Class Note #11

Date: 02/15/2006

[Overall Information]
In today’s class, we studied a new algorithm, Kruskal’s algorithm, solving the Minimum Spanning Tree problem. We proved its correctness, and analyzed the running time of different implementations.

[Announcements]
(1) Homework No.3 has been assigned. You can turn in the homework to the professor or the TA if you want some feedback.

(2) Quiz#2 is rescheduled for 02/27/2006 due to the conflict with the Presidents Day. There will be no lecture and no TA office hours on Presidents Day (next Monday).

(3) There will be an interesting talk about the future of Microprocessor Architecture this Friday from 3 to 5pm at GER, given by Dr. John Hennessy, a leading figure in this area.
At the beginning of today’s lecture, the class briefly reviewed the Minimum Spanning Tree problem we studied last time. We went over the two properties we used and the algorithm we studied.

In order to derive a different (Kruskal's) algorithm, we started from an example. Last time, we observed that the cheapest edge should always be in the minimum spanning tree. But once we have decided to add that edge, the two endpoints are already connected, so we may consider them the same node for all practical purposes. (The resulting new “node” inherits all the edges of both of the nodes that were merged to form it – in practice, we can retain only the cheapest among parallel edges.) This is called “contracting” the edge. In the remaining graph, we now know that we must add the cheapest edge, which we can in turn contract. We can keep doing that (and contracting edges) until only one node is left.
When thinking about this repeated contraction algorithm, we see a different way to phrase it. If we look at the edges included up to any certain point, then the set of nodes that ended up being contracted into a given “pseudo-node” is exactly a connected component. In fact, all remaining pseudo-nodes correspond exactly to connected components. And because we delete parallel edges (and self loops), we make sure that all edges considered are only between different components, never within one. (It's a good thing we do that, as we should not create cycles.) This suggests the following reformulation, which is known as Kruskal's algorithm.

First, we want to prove that the algorithm is correct. Correctness here means that:

(1) All the nodes are included and connected;

(2) There is no cycle;

(3) The solution is the cheapest.
3. Because the algorithm explicitly avoids putting in any edges that would create a cycle, the second statement (“no cycle”) is trivial.

To prove that the solution is a connected graph, let’s assume (for contradiction) that the solution contains two components $C_1, C_2$ and they are not connected to each other. However, in the original graph, there must exist at least one edge $e$ connecting the two components (because we assumed that the original graph was connected). The only reason why $e$ is excluded later is $e$ will bring a cycle. But since no edge between $C_1$ and $C_2$ was added before, $e$ can not bring any cycle into the solution. Thus, there cannot have been two (or more) connected components.

4. Next, we want to prove that each edge the algorithm includes is in the optimal solution. The “cut property” was used to prove this statement. The way we plan to use it is by giving, for each edge $e$, some cut for
which $e$ is cheapest. That is enough to ensure that $e$ must be in the optimal solution. So let's focus on some edge $e$. When $e$ is added, it connects two components $C_1$ and $C_2$. Then, two cuts we could consider (in addition to potentially others) are the ones between $C_1$ and its complement, or between $C_2$ and its complement. $e$ is the cheapest edge across these cuts because if there existed a cheaper edge $e'$, it should have been added before, and $e$ would now create a cycle, and not be added.

5. After the correctness proof, the class moved to running time analysis.
The key step here is to decide whether adding an edge will create a cycle. One way to do this is to check, every time we are looking at an edge \( e=(u,v) \), whether \( u \) and \( v \) are connected. DFS or BFS can be used to fulfill the task in time \( O(n) \) (because there are at most \( n \) edges in the graph). Then, the running time is \( \Theta(m \log(m) + mn) \). The first term comes from sorting the edges, and the second from running a BFS for each edge.

6. The previous implementation seems a bit wasteful. We start a BFS from scratch, even though only one edge has been added. A better implementation would keep track, for each node \( v \), of the component \( r[v] \) it belongs to. A cycle would be created due to the coming of edge \( (u, v) \) if and only if \( r[u] = r[v] \). The test can then be performed with just two array lookups, in constant time. On the other hand, every time two components merge, we must change the component designation for all nodes in one of the components. This would be done with one loop over all nodes, resetting \( r[w] \) to \( r[v] \) for all nodes \( w \) which originally had \( r[w] = r[u] \). This loop takes \( \Theta(n) \) time. However, it is only called when an edge is actually added, which only happens \( O(n) \) times. Thus, the running time becomes \( \Theta(m \log(m) + n^2) \), as the \( O(m) \) array lookups are dominated by the \( m \log(m) \) term for sorting.
The above implementation suffers from overwriting a lot of values multiple times. Instead, we would like to be able to say something like “every node that previously was in component i is now in component j”.

We can do the equivalent of this by using pointers. The idea is that each node has a pointer to a “representative” in its component. The only node in a component not having a representative is called the leader of this component. So each component is, in fact, represented by a leader node.

In order to decide whether two nodes are in the same component, we follow the pointers of representatives until we reach the leaders, and test if the leaders are the same.
Initially, each node is its own component, so it is its own leader as well. When merging two components under this implementation, we can just make the leader of one component the leader of the other component. This only involves adding one pointer, and thus runs in constant time. So the bottleneck is really the time for the lookup of a leader, which is proportional to the “height” $h$ of the pointer structure, the longest path from a node to its leader. Thus, the merging rule (which leader points to the other) should be such that this height does not grow fast.

A relatively easy to analyze approach is to keep track of the size of components and always link the leader of the smaller component to the leader of larger size component (and update the component size of the larger component to the sum, of course).
To bound the maximum height under this approach, look at some node v, and the distance it has to its leader. Whenever that distance actually increases, it must have been because the component merged another component, and that other component was at least as large as v's. Thus, if we look at v, it must now be in a component at least twice the size. Thus, every time a node's distance to its leader increases, the component size must have doubled. Since there are at most n nodes in the graph, and thus in any component, the height can never exceed log (n). Thus, each lookup takes at most \( \Theta(\log(n)) \) steps, and is invoked at most m times, so
the total running time is $\Theta(m \log(m))$.

10. The kind of data structure we have been developing so far is called “Union-Find” structure, because the operations it supports are to take the union of sets (merging components), and to look up which set an element is in (find, which we use to find the leader). These structures are important for other applications as well, and hence there is interest in speeding up the operations. We already noticed that the smaller the
height of the tree structure, the faster the lookup operations. On the other hand, setting a lot of pointers (for instance, to point directly to the leader) takes more time for the merging step. A good compromise is achieved by “Path Compression”: whenever the algorithm looks up a leader of a node v, all the intermediate nodes on the path to the leader change their representative pointer to point directly to the leader. This only doubles the lookup time, but might speed up future lookups significantly. The rule is actually not too difficult to implement, but the analysis gets quite complicated (see Chapter 21 in the textbook). One can show that the running time of m operations is now O(m log*(n)). (In fact, it is even faster, namely O(m \(\alpha(n)\)), where \(\alpha(n)\) is the Inverse Ackerman function.) This is very slow growing. The running time for Kruskal's Algorithm is then O(m log m + m \(\alpha(n)\)), so the sorting time dominates the running time.

In talking about Minimum Spanning Tree algorithms, there are actually, despite a lot of interesting results, still some major open questions. In particular, it is unknown if there is an O(m+n) deterministic algorithm for finding a Minimum Spanning Tree (there is a randomized algorithm with expected such running time). Perhaps slightly easier is if there is an O(m+n) algorithm under the additional assumption that the edges are already sorted by weight. Notice that the sorting is the bottleneck for Kruskal's algorithm, and the running time
O(m α(n)) is “almost” linear once the sorting is done. But not quite, so it would be interesting to see whether it can be brought down to linear.