[Overall Information]

In this class, we proved the correctness of Breadth-First-Search algorithm with the help of a lemma.

The first quiz of CSCI303 at Spring 2006 will take place in next class. Please make sure to show up on time.

[During the Lecture]

1. A few new announcements were made:

(1) The solution of homework#1 was distributed at the end of the class. For those who actually turned their homework in, the graded homework will probably be returned during next class.

(2) Quiz#1 will take place at the beginning of next class. It is a closed-book test and calculators are allowed.

(3) Students are encouraged to use the blackboard online system to
discuss class-related questions.

2. After the class began, we continued our study of Breadth-First-Search (BFS) algorithm. After the pseudocode was written onto the whiteboard, our first task is to prove the correctness of BFS.

This time, what do we mean by “correct”? For BFS algorithm, the correct lie in three aspects:

(1) All nodes that are reachable should be explored;

(2) For all explored nodes we are able to tell their shortest paths by backtracking;

(3) For arbitrary explored node \( v \), the shortest path distance can be found at \( d[v] \).
3. To prove the three properties, our strategy is trying to use induction to show that for each $i$, all nodes $v$ with distance less than or equal to $i$ satisfy the above three requirements.

For the base case when $i=0$, $s$ is the only node. $d[s]$ is set to 0, which is the correct distance. It does not need $\text{pred}(s)$ and it is explored. So all the three requirements are met.
4. Next, assume that all the nodes with \( \text{dist}(s,v) \) less than or equal to \( i \) are “correct”. Let \( v \) be an arbitrary node with \( \text{dist}(s,v) = i+1 \). Consequently, there exists a path with \((i+1)\) hops. If \( u \) is the last node before \( v \) on the shortest path, then it is at distance \( i \), so we can apply the induction hypothesis. Among all such nodes \( u \) (there may be multiple shortest paths from \( s \) to \( v \)), we let \( u \) denote the one that BFS explores first. We would like to claim that when \( v \) is enqueued by BFS, it is as a result of exploring this particular \( u \). To prove this, we need a lemma about how the queue operates.
The lemma states that the distance from s to unexplored nodes is at least as large as the distance from s to the explored nodes, and within the queue, the distances from s are non-decreasing. Furthermore, and most importantly, the last node of the queue is at most one hop further away from s than the first node of the queue.

Using this lemma, we can prove (by contradiction) that v is enqueued as a result of exploring u. Thus, BFS does the following when it enqueues
v:

(1) put \( v \) into the queue and mark is as “enqueued”.

(2) set \( d[v] = d[u] + 1 = \text{dist}(s, u) + 1 = \text{dist}(s, v) \). (The middle equality follows because \( d[u] \) contains the correct value by induction hypothesis.)

(3) set \( \text{pred}[v] = u \), which is also correct because \( u \) is the last hop on a shortest \( s-v \) path.

So the correctness is proved by induction.

6. One remaining problem is to prove the lemma. It was proved also by induction.
The key step here is that the newly explored node $v$ is head of the queue, so it has a distance less than or equal to everything else in the queue. All elements in the queue have distance at most $d(s, v)+1$ while the newly appended elements (if any) have distance $d(s, v)+1$. As a result, the queue is still ordered, and all explored nodes still have distances at most those of nodes in the queue.

Above all, we finished the whole correctness proof of BFS.
7. At last, the class talked briefly about the difference between Breadth-First-Search (BFS) and Depth-First-Search (DFS). The running time of BFS is optimal, so one wonders why we should ever use DFS instead. One advantage of DFS lies in the fact that it frequently saves memory, because it only maintains a stack with as many elements as the “depth” of the graph. This is particularly important when the graph is not given explicitly, but only implicitly, for instance as a state space of a puzzle like Rubik's Cube. In addition, DFS is more natural for some application areas, such as family trees, trees representing graphic objects clustered by proximity, or object relationships in databases. In all of these, it makes sense to explore nearby nodes together.
roots of the queue

\[ a(3, u) + 1 \]

(1) \ref{a(3, u) + 1} \qquad \text{with } i = n

(1) \ref{a(3, u) + 1} \text{ of the queue

\[ a(3, u) + 1 \]