Chapter 23

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 23.1-1

Suppose that A is an empty set of edges. Then, make any cut that has (u, v) crossing it. Then, since that edge is of minimal weight, we have that (u, v) is a light edge of that cut, and so it is safe to add. Since we add it, then, once we finish constructing the tree, we have that (u, v) is contained in a minimum spanning tree.

Exercise 23.1-2

Let G be the graph with 4 vertices: u, v, w, z. Let the edges of the graph be $(u, v), (u, w), (w, z)$ with weights 3, 1, and 2 respectively. Suppose A is the set $\{(u, w)\}$. Let $S = A$. Then S clearly respects A. Since G is a tree, its minimum spanning tree is itself, so A is trivially a subset of a minimum spanning tree. Moreover, every edge is safe. In particular, (u, v) is safe but not a light edge for the cut. Therefore Professor Sabatier's conjecture is false.

Exercise 23.1-3

Let T_0 and T_1 be the two trees that are obtained by removing edge (u, v) from a MST. Suppose that V_0 and V_1 are the vertices of T_0 and T_1 respectively. Consider the cut which separates V_0 from V_1. Suppose to a contradiction that there is some edge that has weight less than that of (u, v) in this cut. Then, we could construct a minimum spanning tree of the whole graph by adding that edge to $T_1 \cup T_0$. This would result in a minimum spanning tree that has weight less than the original minimum spanning tree that contained (u, v).

Exercise 23.1-4

Let G be a graph on 3 vertices, each connected to the other 2 by an edge, and such that each edge has weight 1. Since every edge has the same weight, every edge is a light edge for a cut which it spans. However, if we take all edges we get a cycle.

Exercise 23.1-5

Let A be any cut that causes some vertices in the cycle on once side of the cut, and some vertices in the cycle on the other. For any of these cuts, we know that the edge e is not a light edge for this cut. Since all the other cuts wont have the edge e crossing it, we won't have that the edge is light for any of those cuts either. This means that we have that e is not safe.

Exercise 23.1-6

Suppose that for every cut of the graph there is a unique light edge crossing the cut, but that the graph has 2 spanning trees T and T'. Since T and T' are distinct, there must exist edges (u, v) and (x, y) such that (u, v) is in T but not T' and (x, y) is in T' but not T. Let $S = \{u, x\}$. There is a unique light edge which spans this cut. Without loss of generality, suppose that it is not (u, v). Then we can replace (u, v) by this edge in T to obtain a spanning tree of strictly smaller weight, a contradiction. Thus the spanning tree is unique.

For a counter example to the converse, let $G = (V, E)$ where $V = \{x, y, z\}$ and $E = \{(x, y), (y, z), (x, z)\}$ with weights 1, 2, and 1 respectively. The unique minimum spanning tree consists of the two edges of weight 1, however the cut where $S = \{x\}$ doesn't have a unique light edge which crosses it, since both of them have weight 1.

Exercise 23.1-7

First, we show that the subset of edges of minimum total weight that connects all the vertices is a tree. To see this, suppose not, that it had a cycle. This would mean that removing any of the edges in this cycle would mean that the remaining edges would still connect all the vertices, but would have a total weight that's less by the weight of the edge that was removed. This would contradict the minimality of the total weight of the subset of vertices. Since the subset of edges forms a tree, and has minimal total weight, it must also be a minimum spanning tree.

To see that this conclusion is not true if we allow negative edge weights, we provide a construction. Consider the graph K_3 with all edge weights equal to -1. The only minimum weight set of edges that connects the graph has total weight -3, and consists of all the edges. This is clearly not a MST because it is not a tree, which can be easily seen because it has one more edge than a tree on three vertices should have. Any MST of this weighted graph must have weight that is at least -2.

Exercise 23.1-8

Suppose that L' is another sorted list of edge weights of a minimum spanning tree. If $L' \neq L$, there must be a first edge (u, v) in T or T' which is of smaller weight than the corresponding edge (x, y) in the other set. Without

loss of generality, assume (u, v) is in T. Let C be the graph obtained by adding (u, v) to L'. Then we must have introduced a cycle. If there exists an edge on that cycle which is of larger weight than (u, v), we can remove it to obtain a tree C' of weight strictly smaller than the weight of T', contradicting the fact that T' is a minimum spanning tree. Thus, every edge on the cycle must be of lesser or equal weight than (u, v). Suppose that every edge is of strictly smaller weight. Remove (u, v) from T to disconnect it into two components. There must exist some edge besides (u, v) on the cycle which would connect these, and since it has smaller weight we can use that edge instead to create a spanning tree with less weight than T, a contradiction. Thus, some edge on the cycle has the same weight as (u, v). Replace that edge by (u, v). The corresponding lists L and L' remain unchanged since we have swapped out an edge of equal weight, but the number of edges which T and T' have in common has increased by 1. If we continue in this way, eventually they must have every edge in common, contradicting the fact that their edge weights differ somewhere. Therefore all minimum spanning trees have the same sorted list of edge weights.

Exercise 23.1-9

Suppose that there was some cheaper spanning tree than T'. That is, we have that there is some T'' so that $w(T'') < w(T')$. Then, let S be the edges in T but not in T'. We can then construct a minimum spanning tree of G by considering $S \cup T''$. This is a spanning tree since $S \cup T'$ is, and T'' makes all the vertices in V' connected just like T' does. However, we have that $w(S \cup T'') = w(S) + w(T'') < w(S) + w(T') = w(S \cup T') = w(T)$. This means that we just found a spanning tree that has a lower total weight than a minimum spanning tree. This is a contradiction, and so our assumption that there was a spanning tree of V' cheaper than T' must be false.

Exercise 23.1-10

Suppose that T is no longer a minimum spanning tree for G with edge weights given by w'. Let T' be a minimum spanning tree for this graph. Then we have we have $w'(T') < w(T) - k$. Since the edge (x, y) may or may not be in T' we have $w(T') \leq w'(T') + k < w(T)$, contradicting the fact that T was minimal under the weight function w.

Exercise 23.1-11

If we were to add in this newly decreased edge to the given tree, we would be creating a cycle. Then, if we were to remove any one of the edges along this cycle, we would still have a spanning tree. This means that we look at all the weights along this cycle formed by adding in the decreased edge, and remove the edge in the cycle of maximum weight. This does exactly what we want since we could only possibly want to add in the single decreased edge, and then, from there we change the graph back to a tree in the way that makes its total weight

minimized.

Exercise 23.2-1

Suppose that we wanted to pick T as our minimum spanning tree. Then, to obtain this tree with Kruskal's algorithm, we will order the edges first by their weight, but then will resolve ties in edge weights by picking an edge first if it is contained in the minimum spanning tree, and treating all the edges that aren't in T as being slightly larger, even though they have the same actual weight. With this ordering, we will still be finding a tree of the same weight as all the minimum spanning trees $w(T)$. However, since we prioritize the edges in T, we have that we will pick them over any other edges that may be in other minimum spanning trees.

Exercise 23.2-2

At each step of the algorithm we will add an edge from a vertex in the tree created so far to a vertex not in the tree, such that this edge has minimum weight. Thus, it will be useful to know, for each vertex not in the tree, the edge from that vertex to some vertex in the tree of minimal weight. We will store this information in an array A, where $A[u] = (v, w)$ if w is the weight of (u, v) and is minimal among the weights of edges from u to some vertex v in the tree built so far. We'll use $A[u].1$ to access v and $A[u].2$ to access w.

Algorithm 1 PRIM-ADJ(G,w,r)

```
Initialize A so that every entry is (NIL, ∞)
T = {r}
for i = 1 to V do
    if Adj[r, i] ≠ 0 then
        A[i] = (r, w(r, i))
    end if
end for
for each u ∈ V − T do
    k = min_i A[i].2
    T = T ∪ {k}
    k.π = A[k].1
    for i = 1 to V do
        if Adj[k, i] ≠ 0 and Adj[k, i] < A[i].2 then
            A[i] = (k, Adj[k, i])
        end if
    end for
end for
```

Exercise 23.2-3

Prim's algorithm implemented with a Binary heap has runtime $O((V + E)\lg(V))$, which in the sparse case, is just $O(V\lg(V))$. The implementation with Fibonacci heaps is $O(E + V\lg(V)) = O(V + V\lg(V)) = O(V\lg(V))$. So, in the sparse case, the two algorithms have the same asymptotic runtimes.

In the dense case, we have that the binary heap implementation has runtime $O((V + E)\lg(V)) = O((V + V^2)\lg(V)) = O(V^2\lg(V))$. The Fibonacci heap implementation however has a runtime of $O(E+V\lg(V)) = O(V^2+V\lg(V)) = O(V^2)$. So, in the dense case, we have that the Fibonacci heap implementation is asymptotically faster.

The Fibonacci heap implementation will be asymptotically faster so long as $E = \omega(V)$. Suppose that we have some function that grows more quickly than linear, say f, and $E = f(V)$. The binary heap implementation will have runtime $O((V + E)\lg(V)) = O((V + f(V))\lg(V)) = O(f(V)\lg(V))$. However, we have that the runtime of the Fibonacci heap implementation will have runtime $O(E + V\lg(V)) = O(f(V) + V\lg(V))$. This runtime is either $O(f(V))$ or $O(V\lg(V))$ depending on if $f(V)$ grows more or less quickly than $V\lg(V)$ respectively. In either case, we have that the runtime is faster than $O(f(V)\lg(V))$.

Exercise 23.2-4

If the edge weights are integers in the range from 1 to $|V|$, we can make Kruskal's algorithm run in $O(E\alpha(V))$ time by using counting sort to sort the edges by weight in linear time. I would take the same approach if the edge weights were integers bounded by a constant, since the runtime is dominated by the task of deciding whether an edge joins disjoint forests, which is independent of edge weights.

Exercise 23.2-5

If there the edge weights are all in the range $1, \ldots, |V|$, then, we can imagine adding the edges to an array of lists, where the edges of weight i go into the list in index i in the array. Then, to decrease an element, we just remove it from the list currently containing it(constant time) and add it to the list corresponding to its new value(also constant time). To extract the minimum wight edge, we maintain a linked list among all the indices that contain non-empty lists, which can also be maintained with only a constant amount of extra work. Since all of these operations can be done in constant time, we have a total runtime $O(E+V)$.

If the edge weights all lie in some bounded universe, suppose in the range 1 to W. Then, we can just vEB tree structure given in chapter 20 to have the two required operations performed in time $O(\lg(\lg(W)))$, which means that the total runtime could be made $O((V + E)\lg(\lg(W)))$.

Exercise 23.2-6

For input drawn from a uniform distribution I would use bucket sort with

Kruskal's algorithm, for expected linear time sorting of edges by weight. This would achieve expected runtime $O(E\alpha(V))$.

Exercise 23.2-7

We first add all the edges to the new vertex. Then, we preform a DFS rooted at that vertex. As we go down, we keep track of the largest weight edge seen so far since each vertex above us in the DFS. We know from exercise 23.3-6 that in a directed graph, we don't need to consider cross or forward edges. Every cycle that we detect will then be formed by a back edge. So, we just remove the edge of greatest weight seen since we were at the vertex that the back edge is going to. Then, we'll keep going until we've removed one less than the degree of the vertex we added many edges. This will end up being linear time since we can reuse part of the DFS that we had already computed before detecting each cycle.

Exercise 23.2-8

Professor Borden is mistaken. Consider the graph with 4 vertices: a, b, c, and d. Let the edges be $(a, b), (b, c), (c, d), (d, a)$ with weights 1, 5, 1, and 5 respectively. Let $V_1 = \{a, d\}$ and $V_2 = \{b, c\}$. Then there is only one edge incident on each of these, so the trees we must take on V_1 and V_2 consist of precisely the edges (a, d) and (b, c), for a total weight of 10. With the addition of the weight 1 edge that connects them, we get weight 11. However, an MST would use the two weight 1 edges and only one of the weight 5 edges, for a total weight of 7.

Problem 23-1

a. To see that the second best minimum spanning tree need not be unique, we consider the following example graph on four vertices. Suppose the vertices are $\{a, b, c, d\}$, and the edge weights are as follows:

	a	b	c	d
a	–	1	4	3
b	1	–	5	2
c	4	5	–	6
d	3	2	6	–

Then, the minimum spanning tree has weight 7, but there are two spanning trees of the second best weight, 8.

b. We are trying to show that there is a single edge swap that can demote our minimum spanning tree to a second best minimum spanning tree.

In obtaining the second best minimum spanning tree, there must be some cut of a single vertex away from the rest for which the edge that is added is not light, otherwise, we would find the minimum spanning tree, not the second best minimum spanning tree. Call the edge that is selected for that cut for

the second best minimum spanning tree (x, y). Now, consider the same cut, except look at the edge that was selected when obtaining T, call it (u, v). Then, we have that if consider $T - \{(u, v)\} \cup \{(x, y)\}$, it will be a second best minimum spanning tree. This is because if the second best minimum spanning tree also selected a non-light edge for another cut, it would end up more expensive than all the minimum spanning trees. This means that we need for every cut other than the one that the selected edge was light. This means that the choices all align with what the minimum spanning tree was.

c. We give here a dynamic programming solution. Suppose that we want to find it for (u, v). First, we will identify the vertex x that occurs immediately after u on the simple path from u to v. We will then make $max[u, v]$ equal to the max of $w((u, x))$ and $max[w, v]$.Lastly, we just consider the case that u and v are adjacent, in which case the maximum weight edge is just the single edge between the two. If we can find x in constant time, then we will have the whole dynamic program running in time $O(V^2)$, since that's the size of the table that's being built up.

To find x in constant time, we preprocess the tree. We first pick an arbitrary root. Then, we do the preprocessing for Tarjan's off-line least common ancestors algorithm(See problem 21-3). This takes time just a little more than linear, $O(|V|\alpha(|V|))$. Once we've computed all the least common ancestors, we can just look up that result at some point later in constant time. Then, to find the w that we should pick, we first see if $u = LCA(u, v)$ if it does not, then we just pick the parent of u in the tree. If it does, then we flip the question on its head and try to compute $max[v, u]$, we are guaranteed to not have this situation of $v = LCA(v, u)$ because we know that u is an ancestor of v.

d. We provide here an algorithm that takes time $O(V^2)$ and leave open if there exists a linear time solution, that is a $O(E + V)$ time solution. First, we find a minimum spanning tree in time $O(E + V \lg(V))$, which is in $O(V^2)$. Then, using the algorithm from part c, we find the double array max. Then, we take a running minimum over all pairs of vertices u, v, of the value of $w(u, v) - max[u, v]$. If there is no edge between u and v, we think of the weight being infinite. Then, for the pair that resulted in the minimum value of this difference, we add in that edge and remove from the minimum spanning tree, an edge that is in the path from u to v that has weight max[u,v].

Problem 23-2

a. We'll show that the edges added at each step are safe. Consider an unmarked vertex u. Set $S = \{u\}$ and let A be the set of edges in the tree so far. Then the cut respects A, and the next edge we add is a light edge, so it is safe for A. Thus, every edge in T before we run Prim's algorithm is safe for T. Any edge that Prim's would normally add at this point would have to connect two

of the trees already created, and it would be chosen as minimal. Moreover, we choose exactly one between any two trees. Thus, the fact that we only have the smallest edges available to us is not a problem. The resulting tree must be minimal.

b. We argue by induction on the number of vertices in G. We'll assume that $|V| > 1$, since otherwise MST-REDUCE will encounter an error on line 6 because there is no way to choose v. Let $|V| = 2$. Since G is connected, there must be an edge between u and v, and it is trivially of minimum weight. They are joined, and $|G'.V| = 1 = |V|/2$. Suppose the claim holds for $|V| = n$. Let G be a connected graph on $n + 1$ vertices. Then $G'.V \leq n/2$ prior to the final vertex v being examined in the for-loop of line 4. If v is marked then we're done, and if v isn't marked then we'll connect it to some other vertex, which must be marked since v is the last to be processed. Either way, v can't contribute an additional vertex to $G'.V$, so $|G'.V| \leq n/2 \leq (n + 1)/2$.

c. Rather than using the disjoint set structures of chapter 21, we can simply use an array to keep track of which component a vertex is in. Let A be an array of length $|V|$ such that $A[u] = v$ if $v = FIND - SET(u)$. Then FIND-SET(u) can now be replaced with $A[u]$ and UNION(u, v) can be replaced by $A[v] = A[u]$. Since these operations run in constant time, the runtime is $O(E)$.

d. The number of edges in the output is monotonically decreasing, so each call is $O(E)$. Thus, k calls take $O(kE)$ time.

e. The runtime of Prim's algorithm is $O(E + V \lg V)$. Each time we run MST-REDUCE, we cut the number of vertices at least in half. Thus, after k calls, the number of vertices is at most $|V|/2^k$. We need to minimize $E + V/2^k \lg(V/2^k) + kE = E + \frac{V \lg(V)}{2^k} - \frac{Vk}{2^k} + kE$ with respect to k. If we choose $k = \lg \lg V$ then we achieve the overall running time of $O(E \lg \lg V)$ as desired. To see that this value of k minimizes, note that the $\frac{Vk}{2^k}$ term is always less than the kE term since $E \geq V$. As k decreases, the contribution of kE decreases, and the contribution of $\frac{V \lg V}{2^k}$ increases. Thus, we need to find the value of k which makes them approximately equal in the worst case, when $E = V$. To do this, we set $\frac{\lg V}{2^k} = k$. Solving this exactly would involve the Lambert W function, but the nicest elementary function which gets close is $k = \lg \lg V$.

f. We simply set up the inequality $E \lg \lg V < E + V \lg V$ to find that we need $E < \frac{V \lg V}{\lg \lg V - 1} = O\left(\frac{V \lg V}{\lg \lg V}\right)$.

Problem 23-3

a. To see that every minimum spanning tree is also a bottleneck spanning tree. Suppose that T is a minimum spanning tree. Suppose there is some edge in it (u, v) that has a weight that's greater than the weight of the bottleneck

spanning tree. Then, let V_1 be the subset of vertices of V that are reachable from u in T, without going though v. Define V_2 symmetrically. Then, consider the cut that separates V_1 from V_2. The only edge that we could add across this cut is the one of minimum weight, so we know that there are no edge across this cut of weight less than $w(u, v)$. However, we have that there is a bottleneck spanning tree with less than that weight. This is a contradiction because a bottleneck spanning tree, since it is a spanning tree, must have an edge across this cut.

b. To do this, we first process the entire graph, and remove any edges that have weight greater than b. If the remaining graph is selected, we can just arbitrarily select any tree in it, and it will be a bottleneck spanning tree of weight at most b. Testing connectivity of a graph can be done in linear time by running a breadth first search and then making sure that no vertices remain white at the end.

c. Write down all of the edge weights of vertices. Use the algorithm from section 9.3 to find the median of this list of numbers in time $O(E)$. Then, run the procedure from part b with this median value as the one that you are testing for there to be a bottleneck spanning tree with weight at most. Then there are two cases:

   First, we could have that there is a bottleneck spanning tree with weight at most this median. Then just throw the edges with weight more than the median, and repeat the procedure on this new graph with half the edges.

   Second, we could have that there is no bottleneck spanning tree with at most that weight. Then, we should run the procedure from problem 23-2 to contract all of the edges that have weight at most this median weight. This takes time $O(E \lg(\lg(V)))$ and then we are left solving the problem on a graph that now has half the vertices.

Problem 23-4

a. This does return an MST. To see this, we'll show that we never remove an edge which must be part of a minimum spanning tree. If we remove e, then e cannot be a bridge, which means that e lies on a simple cycle of the graph. Since we remove edges in nonincreasing order, the weight of every edge on the cycle must be less than or equal to that of e. By exercise 23.1-5, there is a minimum spanning tree on G with edge e removed.

   To implement this, we begin by sorting the edges in $O(E \lg E)$ time. For each edge we need to check whether or not $T - \{e\}$ is connected, so we'll need to run a DFS. Each one takes $O(V + E)$, so doing this for all edges takes $O(E(V + E)$. This dominates the running time, so the total time is $O(E^2)$.

b. This doesn't return an MST. To see this, let G be the graph on 3 vertices a, b, and c. Let the eges be (a, b), (b, c), and (c, a) with weights 3, 2, and 1 respectively. If the algorithm examines the edges in their order listed, it will take the two heaviest edges instead of the two lightest.

   An efficient implementation will use disjoint sets to keep track of connected components, as in MST-REDUCE in problem 23-2. Trying to union within the same component will create a cycle. Since we make $|V|$ calls to MAKE-SET and at most $3|E|$ calls to FIND-SET and UNION, the runtime is $O(E\alpha(V))$.

c. This does return an MST. To see this, we simply quote the result from exercise 23.1-5. The only edges we remove are the edges of maximum weight on some cycle, and there always exists a minimum spanning tree which doesn't include these edges. Moreover, if we remove an edge from every cycle then the resulting graph cannot have any cycles, so it must be a tree.

   To implement this, we use the approach taken in part (b), except now we also need to find the maximum weight edge on a cycle. For each edge which introduces a cycle we can perform a DFS to find the cycle and max weight edge. Since the tree at that time has at most one cycle, it has at most $|V|$ edges, so we can run DFS in $O(V)$. The runtime is thus $O(EV)$.