COMPSCI 330: Design and Analysis of Algorithms **Mar 4, 2020**

Lecture 14: Minimum Spanning Tree II

Lecturer: Rong Ge *Scribe: Haoming Li*

Overview

In the previous lecture, we introduced the definition of a Minimum Spanning Tree (MST), as well as some properties of it. In this lecture, we will prove the Key Property (aka. Cut Property, worded differently in the DPV textbook) and derive Prim's Algorithm, an algorithm for finding a MST. We will also discuss Kurskal's Algorithm that does the same job.

14.1 Prim's Algorithm

Say that in the process of building a MST, we have already chosen some edges and are so far on the right track. Which edge should we add next? The following lemma gives us a lot of flexibility in our choice

14.1.1 Key Property

Lemma 14.1 (Key Property) *Suppose F is a subset of edges of a MST T. Pick any cut $(S, \bar{S})$ that does not intersect with F, and let e be (any) edge with minimum cost in this cut, then $F \cup \{e\}$ is a subset of edges of an MST T' (which may not be the same as T).*

A cut is any partition of the vertices into two groups, S and $\bar{S} = V - S$. What this property says is that, when building a MST, it is always safe to add the lightest edge across any cut (that is, between a vertex in S and one in $\bar{S}$), provided F has no edges across the cut. Let prove why this holds.

Proof: Edges F are part of some MST T; if the new edge e also happens to be part of T, then there is nothing to prove. So assume e is not in T.

Adding e to T will form a cycle, call it C. We know cycle C must intersect with cut $(S, \bar{S})$ in an even number of edges, so there must be another edge $e' \in C$ that also crosses cut $(S, \bar{S})$. We will swap e and e'. Define $T' = (T \backslash \{e'\}) \cup \{e\}$. We have $cost(T') = cost(T) - w(e') + w(e) \leq cost(T)$, since $w(e) \leq w(e')$ by the assumption that e is a min-cost edge across the cut. This means that T' is a MST, and that $F \cup \{e\} \subseteq T'$ ■

14.1.2 Prim's Algorithm

What Key Property tells us in most general terms is that any algorithm conforming to the following greedy schema is guaranteed to work.

```
Algorithm: Prim($G = (V, E)$)
Initialize the tree $F$ to be empty;
while $|F| < |V| - 1$ do
    Find a cut that does not have any edge in the current tree;
    Add the min cost edge of the cut to the tree;
end
```

With the help of Key Property, we can quickly prove the correctness of Prim's Algorithm by induction.

Proof:

Inductive Hypothesis: At iteration i, the edges selected by the algorithm is a subset of some MST.

Base Case: When $i = 0$, the set of edges selected is empty.

Induction Step: see Key Property. ■

When implementing Prim's Algorithm, we want to efficiently find 1) a cut that does not go through any edges we have chosen, and 2) a min-cost edge in the cut. We can choose the cut such that $F = S$, and use a data structure similar to that in Dijkstra's algorithm. The only difference is in the key values by which the priority queue is ordered. In Prim's algorithm, the value of a node is the weight of the lightest incoming edge from set S, whereas in Dijkstra's it is the length of an entire path to that node from the starting point. In particular, we can maintain $dis[u]$ for all u not visited, where $dis[u]$ is the min-cost of an edge (u, v) between u and a visited vertex v.

Nonetheless, the two algorithms are similar enough that they have the same running time, which depends on the particular priority queue implementation. A naive implementaion of Prim's runs in $O(n^2)$, whereas a Fibonacci heap version runs in $O(m + n \log n)$.

14.2 Kruskal's Algorithm

Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from E according to the following rule: **Repeatedly add the next lightest edge that doesn't produce a cycle.**

In other words, it constructs the tree edge by edge and, apart from taking care to avoid cycles, simply picks whichever edge is cheapest at the moment. The proof of correctness of this algorithm centers on showing that the output is a special case of general MST. We are going to show that, every time Kruskal's adds an edge (u, v), we can find a cut $(S, \bar{S})$ where $u \in S$ and $v \in \bar{S}$, such that the cut does not contain any previous edges and that (u, v) is the min-cost edge in the cut.

Proof: Consider when Kruskal's adds an edge (u, v). Let S be the set of vertices that are connected to u using edges already selected by Kruskal's. By design of $v \notin S$, therefore we only need to show that (u, v) is the min-cost edge between $(S, \bar{S})$.

Assume towards contradiction that there is an edge e that crosses the cut $(S, \bar{S})$ such that $w(e) < w((u, v))$. By Kruskal's, edge e is going to be considered before (u, v). When e was considered, $(S, \bar{S})$ were not connected. Therefore, e cannot create a cycle. Hence Kruskal's must have selected e. This is a contradiction. ■

For each edge, we need to check whether adding the edge creates a cycle. This can be done very efficiently in $O(\log n)$ time via a union-find data structure. The idea is that we maintain disjoint sets of the vertices (starting with every vertex being in its own set, which corresponds to the case that none of the vertices are connected) and perform two operations iteratively:

- union(): merges two sets.
- find(): given a vertex, find the set it belongs to. i.e. if u, v are in the same set, then find(u)=find(v); if not, then find(u)≠find(v).

We will implement each set as a tree with directed edges, where every vertex maintain a pointer to its parent. find(v) is therefore finding the root node of the tree v is in, which takes $O(\log n)$ time. union($tree_1$, $tree_2$) is therefore finding the root nodes of two trees and point one to the other; the latter takes $O(1)$ time.

The action of adding a new edge (u, v) in Kruskal's can therefore be implemented as union(find(u), find(v)). This implementation allows us to check for whether adding (u, v) creates a cycle in $O(\log n)$ time.

As a whole, Kruskal's takes $m \log m = O(m \log n)$ time to sort edges by weight, takes $2m = O(m)$ find() operations, and $n - 1$ union() operations. Hence the total running time is $O(m \log n)$.

14.3 Comparison of Running Time

Prim seems to be always faster ($O(m + n \log n)$ vs. Kruskal's $O(m \log n)$). However, the $O(m + n \log n)$ version of Prim is not very easy to implement, and has a large hidden constant. If you use a regular binary heap, the running time are the same, and Kruskal's is usually faster in practice, and easier to implement. If the graph is dense, $O(n^2)$ version of Prim is easy to implement and faster than Kruskal's.