CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 18

November 7, 2019

Instructor: Sepehr Assadi

1 Algorithms for MST

Recall that in the previous lecture, we gave a meta-algorithm for MST that repeatedly picks a *safe* edge in a forest F until F becomes a tree; the definition of safe edges then allowed us to prove in this case the output of the algorithm is an MST.

We also proved the following general theorem regarding safe edges:

Theorem 1. *Let $G(V,E)$ be any undirected connected graph and F be any MST-good forest of G which is not a tree. Suppose $(S, V-S)$ is any cut in F with no cut edges*[1]*. Then the edge e with* minimum weight *among the cut edges of $(S, V-S)$* in G *is a safe edge with respect to F.* [2]

In this lecture and the next one, we use these findings to design two different algorithms for finding an MST. Both algorithms implement the strategy suggested by the meta-algorithm and pick their safe edges according to Theorem 1. The proof of correctness of both algorithms then simply boils down to proving that the edges picked by the algorithm indeed satisfy the property of Theorem 1 and are thus safe.

1.1 Kruskal's Algorithm

Kruskal's algorithm works as follows:

1. Sort the edges of G in increasing (non-decreasing) order of their weights.
2. Let $F = \emptyset$.
3. For $i = 1$ to m (in the sorted ordering of edges):
   (a) If adding e_i to F does *not* create a cycle, let $F \leftarrow F \cup \{e_i\}$.
4. Return F.

We now prove the correctness of this algorithm and see how to implement the step of this algorithm that needs to check whether e_i creates a cycle or not in F efficiently, using the *union-find* data structure.

Proof of Correctness: Consider the forest F maintained by the algorithm. We prove that if F is MST-good at some iteration i and in that iteration we inserted an edge e_i to F, then e_i was safe with respect to F. This then implies that we only added safe edges to F. Moreover, the output of this algorithm is always a tree since whenever we see an edge that does not create a cycle we add it to F (and since G was connected, we will find a tree eventually this way). That means that we added $n-1$ safe edges. This would imply the correctness of the algorithm as we now can say that this algorithm is an implementation of the meta-algorithm discussed in the previous lecture which we already proved its correctness.

[1]Such a cut always exists since F is not connected yet
[2]Such an edge always exists since G is connected.

Consider the iteration i in which we inserted the edge e_i to F. Let $e_i = \{u, v\}$. Since adding e_i to F did *not* create a cycle, we know that u and v were *not* in the same connected component of F. We can now define the cut $(S, V - S)$ where S is the connected component of u in F. Firstly, we know that none of the cut edges of G in $(S, V - S)$ belong to F (otherwise S would not have been a connected component of F). By discussion above we know that e_i is a cut edge of G in $(S, V - S)$ because $u \in S$ and $v \in V - S$. Finally, we know that e_i has the minimum weight among all cut edges of $(S, V - S)$: this is because at this point we know that e_i is the first cut edge of $(S, V - S)$ that we have visited in the algorithm (otherwise, we would have picked that edge sooner as it will not create a cycle in F, thus making it impossible for S to be a connected component of F); since the edges are sorted in non-decreasing order of their weights, we have that e_i has the minimum weight among cut edges of $(S, V - S)$.

We are now done since we can apply Theorem 1 and argue that e_i is a safe edge. This finalizes the proof.

Runtime Analysis: The first line of the algorithm takes $O(m \log m)$ time to sort all the edges (using, say, merge sort). We then have m iteration, each iteration involve deciding whether adding the edge e_i to the graph creates a cycle or not. The easiest way to implement this step is to run DFS/BFS – that will take $O(n+m)$ time per each iteration, making the total runtime of the algorithm $O(m \cdot (n+m)) = O(mn+m^2) = O(m^2)$ since we know that $m \geq n-1$ as G was connected (any connected graph needs at least $n-1$ edges).

However, this is *too inefficient* and we can in fact implement this line, using proper data structures called *union-find* (which we will describe below), in only $O(\log n)$ time with a preprocessing of $O(n)$ time – that will make the total runtime $O(n+m\log m+m\log n) = O(m \log m)$ (as again $m \geq n-1$). It is worth pointing out that we can implement that step *much faster* almost (but not quietly) in constant time; however that will not reduce the runtime as we still need to sort the edges in $O(m \log m)$ time in general.

Thus, overall, the runtime of the algorithm is $O(m \log m)$.

Detour: Union-Find Data Structure

We now describe the union-find data structure, also called disjoint-set data structure, or merge-set data structure. The goal of this data-structure is to maintain a collection of *disjoint* subsets $S_1, \ldots, S_k$ from a universe $U := \{1, \ldots, n\}$, allows us to find for each element $e \in U$, which of these sets e belong to, and to *merge*, or *union*, any two given sets S_i, S_j together, namely, remove S_i, S_j and replace them with $S_i \cup S_j$.

We now define the data structure formally. To do so, we simply need to define the operations supported by the data structure:

- `preprocess`(n): Set the universe $U = \{1, \ldots, n\}$. Create $k = n$ sets $S_1, \ldots, S_k$ where $S_i = \{i\}$. The `preprocess` operation should always be called before any other operation in the data structure; calling it again also will 'restart' the data structure (so it should only be called once at the beginning).
- `find`(e): Given an element $e \in U$, return the index of the set S_j where $e \in S_j$ (return j).
- `union`(i, j): Given index of two sets i, j, replace S_i, S_j with $S_i \cup S_j$ (with index of the new set becoming $\min\{i, j\}$).

These three are all the operations we require from our data structure.

Example:

- Suppose we first run `preprocess`(6) which creates $U = \{1, 2, 3, 4, 5, 6\}$ and the following $k = 6$ sets:
$$S_1 = \{1\}, \quad S_2 = \{2\}, \quad S_3 = \{3\}, \quad S_4 = \{4\}, \quad S_5 = \{5\}, \quad S_6 = \{6\}.$$
- Running `union`$(2, 3)$ results in:
$$S_1 = \{1\}, \quad S_2 = \{2, 3\}, \quad S_4 = \{4\}, \quad S_5 = \{5\}, \quad S_6 = \{6\}.$$

- Running $\texttt{union}(1,4)$ results in:

$$S_1 = \{1,4\}\,, \quad S_2 = \{2,3\}\,, \quad S_5 = \{5\}\,, \quad S_6 = \{6\}\,.$$

- At this point:

$$\texttt{find}(1) = 1, \quad \texttt{find}(2) = 2, \quad \texttt{find}(3) = 2, \quad \texttt{find}(4) = 1, \quad \texttt{find}(5) = 5, \quad \texttt{find}(6) = 6.$$

- We can further run $\texttt{union}(1,2)$ which results in:

$$S_1 = \{1,2,3,4\}\,, \quad S_5 = \{5\}\,, \quad S_6 = \{6\}\,.$$

  . . .

Implementation: There are different ways of implementing such a data structure and this problem was studied extensively in 60's, 70's, and 80's and at this point is very well-understood. We will not go over these implementations in this course (but you are strongly encouraged to check Chapter 21 of CLRS book for mind boggling aspects of these implementations). For our purpose, we only state the following bounds that correspond to the (almost) easiest implementation of this data structure (but again to emphasize, there are much better implementations as well). There is an implementation of the union-find data structure with:

- $\texttt{preprocess}(n)$ running in $O(n)$ time;
- $\texttt{find}(e)$ running in $O(\log n)$ time for every element $e \in U$;
- $\texttt{union}(i,j)$ running in $O(\log n)$ time also for every two given indices i,j of the sets.

How to Use Union-Find to Speed-Up Kruskal's Algorithm?

We now implement Kruskal Algorithm using the union-find data structure. The idea is to maintain the connected components of F as sets $S_1,\ldots,S_k$ inside the union-find data structure, update them accordingly throughout the algorithm, and use them to find whether an edge creates a cycle or not inside F.

1. Sort the edges of G in increasing (non-decreasing) order of their weights.
2. Create a union-find data structure D with $\texttt{preprocess}(n)$ (with universe $U = \{v_1,\ldots,v_n\}$).
3. Let $F = \emptyset$.
4. For $i = 1$ to m (in the sorted ordering of edges):
   (a) Let $e_i = (u_i, v_i)$. Let $a = D.\texttt{find}(u_i)$ and $b = D.\texttt{find}(v_i)$.
   (b) If $a = b$ continue to the next edge.
   (c) Otherwise, $F \leftarrow F \cup \{e_i\}$ and run $D.\texttt{union}(a,b)$.
5. Return F.

To show that this algorithm works we only need to show that we will add e_i to F if and only if F adding e_i does *not* make a cycle – this way, this algorithm will be identical to what described above.

To see how this algorithm works, simply consider the sets $S_1,\ldots,S_k$ maintained by the data structure. We prove by induction over index i of the for-loop that $S_1,\ldots,S_k$ at any point corresponds to the connected components of F (this is our induction hypothesis). The base case of the induction, say for simplicity when $i = 0$, corresponds to the beginning when $F = \emptyset$; in this case the sets in the data structure D are singleton sets (by definition of $\texttt{preprocess}$) which are exactly the connected components of F.

We now prove the induction step. Suppose up until some iteration i, we have connected components and sets in D equal to $S_1, \ldots, S_k$. We prove that after this iteration also this continues to hold. If we skip the edge e_i, we are neither changing D nor F so connected components of F and sets in D remain the same and thus by induction hypothesis we are correct. If we add e_i to F, we the two components containing endpoints of this edge will become the same component in F (they are now connected); at the same time, running D.`union` ensures that the corresponding sets in D will also be merged. Thus, in this case also, the connected components of F correspond to sets in D, proving the induction step.

Finally, note that in the algorithm, whenever we skip an edge e_i, it is because both end points of e_i belong to the same connected component (using above argument): since adding an edge to a connected component *always* makes a cycle, we should indeed skip adding this edge. On the other hand, when we decide to add the edge e_i to F, it is because endpoints of e_i belong to two different connected components (again using above argument): since adding an edge between two different connected components *never* create a cycle, we should indeed add this edge to F. This concludes the proof.

1.2 Prim's Algorithm

In the next lecture, we will go over Prim's algorithm for finding an MST.