COMPSCI 330: Design and Analysis of Algorithms	3/01/2016

Minimum Spanning Tree

Lecturer: Debmalya Panigrahi	*Scribe: Tianqi Song, Yilun Zhou, Nat Kell*

1 Overview

This lecture introduces basic concepts and two algorithms for minimum spanning tree: Kruskal's algorithm and Prim's algorithm.

2 Minimum Spanning Tree

Definition 1. *Given an undirected weighted connected graph $G = (V, E)$, a spanning tree is a subgraph $G' = (V, E')$ of G, where $E' \subseteq E$, such that G' is connected and acyclic.*

Definition 2. *A minimum spanning tree (MST) is a spanning tree with minimum total weight.*

2.1 Generic Property of Minimum Spanning Tree

Lemma 1. *Given an undirected weighted connected graph $G = (V, E)$, for any $S \subseteq V$, the lightest edge cross the cut $(S, V \setminus S)$ is included in any minimum spanning tree.*

Proof. Let T be a minimum spanning tree. Let the lightest edge cross the cut $(S, V \setminus S)$ be (u, v), where $u \in S$ and $v \in V \setminus S$. If T does not contain (u, v), we can find an edge $e \neq (u, v)$ in T which fulfills: (1) e is in the path from u to v and (2) e is an edge cross the cut $(S, V \setminus S)$. Such an edge has to exist because T is a spanning tree. We construct another spanning tree T' by deleting edge e from T and adding edge (u, v), and then T' has a smaller total weight which implies that T is not a minimum spanning tree. $\square$

2.2 Kruskal's Algorithm

The pseudocode is:

Algorithm 1 Kruskal's Algorithm

1: **function** KR($G = (V, E)$)
2:     $E' = \emptyset$
3:     $\forall v \in V$, initial a singleton set $\{v\}$
4:     Sort edges in nondecreasing order
5:     **for** each edge $(u, v) \in E$, taken in nondecreasing order **do**
6:         **if** u and v are not in the same set **then**
7:             $E' = E' \cup (u, v)$
8:             UNION(S_u, S_v), where $u \in S_u$ and $v \in S_v$
9:     Return E'

Lemma 2. *$G' = (V, E')$ is a spanning tree.*

Proof. If G' is not connected, there exist edges that should be selected by the algorithm but not in E', contradiction. Line 6 guarantees that G' is acyclic. □

Lemma 3. *After each selection of an edge by Kruskal's algorithm, there exists a minimum spanning tree $T = (V, E_t)$ such that $E' \subseteq E_t$.*

Proof. We prove it by induction. For the base case when $E' = \emptyset$, it is true. Assume that there exists a minimum spanning tree $T_n = (V, E_n)$ such that $E' \subseteq E_n$ when E' has n edges. For the $(n+1)th$ selection e_{n+1}, we add e_{n+1} to T_n. If there is no cycle, T_n is the tree that we want. If there is a cycle, there exists an edge e in the cycle such that $e \notin E'$. The weight of e must be not smaller than the weight of e_{n+1}, otherwise, e should have been selected by the algorithm. Therefore, the tree constructed by adding e_{n+1} to T_n and deleting e from T_n is also a minimum spanning tree. □

Theorem 4. $G' = (V, E')$ *is a minimum spanning tree.*

Proof. Directly from lemma 1 and lemma 2. □

2.2.1 Running Time

Line 4 of Kruskal's algorithm takes $O(m \log n)$ time since we have to sort all the edges, but the running time of the remainder of the operations depends on the implementation of the FIND and UNION and operations, i.e., given a collection of sets of elements, how do we find the set to which an element belongs (which we need to do on line 5), and given two collection of elements, how do we merge the collections into a single set (which we do on line 8). We will discuss how to implement these operations in Section 3. As we'll see, we can implement these operations such that the rest of procedure takes $O(m \log n)$ time (and so the entire procedure takes $O(m \log n)$ time). However, if we assume we are given the edges in sorted order, we can actually achieve a faster running time if we implement these operations carefully.

2.3 Prim's Algorithm

The pseudocode is:

Algorithm 2 Prim's Algorithm

1: **function** $\text{PR}(G = (V, E))$
2:     $c[s] = 0$
3:     $\forall v \neq s \in V, c[v] = +\infty, prev[v] = NIL$
4:     $E' = \emptyset$
5:     $H = V$
6:     **while** $H \neq \emptyset$ **do**
7:         $u = deletemin(H)$
8:         $E' = E' \cup (prev[u], u)$, if $u \neq s$
9:         **for** all $(u, v) \in E$, where $v \in H$ **do**
10:            **if** $c[v] > l(u, v)$ **then**
11:                $c[v] = l(u, v)$
12:                $prev[v] = u$

2.3.1 Running Time

Prim's algorithm has the same running time as Dijkstra's algorithm, $O(|E|\log|V|)$, by binary heap. It can be improved to $O(|E|+|V|\log|V|)$ by Fibonacci heap.

2.3.2 Correctness Proof

We prove that, after each selection of an edge by Prim's algorithm, there exists a minimum spanning tree $T = (V, E_t)$ such that $E' \subseteq E_t$. We prove it by induction. For the base case when $E' = \emptyset$, it is true. Assume that there exists a minimum spanning tree $T_n = (V, E_n)$ such that $E' \subseteq E_n$ when E' has n edges. For the $(n+1)th$ selection e_{n+1}, we add e_{n+1} to T_n. If there is no cycle, T_n is the tree that we want. If there is a cycle, there exists an edge $e \neq e_{n+1}$ in the cycle such that e only has one endpoint in $V \setminus H_n$, where H_n denotes H after n selections, because e_{n+1} only has one endpoint in $V \setminus H_n$. The weight of e must be not smaller than the weight of e_{n+1}, otherwise, e should be selected by the algorithm instead of e_{n+1}. Therefore, the tree constructed by adding e_{n+1} to T_n and deleting e from T_n is also a minimum spanning tree.

3 The Union-Find Data Structure

3.1 Storing Components for Kruskal's Algorithm

For a weighted graph $G = (V, E)$ where w_e denotes the weight of edge $e \in E$, recall Kruskal's algorithm for computing a minimum spanning tree (MST) of G. At a high level, we begin Kruskal's algorithm by initializing each vertex to be in its own component. Then in order of increasing edge weight, we repeatedly add edges to the tree if they merge two of the current components together (i.e., if $e = (u, v)$ is the edge we are considering, we add edge e to our tree if u is currently in a different component than v). Once all vertices lie in the same component, we argued that the resulting structure does in fact give us a MST.

However, when we previously outlined the pseudo-code for Kruskal's, we glossed over how to represent these collections of vertex sets in memory. On the iteration where we consider adding edge $e = (u, v)$, we need to quickly find out if u and v belong to the same component, and if they do not, we need to merge these components together.

To perform such queries and operations, we will implement a *union-find* data structure. A union-find data structure D is defined over a set of n elements $U = \{x_1, \ldots x_n\}$ and maintains a collection of disjoint subsets $S_1, \ldots S_h$ to which these elements belong, where $1 \leq h \leq n$. As in our scenario above, every element is in its own subset when D is initialized. D then supports the following two operations:

- FIND(x): return S_i such that $x \in S_i$ (in an actual implementation, we would likely just return the representative element for set S_i).

- UNION(S_i, S_j): Replace S_i and S_j with $S_i \cup S_j$ in the set system.

So for Kruskal's algorithm, we initialize a union-find data structure over the vertices. For each edge $e = (u, v)$, if FIND(u) $\neq$ FIND(v), then we call UNION(FIND(u), FIND(v)) to merge u and v's components (otherwise, we move on to the next edge). In total we will issue at most $2m$ FIND queries and always perform n UNION operations.

3.2   Implementing Union-Find

We now turn to the details of implementing $\text{FIND}(x)$ and $\text{UNION}(S_i, S_j)$ efficiently. Our first implementation decision is to how to represent the "labels" for each set. Here, we will use elements as representatives: At any given time, there will be a unique $x \in S_i$ which we will return as the label of S_i whenever we call $\text{FIND}(y)$ for any $y \in S_i$ (in the following implementations, we will make it clear how each representative is determined/maintained).

3.2.1   Union-Find with Linked Lists

The most obvious way to represent the set system is to just use a collection of linked lists. For each set S_i, we have a corresponding linked list L_i which contains the elements in S_i. The representative of L_i will just be element at the head of the list, which is then preceded by the the rest elements in S_i through a sequence of pointers. To execute $\text{FIND}(x)$, we start at x and follow the path of pointers leading to the head and then return it as the label of the set. Note that since there can be $\Omega(n)$ elements in a set, we might have to traverse $\Omega(n)$ links to reach a set's label; therefore when using linked list, $\text{FIND}(x)$ runs in $\Theta(n)$ time in the worst case. UNION operations, however, are quite simple. To implement $\text{UNION}(S_i, S_j)$, we just make the head of L_i point to the tail of L_j (or vice versa). Since we can store head/tail metadata along with the head of a list, UNION is an $O(1)$ time operation.

As noted above, a given run of Kruskal's may do $2m$ $\text{FIND}(e)$ queries, which gives us a $\Theta(mn)$ time algorithm in the worst case. When we first presented Kruskal's algorithm, we claimed a running time of $O(m \log n)$; therefore, using this linked list implementation will not suffice.

3.2.2   Union-Find with Trees

If we want to maintain the property that UNION operations still take $O(1)$ time, a natural improvement to this linked list scheme is to instead maintain a set of trees. Now, a set S_i corresponds to a tree T_i, where the representative of the set is at the root. To implement $\text{UNION}(S_i, S_j)$, we make T_i a subtree of T_j by making the root of T_j the parent of the root of T_i. Note that this implies that each tree is *not* necessarily binary since a fixed root r can participate in several UNION operations (it is possible that each UNION results in another subtree rooted at r).

$\text{FIND}(x)$ still works in the exact same way—we simply start at x and follow a path up the corresponding tree via parent pointers until we reach the root. Our hope is that if each tree structure remains balanced, then we can bound the longest path from node to root when doing a FIND query. However, our current specifications do not ensure balance. For example, consider the sequence of n unions

$$\{x_1\} \cup \{x_2\}$$
$$\{x_3\} \cup \{x_1, x_2\}$$
$$\{x_4\} \cup \{x_1, x_2, x_3\}$$
$$\vdots$$
$$\{x_n\} \cup \{x_1, \ldots, x_{n-1}\}.$$

Informally, we grow one particular set in the set system, and then with each UNION we add one of the remaining singleton sets to this growing set. When we perform $\text{UNION}(S_i, S_j)$ in this scheme, note that we are arbitrarily picking which root (the root of T_i or the root T_j) becomes the new root when we combine T_i and T_j. Thus in the above example, it is possible that when we merge $S = \{x_i\}$ with $S' = \{x_1, \ldots, x_{i-1}\}$,

we use x_i as the new root each time. If we are unfortunate enough to have this sequence of events happen for each union, then the resulting tree structure will just be an n element linked list (and therefore it is still possible for FIND(x) to take $\Omega(n)$ time).

A straight forward way to fix this pitfall is to do what is called *union-by-depth*. For each tree T_i, we keep track of its depth d_i, or the longest path from the root to any node in the tree. Now when we perform a UNION, we check to see which tree has the larger depth and then use the root of this tree as the new root. Note that this extra information can be easily stored and updated with the root of each tree: If we call UNION(S_i, S_j) and $d_i \leq d_j$, then the root of T_j becomes to root of $T_i \cup T_j$, and we update the depth of $T_i \cup T_j$ to be $\max(d_j, d_i + 1)$ (note this max is only necessary in the case where $d_i = d_j$—otherwise, the depth of the combined tree is no larger than depth of T_j).

What does "union-by-depth" buy us? The following theorem establishes that this feature does indeed balance the trees in the set system.

Theorem 5. *For a tree implementation of the union-find data structure that uses union-by-depth, any tree T (representing set S_i in the set system) with depth d contains at least 2^d elements.*

Proof. We do a proof by induction on the tree depth d. Since a tree T with depth 0 has has $2^0 = 1$ elements, the base case is trivial. For the inductive step, assume that the hypothesis holds for all trees with depth $k-1$, i.e., any tree with depth $k-1$ contains at least 2^{k-1} nodes. Observe that in order to build a tree T with depth k, we must merge together two trees T_i and T_j that both have depth $k-1$; otherwise, we would either have:

1. Both T_i and T_j have depth strictly less than $k-1$. Since the depth of $T_i \cup T_j$ can be no more $\max(d_j, d_i) + 1$, the combined tree $T_i \cup T_j$ can have depth at most $k-1$ (note this is true regardless of whether we use union-by-depth).

2. Exactly one tree has depth $k-1$; without loss of generality, suppose $d_j = k-1$ and $d_i < k-1$. Since we are using union-by-depth, we will make the root of $T_i \cup T_j$ the root of T_j. Since $d_i < k-1$, the length of any path from this new root of to any node in T_i can be at most $k-1$. Since T_j has depth $k-1$ and no node in within this subtree changes depth in $T_i \cup T_j$, the depth of the combined tree is exactly $k-1$.

Therefore, assume $d_i = d_j = k-1$; we can then apply our inductive hypothesis to both T_i and T_j to obtain:
$$|T| = |T_i \cup T_j| = |T_i| + |T_j| \geq 2^{k-1} + 2^{k-1} = 2^k,$$
as desired.

$\square$

Theorem 5 implies that any tree with n elements can have depth at most $\log n$ (the theorem implies $n \geq 2^d$ where d is the depth of the largest tree/subset, implying $\log n \geq d$). Therefore, FIND(x) runs in $O(\log n)$ when using union-by-depth. From Kruskal's perspective, this gives us the desired running time. The initial sort we do on the edge weights takes $O(m \log m) = O(m \log n^2) = O(m \log n)$ time. We then do n UNIONs that each take $O(1)$ time and $2m$ FINDs that each take $O(\log n)$ time. Therefore, the overall running time of Kruskal's using this implementation is $O(m \log n) + O(n) + O(m \log n) = O(m \log n)$.

3.2.3 Union-Find with Stars

Although doing a tree implementation that uses union-by-depth gave us the desired asymptotic running time of $O(m\log n)$, it is a bit unsettling that UNIONs take constant time and FINDs could take $\Omega(\log n)$ time. Since $n = O(m)$ for any graph where we want to find a spanning tree, it seems a bit wasteful that our implementation gives us a faster running time for the function we call fewer times (recall we perform n UNIONs and at most $2m$ FINDs). Therefore in this section, we will look at an implementation where we force each FIND to take $O(1)$ time, but as a result make UNIONs operations more expensive (but hopefully by not by too much).

The most naive way to achieve $O(1)$-time FINDs is to represent sets as *star graphs*. A star graph is simply a tree with a designated a center node such that every other node in the graph is a leaf that is only adjacent (or points) to this center node. Thus, we will maintain that each tree T_i that represents a set S_i is a star graph, where the center node of T_i is the representative of S_i. Clearly with this scheme, when we call $\text{FIND}(x)$ we must only traverse at most 1 link to reach the representative node, and therefore the running time of $\text{FIND}(x)$ is $O(1)$.

However to maintain this star graph structure, we will need to take more time when we make a UNION call. If we have two star graphs T_i and T_j that we want to merge, we first need to pick which representative element we will use for $T_i \cup T_j$ (just like for our previous implementation with balanced trees). If we pick T_i's center c_i to be the new center, we then need to iterate through every element $x \in T_j$ and make x point to c_i. Since T_j could have $\Omega(n)$ elements, this operation could take $\Omega(n)$ time. Therefore if we do n UNION operations, our running time for Kruskal's is now $\Theta(n^2)$ (which could be worse than $O(m\log n)$).

To avoid this problem, we will use a rule that is similar to union-by-depth. Namely, we will use *union-by-size*. Namely, if we are given two star graphs T_j and T_i, we will dissemble the smaller of the two sets and make these elements point to the center of the larger set (and leave the star graph in the larger graph untouched).

To analyze the speedup obtained from doing union-by-size, we use a charging argument to do an amortized analysis over the n UNIONs performed by Kruskal's. We use the folioing charging scheme: Any time we merge two trees T_s and T_ℓ such that $|T_s| \leq |T_\ell|$, we will simply put a unit of charge on each element in T_s (remember that we are taking the elements of T_s and changing their pointers to the center of T_ℓ). Note that for all $x \in T_s$, x now belongs to a set that is twice as large. We also know that for $x \in U$, the set to which x belongs can double at most $\log n$ times (the size of final merged set is n); therefore, the charge on a given element x can be at most $\log n$ after n unions. Since the total time needed over all n unions is equal to the total charge distributed over the elements, the time it takes to make n UNION calls is $O(n\log n)$. Note that even though Kruskal's algorithm still runs in $O(m\log n)$ time since we must initially sort the edges, we have reduced the time it takes to execute Kruskal's merging procedure to $O(n\log n + m)$.

3.2.4 Optimal Union-Finds: Path Compression and Union-by-Rank

We will now outline the best scheme for implementing a union-find data structure. This implementation will be more akin to what we saw in Section 3.2.2 when we used balanced trees to represent our set system. The main feature we will add to this implementation is what is known as *path compression*, which will attempt to make our trees more "star-graph-like" whenever we make a FIND call (so in some sense, we are combining the strategies in sections 3.2.2 and 3.2.3).

More specifically, whenever we call $\text{FIND}(x)$ where $x \in S_i$, we will follow some path P from x to the root r_i of S_i. For each element $y \in P$, we now know that y belongs to set S_i; therefore at this point, it makes sense to make each of these elements point directly to r_i. $\text{FIND}(x)$ with path compression does exactly this

modification, and therefore after the procedure completes, r_i and all the elements along P now form a star graph in T_i. Note that it is not too hard to implement $\text{FIND}(x)$ such that it returns r_i, makes every element in P point directly to r_i, and runs in $O(|P|)$ time.

To implement UNION, we essentially still use union-by-depth. We still merge components using the same rule (we make the tree with the smaller depth a subtree of the tree with the larger depth). Note, however, that because we might have path compressing FIND calls in-between UNION calls, we might compress a path that defined the depth a given tree T_i. In such a case, d_i no longer accurately stores the depth of d_i.

How does one fix this issue? The answer is that we do not. Instead, we just call this d_i the *rank* of T_i and use it in the same way we would in our union-by-depth scheme. It turns out that using these two features in combination gives us an extremely good bound over n UNIONs and $2m$ FINDs. Clearly, the n UNION operations still take $O(n)$ time in total since each UNION call takes $O(1)$ time. Using an advanced amortized analysis, one can show that the $2m$ $\text{FIND}(x)$ queries take $O(m \cdot \log^*(n))$ time, where $\log^*(n)$ is number of times we need to "log" the number n for it to become less than 1 (so for example, $\log^* \left(2^{2^{2^2}} \right) = 4$, since $\log \log \log \log \left(2^{2^{2^2}} \right) = 1$). Note that his function grows extremely slowly. For example, $\log^*(10^{80}) = 4$, and 10^{80} is roughly the number of atoms in the observable universe. So for practical purposes, we can think of $\log^*(n)$ as a constant; however we still express the running time as $O(m \cdot \log^* n)$ to be theoretically correct, since $\log^*(n)$ is still a function that tends towards infinity as n goes to infinity.