Version 1.0 ***due:*** *Mon, November 4 @ 11:59pm*

1 Introduction

In this assignment you will find minimum spanning trees. That's it!

2 Files

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

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

from a terminal window. You should see the following files:

1. `segmentlab.pdf`
2. `lib/`
3. `support/`
4. `Makefile`
5. `bobbfile.py`
6. `config.py`
7. `sources.cm`
8. `images/`
9. * `MkBoruvkaMST.sml`
10. * `MkBoruvkaSegmenter.sml`

You should only modify the last 2 files, denoted by *, and (if you would like) the `images/` directory. Additionally, you should create a file called `written.pdf` which contains the answers to the written part of the assignment.

3 Submission

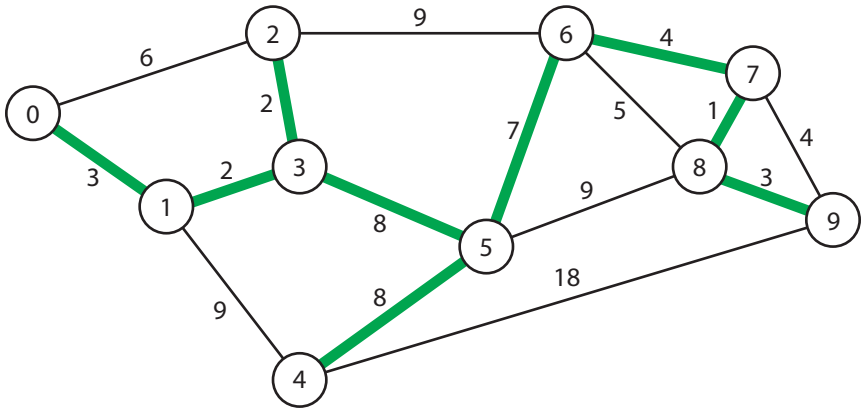
To submit your assignment: open a terminal, `cd` to the `segmentlab-handout` folder, and run `make`. Alternatively, run `make package`, open the Autolab webpage and submit the `handin.tgz` file via the "Handin your work" link.

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



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 implement the parallel MST algorithm, otherwise known as Borůvka's algorithm.

4.1 Logistics

4.1.1 Representation

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`. 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. **The MST produced by your solution should only have edges in one direction.** You may find the following function useful:

```
Random210.flip :  Random210.rand -> int -> int seq
```

where `flip r n` takes a `Random210.rand`, r and produces a n-length sequence of random bits. You may assume that `flip r n` has $O(n)$ work and $O(\log n)$ span. Use `Random210.fromInt` and `Random210.next` to generate seed values for each round of your algorithm.

4.2 Specification

Task 4.1 (40%). Implement the function

```
MST : edge seq * int -> edge seq
```

in `MkBoruvkaMST.sml` where `MST (E, n)` computes the minimum spanning tree of the graph represented by the input edge sequence E using Borůvka's Algorithm. There will be n vertices in the graph, labeled from 0 to $n-1$. Recall that you do not need to manually reverse edges – assume that if the edge

$$(x, y, w)$$

appears in the input, then so does

$$(y, x, w)$$

. For full credit, your solution must have *expected* $O(m \log n + n)$ work and *expected* $O(\log^k n)$ span for some k, where n is the number of vertices and m is the number of edges.

You may find the pseudocode given in lecture or the component labeling code covered in recitation useful for your implementation. Keep in mind that directly translating pseudocode to SML will not result in the cleanest, nor most efficient solution. For reference, our solution is under 50 lines long and only has one recursive helper. As a hint, recall that the `SEQUENCE` library function

```
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.

4.3 Testing

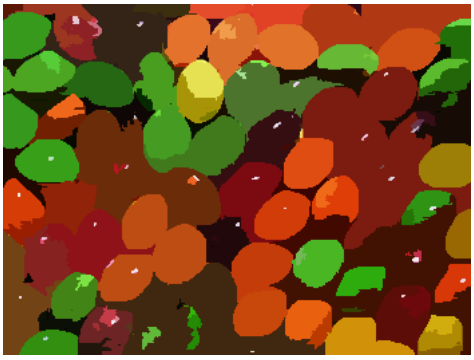
You will not submit test cases for this lab, but we provide you with an easy way to test your implementation of MST. Note that this is **different** from the way you have tested in the past.

As before, you should add test cases (inputs to your MST function) to the `testsMST` list in `Tests.sml`. Then, instead of starting the REPL normally, just do the following:

```
shell> ./run
smlnj> CM.make "sources.cm";
smlnj> use "testing";
smlnj> Tester.testMST ();
```

5 Image segmentation

In the previous section, we implemented Boruvka's algorithm to find the MST of the graph. 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.

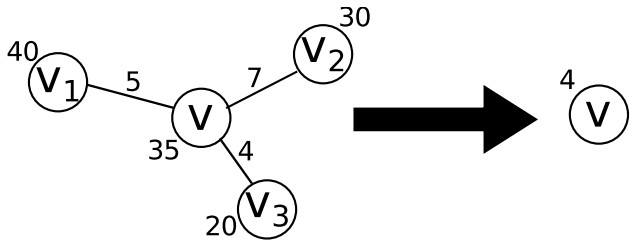


The first image of the Skittles is the original image. The second image is the image after the image segmentation process has been applied to the image. As you can see, by using image segmentation, specific pixel regions have been combined to form a different region with a color consisting of some combination of the pixels in the older regions.

5.1 Algorithm

In order to implement an image segmentation process, we will represent the image as a weighted graph, where vertices are pixels, and edge weights depend on the difference between the colors of the endpoints. We will compute this graph for you and pass it to your function.

We can modify Boruvka's algorithm to assign some number of "credits" to each vertex. These credits represent a sort of currency which a vertex can spend in order to contract with other vertices. When we contract a star, the resulting vertex then has credit equal to the minimum credit of the endpoints of the contracted edges minus the sum of the weights of the contracted edges. A simple example follows.



The caveat to this, however, is that we would like the credits to stay positive as much as possible. After each iteration of Boruvka's, you should ensure that that you remove all edges where the minimum credit of the endpoints is less than the weight of the edge (as contracting such an edge would cause credits to go negative). That is, after each iteration, for every edge uv, keep it in the graph iff $\min(c_u, c_v) - w_{uv} \geq 0$, and remove it otherwise.

The algorithm continues until there are no remaining edges, at which point we will have a collection of vertices, representing a partition of the graph into trees. Each vertex in the original graph has been contracted into exactly one of these final vertices. You should return a sequence, $\langle s_0, s_1, \ldots, s_{|V|} \rangle$, of length $|V|$, where s_i is the label of the final vertex into which vertex i from the original graph is contracted. That is s_i represents the tree in the aforementioned partition which contains vertex i.

Recall that the initial credits represent a sort of "currency" that a vertex can spend in order to contract with other vertices (where the cost of contraction is the weight of the contracted edge). This means that running the algorithm with more initial credits per vertex will result in more contraction and fewer final segments, while running it with fewer initial credits will result in a lot of final segments. This will be useful when testing your code at the end.

5.2 Specification

Task 5.1 (25%). Implement the function

```
findSegment: (edge Seq.seq * int) -> int -> vertex Seq.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. An edge Seq.seq is defined as a sequence of `vertex * vertex * weight`, where the type of `vertex` and the type of `weight` are both `int`. These types can also be found in `support/SEGMENTER.sig`. Once again, you may assume that the input edge sequence already contains edges in both directions (as we are dealing with an undirected graph).

Your solution must have *expected* $O(m \log^2 n)$ work and *expected* $O(\log^k n)$ span for some k, where n is the number of vertices and m is the number of edges.

5.3 Testing

Testing for this lab is once again quite different from previous labs. As before, we will run `./run` from the shell to start the SML REPL. We will then run `CM.make "sources.cm"` to compile our code, and then `use "testing"` to load the `Tester` structure. Then you will run

```
Tester.testSegmenter ("images/inputImage.png", "outputImage.png", 1000);
```

The first argument is a string locating the image which you want to segment, the second argument is where you want to save the final segmented image, and the third argument is the number of initial credits per vertex. Some sample input images and outputs have been provided in `images/`. Note that your output may not be exactly the same, as there is randomness involved.

6 Written Problems

Task 6.1 (10%). **Second-best is good enough for my MST.** Let $G = (V, E)$ be a simple, connected, undirected graph $G = (V, E)$ with $|E| \geq 2$ and *distinct* edge weights. We know for a fact that the smallest (i.e., least heavy) edge of G must be in the minimum spanning tree (MST) of G. Prove that the 2^{nd} smallest edge of G must also be in the minimum spanning tree of G.

Task 6.2 (10%). **I Prefer Chalk.** There is a very unusual street in your neighborhood. This street forms a perfect circle, and there are $n \geq 3$ houses on this street. As the unusual mayor of this unusual neighborhood, you decide to hold an unusual lottery. Each house is assigned a random number $r \in_R [0, 1]$ (drawn uniformly at random). Any house that receives a larger number than **both** of its two neighbors wins a prize package consisting of a whiteboard marker and two pieces of chalk in celebration of education. What is the expected number of prize packages given? Justify your answer.

Task 6.3 (15%). **It's *Probably* Linear.** For all $n > 1$, let X_n be a random variable with $X_n \leq n$. Let f be a non-decreasing function satisfying

$$f(n) \leq f\left(X_n\right) + \Theta(n), \qquad \text{where } f(1) = 1.$$

Prove that if $\mathbf{E}\left[X_n\right] = n/3$, then $\overline{f}(n) = \mathbf{E}\left[f(n)\right] \in O(n)$.
Hint: what is $\mathbf{Pr}[X_n \geq 2n/3]$? Use Markov's Inequality, covered in recitation 9.