Lecture 39

Minimum Spanning Tree

Consider the problem of connecting different locations to a network. It may be sufficient that all nodes are reachable, but there need not be any redundancy in the system. Some examples may be connecting houses to the city water system, or connecting cities to the electrical grid. In these cases, we want to ensure connectivity of the graph, but we also want to minimize the cost of the infrastructure.

Some more definitions that are relevant to the problem:

A **subgraph** G' is a subgraph of G if V' $\subset$ V and E' $\subset$ E.

Spanning Tree a tree which contains every vertex of a connected graph G.

Minimum Spanning Tree spanning tree with the smallest total weight.

The problem described above reduces to the problem of calculating the minimum spanning tree of a graph. There are two obvious and straightforward ways to do this, and it turns out that they both work!

Prim-Jarník

Prim-Jarník Algorithm:

1. Select start node
2. Repeat until V - 1 edges chosen:
   (a) Add cheapest connected edge that does not form a cycle

Often called Prim's algorithm, it was originally discovered by Vojtěch Jarník in 1930, and independently by Robert Prim in 1957. The basic idea is that we start with some node, and build a single tree by adding adjacent edges at each step. Each edge must add a new vertex to our tree, and thus cannot result in a cycle.

How do we know that the loop iterates exactly $|V| - 1$ times?

Prim-Jarník run time

When we discuss run time, we must consider the different ways of representing our graph – an adjacency matrix or an adjacency list.

Adjacency matrix

For an adjacency matrix implementation, our matrix is size $|V|$ x $|V|$, with each row representing the *from* node of an edge, each column representing the *to* node of and edge, and the value stored in the matrix representing the weight of the edge.

At each step, we maintain an array of vertices A_{out} that are not yet in the tree, and the cost of adding them to the tree in the next step. We also maintain a map of vertices and their parent (the node they are connected to that caused them to be added to the tree).

We begin the algorithm by adding the start node to the current tree (that is, the current tree consists of one vertex and no edges). We traverse the row in the matrix for the start node, and if there is an edge from another vertex to the start node, we save its weight for that adjacent vertex in A_{out}, and we save the start node as its parent. If there is no edge, we save the value ∞ for that vertex and `nil` for its parent.

Note that our set up described above has a cost of $O(|V|)$, since there are $|V| - 1$ vertices not in the tree.

Now, at each step, we find the vertex with the cheapest cost to add to the tree, and add it. We traverse the row in the matrix for the new vertex, looking for a cheaper way to add each "out" vertex to the new tree. If we find one, we update A_{out} and the parent map accordingly. Note that each step takes $O(|V|)$ time, since we must traverse an entire row of the matrix.

Thus, the total time for the adjacency matrix implementation is $O(|V|)$ steps times $O(|V|)$ work per step, or $O(|V|^2)$

Adjacency list

For the adjacency list implementation, we use a priority queue (i.e., a min-heap) to keep track of the cheapest edge so far.

We begin by adding the start node to the tree. We then build the priority queue with the other $|V| - 1$ vertices, storing the weight of their edge to the start node as their priority, and the start node as their parent (as before, we use ∞ for their priority and `nil` for their parent if they are not connected to the start node).

At each step, we select the next node to add to the tree. This takes constant time to get the vertex, but $O(\log(|V|))$ time to reheapify.

For the added vertex, we must traverse its adjacency list looking for better edges for each of the vertices remaining in the priority queue, and updating their values and parent map as necessary. The number of times that we do this at each step depends on the nature of the graph, but we know that the total number for the whole algorithm is $2|E|$ (because each edge has two adjacency list entries – one for the source node and one for the destination).

Each examination of an edge requires (in the worst case) a change to the priority of a node in the heap, and therefore another reheapification. Thus, the total run time for the algorithm is the work to select and reheapify the next vertex plus the work to update the heap for the adjacent edges, or:

$$O(|V|\log(|V|)) + O(|E|\log(|V|)) = O((|V| + |E|\log(|V|)) = O(|E|\log(|V|))$$

We get $O(|E|\log(|V|))$ since we know that $|E|$ is $O(|V|)$ at least.

Adjacency matrix versus adjacency list

Which approach is better? It depends on how sparse the graph is. If the graph is very sparse, then $|E|$ is $O(|V|)$. The runtime of the adjacency list implementation is then $O(|V| \log(|V|))$, which is better than the adjacency matrix time of $O(|V|^2)$.

However, if the graph is very dense, then $|E|$ is $O(|V|^2)$, and the adjacency list implementation is $O(|V|^2 \log(|V|))$, which is worse than the adjacency matrix implementation.