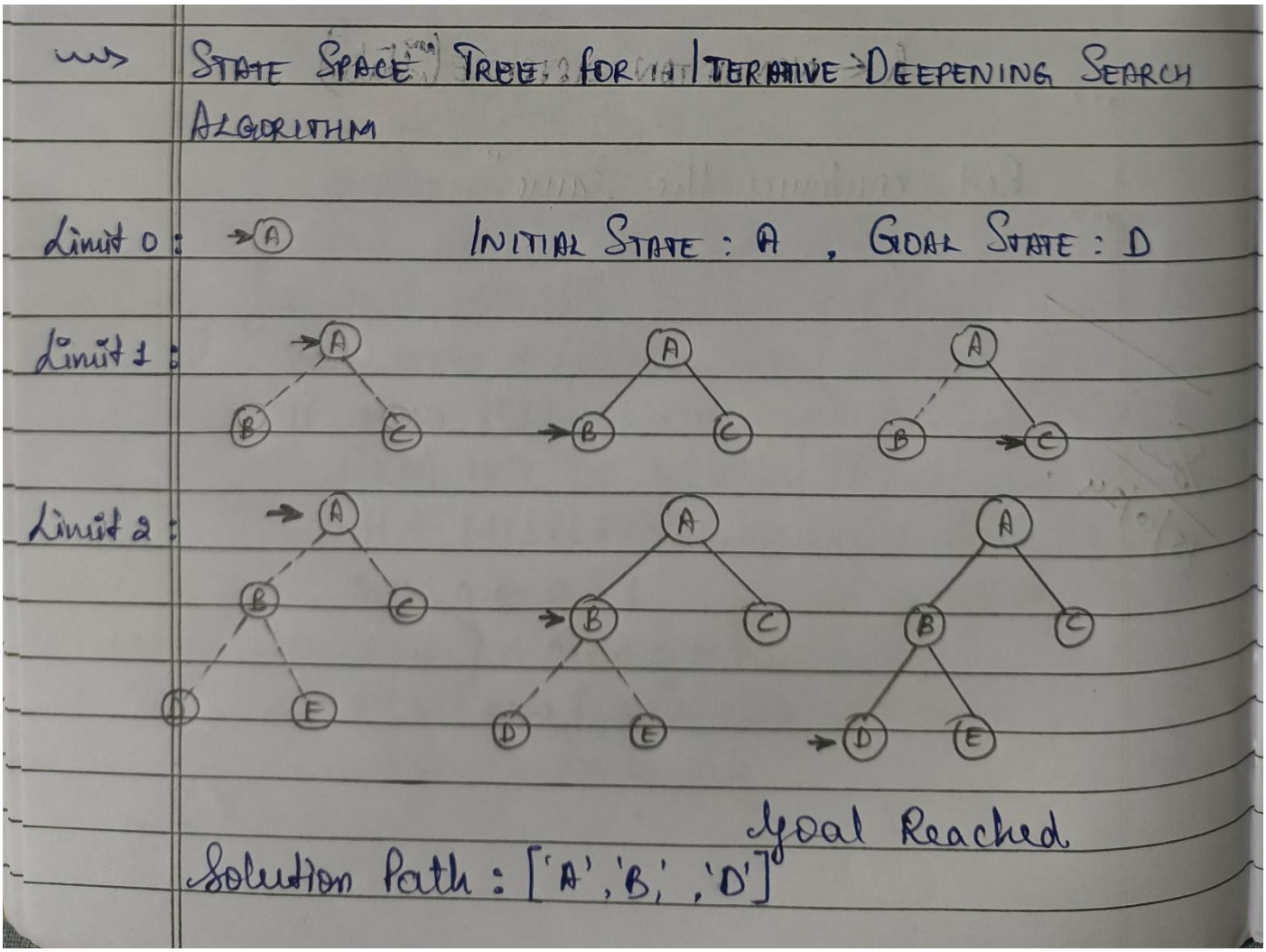
DATE: PAGE:

```
function D-F-S(problem, limit) returns path or cutoff
    frontier ← [NODE(problem.initial_state)]
    explored ← empty_set
    cutoff_occurred ← false
    while frontier is not empty do
        node ← POP(frontier)
        if problem.IS-GOAL(node.state) then
            return node.PATH(), explored.
        if node.state not in explored then
            explored.add(node.state)
            if LENGTH(node.PATH()) - 1 < limit then
                for child in problem.EXPAND(node.state) do
                    APPEND frontier with NODE(child, node)
            else
                cutoff_occurred ← true
    return "cutoff" if cutoff_occurred else null, explored

class GRAPH-PROBLEM
    function __INIT__(initial_state, goal_state, adjacency_list)
        self.initial_state ← initial_state
        self.goal_state ← goal_state
        self.adjacency_list ← adjacency_list

    function IS-GOAL(state) returns boolean
        return state == self.goal_state

    function EXPAND(state) returns list,
        return [neighbor for neighbor in self.adjacency
            _list.get(state, [])].
```

DATE: PAGE:

```
function GET-GRAPH-FROM-INPUT() returns GRAPH-PROBLEM
    adjacency_list ← empty_dictionary
    initial_state ← INPUT("Enter the initial state:").strip()
    goal_state ← INPUT("Enter the goal state:").strip()
    print "Enter the AD list for graph."
    print "Type 'done' when finished."
    while true do
        node ← INPUT("Enter node:").strip()
        if node.lower() == "done" then
            break
        neighbors_input ← INPUT("Enter N of "+node+"
        separated by spaces:").strip()
        neighbors ← neighbors_input.split()
        adjacency_list[node] ← [neighbor.strip() for
            neighbor in neighbors]
    return GRAPH-PROBLEM(initial_state, goal_state,
                         adjacency_list)

main
    problem ← GET-GRAPH-FROM-INPUT()
    solution ← I-D-S(problem)
    if solution is not null then
        print "Solution Path:", solution
    else
        print "No solution found."
```

STATE SPACE TREE



CODE

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

    def path(self):
        node, result = self, []
        while node:
            result.append(node.state)
            node = node.parent
        return result[::-1]

def iterative_deepening_search(problem):
    depth = 0
    while True:
        print(f"Exploring depth: {depth}")
        result, _ = depth_limited_search(problem, depth)
        if result is not None and result != 'cutoff':
            return result
        depth += 1

def depth_limited_search(problem, limit):
    frontier = [Node(problem.initial_state)]
    explored = set()
    cutoff_occurred = False

    while frontier:
        node = frontier.pop()

        if problem.is_goal(node.state):
            return node.path(), explored

        if node.state not in explored:
            explored.add(node.state)
            if len(node.path()) - 1 < limit:
                for child in problem.expand(node.state):
                    frontier.append(Node(child, node))
            else:
                cutoff_occurred = True

    return 'cutoff' if cutoff_occurred else None, explored

class GraphProblem:
    def __init__(self, initial_state, goal_state,
adjacency_list):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.adjacency_list = adjacency_list

    def is_goal(self, state):
        return state == self.goal_state

    def expand(self, state):
        return [neighbor for neighbor in
self.adjacency_list.get(state, [])]

def get_graph_from_input():
    adjacency_list = {}
    initial_state = input("Enter the initial state: ").strip()
    goal_state = input("Enter the goal state: ").strip()

    print("Enter the adjacency list for the graph
(neighbors of each node).")
    print("Type 'done' when finished.")

    while True:
        node = input("Enter node (or 'done' to finish):
").strip()
        if node.lower() == 'done':
            break
        neighbors_input = input(f"Enter neighbors of
{node} separated by spaces: ").strip()
        neighbors = neighbors_input.split()
        adjacency_list[node] = [neighbor.strip() for
neighbor in neighbors]

    return GraphProblem(initial_state, goal_state,
adjacency_list)

if __name__ == "__main__":
    problem = get_graph_from_input()
    solution = iterative_deepening_search(problem)

    if solution:
        print("Solution Path:", solution)
    else:
        print("No solution found.")
```

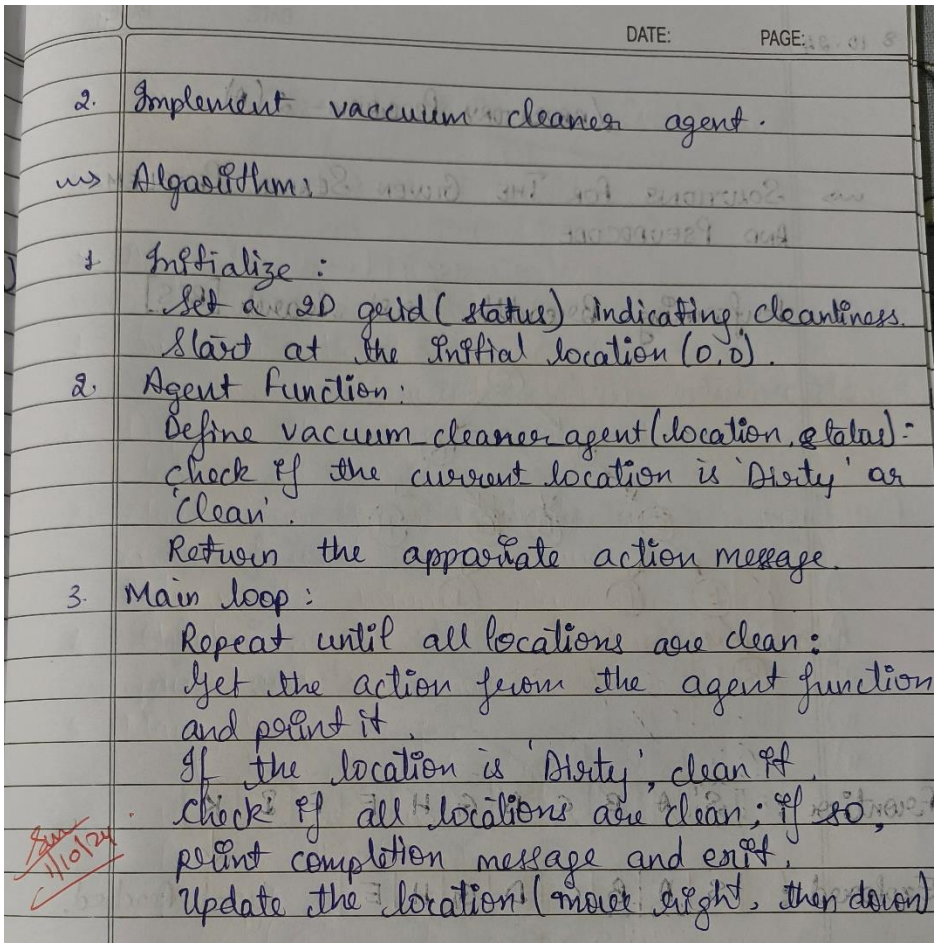
OUTPUT

```
Enter the initial state: A
Enter the goal state: G
Enter the adjacency list for the graph (neighbors of each node).
Type 'done' when finished.
Enter node (or 'done' to finish): A
Enter neighbors of A separated by spaces: B C
Enter node (or 'done' to finish): B
Enter neighbors of B separated by spaces: D E
Enter node (or 'done' to finish): D
Enter neighbors of D separated by spaces: H I
Enter node (or 'done' to finish): C
Enter neighbors of C separated by spaces: F G
Enter node (or 'done' to finish): F
Enter neighbors of F separated by spaces: K
Enter node (or 'done' to finish): done
Exploring depth: 0
Exploring depth: 1
Exploring depth: 2
Solution Path: ['A', 'C', 'G']
```

LABORATORY PROGRAM – 4

IMPLEMENT VACUUM CLEANER AGENT

PSEUDOCODE OR ALGORITHM



DATE: PAGE:

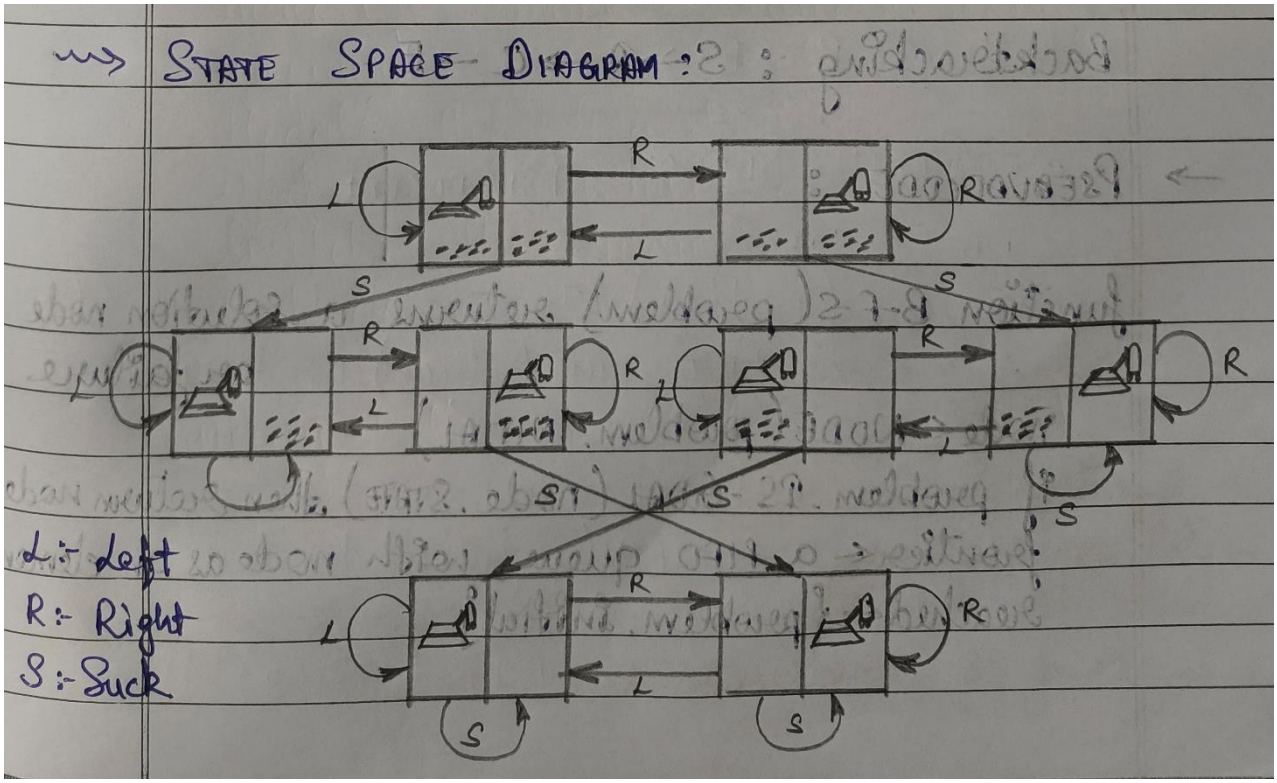
2. Implement vaccuum cleaner agent.

=> Algorithm:

1. Initialize:
   Set a 2D grid (status) indicating cleanliness.
   Start at the initial location (0,0).
2. Agent Function:
   Define vacuum_cleaner_agent(location, status):
   Check if the current location is 'Dirty' or 'Clean'.
   Return the appropriate action message.
3. Main loop:
   Repeat until all locations are clean:
   Get the action from the agent function and print it.
   If the location is 'Dirty', clean it.
   Check if all locations are clean; if so, print completion message and exit.
   Update the location (move right, then down)

11/10/24

STATE SPACE TREE



CODE

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter the initial location of the vacuum cleaner (A or B): ")
    status_input = input(f"Enter the status of room {location_input} (0 for Clean, 1 for Dirty): ")
    status_input_complement = input(f"Enter the status of the other room ({'B' if location_input == 'A' else 'A'}) (0 for Clean, 1 for Dirty): ")

    # Set the initial state based on user input
    initial_state = {
        'A': status_input if location_input == 'A' else status_input_complement,
        'B': status_input_complement if location_input == 'A' else status_input
    }

    print("\nInitial Location Condition:", initial_state)

    if location_input == 'A':
        print("\nVacuum cleaner is placed in room A.")

        if status_input == '1':
            print("Room A is Dirty. Cleaning room A...")
            goal_state['A'] = '0'
            cost += 1
            print("Cost for cleaning room A:", cost)

            if status_input_complement == '1':
                print("\nRoom B is also Dirty. Moving to room B...")
                cost += 1
                print("Cost for moving to room B:", cost)
                print("Cleaning room B...")
                goal_state['B'] = '0'
                cost += 1
                print("Cost for cleaning room B:", cost)
            else:
                print("\nRoom B is already Clean. No further action needed.")

        else:
            print("Room A is already Clean.")

            if status_input_complement == '1':
                print("\nRoom B is Dirty. Moving to room B...")
                cost += 1
                print("Cost for moving to room B:", cost)
                print("Cleaning room B...")
                goal_state['B'] = '0'
                cost += 1
                print("Cost for cleaning room B:", cost)
            else:
                print("\nRoom B is already Clean. No further action needed.")

    else:
        print("\nVacuum cleaner is placed in room B.")

        if status_input == '1':
            print("Room B is Dirty. Cleaning room B...")
            goal_state['B'] = '0'
            cost += 1
            print("Cost for cleaning room B:", cost)

            if status_input_complement == '1':
                print("\nRoom A is also Dirty. Moving to room A...")
                cost += 1
                print("Cost for moving to room A:", cost)
                print("Cleaning room A...")
                goal_state['A'] = '0'
                cost += 1
                print("Cost for cleaning room A:", cost)
            else:
                print("\nRoom A is already Clean. No further action needed.")

        else:
            print("Room B is already Clean.")

            if status_input_complement == '1':
                print("\nRoom A is Dirty. Moving to room A...")
                cost += 1
                print("Cost for moving to room A:", cost)
                print("Cleaning room A...")
                goal_state['A'] = '0'
                cost += 1
                print("Cost for cleaning room A:", cost)
            else:
                print("\nRoom A is already Clean. No further action needed.")

    print("\nGOAL STATE:", goal_state)
    print("Total cost for cleaning:", cost)

vacuum_world()
```

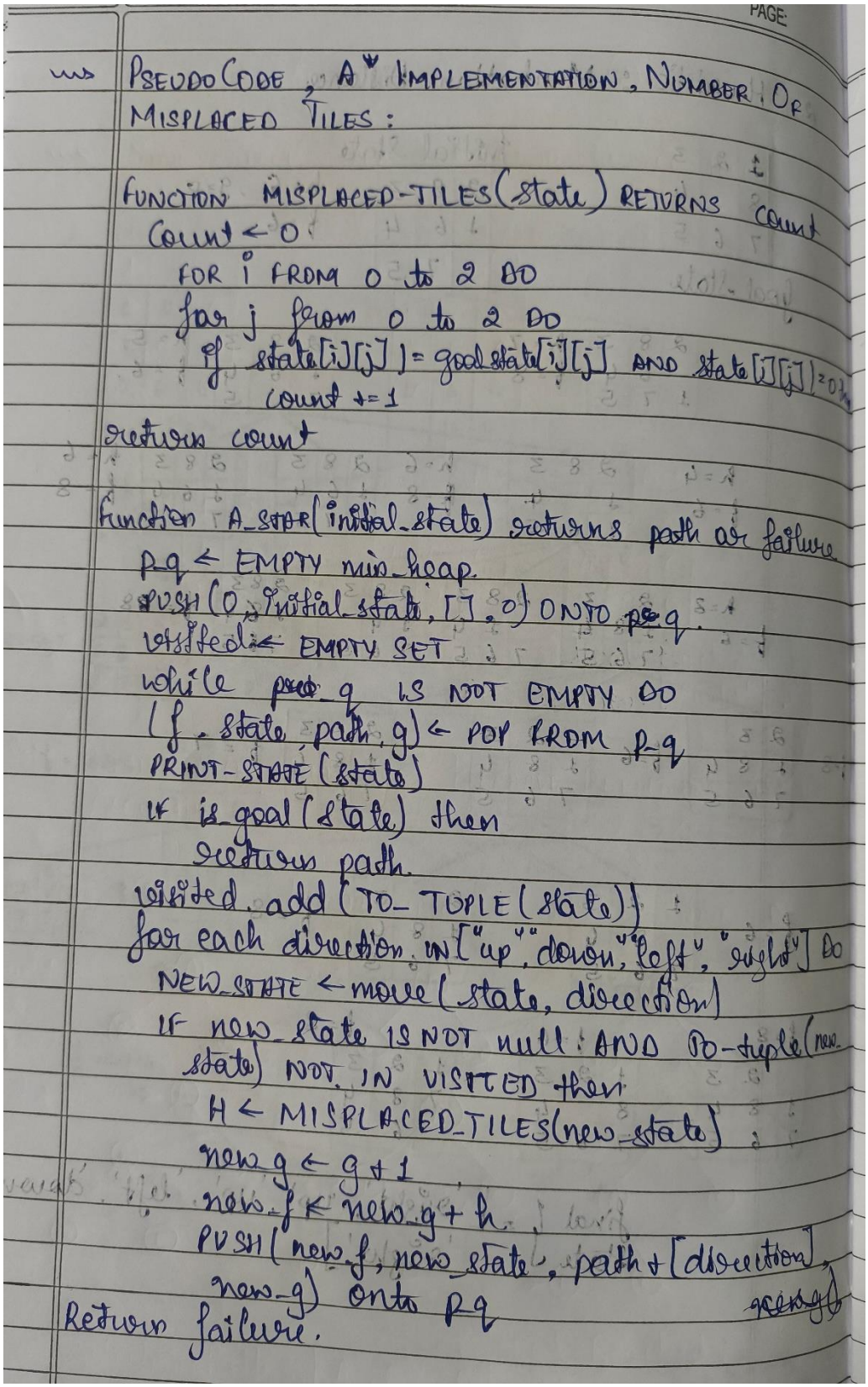
OUTPUT

```
Enter the initial location of the vacuum cleaner (A or B): A
Enter the status of room A (0 for Clean, 1 for Dirty): 1
Enter the status of the other room (B) (0 for Clean, 1 for Dirty): 1

Initial Location Condition: {'A': '1', 'B': '1'}

Vacuum cleaner is placed in room A.
Room A is Dirty. Cleaning room A...
Cost for cleaning room A: 1

Room B is also Dirty. Moving to room B...
Cost for moving to room B: 2
Cleaning room B...
Cost for cleaning room B: 3

GOAL STATE: {'A': '0', 'B': '0'}
Total cost for cleaning: 3
```

LABORATORY PROGRAM – 5

IMPLEMENT A* SEARCH ALGORITHM

PSEUDOCODE OR ALGORITHM



PSEUDO CODE A* IMPLEMENTATION, NUMBER OF MISPLACED TILES:

```
FUNCTION MISPLACED-TILES(state) RETURNS count
  Count ← 0
    FOR i FROM 0 to 2 DO
    for j from 0 to 2 DO
      if state[i][j] = goal_state[i][j] AND state[i][j] ≠ 0
          count += 1
  return count

function A_STAR(initial_state) returns path or failure
  pq ← EMPTY min_heap
  PUSH (0, initial_state, [], 0) ONTO pq
  visited ← EMPTY SET
  while pq IS NOT EMPTY DO
  (f, state, path, g) ← POP FROM pq
  PRINT-STATE (state)
  IF is_goal (state) then
     return path
  visited.add (TO_TUPLE (state))
  for each direction in ["up", "down", "left", "right"] DO
    NEW_STATE ← move (state, direction)
    IF new_state IS NOT null AND to-tuple(new_state) NOT IN VISITED then
      H ← MISPLACED_TILES(new_state)
      new_g ← g + 1
      new_f ← new_g + h
      PUSH (new_f, new_state, path + [direction], new_g) onto pq
Return failure.
```

DATE: PAGE:

PSEUDOCODE, A* IMPLEMENTATION, MANHATTAN DISTANCE

```
Function MANHATTAN-DISTANCE(state) returns distance
    Distance ← 0
    for i from 0 to 2 do
        for j from 0 to 2 do
            if state[i][j] != 0 then
                goal_i, goal_j ← position of state[i][j] in goal state
                if state[i][j] == 8 then
                    goal_i, goal_j ← 1, 1
                Distance += ABS(goal_i - i) + ABS(goal_j - j)
    return distance
```

Function A_STAR(initial_state) returns path or failure

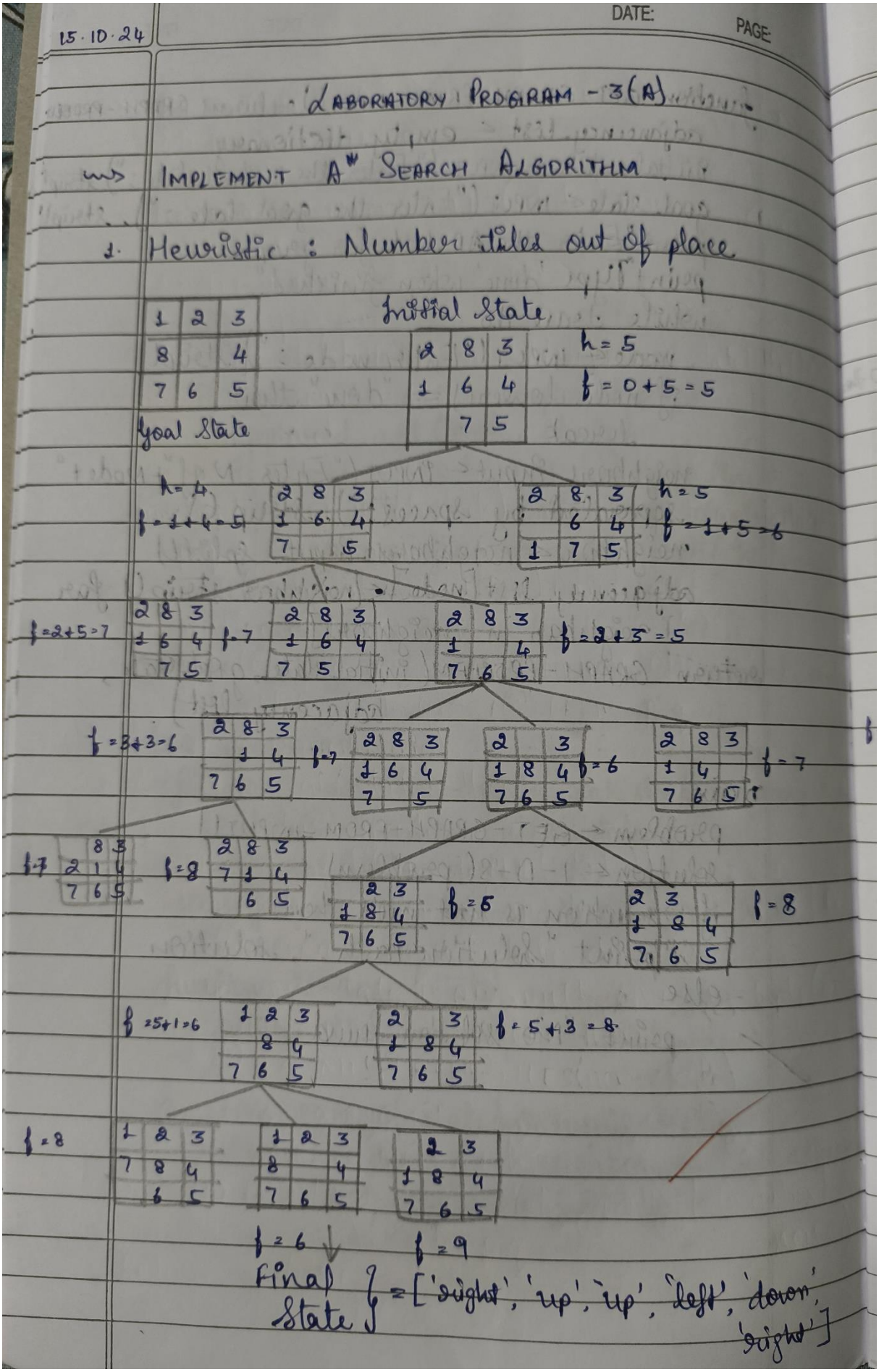
Similar to the pseudocode of misplaced tiles, but change in heuristic function

```
h ← MANHATTAN-DISTANCE(state)
```

Rest continues the same

15/10/24

STATE SPACE TREE



DATE: PAGE:

2. Heuristic : Manhattan Distance

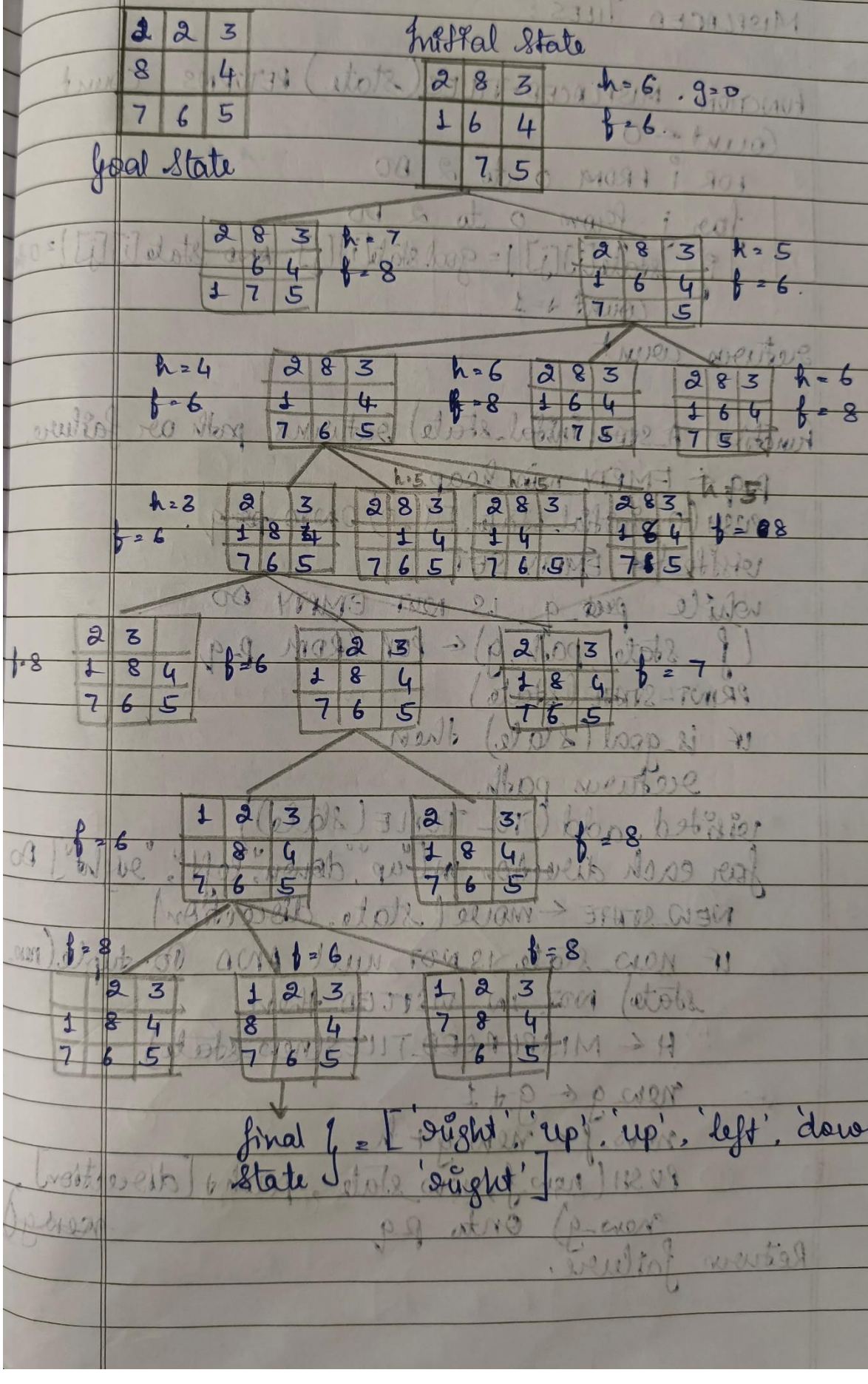
Goal State

1	2	3
8		4
7	6	5

Initial State

2	8	3
1	6	4
	7	5

h=6, g=0, f=6



CODE - G(N): DEPTH OF THE NODE, H(N): NUMBER OF MISPLACED TILES

```
import heapq

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-
1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] =
new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-
1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] =
new_state[i][j+1], new_state[i][j]
    else:
        return None
    return new_state

def is_goal(state):
    return state == goal_state

def misplaced_tiles(state):
    return sum(1 for i in range(3) for j in range(3)
               if state[i][j] != goal_state[i][j] and state[i][j]
!= 0)

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [],
0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(tuple(map(tuple, state)))

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state))
not in visited:
                h = misplaced_tiles(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f,
new_state, path + [direction], new_g))

    return None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for
empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1}
(space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain
exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
```

OUTPUT - G(N): DEPTH OF THE NODE, H(N): NUMBER OF MISPLACED TILES

```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

Exploring state in A*:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

CODE - G(N): DEPTH OF THE NODE, H(N): MANHATTAN DISTANCE

```
import heapq

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

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return (i, j)

def move(state, direction):
    new_state = [row[:] for row in state]
    i, j = find_zero(state)
    if direction == "up" and i > 0:
        new_state[i][j], new_state[i-1][j] = new_state[i-
1][j], new_state[i][j]
    elif direction == "down" and i < 2:
        new_state[i][j], new_state[i+1][j] =
new_state[i+1][j], new_state[i][j]
    elif direction == "left" and j > 0:
        new_state[i][j], new_state[i][j-1] = new_state[i][j-
1], new_state[i][j]
    elif direction == "right" and j < 2:
        new_state[i][j], new_state[i][j+1] =
new_state[i][j+1], new_state[i][j]
    else:
        return None
    return new_state

def is_goal(state):
    return state == goal_state

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j] - 1, 3)
                if state[i][j] == 8:
                    goal_i, goal_j = 1, 1
                distance += abs(goal_i - i) + abs(goal_j - j)
    return distance

def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, [],
0))
    visited = set()

    while priority_queue:
        f, state, path, g = heapq.heappop(priority_queue)
        print("Exploring state in A*:")
        print_state(state)

        if is_goal(state):
            return path

        visited.add(tuple(map(tuple, state)))

        for direction in ["up", "down", "left", "right"]:
            new_state = move(state, direction)
            if new_state and tuple(map(tuple, new_state))
not in visited:
                h = manhattan_distance(new_state)
                new_g = g + 1
                new_f = new_g + h
                heapq.heappush(priority_queue, (new_f,
new_state, path + [direction], new_g))

    return None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_initial_state():
    print("Enter the initial state of the 8-puzzle (0 for
empty space):")
    initial_state = []
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1}
(space-separated): ").strip().split()))
        if len(row) != 3:
            raise ValueError("Each row must contain
exactly 3 numbers.")
        initial_state.append(row)
    return initial_state

if __name__ == "__main__":
    initial_state = get_initial_state()
    print("Initial State:")
    print_state(initial_state)
    print("Solving using A* search:")
    a_star_solution = a_star(initial_state)

    if a_star_solution:
        print("A* Solution:", a_star_solution)
    else:
        print("No solution found with A*.")
```

OUTPUT - G(N): DEPTH OF THE NODE, H(N): MANHATTAN DISTANCE

```
Enter the initial state of the 8-puzzle (0 for empty space):
Enter row 1 (space-separated): 2 8 3
Enter row 2 (space-separated): 1 6 4
Enter row 3 (space-separated): 0 7 5
Initial State:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Solving using A* search:
Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[0, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

Exploring state in A*:
[2, 8, 3]
[1, 6, 4]
[7, 5, 0]

Exploring state in A*:
[2, 8, 3]
[0, 6, 4]
[1, 7, 5]

Exploring state in A*:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
```

```
Exploring state in A*:
[2, 8, 3]
[1, 5, 6]
[7, 4, 0]

Exploring state in A*:
[0, 8, 3]
[2, 6, 4]
[1, 7, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 4]
[0, 6, 5]

Exploring state in A*:
[1, 2, 3]
[7, 8, 6]
[0, 5, 4]

Exploring state in A*:
[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* Solution: ['right', 'up', 'up', 'left', 'down', 'right']
```

LIKHITH M (1BM22CS135)