Lab 2: Vacuum Cleaner and 8 puzzle using BFS,DFS

1. **Vacuum Cleaner**

**Input:**

print("USN: 1BM22CS332")

print("V KARUNESHWAR REDDY")

class VacuumCleaner:

def \_\_init\_\_(self):

# Initialize places A and B as either 'Dirty' or 'Clean'

self.places = {'A': 'Dirty', 'B': 'Dirty'}

# Start the vacuum cleaner at place A

self.current\_position = 'A'

def check(self):

# Check if the current position is dirty

if self.places[self.current\_position] == 'Dirty':

print(f"Place {self.current\_position} is Dirty.")

return True

else:

print(f"Place {self.current\_position} is Clean.")

return False

def suck(self):

# Clean the current position if it's dirty

if self.check():

print(f"Cleaning place {self.current\_position}.")

self.places[self.current\_position] = 'Clean'

else:

print(f"Place {self.current\_position} is already clean.")

def move(self):

# Move to the other place

self.current\_position = 'B' if self.current\_position == 'A' else 'A'

print(f"Moving to place {self.current\_position}.")

def start\_cleaning(self):

# Start the cleaning process

for \_ in range(2): # Loop twice to cover both places

self.suck() # Clean the current position if dirty

self.move() # Move to the other position

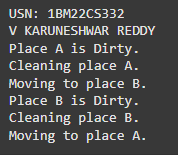
# Create a vacuum cleaner instance

vacuum = VacuumCleaner()

# Start the cleaning process

vacuum.start\_cleaning()

**Output:**



**B) 8 puzzle problem**

**i) BFS**

**Input:**

print("1BM22CS332")

print("V KARUNESHWAR REDDY")

from collections import deque

def solve\_8puzzle\_bfs(initial\_state):

"""

Solves the 8-puzzle using Breadth-First Search.

Args:

initial\_state: A list of lists representing the initial state of the puzzle.

Returns:

A list of lists representing the solution path, or None if no solution is found.

"""

def find\_blank(state):

"""Finds the row and column of the blank tile (0)."""

for row in range(3):

for col in range(3):

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

return row, col

def get\_neighbors(state):

"""Generates possible neighbor states by moving the blank tile."""

row, col = find\_blank(state)

neighbors = []

# Possible moves: Up, Down, Left, Right

if row > 0: # Up

new\_state = [r[:] for r in state]

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

neighbors.append(new\_state)

if row < 2: # Down

new\_state = [r[:] for r in state]

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

neighbors.append(new\_state)

if col > 0: # Left

new\_state = [r[:] for r in state]

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

neighbors.append(new\_state)

if col < 2: # Right

new\_state = [r[:] for r in state]

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

neighbors.append(new\_state)

return neighbors

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

# Print initial and goal states

print("Initial State:")

for row in initial\_state:

print(row)

print("\nGoal State:")

for row in goal\_state:

print(row)

print("\nStarting BFS...\n")

queue = deque([(initial\_state, [])])

visited = set()

while queue:

current\_state, path = queue.popleft()

# Check if the goal state is reached

if current\_state == goal\_state:

return path + [current\_state]

# Mark the current state as visited

visited.add(tuple(map(tuple, current\_state)))

# Explore neighbors

for neighbor in get\_neighbors(current\_state):

if tuple(map(tuple, neighbor)) not in visited:

queue.append((neighbor, path + [current\_state]))

return None # No solution found

# Example usage:

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

solution = solve\_8puzzle\_bfs(initial\_state)

if solution:

print("\nSolution found:")

for state in solution:

for row in state:

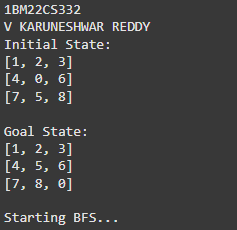
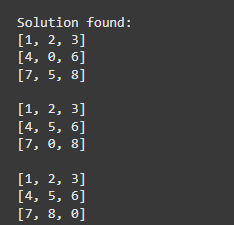
print(row)

print()

else:

print("No solution found.")

**Output:**

**ii) DFS**

**Input:**

print("1BM22CS332")

print("V KARUNESHWAR REDDY")

from collections import deque

def solve\_8puzzle\_dfs(initial\_state):

"""

Solves the 8-puzzle using Depth-First Search.

Args:

initial\_state: A list of lists representing the initial state of the puzzle.

Returns:

A list of lists representing the solution path, or None if no solution is found.

"""

def find\_blank(state):

"""Finds the row and column of the blank tile (0)."""

for row in range(3):

for col in range(3):

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

return row, col

def get\_neighbors(state):

"""Generates possible neighbor states by moving the blank tile."""

row, col = find\_blank(state)

neighbors = []

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right

for dr, dc in directions:

new\_row, new\_col = row + dr, col + dc

if 0 <= new\_row < 3 and 0 <= new\_col < 3:

new\_state = [r[:] for r in state]

new\_state[row][col], new\_state[new\_row][new\_col] = new\_state[new\_row][new\_col], new\_state[row][col]

neighbors.append(new\_state)

return neighbors

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

# Print initial and goal states

print("Initial State:")

for row in initial\_state:

print(row)

print("\nGoal State:")

for row in goal\_state:

print(row)

print("\nStarting DFS...\n")

stack = [(initial\_state, [])]

visited = set()

while stack:

current\_state, path = stack.pop()

# Convert state to tuple for easy set comparison

state\_tuple = tuple(map(tuple, current\_state))

# Skip if already visited

if state\_tuple in visited:

continue

# Mark as visited

visited.add(state\_tuple)

# Check if the goal state is reached

if current\_state == goal\_state:

return path + [current\_state]

# Explore neighbors

for neighbor in get\_neighbors(current\_state):

stack.append((neighbor, path + [current\_state]))

return None # No solution found

# Example usage:

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

solution = solve\_8puzzle\_dfs(initial\_state)

if solution:

print("\nSolution found:")

for state in solution:

for row in state:

print(row)

print()

else:

print("No solution found.")

**Output:**

