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

Lecture 19

November 11, 2019

Instructor: Sepehr Assadi

1 Algorithms for MST

Recall the notion of *safe* edges, the meta-algorithm for MST, and the following theorem about safe edges from previous lectures:

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]

We used these in the previous lecture to design Kruskal's algorithm, which we quickly review below. This lecture then focuses on another implementation of the meta-algorithm for MST, namely, Prim's algorithm.

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 proved the correctness of this algorithm in the previous lecture. We also showed how to implement this algorithm in $O(m \log m)$ time using the union-find data structure.

1.2 Prim's Algorithm

We now go over Prim's algorithm for finding an MST. As can be easily observed in Kruskal's algorithm, we are maintaining *multiple* connected component of the forest F simultaneously and update them (merge them together in the algorithm). On the other hand, Prim's algorithm only contains *one non-trivial* connected component (all other components are singleton vertices) which we "grow" until it contains the entire graph. The strategy for growing the component is also very basic: we simply pick the edge with minimum weight that goes out of this component in every step. Prim's algorithm is formally as follows:

[1] Such a cut always exists since F is not connected yet

[2] Such an edge always exists since G is connected.

Prim's Algorithm:

1. Let $mark[1:n] = FALSE$ and s be an *arbitrary* vertex of the graph.
2. Let $F = \emptyset$.
3. Set $mark[s] = TRUE$ and let S (initially) be the set of edges incident on s.
4. While S is non-empty:
   (a) Let $e = \{u, v\}$ be the *minimum weight* edge in S and remove e from S.
   (b) If $mark[u] = mark[v] = TRUE$ ignore this edge and go to the next iteration of the while-loop.
   (c) Otherwise, without loss of generality, assume $mark[v] = FALSE$ (if $mark[u] = FALSE$ simply switch the name of u,v below).
   (d) Set $mark[v] = TRUE$, add all edges incident on v to S, and add $\{u, v\}$ to F.
5. Return F.

Proof of Correctness: There are two steps in proving the correctness of Prim's algorithm: (1) it outputs a spanning tree, and (2) the weight of this spanning tree is minimum, namely, it is an MST. The proof of first part is identical to the proof of correctness of BFS and DFS (note that the only difference between these algorithms is that in BFS we stored vertices in a queue and removed them while here we are storing the edges and then examine their unmarked endpoints). We thus prove the second (and the main) part below.

The plan as before is to show that Prim's algorithm implements the meta-algorithm discussed earlier by picking safe edge. This in turn is done by proving that any edge $\{u, v\}$ added to F satisfies the properties of Theorem 1 and thus is a safe edge.

At the beginning of each iteration of the while-loop, define C as the set of all marked vertices. By part (1) of the argument (and correctness of DFS/BFS), C is the connected component of s in the forest F at the beginning of this iteration. Consider the cut $(C, V - C)$ in this iteration. This cut has no cut-edges in F by definition as C is a connected component. We claim that the edge e chosen in this iteration, either has both its endpoints marked TRUE, or is the minimum weight edge among the cut-edges of $(C, V - C)$ in G which implies that this edge is a safe edge by Theorem 1. As such, we only add an edge e to F if this edge is safe.

We prove this by showing that *all* cut-edges of $(C, V - C)$ in G belong to the set S at the beginning of this iteration: whenever a vertex v joins C (by setting $mark[v] = TRUE$) we add all edges incident on v to S and we definitely never removed any cut-edge of $(C, V - C)$ before as otherwise we already added those edges to F and C would not be a connected component of F anymore (note that for a cut-edge only one endpoint can be marked TRUE). Since we are returning a minimum weight edge from S at this step, we know that weight of $\{u, v\}$ is smaller than weight of all edges in S and in particular all cut-edges of $(C, V - C)$ in G. This proves that $\{u, v\}$ is a minimum weight edge in the cut $(C, V - C)$.

Runtime Analysis: Similar to Kruskal's algorithm, there is a direct (but not so efficient) way of implementing this algorithm. Simply store the set S in an array or a linked list and then in each iteration of the while-loop, spend $O(|S|) = O(m)$ time to find the minimum weight edge of S. It is easy to see that in this case the total runtime of the algorithm would be $O(n + m + m^2) = O(m^2)$. However, we can implement this algorithm more cleverly in $O(m \log m)$ time using a standard data structure, namely, a *min-heap*.

Detour: Min-Heap Data Structure

We now briefly go over the *min-heap* data structure that you most likely have seen in your previous courses. The goal of this data structure is to maintain a collection of numbers (or more generally *(key, value) pairs*), allow us to add more numbers to this collection, and extract the *minimum* number from it.

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

- `preprocess`: Create an empty set $U = \emptyset$ which will (later on) contain the numbers we insert to the heap. 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).
- `add`(e): Adds the element e to U.
- `extract-min`: Remove the minimum number from U and returns it.
- `size`: returns the number of elements in the heap.

These three are all the operations we require from our data structure (although some variants of min-heap support more functionalities as well).

Implementation: There is a simple way of implementing the min-heap data structure that many of you have probably seen already (see Chapter 6.1 of CLRS for a refresher). For our purpose, we only state the following bounds that follow from this simple implementation:

- `preprocess` runs in $O(1)$ time;
- `add`(e) runs in $O(\log n)$ time where n is the number of the elements in the heap;
- `extract-min` runs in $O(\log n)$ time where n is the number of elements in the heap;
- `size` runs in $O(1)$ time.

How to Use Min-Heap to Speed-Up Prim's Algorithm?

We run the following algorithm by replacing the set S of edges with a min-heap H.

1. Let $mark[1:n] = FALSE$ and s be an *arbitrary* vertex of the graph.
2. Let $F = \emptyset$ be the maintained forest and set $mark[s] = TRUE$.
3. Let H be a min-heap data structure: Call H.`preprocess` and H.`add`(e) for every edge e incident on s.
4. While H.`size` > 0:
   (a) Let $e = \{u, v\} = H$.`extract-min`.
   (b) If $mark[u] = mark[v] = TRUE$ ignore this edge and go to the next iteration of the while-loop.
   (c) Otherwise, without loss of generality, assume $mark[v] = FALSE$ (if $mark[u] = FALSE$ simply switch the name of u,v below).
   (d) Set $mark[v] = TRUE$, call H.`add`(e) for all edges e incident on v, and add $\{u, v\}$ to F.
5. Return F.

It is immediate to verify that this algorithm is the same exact implementation of Prim's algorithm above with the only difference being that the set S replaced by a min-heap H. The runtime of this algorithm is now $O(\log m)$ by each iteration of the while-loop to extract the minimum and $O(\deg(v) \cdot \log m)$ for each vertex v to insert all edges incident to v in the heap (note that size of the min-heap never gets more than $O(m)$). This means that the total runtime is $O(m \log m)$ as desired.