Recitation 10 (11/7 - 11/11)

Boruvka's Algorithm

An algorithm to find the minimum spanning tree for a graph with distinct edge weights (none of the edges have the same value).

The goal of the algorithm is to connect "components" using the shortest edge between components. It begins with all of the vertices considered as separate components.

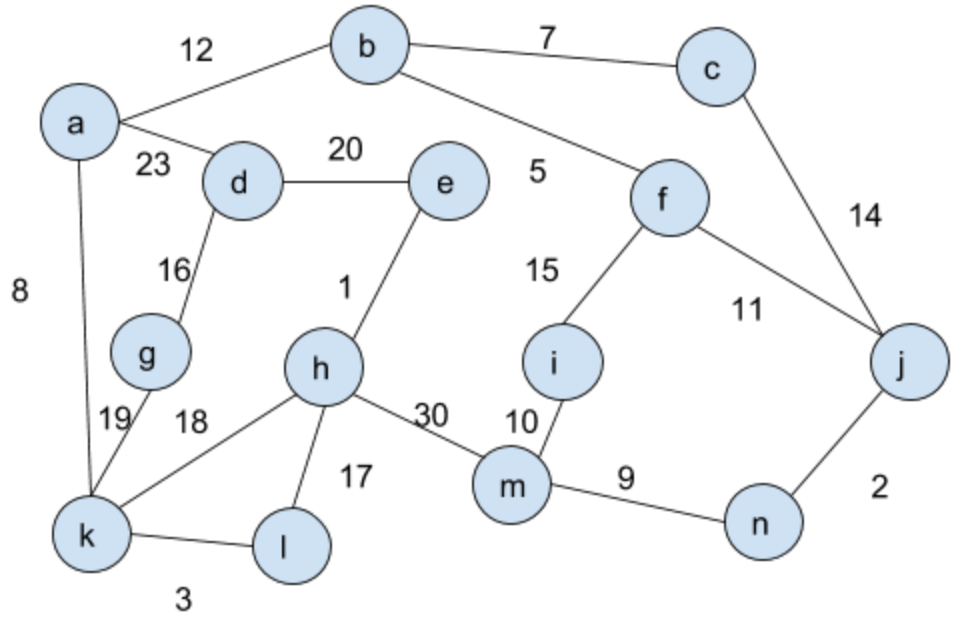
Pseudocode

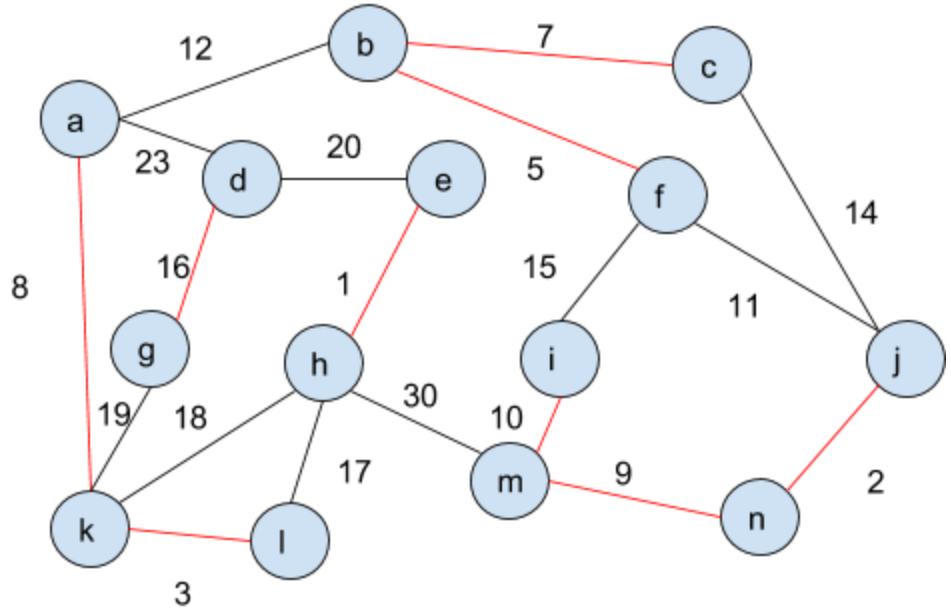
Courtesy of wikipedia

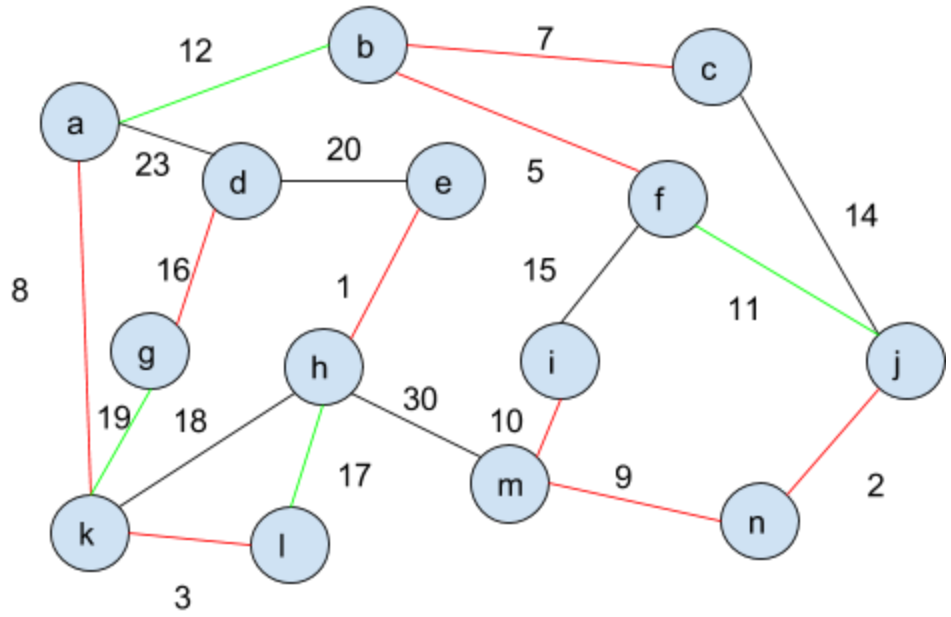
```
Input: A connected graph G whose edges have distinct weights
1  Initialize a forest T to be a set of one-vertex trees, one for each vertex of the graph.
2  While T has more than one component:
3    For each component C of T:
4      Begin with an empty set of edges S
5      For each vertex v in C:
6        Find the cheapest edge from v to a vertex outside of C, and add it to S
7      Add the cheapest edge in S to T
8    Combine trees connected by edges to form bigger components
9  Output: T is the minimum spanning tree of G.
```

NOTE that an edge can be selected twice (by two different components). This is fine.

Example


a
b
c
d
e
f
g
h
i
j
k
l
m
n
12
7
23
20
5
14
8
16
1
15
11
19
18
30
10
17
9
2
3


a
b
c
d
e
f
g
h
i
j
k
l
m
n
12
7
23
20
5
14
8
16
1
15
11
19
18
30
10
17
9
2
3



At this point, we are done. There are two more examples on the Wikipedia page.

Complexity

Note that the while loop will halve the number of components each time. We start with *v* components (the number of vertices), so the while loop runs in log(*v*) time.

In each iteration of the loop, we look through the edges, taking *e* time.

This can be said to have O(e * log(v)), or O(m * log(n)) time.

Proof Idea

(optional)
http://www.csee.wvu.edu/~ksmani/courses/fa01/random/lecnotes/lec11/MST.pdf

Boruvka's Algorithm is based upon the following lemma:
Let v ∈ V be any vertex in G. The minimum spanning tree for G must contain the edge (v, w) that is the minimum weight edge incident on v.

This can be proved by contradiction. Assume the MST does not contain the minimum weight edge incident on v. If so, there must be another edge connecting v to our MST. However, if we remove that edge, and add the minimum weight edge (we need to prevent cycles, this is an

MST), then the total edge values would be less than or equal to the tree with the non-minimum edge. This is a contradiction.

Additionally, at each contraction, the algorithm creates a forest in the original graph.
This forest does not have any edges that would not be in the MST.
Finally, the algorithm runs until there is only one component. This means the final component's edges must correspond with the MST's edges.

Recurrence Relations (5.2)

The idea:

- Divide - and - conquer algorithms - recursively call on *q* subproblems of size n/2 each and combine results in linear time (think merge sort).

What is the time complexity?

Remember, we have *q* subproblems in each recursive call, and they each have size *n/2*.
From the book:

(5.3) *For some constant c,*

$$T(n) \leq qT(n/2) + cn$$

when $n > 2$, *and*

$$T(2) \leq c.$$

This means that the time for each recursive call can be broken down into:

- A factor of the lower call plus
- A constant factor

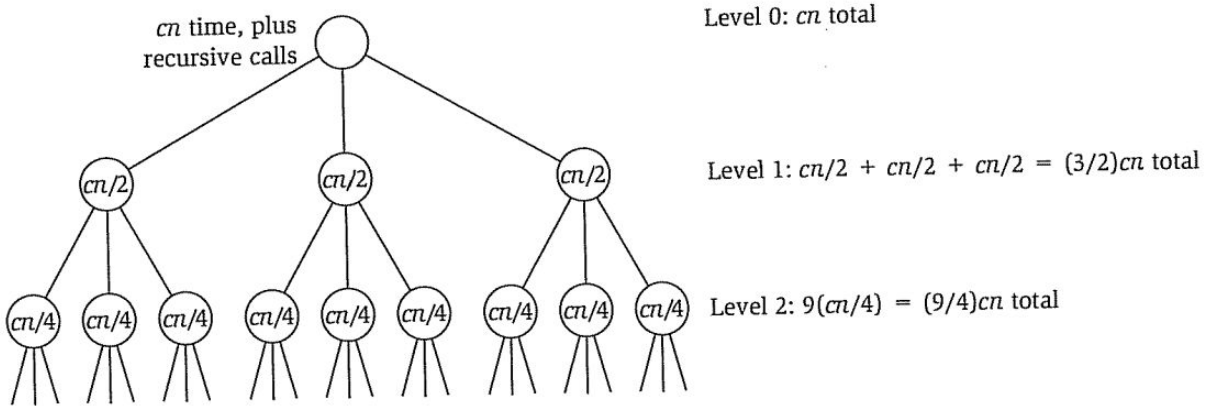
And this is just the bound! It will never be greater, but may be less.

But how can we determine this?

NOTE that the cases are different for q > 2, q = 2, q = 1. Case q > 2 will be covered now.

Case q > 2

At this point, we will follow the book's example of q = 3. This can be extended to larger q's if desired.



Unrolling:

1. First level, have problem of size *n*, which will take *cn* time (we stated that the combination is in linear time) plus the total for all previous levels.
2. Next level, we have q = 3 problems. Each of these problems has size *n/2* (again from our statement for recurrence relations). Each of these takes a time to combine of *cn/2*, plus the recursive calls. The total will be *(q/2)cn* time.
3. For the third level, we have q^3 = 9 problems, each of size *n/4.* The total is *(q^2/4)cn*

The Pattern

At any arbitrary level *j*, we have q^j instances, each of size n/2^j. Thus the total work at this level is q^j(cn/2^j) = (q/2)^j * cn.

We can sum this over the levels.

$$T(n) \leq \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j cn = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{q}{2}\right)^j$$

For anyone interested in more in-depth math, I recommend reading chapter 5.2. Everything is simplified, and corners are cut because we are only looking for an upper bound. The result:

$$O(n^{\log_2 q})$$

Case q = 1:

Each level in our unrolling will have no *q* value. Instead, it will just have time *cn/2* plus the time for the previous.

At an arbitrary level *j*, there is one instance, size *n/2^j* and contributes *cn/2^j* to the running time.

$$T(n) \leq \sum_{j=0}^{\log_2 n-1} \frac{cn}{2^j} = cn \sum_{j=0}^{\log_2 n-1} \left(\frac{1}{2^j}\right)$$

If you remember from calculus, this summation will never be greater than 2. Thus, we can say, for case q = 1:

$$T(n) \leq 2cn = O(n).$$

Case of two was covered in the previous chapter.