Class Note #06

Date: 01/30/2006

[Overall Information]

Based on the grading result of quiz#1, we reviewed basic concepts we learnt about Big-O notation.

After that, the new problem we studied in today’s class is Topological Sort. The algorithm was given and the correctness was proved.

[During the Lecture]

1. Quiz#1 has been graded and the graded test papers were returned after today’s class. The overall grades are not very satisfying. If you have any question about your quiz#1 grading, please come to the professor’s office hours (SAL 232 Tuesday, 10:30-12:00).

Quiz#2 is scheduled on 02/08/2006.

There will be an interesting guest talk about matching problem on 02/02/2006. Students are encouraged to attend it.
From the grading result of quiz#1, it seems that some students are still not very clear about Big-O notation. Since this part of knowledge is very fundamental for this course, we spent some time reviewing the Big-O notation at the beginning of today’s class. If you still have questions about Big-O notation, you may have a look at class notes NO.2 located at the following link:

http://www-rcf.usc.edu/~dkempe/CS303/notes02.pdf

We also went over some reasons why this notation is so popular and important:

(1) Its simplicity and brevity;

(2) Some time, we don’t know how to calculate T(n) exactly and directly. Or we don’t know what’s the worst-case for certain algorithm. But we can get upper and lower bounds on T(n), and if they match to withing constants, it's good enough in Big-O notation.
3. Whenever we analyze an algorithm's running time, what we talk about is its worst-case running time $T(n)$. Proving an upper bound of $f(n)$ on $T(n)$ means that even the worst case does not take more than $f(n)$ steps, so the algorithm never takes more than $f(n)$ steps. However, it is always possible that our analysis was “sloppy”: that in reality, the algorithm doesn't even come close to using $f(n)$ steps on any input. Thus, to learn more about the input, we also want to lower-bound $T(n)$. Ideally, we would like to know what the worst case of the algorithm is, so we can calculate $T(n)$ directly. But that is often difficult (or
impossible). Instead, we can look for a “bad” case, and calculate the running time \( g(n) \) for that case. We then know that \( T(n) \) is at least \( g(n) \). If the “bad” case were actually the worst one, then these would be equal, but at least, we have proved that the worst case must be at least as bad as \( g(n) \).

4. In oder to make any statement such as “\( T(n) \) is \( O(f(n)) \)” or “\( T(n) \) is \( \Omega(g(n)) \)” meaningful, we have to exhibit a bad case (and compute an upper bound) for each input size \( n \). If we only had one example of size 20, on which the algorithm takes 400 steps, we don't know if the running time is \( n^2 \) or \( 20^n \) for all \( n \). So lower bounds always come as families of lower bounds: one for each \( n \).
4. Notice that if we have found upper and lower bounds \( f \) and \( g \) (in Big-O notation) with \( f = g \), we have also proved that the “bad” input we had was actually the worst case (except for constant factors), and the analysis was tight (up to constant factors). However, if \( f \neq g \), we have only found out a range for \( T(n) \), which implies that there is something wrong with either our loop analysis or the “bad” case (it is not bad enough). Sometimes, getting \( f \) and \( g \) to match is quite difficult, and sometimes, it is not so important for them to match. Often, upper bounds are more important, but having matching upper and lower bounds is always best.

5. Several useful formulas were given at the end of this review section, including:

(1) if \( f = O(g) \), then \( g = \Omega(f) \);

(2) if \( f = \Theta(g) \), then \( g = \Theta(f) \);
(3) \( n^k = O(n^m) \), for \( m > k \);

(4) \( (\log(n))^k = O((n^m)) \) for any \( k \) and \( m \);

(5) \( n^k = O(c^n) \), for any \( c > 1 \) and any \( k \);

…

For the loop analysis part, given a loop of \( i \) from \( a \) to \( b \), which takes time \( O(f(i)) \) for each iteration, the loop in total will take \( O(f(a) + f(a+1) + f(a+2) + … + f(b)) = O(\sum f(i)) \). When it comes to specific problems, we first need to calculate \( f(i) \), and then compute the sum.

After reviewing above knowledge, the class continued with graph-related topics. The problem we were to study is Topological Sort. Suppose we have a graph, in which edge \( = (u, v) \) means \( u \) must be done before \( v \) can be done (a real-life example is pre-requisites among
courses). Our goal is to find a valid ordering of the nodes satisfying:

1. If there is an edge $(u, v)$, then $u$’s number must be smaller than $v$’s.
2. Also detect possible conflict cycles.

7. To solve the problem, the key idea here is like the following.

Start from the node that does not have any incoming edge. That node can surely get the lower number without any conflicts. Then in each step, remove the nodes (as well as the connecting edges) that have already been assigned a number, because they will not cause any more conflicts in the future.

So the pseudocode of Topological Sort is as follows (here, the $d[]$ variables keep track of the number of the remaining number of incoming
edges for nodes, so we don't have to recount the edges every time):

8. Finally, the class moved to correctness proof. What we need to prove is if all nodes are numbered by the above algorithm, their number must be correct, and if some node remains unlabeled, then there must be cycles.

(1) Assume there is an edge \((v,u)\) and \(v\) has a higher number. Then, \(u\) is processed before \(v\), but it cannot be put in the queue until its edge from \(v\) was subtracted. This gives a contradiction.

(2) After running the algorithm, if there are unprocessed notes left, then they all have \(d[v] \geq 1\). So all of them have at least one incoming edge from some other unprocessed nodes. With the help of this property, we are able to construct a path of arbitrary length: start with some node, next
take some node that has an edge to it (which exists, because $d[v] > 0$), then take a node with an edge to that one, and so forth. After at most $n+1$ nodes, one must repeat, so we have a cycle.

**Conclusion:** Prove that if all nodes are numbered, then the numbering is correct; otherwise, there is a cycle.

1. **Contradiction:** Assume that there is an edge $e(v,v')$, and $v'$ has a higher number.
   - Then, it was processed before $v$, but it is not yet in the queue until $d[v']=0$, which only happens once a node is processed. By contradiction, this cannot happen.

2. **If not all nodes are numbered, there is a cycle:**
   - If there are unprocessed nodes, let all of them have $d[v]=1$, so they all have at least one incoming edge from some other unprocessed node.

   Construct a cycle as follows:
   - Take an arbitrary (unprocessed) node $v$.
     - Let $u'$ be any node with an edge to $v$, where because $d[u']=1$.
   - For node $u'$, let $v''$ be an unprocessed node with an edge to $v''$ (exists because $d[u'']=1$).
   - So we continue as long as we want - after $n+1$ steps, some node $v_n$ must have repeated.
   - That gives a cycle.

   So the algorithm has worked in naming a cycle exists.