MDS573C – Image and Video Analytics

`Lab Exercise 1`

- Assume a binary image size 5 x 5 and perform the following: Let V ={1} be the set of intensity value to define the adjacency.
- (a) Find the following digital paths and print the path traversal from any source pixel ‘p’ to any other source pixel ‘q’ 4-Path, 8-Path and m-Path
- (b) Find all the regions present in the image and declare the adjacent regions and disjoint regions.

`Definitions:`

- 4-Path: Two pixels are adjacent if they are horizontally or vertically neighbors.
- 8-Path: Two pixels are adjacent if they are neighbors horizontally, vertically, or diagonally.
- m-Path: It's a mixed path, which combines both 4-path and 8-path depending on the scenario (generally avoids ambiguities that arise from diagonal adjacency in certain configurations).

`Regions:`

- Regions are groups of connected pixels based on adjacency (4-path or 8-path).
- Disjoint regions are separate groups of connected pixels.

In [1]:

```
import numpy as np
from collections import deque

# Define a binary image (5x5) where 1 is part of the region and 0
image = np.array([[1, 0, 0, 1, 0],
                  [1, 1, 0, 0, 0],
                  [0, 0, 0, 1, 1],
                  [0, 1, 1, 0, 1],
                  [0, 0, 1, 0, 0]])

# Define possible movement directions for 4-path and 8-path
four_path_moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down,
eight_path_moves = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1),

# Function to check if a pixel is within bounds and is part of th
def is_valid(image, x, y, visited):
    return 0 <= x < image.shape[0] and 0 <= y < image.shape[1] an

# Breadth-First Search for finding the path
def bfs(image, start, end, moves):
    queue = deque([start])
    visited = np.zeros_like(image, dtype=bool)
    visited[start[0], start[1]] = True
    parent = {start: None}

    while queue:
        current = queue.popleft()
        if current == end:
            # Reconstruct path
            path = []
            while current is not None:
                path.append(current)
                current = parent[current]
            path.reverse()
            return path

        for move in moves:
            new_x, new_y = current[0] + move[0], current[1] + mov
            if is_valid(image, new_x, new_y, visited):
                visited[new_x, new_y] = True
                parent[(new_x, new_y)] = current
                queue.append((new_x, new_y))

    return None  # No path found

# Find regions using DFS
def find_regions(image, moves):
    visited = np.zeros_like(image, dtype=bool)
    regions = []

    def dfs(x, y, region):
        stack = [(x, y)]
        visited[x, y] = True
        region.append((x, y))

        while stack:
            current_x, current_y = stack.pop()
            for move in moves:
                new_x, new_y = current_x + move[0], current_y + m
                if is_valid(image, new_x, new_y, visited):
                    visited[new_x, new_y] = True
```

```
                    stack.append((new_x, new_y))
                    region.append((new_x, new_y))

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):
            if image[i, j] == 1 and not visited[i, j]:
                region = []
                dfs(i, j, region)
                regions.append(region)

    return regions

# Part (a): Finding the paths
start_pixel = (0, 0)  # Source pixel 'p'
end_pixel = (3, 2)    # Destination pixel 'q'

# 4-Path traversal
four_path = bfs(image, start_pixel, end_pixel, four_path_moves)
print(f"4-Path from {start_pixel} to {end_pixel}: {four_path}")

# 8-Path traversal
eight_path = bfs(image, start_pixel, end_pixel, eight_path_moves)
print(f"8-Path from {start_pixel} to {end_pixel}: {eight_path}")

# Mixed Path (m-Path)
m_path = bfs(image, start_pixel, end_pixel, four_path_moves + eig
print(f"m-Path from {start_pixel} to {end_pixel}: {m_path}")

# Part (b): Finding regions and identifying adjacent and disjoint
four_regions = find_regions(image, four_path_moves)
eight_regions = find_regions(image, eight_path_moves)

print(f"4-Path Regions: {four_regions}")
print(f"8-Path Regions: {eight_regions}")

# Declare adjacent and disjoint regions
def region_adjacency(regions, moves):
    adjacent_regions = []
    for i, region1 in enumerate(regions):
        for j, region2 in enumerate(regions):
            if i != j:
                for (x1, y1) in region1:
                    for move in moves:
                        x2, y2 = x1 + move[0], y1 + move[1]
                        if (x2, y2) in region2:
                            adjacent_regions.append((i, j))
                            break
                    else:
                        continue
                    break
    return adjacent_regions

four_adjacent_regions = region_adjacency(four_regions, four_path_
eight_adjacent_regions = region_adjacency(eight_regions, eight_pa

print(f"4-Path Adjacent Regions: {four_adjacent_regions}")
print(f"8-Path Adjacent Regions: {eight_adjacent_regions}")
```

```
4-Path from (0, 0) to (3, 2): None
8-Path from (0, 0) to (3, 2): None
m-Path from (0, 0) to (3, 2): None
4-Path Regions: [[(0, 0), (1, 0), (1, 1)], [(0, 3)], [(2, 3), (2,
4), (3, 4)], [(3, 1), (3, 2), (4, 2)]]
8-Path Regions: [[(0, 0), (1, 0), (1, 1)], [(0, 3)], [(2, 3), (2,
4), (3, 2), (3, 4), (3, 1), (4, 2)]]
4-Path Adjacent Regions: []
8-Path Adjacent Regions: []
```

`Explanation:`

- `Paths:`
- We use Breadth-First Search (BFS) to find the shortest path between two points for both 4-path, 8-path, and m-path.
- The traversal is done using defined neighbor movements for each type of path.
- `Regions:`
- We identify regions using Depth-First Search (DFS). Regions are groups of connected pixels that share a common adjacency relation (either 4-path or 8-path).
- Adjacent and disjoint regions are determined by checking if any pixel in one region is adjacent to any pixel in another region based on the allowed movements (4-path or 8-path).

`Output:`

The code will output:

- The digital paths (4-path, 8-path, m-path) from a source pixel p to another pixel q.
- The regions in the image based on both 4-path and 8-path.
- The adjacency relations between regions based on their neighbors.

Interpretation and Analysis of the Results:

`Part (a) - Digital Paths`

- 4-Path: The 4-path restricts movement between pixels to horizontal and vertical directions (up, down, left, right). This type of path ensures that only adjacent pixels in the strictest sense (not diagonal) are considered connected.
- `Interpretation:` The 4-path tends to result in a longer traversal for most images because it excludes diagonal moves, limiting the direct connectivity between certain pixels. This constraint makes the paths more step-by-step, which can increase the distance between pixels, especially if they are diagonally separated in the image.
- `Key Insight:` 4-path adjacency is useful when diagonal connections are undesirable (e.g., in certain medical images, where the definition of a region needs to be conservative).
- `8-Path:` The 8-path expands the idea of connectivity to include diagonal movements. This makes the path between two pixels shorter in many cases because diagonal moves can help traverse the image more efficiently.
- `Interpretation:` With 8-path, the traversal tends to be faster, and the path is shorter because diagonal neighbors provide more flexibility. It mimics a more natural connectivity (as we would expect in most real-life images).

- `Key Insight:` The 8-path is particularly useful in scenarios where diagonal connectivity is relevant or acceptable, such as when segmenting complex objects in images, as it can capture more pixel connections.
- `m-Path:` The m-path (mixed-path) combines both 4-path and 8-path, and is typically used to resolve situations where diagonal paths can introduce ambiguity. This path offers more flexibility while avoiding unnecessary diagonal connections that could merge regions that should remain separate.
- `Interpretation:` The m-path can adapt to different situations, using diagonal moves when appropriate but avoiding them if they would create visual or structural issues. It attempts to balance the strictness of 4-path and the flexibility of 8-path.
- `Key Insight:` The m-path is useful when we want to avoid ambiguities in segmentation tasks but still need flexibility in pixel connectivity.

`Part (b) - Regions`

- `Region Identification:` In our binary image, we identified regions as connected sets of pixels with intensity value 1. These regions depend on the type of path used to define connectivity (4-path or 8-path).

- `4-Path Regions:`

Regions are generally smaller and more fragmented because of the restricted connectivity rules. This can result in multiple small regions being defined, even if they are diagonally connected.

- `8-Path Regions:` Regions are more likely to be larger and more unified since diagonal connections are allowed. As a result, pixels that might have been separated in the 4-path regions are now part of the same region.
- `Example:`

In the provided image, for 4-path, regions like the cluster of 1s near the bottom left and the cluster near the top right might be considered separate regions. For 8-path, these clusters might be connected diagonally, resulting in a single larger region.

- `Key Insight:` In practice, 8-path regions are often more relevant in applications where diagonal connections between objects are meaningful (e.g., detecting connected objects in natural images). However, for applications like grid-like data (city layouts, pixelated structures), 4-path regions provide more precise results.
- `Adjacent and Disjoint Regions:` After identifying the regions, we determine whether two regions are adjacent or disjoint. Adjacent regions share a boundary pixel, while disjoint regions do not.
- `Interpretation:` In real-world applications, knowing whether two regions are adjacent can help in merging regions or determining relationships between objects. For instance, in image processing tasks, adjacent regions might represent parts of a single larger object that were separated due to noise or other factors.
- `Key Insight:` Identifying adjacent and disjoint regions is critical for tasks like region

In [ ]:

```
1
```