Lecture 16: Dynamic Programming (cont.)

During the first part of the lecture, I described how the LCS dynamic programming table is filled in, and how to find the LCS by trace-back. Read the textbook if you have doubts.

The next problem is matrix chain multiplication. Again, this is a problem that is well explained in the textbook. I want to emphasize several things. First, finding the subproblems is often one of the most important aspects of using dynamic programming. For this problem, we define \( M[i,j] \) (where \( i \leq j \)) as the smallest amount of computation needed to multiple matrices \( A_i \ldots A_j \). This is natural: maybe the optimal solution will put parenthesis right before \( A_i \) and after \( A_j \). Of course, we are not sure, but the key idea of DP is to compute the results for all the subproblems and then figure out the overall solution from these subproblems. I will skip the rest of details (since the textbook explains well). I suggest you to experiment with small examples to understand why the proposed DP algorithm works.

Lecture 17: Basic Graph Algorithms: BFS and DFS

Graph is one of the most important concepts in algorithms. We will spend several weeks on algorithms for graphs. In this lecture, we study two basic algorithms for graph search: breadth-first search (BFS) and depth-first search (DFS). Many of you have known these two algorithms before. But I suggest you to refresh your memory and also think a little deeper about these two algorithms.

First, BFS can find the distances of nodes to a source node. Here, distance just means the smallest number of edges connecting the two nodes. See the textbook for the BFS algorithm. In class, I presented a simpler version of the algorithm: no coloring is used since we just use the distance to check whether a node is visited before. The key question on BFS is why the distances set by the BFS algorithm are correct. The correctness is based on three properties that holds for all \( d = 0, 1, 2, \ldots \) at some point of time (note it is some time, not always): (1) each node with distance (from source \( s \)) \( \leq d \) is set properly, (2) all other nodes are not discovered (i.e. distance is set to \( \infty \)), (3) the queue contains precisely the nodes with distance \( d \). We can see why these hold by using induction. At \( d = 0 \), this is almost trivial: we only have \( s \) in stack (whose distance is 0, and \( s \) is the only such node). Then assume at some point of time, these three properties hold. Then, we can show that there exists some moment where the three properties hold for \( d + 1 \). Let us focus on one part: the distance set by the first node \( u \) of the queue is correct. Note in this case, a newly discovered node \( v \) (distance currently set to \( \infty \)) will be set to distance \( d + 1 \). Why is this action correct: can \( dist(v) \geq d + 2 \) or \( dist(u) \leq d \)? First, \( dist(v) \) must be at most \( d + 1 \) because \( v \) is reachable in one hop from a distance \( d \) node. Second, \( dist(v) \) can not be \( d \) or smaller because by the induction assumption, all such nodes (with distance \( d \) or smaller) have already been set (and thus \( v \)'s distance can not be \( \infty \). We are not quite done yet, but I hope you have seen where I am leading to. You can finish the details yourself.

Then we come to DFS, which is like exploring the maze. Just like exploring the maze, we must try to avoid going in cycles. The tools for this is (a) chalk for marking visited places, and (b) a roll of strings to backtrack. These are easily done by having a flag for each graph node (which is marked when it is visited), and by recursion. In class, I described a slightly simpler algorithm than the one in textbook. There, we just use a flag visited(u) to indicate whether u is visited or not (instead of colors in the textbook). The rest of algorithm is the same as the textbook (read the section if you did not write down all the steps in class). A key utility of DFS is timestamp, which tells when a node is initially discovered and finished. This time interval \([u.d, u.f]\) is often quite useful for some problems. We will continue on this next week.