CS61B Summer 2006
Instructor: Erin Korber
Lecture 29: 15 Aug.

Graphs - Minimum Spanning Trees

Let G = (V, E) be an undirected graph. A *spanning tree* T = (V, F) of G is a graph containing the same vertices as G, and $|V| - 1$ edges of G that form a tree. (Hence, there is exactly one path between any two vertices of T.) If G is not connected, then T is not a tree; rather, it is a *forest*, or collection of trees, having as many trees as G has connected components.

If G is weighted, then a *minimum spanning tree* T of G is a spanning tree of G whose total edge weight is minimal. In other words, no other spanning tree of G has a smaller total weight.

In graphs with no negative edge weights, the minimum spanning tree is also the minimum-cost subgraph that connects all the vertices (since any graph with cycles would necessarily have higher cost).

The most obvious application of minimum spanning trees is in designing networks for some kind. For example, if a cable company was trying to provide cable service to a group of houses (nodes in the graph), the minimum spanning tree would be the minimum-cost way to connect them.

The following fact will be essential correctness of both of the algorithms that we discuss.

FACT: Let G be a weighted connected graph, and let V1 and V2 be a *partition* of the vertices of G such that V1 and V2 are disjoint, nonemtpy sets. Let edge e be the lowest-weight edge connecting a vertex in V1 to a vertex in V2. Then e is part of the minimum spanning tree.

Proof: Let T be a minimum spanning tree of G. If T does not contain e, then T + e must contain a cycle (since in T, there is already a path from every node to every other node). Therefore, there is some edge f in this path with one endpoint in V1 and the other in V2. By the choice of e, we know that e.weight $\leq$ f.weight. If we remove f from T + e, we have a spanning tree whose total weight is $\leq$ T's weight. Since T was a *minimum* spanning tree, the new tree T + e - f must also be a minimum spanning tree.

1 Kruskal's algorithm

Kruskal's algorithm uses the preceeding fact to build the minimum spanning tree by joining together smaller subtrees.

The algorithm (on graph G = (V,E)):

- Create a new graph T with the same vertices as G, but no edges. (so T is a forest)
- Make a list Q containing all the edges in E, sorted by weight
- for each edge e = (u,v) in Q:

  * if there is not already a path in T from u to v, add e to T (which will combine two trees into a single tree) (else , we do nothing with e).

When the algorithm terminates, T will be a minimum spanning tree of G.

1.1 Running time

The running time of Kruskal's algorithm depends on two things: the amount of time to sort the list of edges, and the time to determine if there is already a path in F between two nodes. We know we can sort the edge list in $O(|E| \log |E|)$ time (or perhaps even $O(|E|)$ if they can be sorted with radix or counting sort).

The simplest way to determine if u and v are already connected is by traversing T (either depth- or breadth- first) starting from u and seeing if we reach v. But this could take up to $\Theta(|V|^2)$ time. Tomorrow, we'll learn a method for solving the problem quickly, in $O(\log |E|)$ time, so that the $|E|$ iterations will together take $O(|E| \log |E|)$ time. Therefore, the overall running time of Kruskal's algorithm will be in $O(|E| \log |E|)$.

Furthermore, since E is at most $|V|^2$, $\log |E|$ is at most $2 \log |V|$. So the running time is in $O(|E| \log |V|)$.

2 Prim's algorithm

Prim's algorithm runs very similarly to Dijkstra's algorithm. We begin with a single root vertex r, and add edges connecting new vertices until they are all connected. As in Dijkstra's algorithm, we'll use a "distance" variable for each vertex to keep track of the shortest edge so far for joining that vertex to the result tree T. We'll also track which edge that is.

The algorithm (on graph G = (V,E)):

- For all v in V, set v.dist = infinity, v.theEdge = null.
- Create an empty graph T
- Choose a vertex r from V. Set r.dist = 0.
- Make a priority queue Q of all the vertices, with distance as the priority.
- While Q is not empty:
  - u = Q.removeMin
  - Add vertex u and edge u.theEdge to T.
  - For each vertex w such that e = (u,w) is in E and w is still in Q
    - if e.weight < w.dist
      - Set w.dist = e.weight
      - Set w.theEdge = e

When the algorithm terminates, T will be a minimum spanning tree of G.

2.1 Running time

The analysis of the running time of Dijkstra's algorithm will in general apply to Prim's algorithm also. If we implement Q as a binary heap, we can perform removeMin and updatePriority in log(n) time. So the total time for the $|V|$ removeMin operations will be in $O(|V| \log |V|)$. The total time for the $|E|$ updatePriority operations will be in $O(|E| \log |V|)$. So the total running time is in $O((|V| + |E|) \log |V|)$, which is $O(|E| \log |V|)$.