Homework06

Parallel and Sequential Algorithms COMP 312 (Spring '17)

1 Introduction

In this homework, you will practice implementing a parallel weighted graph algorithm, Borůvka's algorithm for minimum spanning trees. Then, you will extend this algorithm to do a form of image segmentation, a process that divides an image into many sets of pixels. Segmentation has many useful applications, one of which is a *posterization* effect.

2 Files

After downloading the assignment tarball, extract the files by running:

```
tar -xf homework06-handout.tar
```

from a terminal window. Some of the files worth looking at are listed below. The files denoted by * will be submitted for grading.

1. * `MkBoruvkaSegmenter.sml`
2. `Tests.sml`

3 Submission

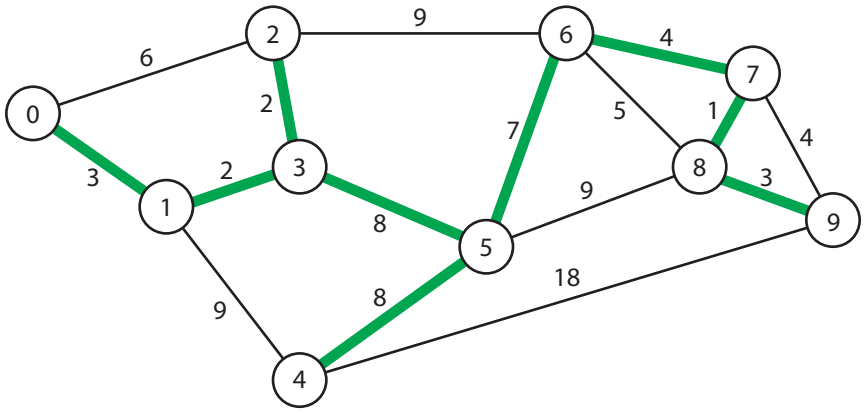
To submit your assignment, copy MkBoruvkaSegmenter.sml to your handin directory, and bring your answers to any written problems to the next course meeting.

4 Overview

Recall that the minimum spanning tree (MST) of a connected undirected graph $G = (V, E)$ where each edge e has weight $w : E \to \mathbb{R}^+$ is the spanning tree T that minimizes

$$\sum_{e \in T} w(e)$$

For example, in the graph below, the MST shown in green has weight 38, which is minimal.



For this homework, you will use the parallel MST algorithm, Borůvka's algorithm, as presented in lecture and Chapter 18 of the textbook.

Image segmentation is defined as the process of "partitioning a digital image into multiple sets of pixels". The goal of image segmentation is to reduce an image into a simpler form, which is often easier to analyze than the original image.

If we segment an image into disjoint, connected sets of pixels, then color each set the same color, we see a posterization effect. An example of this process is shown below.

The left is the original image, the middle is based on segmentation, and the right is based on segmentation where the connected sets are allowed to grow larger.

We can use MSTs to segment images in this way. Specifically, we will

(a) construct a grid graph, where vertices are pixels and edges connect adjacent pixels,

(b) weight each edge with the difference in color of its endpoints (similar colors result in low weight while different colors result in high weight),

(c) compute the minimum spanning tree of the graph while allowing certain "high-weight" edges to be deleted (more on this soon) such that the graph becomes disconnected.

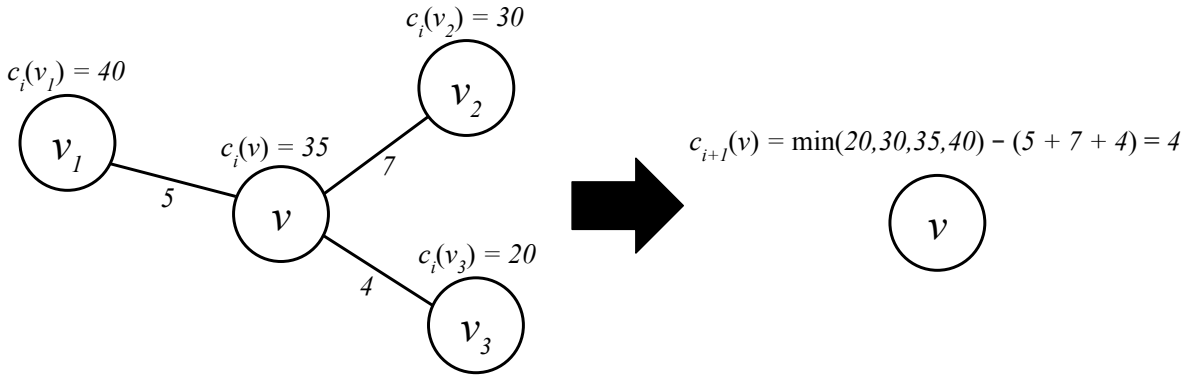
In fact, this process computes a *minimum spanning forest* (MSF) of a now *disconnected* graph, where each MST within the resulting MSF is considered to be one "segment" of the original graph. These steps of image processing have been implemented for you; your job is to do the MST part.

As mentioned above, we represent the image as a weighted, undirected graph. Our image segmentation algorithm is an extension of Borůvka's algorithm. We begin by assigning some number of "credits" to each vertex. These credits are a sort of currency which vertices may spend in order to contract with other vertices. As we will see, the amount of credit correlates with the size of the segments in the output.

When we contract a star, we have to "spend credits." Specifically, on round i, suppose we're contracting a star X consisting of vertices V_X, edges E_X, and center $v \in V_X$. Let $c_i(u) : u \in V_X$ be the credits of these vertices on this round. We update the credit of the center according to the formula

$$c_{i+1}(v) = \min_{u \in V_X} c_i(u) - \sum_{e \in E_X} w(e).$$

An example is shown below. Note that we don't need to update the credits of the satellites, because they have been contracted and are no longer present in the graph.



There is one caveat: when a vertex runs out of credit, it is no longer allowed to contract. So, at the end of each round of Borůvka's algorithm, we remove edges which might cause credits to go negative. Specifically, if E_i' is the set of edges remaining at the end of some round i, then the edge set given as input to round $i + 1$ is

$$\big\{(u, v) \in E_i' \,\big|\, \min(c_{i+1}(u), c_{i+1}(v)) \geq w(u, v)\big\}.$$

Notice that in this formula, we use the values c_{i+1} rather than c_i, because E_i' contains only the edges which remain between star centers.

It turns out that if we assign an initial credit of ∞ to each vertex, then this algorithm is actually identical to traditional Borůvka's, and can still be used to compute MSTs! We will take advantage of this fact to test your code.

5 Implementation Tasks

You are given a graph represented as a number n and a sequence of edges, each of which is a triple (u, v, w) representing an edge from u to v with weight w, where $0 \leq u, v < n$. The input graph is undirected, simple (no self-loops, at most one undirected edge between any two vertices), connected, and has no negative weighted edges. For every (u, v, w) in the input, you can assume the input also contains (v, u, w).

5.1 Borůvka MST

Task 5.1 (70 pts). In `MkBoruvkaSegmenter.sml`, implement the function

val `segment : int → (edge Seq.t * int) → edge Seq.t`

such that it ignores its first `int` argument and computes the MST of the given graph, which is represented by an edge sequence E and the number of vertices n (where vertices are labeled $0 \leq v < n$ within E). For full credit, your algorithm should require $O(|E| \log^2 n)$ work and $O(\log^3 n)$ span in expectation.

Your function should output the sequence of edges in the MST. Unlike the input, you should only include edges pointing in one direction (for each undirected edge in the output, you should either include (u, v, w) or (v, u, w), but not both).

You can follow Algorithm 18.22 in the textbook closely, or, because the graph's vertices are numbers, use sequences instead of tables (in this case you might find the function `Seq.inject` particularly useful).

5.2 Segmentation

Task 5.2 (30 pts). Extend the above MST implementation to implement the cost-based segmentation algorithm described in the previous section. The first input c is the initial credit for each vertex. For full credit, your algorithm should require $O(|E| \log^2 n)$ work and $O(\log^3 n)$ span in expectation. In the case of $c = \infty$, this should be the minimum spanning tree of the input graph.

5.3 Pseudorandomness

A *pseudorandom number generator* is an algorithm which computes sequences of numbers which appear random. These algorithms are in fact deterministic, and are often "seeded" by some user-provided number.

One of the arguments given to the `MkBoruvkaSegmenter` functor is the structure `Rand`, ascribing to `RANDOM210`. This structure implements a *deterministic parallel random number generator* (DPRNG) with splittable seeds. You will need to use `Rand` to implement Borůvka's algorithm. In the support code, there is a helper function `flipCoins n r`, which given n and an input seed r, produces n random booleans along with a fresh seed r'. Its work and span are $O(n)$ and $O(1)$, respectively. Note that due to the deterministic nature of `Rand`, if you call this function twice with the same r, you will get the same sequence of booleans as output both times. This is why the function returns a fresh seed, r', to be used in a subsequent call.

You will need to construct an initial seed for your algorithm with `Rand.fromInt`. You may pass any integer you would like to this function. Changing this number will only result in small aesthetic changes in the output of your segmenter.

6 Testing (And Segmenting!)

There are three ways to test your code.

1. Run your segmenter at the REPL. For example, the following segments the image `images/skittles.png` with an initial credit of 1000, writing the resulting image at `images/my-skittles-1000.png`.

   ```
   - CM.make "segmenting.cm";
   ...
   - SegTester.makeSegsFile ("images/skittles.png", "images/my-skittles-1000.png", 1000);
   ```

   We have provided a few sample images in `images/`, and example segmentations of these in `images/out/` for reference.

2. In `Sandbox.sml`, write whatever testing code you'd like. You can then access the sandbox at the REPL:

   ```
   - CM.make "sandbox.cm";
   ...
   - open Sandbox;
   ```

3. In `Tests.sml`, add test cases according to the instructions given. Then run the autograder:

   ```
   - CM.make "autograder.cm";
   ...
   - Autograder.run ();
   ```

   To test your code, we will use an initial credit of $c = \infty$ and verify that you compute a minimum spanning tree. We will hand-grade segmentation qualitatively.