Lab 7 - *SegmentLab*

Due: *Mon, Mar 31st, 2014 @ 11:59 pm*

Parallel and Sequential Data Structures and Algorithms 15-210 (Spring '14)

1 Introduction

Image segmentation is a process that divides an image into many sets of pixels. In this assignment, you will segment images in parallel using Borůvka's MST algorithm.

Segmentation can be used to implement effects such as posterization – so while you're not quite writing your own implementation of Adobe PhotoShop, this lab will get you 0.0001% of the way there!

2 Files

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

```
tar -xvf segmentlab-handout.tgz
```

from a terminal window. Some of the files worth looking at are listed below. You should only modify the files denoted by *, as these will be the only ones handed in by the submission script.

1. `Makefile`
2. `images/`
3. `Tests.sml`
4. * `MkBoruvkaSegmenter.sml`

Additionally, you should create a file called:

```
written.pdf
```

which contains the answers to the written parts of the assignment.

3 Submission

To submit your assignment to Autolab, open a terminal, `cd` to the `segmentlab` folder, and run:

```
make
```

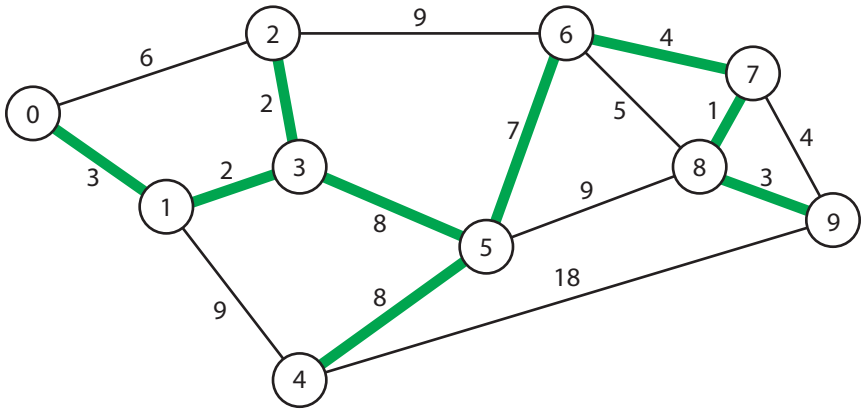
Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "*Handin your work*" link.

4 Borůvka's Algorithm

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.



You should be familiar with Kruskal's and Prim's algorithms for finding minimum spanning trees. However, both algorithms are sequential. For this problem, you will use the parallel MST algorithm, Borůvka's algorithm, as presented in lecture.

5 Image Segmentation

Finding a minimum spanning tree can be used to solve various real world problems, one of which is image segmentation. 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, often that is easier to analyze than the original image. An example of this process is shown below.

The image of the Skittles on the left is the original image. The one on the right is the image after the image segmentation process. As you can see, regions of pixels have all been colored the same color, indicating that they were segmented into the same group.

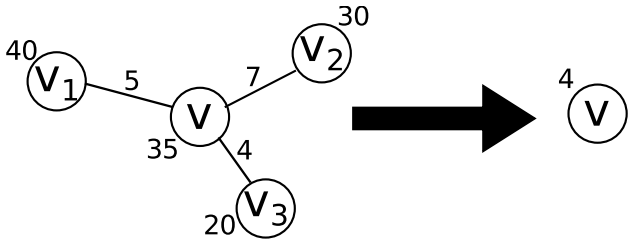
5.1 Algorithm

We will represent the image as a weighted graph, where vertices are pixels, and edge weights represent the difference between the colors of the endpoints. The 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 a vertex can spend in order to contract with other vertices. As we will see, low amounts of initial credit will leave the image largely untouched, while high initial credit will result in lots of segmentation.

When we contract a star, the resulting vertex should have credit equal to the minimum credit of the contracted vertices minus the sum of the weights of the contracted edges. In other words, suppose you have a star X consisting of vertices V_X, edges E_X, and center $v \in V_X$. After contracting X, only v will remain, and it should have credit value

$$c'(v) = \underset{u \in V_X}{\text{MIN}}\, c(u) - \sum_{e \in E_X} w(e)$$

Note that we don't need to update the credit values for the satellite vertices, because they have been contracted and are no longer a part of the graph. Here is an example:



There is one caveat: we want the credits to stay positive as much as possible. After each iteration of Borůvka's with credit updates, you should only keep the edges which can't locally cause the credits to go negative, i.e.

$$E' = \left\{ e \text{ as } (u, v) \;\middle|\; e \in E \text{ and } \text{MIN}(c(u), c(v)) - w(e) \geq 0 \right\}$$

The algorithm continues until no edges remain, at which point we have a collection of vertices that represent a partition of the graph into trees. Each vertex from the original graph has been contracted into exactly one of these final vertices, therefore we can return a mapping between vertices and their representative final vertices.

It turns out that this algorithm is actually equivalent to Borůvka's MST algorithm if we never run out of credit (that is, if we never have to remove an edge to prevent the credit from going negative)!

6 Writing Your Image Segmenter

6.1 Logistics

Vertices will be labeled from 0 to $|V| - 1$. The input graph is both simple (no self-loops, at most one undirected edge between any two vertices) and connected. We will represent both the input and output graphs as edge sequences, where an edge is defined as:

```
type edge = vertex * vertex * weight
```

such that the triple (u, v, w) indicates a directed edge from u to v with edge weight w. The type of `vertex` and `weight` are both `int` (for more details, see `support/SEGMENTER.sig`). For the input, since we will be dealing with undirected graphs, for every edge (u, v, w) there will be another edge (v, u, w) in the sequence.

6.2 Borůvka Segmenter

Task 6.1 (75%). In `MkBoruvkaSegmenter.sml`, implement the function

```
val findSegment : (edge seq * int) -> int -> (vertex seq * edge seq)
```

where `(findSegment (E, n) c)` computes the image segmentation of a picture (that is represented as graph). E is the input edge sequence, n is the number of vertices in the graph, and c is the initial credit for each vertex. The output tuple should be (R, T), where

- $R = \langle r_0, r_1, \ldots r_{n-1} \rangle$, where each r_i is the representative final vertex of original vertex i. This output is used to generate the segmented image, and is ignored by the autograder.
- $T = \langle\, e \mid e \text{ is a contracted edge} \,\rangle$. This output is used to verify that your algorithm computes MSTs properly.

For full credit, your solution must have *expected* $O(m \log^2 n)$ work and *expected* $O(\log^3 n)$ span, where n is the number of vertices and m is the number of edges. For reference, our solution is around 70 lines long with comments and only has one recursive helper.

The correctness of `findSegments` will be evaluated based on the following:

- Correctly implementing Borůvka's : 30%
- Properly segmenting an image: 20%
- Cost bounds: 25%

***Note regarding grading**: For testing that your algorithm correctly computes MSTs, we will exploit the fact that* `findSegments` *is equivalent to traditional Borůvka's if we assign an initial credit of $c = +\infty$. We will only examine T from your output, which should then be a sequence containing only the tree edges of the MST of the graph.*

...And that's it for the programming part of this lab!

6.3 Wait, What?

Yes, we realize that this is a pretty substantial algorithm to implement in one task. So, where do you start? We *highly* recommend first attempting to implement Borůvka's alone – you can do this by ignoring the credits entirely. As long as your output tuple (R, T) contains a correct T, you can test it with `testMST` (see Testing below) or submit it to Autolab for autograding. We only examine R in order to produce the final segmented image.

You might find the following function from `RANDOM210` to be useful:

```
val flip : rand -> int -> (int seq * rand)
```

where `(flip seed n)` evaluates to $(S, seed')$, where S is a sequence of n random bits, and $seed'$ is a new seed. For our purposes, you may assume `flip` has $O(n)$ work and $O(\log n)$ span. Use `fromInt` to generate an initial seed.

Lastly, as a hint, recall that the `SEQUENCE` library function

```
val inject : (int * 'a) seq -> 'a seq -> 'a seq
```

takes a sequence of index-value pairs to write into the input sequence. If there are duplicate indices in the sequence, the *last* one is written and the others are ignored. **Consider presorting the input sequence of edges E from largest to smallest by weight.** Then injecting any subsequence of E will always give priority to the minimum-weight edges.

6.4 Testing

You will not submit test cases for this lab, but we provide you with an easy way to test whether your function returns the MST.

As before, you should add test cases (inputs to `findSegment`) to the `testsMST` list in `Tests.sml`.

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
- Tester.testMST ();
```

To test the image segmentation functionality, run the following commands:

```
$ smlnj
Standard ML of New Jersey v110.xx
- CM.make "sources.cm";
- Tester.testSegmenter ("images/inputImage.png", "outputImage.png", 1000);
```

where `Tester.testSegmenter (in, out, c)` segments the image located at `in` with initial credit `c` and stores the result at the location `out`. If you like to work locally, there is a good chance that the segmenter will not work properly. If this is the case, please run it on a Linux cluster machine instead.

Some sample input images and outputs have been provided in `images/`. The sample output images are named `outputImage-[numCredits].png`, where `numCredits` is the number of initial credits input to get that particular output image. Note that your output may not be exactly the same as the given images, as there is randomness involved.

7 Written Problems

Recall that edge contraction is a graph contraction strategy that partitions the graph into disjoint components consisting of at most 2 vertices each. For arbitrary graphs, we have seen that edge contraction is not a sufficient strategy because it cannot always remove at least a constant fraction of the number of edges (in expectation). However, edge contraction *is* viable in certain situations – for example, with certain kinds of trees.

Suppose you are given a ternary tree T with n vertices and m edges. You want to prove that T can be contracted using edge contraction in expected $O(m)$ work and $O(\log^2 n)$ span.

Task 7.1 (10%). Prove that the number of vertices in T with degree strictly less than 3 is at least $\frac{n}{k}$ for some constant $k > 1$.

Task 7.2 (10%). Describe a process for selecting edges to contract. Prove that in expectation, you will select a constant fraction of edges *at each step*. You should exploit the ternary properties of T. *(Hint: consider why your contraction strategy needs to preserve the ternary-ness of T after each iteration.)*

Task 7.3 (5%). Using the results of the previous part, justify the given cost bounds.