Class Note #10

Date: 02/13/2006

[Overall Information]

In this class, we started studying the Minimum Spending Tree problem. An algorithm solving this problem was given. Its correctness was proved, and its running time was analyzed.

[During the Lecture]

1. After the class began, the statistics of quiz#2 including minimum, maximum and average was given. It seems most students know Big-O notation well now.

The next quiz (quiz#3) will take place on 02/27/2006 (in contrast to what was announced in class). There will be one more homework set before it.
2. Today’s lecture started from network design. The general definition of network design is: buy potential (available) edges at cost \( c_e \geq 0 \), so that the resulting graph \( G(V,E) \) meets certain requirements, in particular requirements such as connecting certain pairs of nodes, or all nodes to each other. More specifically, in the following, we will study the problem where we want to connect all nodes by undirected paths. Thus, what we are looking for is a cheapest subset of edges \( E' \) such that \( (V, E') \) is a connected graph. The total cost of an edge set is the sum of the costs of all edges in the set, and cheapest means that that sum is minimized.

Network Design:

Buy edges at costs \( c_e \geq 0 \)

in a graph \( G = (V,E) \), to meet certain connectivity requirements.

Here: make sure there is an undirected path between each pair of nodes (assume that \( G \) itself was connected.)

Solution is an edge set \( E' \subseteq E \) \( (V,E') \) is connected, and \( E' \) is the cheapest such set, 

\[ c(E') = \sum_{e \in E'} c_e \] is the cost.
The first observation we make is that if all edge costs are non-negative, then without loss of generality, the solution contains no cycles. (Here, “without loss of generality” means that if we had an optimal solution with a cycle, then all edges on that cycle would have cost 0, so we may as well leave on out and get an optimal solution without a cycle).

Because a connected acyclic graph is exactly a tree, this problem is usually known as the Minimum Spanning Tree problem. It has obvious applications in the design of computer or road networks (or their maintenance – it corresponds to keeping a “communication backbone”), but is also very important as a subroutine for many other problems, including the traveling salesman problem.

In thinking about how to come up with an algorithm for this problem, we started with a few observations:

(0) First off, to simplify the presentations (and proofs) a bit, we will assume that all edges have distinct costs. This is not a big deal, but will
have us worried less about breaking ties

(1) If a node has only one edge, that edge must be in the solution.

(2) The edge with the cheapest cost must be in the solution.

The second observation can be proved by first assuming that there is a solution not containing the cheapest edge. If we add the cheapest edge into the tree, we will form a cycle. After that, we can remove a more expensive edge and preserve this cheapest one to form a cheaper solution (which is still connected).

5. Actually, this idea can be generalized.

We define a “cut” to be a partition of the graph's nodes into two sets $S_1$ and its complement $S_2$. We can then state the following lemma:
**Lemma (Cut Property):** For each cut \((S_1, S_2)\), the cheapest edge crossing the cut must be included in the optimal solution (we say that the edge \(e = (u,v)\) crosses the cut if one of its endpoints is in \(S_1\) and the other in \(S_2\).

The proof is similar to the proof of the second observation. We start out with a solution \(T\) which does not contain the cheapest edge \(e = (u,v)\). Because it is connected, there must be a path in \(T\) from \(u\) to \(v\). But because that path starts in \(S_1\) and ends in \(S_2\), it must cross the cut at some edge \((u',v')\). Because that edge is not as cheap as \(e\), removing it and adding \(e\) instead gives us a cheaper solution, which is still connected.
6. For each edge in the graph, there are many cuts it crosses. The second lemma states that if an edge is inside an optimal solution, then it must be the cheapest edge across some cut. This can be proved by the similar “add and remove” strategy.

7. The two lemmas suggest a generic outline of an algorithm we could use to solve this problem. The algorithm adds the cheapest edge across some cut as long as the set T it has does not span all nodes yet. In instantiating the algorithm, we have to decide which cuts the algorithm considers (in which order), and which edges are added as a result.

One way is to start from some node s. Initially, $S_1=\{s\}$ and $S_2=V\setminus\{s\}$. Then for each iteration step, we add the cheapest edge between $S_1$ and $S_2$ and update $S_1$ and $S_2$ accordingly (move the new node of the new edge
from $S_2$ to $S_1$). This algorithm is known as Prim's algorithm, and bears a lot of resemblance to Dijkstra's Shortest Paths algorithm.

8. The correctness of the algorithm is characterized by the following statements:

(1) All nodes are spanned;

(2) There is no cycle;

(3) The cheapest solution is found.

The corresponding proofs are:

(1) At the termination of the algorithm, all the nodes are explored and added;

(2) For each step, we add a new edge, one end point of which is not in $S_1$. Such an operation cannot generate a cycle;

(3) Whenever the algorithm adds an edge $e$, it is the cheapest edge
crossing the cut \((S_1, S_2)\) by definition of the algorithm. Thus, by the cut property, \(e\) must be an edge in the optimal solution. So the solution found by the algorithm is a subset of the optimal solution, and at termination contains the same number of elements as the optimal set. As a result, what the algorithm found is the optimal solution.

Next, the class moved to the runtime analysis for Prim's algorithm. A simple implementation would scan all edges in each iteration step to see if they are crossing the cut and to find the cheapest one among them. There are \(n\) iterations all together, consequently, the algorithm takes \(O(mn)\) time.

The running time can be improved by, for each node \(v \in S_2\), only keeping track of the one cheapest edge \((u,v)\) with \(u \in S_1\). This can be done by updating \(a[v]\) values for all \(v\) not belonging to \(S_1\) and connected to \(v\) whenever node \(v\) is added to \(S_1\).
The operations need for finding the right edge to add are are find-min, insert, delete-min and update. Thus, a heap data structure is a good solution. The running time then becomes $O(m \log(n))$, as each edge causes at most one heap update.