Programming Assignment #2

Programming assignments are to be done individually. You may discuss the problem and general concepts with other students, but there should be no sharing of code. You may not submit code other than that which you write yourself or is provided with the assignment. This restriction specifically prohibits downloading code from the Internet. If any code you submit is in violation of this policy, you will receive no credit for the entire assignment.

The goals of this lab are:

- To recognize algorithms you have learned in class in new contexts.
- Ensure that you understand certain aspects of graphs and greedy algorithms.

Problem Description

You have recently started a job at Seemo, an up and coming autonomous aerial vehicle startup. Seemo's newest aerial vehicle is smaller and lighter than the competition, but this new form factor means that it is unable to keep up computationally with existing larger vehicles on the market.

Your team has isolated the primary bottleneck to be the inefficiency of the salient object detection algorithm, and is working on developing a more computationally efficient algorithm.

First, some high level background for the problem. The general goal of salient object detection is to locate relevant objects within a given image. Let us assume that an image is a 2D-array (or matrix representation) of pixels. Let us also assume that neighboring pixels of similar pixel "intensity" are more likely to belong to the same object.

Figure 1 shows a graph representation of a 4×4 pixel image. The value at each pixel represents the pixel "intensity." Let us take a more detailed look at the pixel with intensity 11. We observe that it is more likely to be part of the same object as the pixel with intensity 10, than it is likely to be part of the same object as the pixel with intensity 1.

Using the intuition that pixels with larger "gaps" in intensity are "farther" apart, we can assign absolute distance weights to edges between neighboring pixels. Figure 2 shows the same image graph, except this time with the distances between pixels shown. Notice that, by definition, there are multiple "paths" between any pair of pixels in this image graph.

Lets call this graph the "intensity graph" of an intensity matrix.

One of the senior engineers on your team has a promising solution to the salient object detection problem in the works[1], and has separated the problem into several pieces.

[1] inspired by a real publication that you can find here if interested: `http://www.shengfenghe.com/uploads/1/5/1/3/15132160/cvpr16_mstsaliency.pdf`

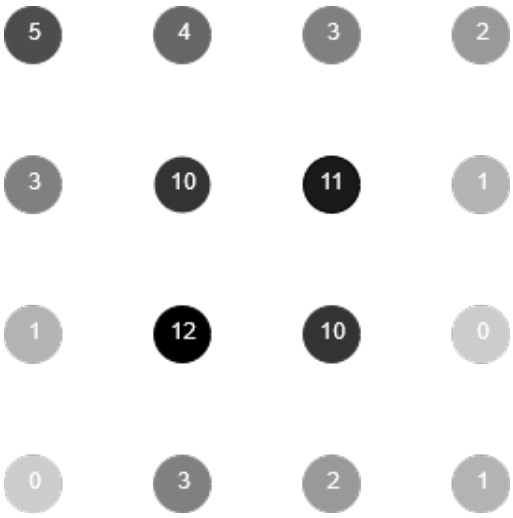

Figure 1: Graph representation of a 4×4 pixel image.

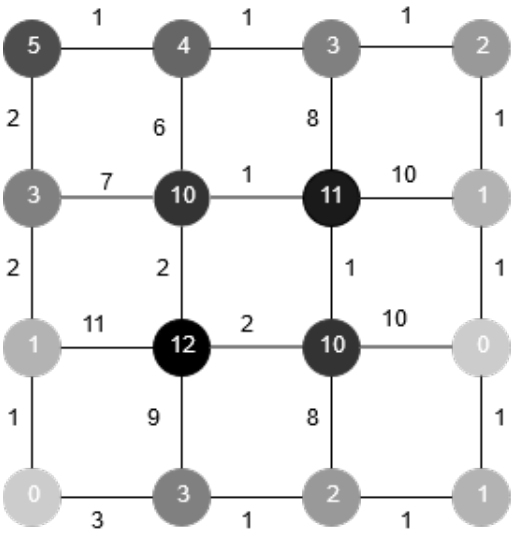

Figure 2: Intensity graph for given input nodes.

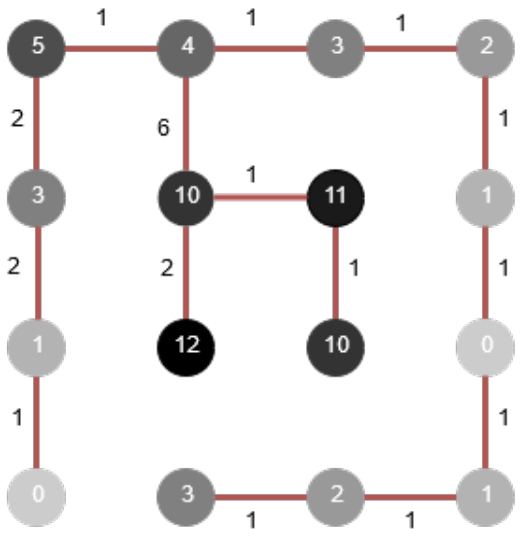

Figure 3: Pruned version of intensity graph.

The engineer's solution relies on computing the shortest path between each pair of pixels in the image. Unfortunately, for larger images, this is proving too computationally expensive. However, the engineer has noted that in practice the chosen path nearly always excludes the larger edge weights[2].

This is where you come in. The senior engineer has tasked you with pruning the search space of possible paths by excluding certain edges from the original image graph. Specifically, the engineer wants you to devise and implement a time efficient algorithm that generates a new graph using a subset of the edges from the original image graph, such that there is still a path between any pair of vertices, and the sum of all the edge weights in the graph is minimized.

Note that the weights of the edges in the new graph must not be modified from the original graph. To depict the pruning process, Figure 3 shows a pruned image graph corresponding to the intensity graph in Figure 2.

Lets call the pruned version of an intensity graph, the "pruned graph".

The engineer recommends that you separate the problem into two parts.

Part 1: Constructing the Intensity Graph [30 points]

Based on the 2D-array of pixel intensity values, implement an algorithm that constructs a representative intensity graph (as shown in Figure 2). In order to provide a compact summary of the graph, you should return the sum of the edge weights in your graph. The specific inputs provided and outputs expected are provided below.

[2]in reality, this is an oversimplification for shortest path computations; however, it is true for certain more complex distance computations outside the scope of this assignment

Input: A 2D-array of values* IMAGE where each element, IMAGE[m][n], represents the pixel intensity for the pixel at row m and column n of the image.

Output: An INTENSITYWEIGHT value* where INTENSITYWEIGHT represents the sum of the weights of all of the edges in the intensity graph that represents IMAGE.

Method Signature: int *constructIntensityGraph*(int[ ][ ] image);

*values are nonnegative integers

Part 2: Constructing the Pruned Graph from the Intensity Graph [50 points]

Modify and extend your work in Part 1, by implementing an algorithm that creates an intensity graph representation and converts it into a pruned graph representation (as shown in Figure 3). You will be heavily penalized for a non-polynomial time (in terms of the number of pixels) algorithm. The specific inputs provided and outputs expected are provided below.

Input: A 2D-array of values* IMAGE where each element, IMAGE[m][n], represents the pixel intensity for the pixel at row m and column n of the image.

Output: A PRUNEDWEIGHT value* where PRUNEDWEIGHT represents the sum of all of the edges in the pruned graph derived from the intensity graph that represents IMAGE.

Method Signature: int *constructPrunedGraph*(int[ ][ ] image);

*values are nonnegative integers

Part 3: Report [20 points]

Write a short report that provides the following information:

(a) From a space complexity and time complexity perspective, does it make more sense to use an adjacency matrix or an adjacency list in order to represent the intensity graph in part 1? Explain your answer.

(b) Describe the algorithm you implemented in part 2 (both text and pseudocode are fine).

(c) In Big-O notation, state the runtime complexity of the algorithm you implemented in part 2, in terms of the number of pixels, p, in a given input image.

Program Details

You will be provided with helper classes that evaluate results and simplify packaging of inputs and outputs as well as code skeletons where you fill in your own solutions. The following Java files are provided:

- Program2.java - This is where you implement your algorithms for part 1 and part 2 in the constructIntensityGraph and constructPrunedGraph methods, respectively. Any variables and data structures that are necessary should be initialized locally to the respective methods, since we will use a single instance of your Program2 class to run all of the test cases. If you leave lingering variables and data structures in the class that persist between calls to these methods, you may fail test cases. You are welcome to add helper methods to the class.
- TestRunner.java - The runner class that evaluates your algorithm on provided test cases. We have provided two sample test cases for each part to help you get started. We have also provided comments depicting the graphs we constructed to help guide you. Note that you are only expected to provide the single value outputs. Note that these test cases are not necessarily indicative of the size or value of inputs used for grading. We highly encourage you to create your own test cases, especially with large input sizes. Our grading process will use a similar runner class with additional test cases. (Hint: you might consider creating your own test generator method to automatically generate large test cases in order to validate that your code terminates.)
- TestCase.java - Class containing an image, the intensityWeight for the graph representing the image, and the prunedWeight for the graph representing the image. The intensityWeight and prunedWeight serve as ground truth values, that can be checked against external results (for example, the methods in your Program2.java).

Note that the only file used to evaluate your code is your Program2.java. While you may add helper methods to the Program2 class, DO NOT include additional java class files (any additional files will be ignored).

Submission Details

You should submit a single zip file titled `eid_lastname_firstname.zip` that contains your Program2.java and a typed pdf report `eid_lastname_firstname.pdf`. Do not put these files in a folder before you zip them (i.e. the files should be in the root of the ZIP archive). Your solution must be submitted via Canvas BEFORE 11:59 pm on April 2, 2019. Absolutely no late assignments will be accepted for any reason.

- Make sure your program compiles on the LRC machines before you submit it. The LRC Machines use Java 1.8, and we will be evaluating the program using Java 1.8. Make sure it compiles and runs correctly with Java 1.8.
- It is **highly recommended** that you do comment your code and follow good programming style. It will be very unlikely that you receive partial credit if there are not descriptive comments and the TAs cannot understand what your code is/should be doing.