1 Overview

Suppose you wanted to write a program to recognize text in an image (OCR), but you didn't have access to large labeled datasets, powerful GPUs, and trained ML models. You might first attempt to isolate individual characters in order to analyse them individually.

In this assignment, you will be writing a program that can identify individual lines of text in an image. To do so, you will use Dijkstra's algorithm for finding shortest paths in weighted graphs. Broadly, this can be broken into the following steps:

1. Read in the image.
2. Represent the image as a graph.
3. Use Dijkstra's algorithm to identify individual lines of text.
4. Process the image accordingly (e.g. split the original image into individual images for each line).

Items 2 and 3 are crucial. For this task, what is an appropriate way to map an image to a graph? And how can Dijkstra's algorithm be used to identify lines of a text in an image?

2 Graph Implementation

You will be implementing a generic weighted graph class. The underlying representation *must be an* ***adjacency list***. In addition to basic graph functionality, you will be implementing a method that uses Dijkstra's algorithm 1 for finding single-source shortest paths.

You are free to use any priority queue implementation that you choose, but if it is not part of the standard JDK, then it must be included in source format and properly *attributed* (including in your write-up).

3 Line Separation

3.1 Images

For simplicity, we will restrict our focus to basic bitmap images. You can use the provided helper class to load the image and get a square 2D matrix of ARGB pixels. Each pixel is represented by 4 bytes, where the most significant byte (MSB) is the alpha channel. We will further assume that there is no transparency, i.e. that the MSB is always 0xFF. The other bytes are treated as typical RGB values, thus 0xFFFFFFFF is white and 0xFF000000 is black. An example pixel is shown in 2.

Algorithm 1 Dijkstra's algorithm

```
1: procedure DIJKSTRA'S(G, w, s)
2:     init(G, s)                    ▷ Preparation such as setting initial shortest path values.
3:     S = ∅
4:     Q = ∅
5:     Q.insert(s)
6:     while Q ≠ ∅ do
7:         u = Q.extractMin()
8:         S = S ∪ {u}
9:         for v ∈ G.adj(u) do
10:            if v.d > u.d + w(u, v) then
11:                v.d = u.d + w(u, v)
12:                Q.decreaseKey(v, v.d)
13:            end if
14:        end for
15:    end while
16: end procedure
```

3.2 Converting To A Graph

There are many ways you could choose to convert the grid of pixels to a graph. An limited visual example can be seen in 3. Given such a representation, a low-weight path between two pixels in the same row on opposing sides of the image may represent natural separation between lines of text. We will use our implementation of Dijkstra's algorithm in our graph class to identify these paths.

A common issue with facsimiles of paper documents is that, even with black and white documents, the space between characters is not perfectly white. See the bottom-center pixel in 3 for an example: is this noise or part of a character? To account for this, a threshold can be given that represents the amount of tolerance for noise in the image or path. However, for the purposes of this assignment, we will assume that the space between text is entirely white. We will also limit ourselves to considering only correctly-aligned monospace text.

4 Classes and Interfaces

You may add helper methods and member variables, but they must be private. Your graph class may be tested independently of the image related classes, and it will only be tested using the given public API.

Important classes and some of their notable methods are given below. There methods not listed here that are required by the interfaces. They must also be implemented correctly.

- `public class WeightedAdjacencyList<T> implements WeightedGraph<T>`

Figure 2: An ARGB pixel with its value in hex.

A directed and weighted graph stored as an adjacency list. In addition to all of the basic graph functionality required by the WeightedGraph interface, you must implement the following two methods:

  - `public WeightedAdjacencyList(List<T> vertices) { /* ... */ }`

    : This constructor initializes the graph with the given list of vertices. This list may be empty.

  - `public Map<T, Long> getShortestPaths(T s) { /* ... */ }`

    : Take in a source vertex and returns a mapping from all reachable vertices to their shortest distance from s.

- `public class CharacterSeparator`

  This class contains the logic for separating rows and columns of text in a given image. You will need to implement the following method:

  - `public static Pair<List<Integer>, List<Integer>> findSeparationWeighted(String path) { /* ... */ }`

    : This method takes as input the (filesystem) path to an image. It returns two lists of integers. The first is a list of integers indicating the rows of pixels that do not contain relevant text in the given image. The second list indicates columns that are between relevant text. This method should only construct exactly one instance of your graph class and use **a constant number of calls to Dijkstra's algorithm**. Do not do non-portable things like prepending your home directory to the given path; this method should work with both relative and absolute paths.

- `public final class BitmapProcessor { /* ... */ }`

  This class is given to you and should not be modified. It has various helper methods for simplifying image processing. The are relatively basic, but they are intended to

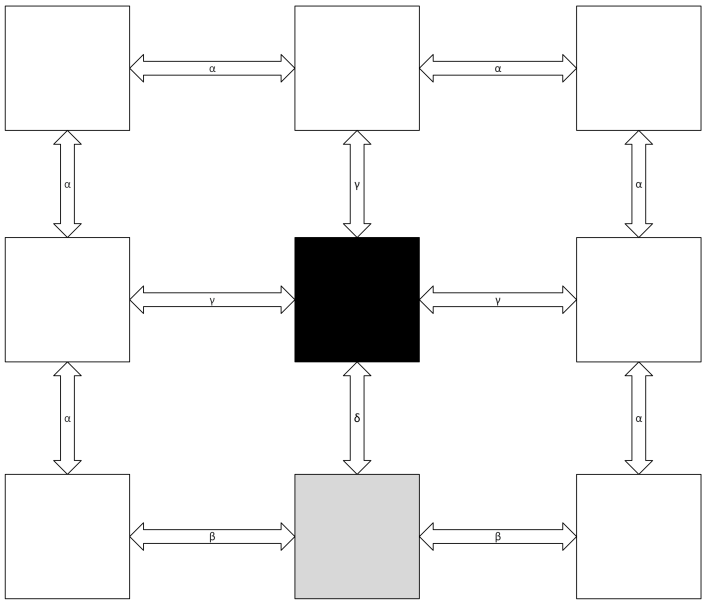

Figure 3: An example of a portion of an image viewed as a weighted graph. Note that $\alpha < \beta < \gamma < \delta$.

suffice for the purposes of this assignment. You may need to use the following two methods it provides:

- `public BitmapProcessor(String path) throws IOException`

  : Creates a bitmap processor object from the given image file.

- `public BitmapProcessor(String path) throws IOException`

  : Gets the 2D grid of pixels in the images as ARGB values with no transparency.

5 Write-up

You are to include a *brief* write-up in with your submission called *writeup.pdf*. In it, you should state the following 3 things:

1. The Java type you chose to represent the adjacency list inside your graph class.
2. How you assigned edge weights when converting the image to a graph.
3. How you solved the proposed problem using a constant number of invocations of Dijkstra's algorithm.
4. The heap implementation you used. If you did not write it yourself and it is not part of the standard Java Class Library, you should mention this, including the license of the code and a link to it.

Please place each in a separate paragraph with a heading clearly denoting the question being answered.

You must state your name, netid, and the assignment number clearly at the top. It must be in PDF format. It must be in the root of the folder your zip unpacks to, i.e. at the same level as the `src` folder.

6 Important

- Your submission must be *your own work*. The only exceptions are the skeleton provided to you and *possibly* the heap implementation.
- Any code submitted that is not your own *must* be properly attributed (at minimum via a `README` file in your submissions root directory), and it must be in source format (not compiled or otherwise). Any code given to you by us for this assignment is excluded from this requirement.
- Your classes may be tested independently of one another. They will be tested using the specified public API.
- You should document your code where appropriate. This includes inline documentation for long or complex functions.
- You may **not**:
  - Change or alter the public API of any given or required types. This includes *class names, package names, method names, return types, class visibility, method visibility, method arguments, interfaces, etc.*
  - Include any external libraries (`jar` or otherwise). This especially includes graph libraries.
  - Submit any code that is not your own, unless otherwise explicitly allowed. Any code given to you by us with this assignment is excluded from this requirement.