COM S 311 Homework 4

Write Up

Name: Alisala Mwamba

Net-ID: mwambama

1. Java Type Used to Represent the Adjacency List:
   In my implementation, the adjacency list is represented using a `HashMap<String, List<Edge>>`. Each key in the map represents a vertex (in this case, pixel coordinates like "i,j"), and the value is a list of edges, where each edge connects to a neighboring vertex with a specified weight. This approach ensures efficient lookups and supports arbitrary numbers of neighbors for each vertex. each vertex `T` is mapped to another map, which in turn holds its neighboring vertices along with their corresponding edge weights. This structure offers several advantages for the following.
   1. Efficient access to both vertices and their neighbors.
   2. Constant-time retrieval of edge weights for any existing connection.
   3. Flexibility and scalability for graphs with varying connectivity and density.

2. Assigning Edge Weights When Converting the Image to a Graph:

Each pixel in the input image is treated as a vertex, and edges are formed between adjacent pixels (up, down, left, right). Edge weights are determined based on pixel values:

1.White pixels (0xFFFFFFFF): Representing whitespace or foreground text areas, transitions between adjacent white pixels are assigned a low weight of 1.

2.All other pixels (e.g., black pixels 0xFF000000): Treated as background or noise, transitions involving these are assigned a higher weight of 100 or more to discourage traversal through these regions. This weighting heuristic guides pathfinding algorithms to prefer paths through whitespace, which effectively helps delineate lines of text and avoids cutting through background or noisy regions.

3. Solving the Problem Using Dijkstra's Algorithm:

To detect and separate lines of text (whitespace-based segmentation), I implemented Dijkstra's algorithm within the getShortestPaths method. The process involves:

1. Initializing a priority queue and a distance map to track the minimum known distance from a source to every vertex.
2. Iteratively updating the shortest paths by exploring neighbors and relaxing edges.
3. Skipping any edge with negative weights, although this scenario is prevented by design.

To ensure efficiency, the algorithm is only run twice per image:

1. Row-wise traversal: A virtual source node is connected to the leftmost white pixels in each row. The goal is to find paths to the rightmost white pixels with a cost equal to the row width minus one.
2. Column-wise traversal: A similar setup connects a virtual source to the topmost white pixels in each column. The objective is to reach the bottommost white pixels.
3. This fixed number of invocations (2) ensures the algorithm remains computationally efficient, even for large images.

4. Heap Implementation Used:

For the priority queue in Dijkstra's algorithm, I used Java's standard PriorityQueue class (java.util.PriorityQueue), which is implemented as a binary heap. Key benefits include:

1. Automatic ordering based on a custom comparator (e.g., comparing current distances).
2. Logarithmic insertion and extraction times, ensuring performance during graph traversal.
3. Being part of the Java Standard Library, it is stable, well-tested, and licensed under the GNU General Public License (GPL) with the Classpath Exception.

And therefore, this combination of techniques that I used such as —using a `HashMap` for adjacency, determining edge weights based on pixel values, and using the Java's built-in `PriorityQueue`—allowed for an efficient solution to identifying text separations in the image.