Heaven's Light is Our Guide

Rajshahi University of Engineering & Technology

Department of

Electrical & Computer Engineering

Lab Report 3

Pathfinding in Binary Images: A Comparative Analysis of Pixel Connectivity

Course Code: ECE 4224

Course Title: Digital Image Processing Sessional

Submitted to:
Oishi Jyoti
Assistant Professor
Dept of ECE, RUET

Submitted by:
Md. Tajim An Noor
Roll: 2010025
Semester: 4th Year Even

Submission Date: January 27, 2026

Contents

Pixel-Level Manipulation and Histogram Analysis

1 Theory and Introduction

In digital image processing, a fundamental operation is establishing relationships between pixels to define regions and boundaries. This relationship is governed by **connectivity**, which determines whether two pixels are considered "neighbors" [1]. Finding the shortest path between two pixels is essential for tasks such as object labeling, region growing, and morphological processing. This experiment investigates three types of connectivity to determine the shortest path between a Start point $S(x, y)$ and an End point $E(x, y)$ in a binary image:

- **4-Connectivity (N_4):** Two pixels are connected if they share a horizontal or vertical edge. Coordinates: $(x \pm 1, y)$ and $(x, y \pm 1)$.
- **8-Connectivity (N_8):** Two pixels are connected if they share an edge or a corner (diagonal). Includes all neighbors in N_4 plus $(x \pm 1, y \pm 1)$.
- **m-Connectivity (N_m):** A modification of 8-connectivity designed to eliminate ambiguous multiple paths. Two pixels p and q are m-connected if:
  1. q is in $N_4(p)$, OR
  2. q is in the diagonal neighborhood $N_D(p)$ **AND** their common 4-neighbors are empty (value 0) [1].

To find the optimal path, the Breadth-First Search (BFS) algorithm is utilized. Since the distance between adjacent pixels is uniform (1 step), BFS guarantees finding the shortest path by exploring the image layer-by-layer [2].

2 Methodology

2.1 Problem Definition

A 4×4 binary image matrix I is analyzed, where 1 represents the object and 0 represents the background.

$$I = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Task: The shortest path from coordinate $(1, 1)$ to $(4, 4)$ (using 1-based indexing) is to be found for all three connectivity types.

2.2 Algorithm Implementation

The Python implementation follows these steps:

1. **Initialization:** The image matrix is defined and coordinates are converted to 0-based indexing ($Start = (0, 0), End = (3, 3)$).
2. **Neighbor Selection:**
   - For **4-Connectivity**, cardinal directions are checked only.
   - For **8-Connectivity**, cardinal and diagonal directions are checked.
   - For **m-Connectivity**, a specific condition is applied: diagonals are allowed only if the intersection of 4-neighbors is 0.
3. **Pathfinding (BFS):** A queue is initialized with the start node. The current path is iteratively dequeued, valid neighbors are checked, and new extended paths are enqueued until the destination is reached.
4. **Visualization:** The resulting paths are plotted on the matrix grid using Matplotlib.

2.3 Python Code

```python
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
from matplotlib.colors import ListedColormap
import os

os.makedirs("./images/output", exist_ok=True)

# 1. Define Image Matrix (4x4)
image = np.array([[1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0], [0, 1, 0, 1]])

# Define Start and End (User requested 1,1 to 4,4 -> Python 0,0 to 3,3)
START = (0, 0)
END = (3, 3)


# Helper: Check if bounds are valid
def is_valid(x, y, shape):
    return 0 <= x < shape[0] and 0 <= y < shape[1]


# Helper: Get value safely
def get_val(img, x, y):
    if is_valid(x, y, img.shape):
        return img[x, y]
    return 0


# 2. Neighbor Finding Logic
def get_neighbors(p, img, mode):
    x, y = p
    neighbors = []
    rows, cols = img.shape

    # Potential moves # N4: (dx, dy)
    moves_4 = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    # Diagonals: (dx, dy)
    moves_diag = [(-1, -1), (-1, 1), (1, -1), (1, 1)]

    # --- 4-Connectivity ---
    if mode == "4":
        for dx, dy in moves_4:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny, img.shape) and img[nx, ny] == 1:
                neighbors.append((nx, ny))

    # --- 8-Connectivity ---
    elif mode == "8":
        # Add all 4-neighbors and Diagonal neighbors
        for dx, dy in moves_4 + moves_diag:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny, img.shape) and img[nx, ny] == 1:
                neighbors.append((nx, ny))

    # --- m-Connectivity ---
    elif mode == "m":
        # Condition 1: q is in N4(p)
        for dx, dy in moves_4:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny, img.shape) and img[nx, ny] == 1:
                neighbors.append((nx, ny))
```

```python

        # Condition 2: q is in Nd(p) AND intersection of N4 is empty
        for dx, dy in moves_diag:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny, img.shape) and img[nx, ny] == 1:
                # Check intersection neighbors (the two shared 4-neighbors)
                # For a diagonal move (dx, dy), the shared neighbors are (x+dx, y) and (x, y+dy)
                n4_1 = get_val(img, x + dx, y)
                n4_2 = get_val(img, x, y + dy)

                # Intersection must be empty (both 0)
                if n4_1 == 0 and n4_2 == 0:
                    neighbors.append((nx, ny))

    return neighbors


# 3. BFS Algorithm for Shortest Path
def bfs_shortest_path(img, start, end, mode):
    queue = deque([[start]])
    visited = set([start])

    while queue:
        path = queue.popleft()
        node = path[-1]

        if node == end:
            return path

        for neighbor in get_neighbors(node, img, mode):
            if neighbor not in visited:
                visited.add(neighbor)
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)

    return None


# 4. Execution & Visualization
connectivities = ["4", "8", "m"]
results = {}

plt.figure(figsize=(12, 5))

for i, mode in enumerate(connectivities):
    path = bfs_shortest_path(image, START, END, mode)
    results[mode] = path

    # Visualization
    ax = plt.subplot(1, 3, i + 1)

    # Create visualization matrix: 0=Black, 1=White, 2=Path(Red)
    vis_img = np.zeros_like(image, dtype=float)
    vis_img[image == 1] = 1.0  # Foreground
    vis_img[image == 0] = 0.0  # Background

    path_len_str = "No Path"
    if path:
        path_len_str = f"Len: {len(path)} pixels"
        # Mark path
        for px, py in path:
            vis_img[px, py] = 0.5  # Grey/Red placeholder

    # Custom plotting overlaying the path manually to ensure it's visible
    ax.imshow(image, cmap="gray", vmin=0, vmax=1)

    if path:
        # Unzip path into x and y lists for plotting lines
        ys, xs = zip(*path)  # x is row (y-axis in plot), y is col (x-axis in plot)
        ax.plot(
            xs, ys, color="red", linewidth=3, marker="o", markersize=8, label="Path"
        )

    ax.set_title(f"{mode}-Connectivity\n{path_len_str}")
    ax.invert_yaxis()  # Match matrix coordinates
    ax.set_xticks(range(4))
    ax.set_yticks(range(4))
    ax.grid(True, color="gray", linestyle="--", linewidth=0.5)

plt.tight_layout()
plt.savefig("./images/output/lab3_pathfinding.png")
# plt.show()

# 5. Text Output
print(f"Start: {(START[0]+1, START[1]+1)}")
print(f"End:   {(END[0]+1, END[1]+1)}\n")

for mode in connectivities:
    path = results[mode]
    print(f"--- {mode}-Connectivity ---")
    if path:
        # Convert to 1-based indexing for display
        path_1based = [(x + 1, y + 1) for x, y in path]
        print(f"Path Found: {path_1based}")
        print(f"Pixel Count: {len(path)}")
    else:
```

```
        print("Path Not Found")
    print("")
```

3 Results

The BFS algorithm was executed for each connectivity mode. The generated paths are visualized below:

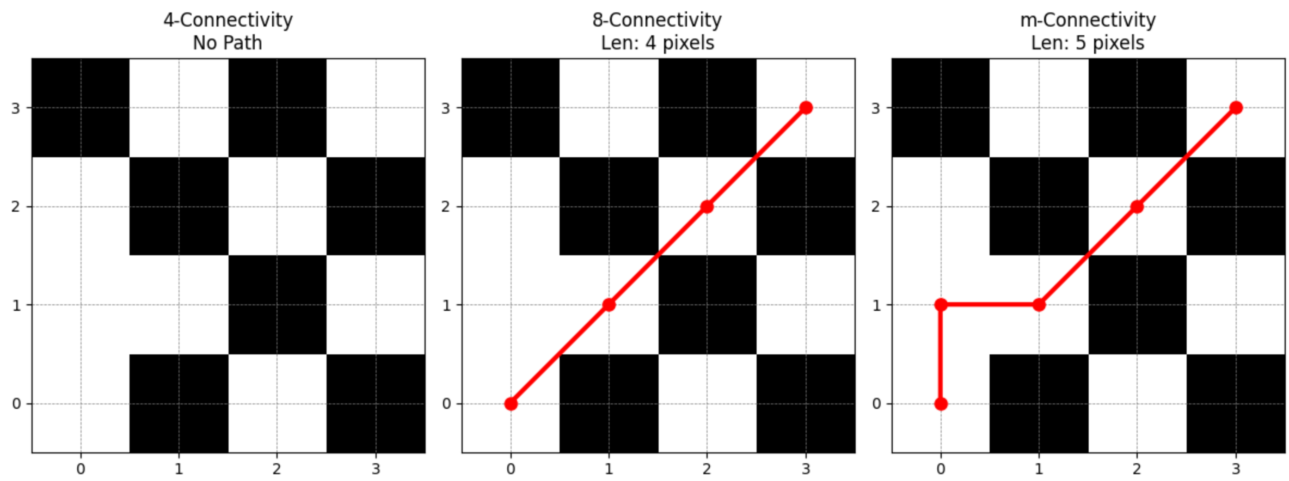

Figure 1: Shortest Path Analysis Results. Left: 4-Connectivity fails to find a path. Center: 8-Connectivity finds the shortest diagonal route. Right: m-Connectivity finds a valid but slightly longer path to resolve ambiguity.

4 Discussion

Distinct results were yielded for each connectivity type, highlighting their geometric properties:

- **4-Connectivity (No Path):** A path failed to be found by the algorithm. As seen in the matrix, the Start pixel $(0, 0)$ and its immediate neighbors form a cluster that is isolated from the destination $(3, 3)$ when diagonals are forbidden. The zeros at $(1, 2)$ and $(2, 1)$ effectively act as a wall.
- **8-Connectivity (Length 4):** The shortest possible path was produced by this mode. By exploiting diagonal connections, the path moved directly from $(0, 0) \to (1, 1) \to (2, 2) \to (3, 3)$. This represents the Chebyshev distance.
- **m-Connectivity (Length 5):** The m-connectivity path was found to be longer than the 8-connectivity path.
  - The move $(0, 0) \to (1, 1)$ was **invalid** in m-connectivity because the shared neighbor $(1, 0)$ is 1 (not empty). This forced the path to take a 4-connected step first: $(0, 0) \to (1, 0) \to (1, 1)$.
  - Subsequent moves $(1, 1) \to (2, 2)$ and $(2, 2) \to (3, 3)$ were valid diagonal m-connections because their shared neighbors were 0.

5 Conclusion

The impact of connectivity definitions on spatial analysis was successfully demonstrated in this lab. It was observed that 4-connectivity is the most restrictive, often failing to connect visually adjacent regions. 8-connectivity is the most permissive and yields the shortest geometric distance. m-connectivity serves as a robust hybrid, allowing diagonal connections only when necessary to prevent the ambiguity of multiple path options. Implementing these logic rules within a BFS framework effectively solved the shortest path problem.

References

[1] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed. Boston, MA, USA: Pearson, 2018.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.