Prim’s MST Algorithm and Priority Queues

Instructor: Richard Peng

Sep 21, 2016

• DISCLAIMER: These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

• Main topics:
  – Prim’s algorithm and how it relates to cut rule.
  – Speedups using priority queue.

• This week in a nutshell:
  – Cut property: edge in MST iff its minimum on some cut.
  – Kruskal’s algorithm: sort by weights, greedy.
  – Prim’s algorithm: start with single vertex, greedily add minimum edge in cut between reached and unreached vertices.
  – Running time: straight implementation $O(nm)$ for Kruskal, $O(n^2)$ for Prim.
  – Union find: merge smaller into bigger, $O(\log n)$ per union, $O(m \log n)$ Kruskal.
  – Priority queue: Prim’s algorithm in $O(m \log n)$ as well.
  – Can also use priority queue to speed up shortest path with positive edge weights, $O(m \log n)$: Dijkstra’s algorithm.

• Comments from Last Time:
  – How to invoke cut rule in Kruskal’s algorithm?
  – What kinds of proofs are needed?

Last two classes we discussed the cut property for minimum spanning trees (MSTs): an edge $e$ is in the MST if it’s the minimum weight across some cut. This is used in Kruskal’s algorithm as follows:

1. The connectivity in the forest before we consider $e$ is precisely the connectivity with all edges of weight $< w_e$, let this graph be $H$.

2. If $u$ and $v$ is disconnected in $H$, then there is a cut in $H$ separating $u$ and $v$.

3. Let this cut be $S$, then $E_H(S, V \setminus H)$ is empty, so $E_G(V, V \setminus H)$ only contains edges with weights $\geq w_e$. 

1
Another algorithm that invokes the cut property far more directly is Prim’s algorithm:

1. Initialize \( S = \{u\} \) for some vertex \( u \)

2. While \( |S| < n \).
   
   (a) Find edge of minimum weight in the cut \( E(S, V \setminus S) \).
   
   (b) Add this edge to the tree, and update \( S \) to include both its endpoints.

The correctness of this algorithm follows from the observation that we always add the edge of minimum weight from the cut \( E(S, V \setminus S) \). It also proves that the cut rule is iff: we only added edges that are minimum in some cut.

This algorithm can be implemented in \( O(nm) \) time by looping through all edges at each step. We can do better by tracking for each vertex \( u \), the minimum weight of a neighbor it has in \( S \). Since \( S \) is growing, each time we add a vertex to \( S \), we can updates this value for all its neighbors in \( O(n) \) time. When we want the minimum weight edge, we can also find this information by looping through all vertices in \( V \setminus S \) in \( O(n) \) time. This gives an \( O(n^2) \) time algorithm.

In the rest of this lecture we will see how to make it run in \( O(m \log n) \) time. This hinges on the observation that an edge enters and leaves \( E(S, V \setminus S) \) exactly once. Let the edge be \( e = uv \), and assume \( u \) enters \( S \) first. Then \( uv \) is on the cut precisely between the steps where \( u \) enters \( S \) and \( v \) enters \( S \). This means we can put all the edges into a priority queue, which is a data structure that supports:

1. insert / delete an element,
2. find the element with minimum key,

There are many ways of implementing a priority queue so all operations are \( O(\log n) \) time. You are not required to know how it operates, just the interface of insert/delete/query.

The main idea of a priority queue is to put the data points in a balanced binary tree so that each node’s value is greater than its parents. This way the root always contains the minimum, and the updates are handled by:

1. Insert: swap with parent as long as you’re less than them.
2. Delete: set value to \( \infty \), swap with lower valued child, repeat.

If we view insert/delete as changing keys, a priority queue also has a simple interpretation through divide and conquer: we partition the data evenly, and store the minimum of each half and repeat. Note that the minimum of a set can be recomputed from the minimum of its two subsets in \( O(1) \) time, assuming that those values are correct.

Every time we update, we update the minimum of all the subsets containing it in a bottom up manner, maintain correctness of the value inductively. As the sizes of the sets are geometrically increasing, each node can belong to at most \( O(\log n) \) of them, giving the \( O(\log n) \) cost per update.