Homework 4

Michael Becker

NetID: 371615257

1. Java Type

For my implementation of the WeightedAdjacency list class, I chose to use a `Map<T, Map<T, Integer>>` data structure. The outer map represents vertices as keys, with each key mapping to an inner map. The inner map has keys that are the neighbor vertices and the values are the weights. This was chosen for multiple reasons:

- It has an efficient run time and time complexity.
- O(1) average time complexity for ops like checking if a vertex exists.
- Directly maps its vertex to its adjacent vertices and edge weights
- Makes traversing easy.

The inner map's key value pairs allow a way for me to get both the vertex's neighbors and edge weights in a single structure.

2. How edge weights were assigned

When I needed to convert the image to a graph for line separation. I created a weighting mechanism that assigns low weights to white spaces and high weights to dark text. This approach was used so that Dijkstra's algorithm would want to go through the white space. The weight process extracts the RBG values and calculates the brightness of each pixel as the average of its RGB value. For each edge between, I'd calculate the brightness of the two pixels. Last of all, I then invert it and scale the brightness. The formula I have created will basically make white pixels with a brightness of 255, have a weight of 0 and Black pixels have a brightness of 0 and a thicker weight. The graph is made with edges between the pixels to create a grid structure where Dijkstra's algorithm can find the path through the white space.

3. How I solved the problem using a constant number of invocations of Dijkstra's algorithm.

My approach to finding the line separations is centered around creating a single graph representation of the image and then making a linear number of calls to Dijkstra's algorithm. The image is represented as a single weight graph where each pixel is a vertex represented by its coordinates. For the horizontal separations, each row I run Dijkstra's algorithm from the leftmost pixel to the right most pixel. The height of the image will be the number of calls. The approach I use requires the total number of calls to Dijkstra's algorithm to be the height and the width. The threshold I have for determining if a path is whitespace is 25% of the total/max weight.

4. Heap Implementation

For my implementation of Dijkstra's algorithm I used PriorityQueue class from the standard Java library (java.util.package). Implementing this way provides a binary min-heap. These are the performances:

- O(log n) for insertion
- O(log n) for extracting the min

I made the the priority queue so it compares vertices based on their current distance in the algorithm. The PriorityQueue is simpler than other heap implementations. It uses the standard library so it was easier to do.