[Overall Information]

In this class, first we continued our last lecture’s topic about Topological Sort and analyzed its running time.

The later part of the class dealt with Shortest Path problem. A simple algorithm we learnt before to solve this problem was discussed.

[During the Lecture]

1. After the class began, a few announcements were made:

(1) Homework NO.2 was assigned.

(2) Quiz#2 is scheduled on Wednesday, 02/08/2006 (one week from today) starting from 8:30.

Most of the quiz#2 questions will still come from homework and based on the grading result of quiz#1, there will be one Big-O notation related problem in quiz#2.

(3) The guest talk the professor mentioned in last lecture will take place tomorrow (02/02/2006) at SSL150 from 3:00 pm. Students who are interested in the current research in algorithm field are encouraged to attend the talk.
2. We study the Topological Sort algorithm. In the last lecture, we proved the correctness of the algorithm. Now, we’d like to analyze the running time.
3. The initialization step takes constant time. To calculate the number of incoming edges for every node, we set all $d[v] = 0$. Then go through all nodes $u$, check all outgoing edges $(u,v)$, and increase the value of $d[v]$. This goes through all nodes and all edges once overall, so the running time is $O(n+m)$. The inner loop also goes through each node (at most), and decreases a counter $d[u]$ for each edge, so it as well takes $O(n+m)$. So the total running time is $O(n+m)$.

To calculate the lower bound of the running time, we only need to focus on the most time-consuming steps (if we leave out the faster steps, the algorithm certainly won't take any less time). The second step always has to go through all of the edges, so it by itself already shows that the running time on each graph is $\Omega(n+m)$. As this matches the upper bound (up to constants), the running time is $\Theta(n+m)$. 
Next, the class moved to Shortest Path problem. Given a graph (directed or undirected) in which $c_e \geq 0$ represents the length (or cost) of edge $e$, the length of a path is the sum of the lengths/costs of all edges along that path. Starting from a node $s$, our goal is to find the shortest path to certain node $t$, or perhaps to all other nodes $t$.

A simple reduction we considered before is to divide each edge $e$ into $c_e$ edges of length 1 each.
5. Here a potential problem we should notice is that if not all the edge lengths are integers, we will not be able to, for example, divide an edge into 2.5 hops. For a graph, of which all the edge lengths are rational numbers, the solution is to multiply all the lengths with their common dominator. Since such multiplication is only to scale the problem, the final result will not be affected.

We may have the intuition that if $c_e$ is very large, it will be divided into a huge number of small hops each with length 1. As a result, this
algorithm, correct as it may be, is very inefficient. However, even though it is inefficient, it looks at first glance as though it might be polynomial, because it is $O(n + \Sigma c_e)$. (We are particularly interested in “polynomial time algorithms”, those whose running time is $T(n)=O(n^k)$ for some $k$. These usually coincide with “practically useful” algorithms.)

When we say “polynomial”, we need to be careful about “polynomial in what”. In analyzing any algorithm, the running time is always expressed as the input size. For arrays or graphs, we usually measure the input size in the number $n$ of array elements, or the numbers $n$ and $m$ of nodes and edges. More generally, we express the input size as the number of bits it takes to write down the input. Since for a decimal number $K$, writing it down in binary only takes $\log_2 K$ bits, writing down $c_e$ takes $\log_2(c_e)$ bits.
6. Let’s come back to our simple algorithm of Shortest Path problem. The input size is \( n + \sum \log_2(c_e) \), and the running time is \( O(n + \sum c_e) \). Because \( c_e \) is exponential in \( \log_2(c_e) \), the algorithm has a polynomial running time in the input numbers, but not the input size. Such algorithms are called pseudo polynomial algorithms. They are efficient only when the input numbers are small.

7. As a particular case where people often confuse polynomial and pseudo-polynomial algorithms, consider factoring a (large) number \( n \). This can be easily done in time polynomial in \( n \), by checking all numbers \( k \) smaller than \( n \), and seeing if they divide \( n \). However, while this is polynomial in \( n \), it is not a polynomial-time algorithm, as
writing down $n$ takes only $\log(n)$ bits. So a polynomial-time algorithm would have to run in time $O(\log(n)^k)$, and whether or not a polynomial time algorithm exists for factoring is one of the big open problems of computer science, with immense importance for cryptography.

7. For the Shortest Path problem, we’d like to find a real polynomial-time algorithm (not pseudo-polynomial). The idea of the new algorithm we were going to study was first illustrated by a specific example.
The important observation was that we cannot simply choose a queue of nodes, as we did for BFS. Rather, we always want to next process the node that has the smallest estimate of distance from s. Thus, our data structure must be able to re-arrange itself when new information about a node's distance from s is gained by examining nodes.

8. The detailed algorithm was given afterwards. The algorithm updates the total length (from starting node to the current node) each time the path is extended and tries to only extend the path in a direction that minimizes the total length.
9. To implement the algorithm, we will need a data structure DS supporting the following operations:

(1) Insert (DS, v, d[v]);

(2) Find-min (DS);

(3) Delete-min (DS);

(4) Update (DS, v, d[v]).

We could use several different data structures. An unsorted array or list is very fast at inserting new elements or updating them (O(1)), but it is slow at finding and deleting the min-element (Ω(\(n\))). A sorted array or list, on the other hand, is very slow at inserting new elements (Ω(\(n\))), but it is fast at finding and deleting the min-element (O(1)). A data structure that does well for all of those operations is a heap. For a heap structure,
the minimum element is always located at the root of the heap while other elements are not totally sorted.

10. At the end of today’s class, a lemma for next class was provided. In one word, the lemma claims that any sub-path of a shortest path P is also a shortest path between the corresponding endpoints. The proof was by contradiction. Assume that there exists a shorter path Q, and then construct a shorter path from s to the end point of the original shortest path by concatenating Q with the second half of the assumed shortest path P. We will use this lemma to prove the correctness of our new Shortest Path algorithm.
Lemma: Let $P$ be a shortest path from $s$ to $t$, and $v \in P$ a node on $P$. Then, the subpath of $P$ from $s$ to $v$ is a shortest $s-v$ path.