CS 218 : Design and Analysis of Algorithms

Lecture 14: Shortest Paths: Forcing a recursion

Lecturer: *Sundar Vishwanathan*
Computer Science & Engineering Indian Institute of Technology, Bombay

1 Problem Statement

Input: A graph with positive weights on the edges. Two vertices s and t.
Output: Shortest path between s and t.

You may have already studied many algorithms for this in the data structures course. Our objective is to start from scratch and see how an algorithm develops. As usual we will only calculate the length of the shortest path. The path itself can be found by doing some additional book-keeping.

We begin with a recurrence. We know that the shortest path from s to t will pass through one of the neighbors of s and so we may recurse on each neighbor and pick the smallest. The procedure FirstShortestpath$(G; s, t)$ returns the shortest path from any vertex u to t in G. The obvious recursion is to pick the minimum of the following:

$$\min_i W(s, u_i) + F\text{irstShortestpath}(G; u_i, t).$$

Here u_i ranges over the neighbors of s. The problem with this recurrence is that the input size does not go down. This procedure will not terminate. For induction to work, the problem size has to go down with each recursive call. This does not happen here and we go around in circles.

Here is a possible fix. We can remove the vertex s from G in the recursion.

$$\min_i W(s, u_i) + F\text{irstShortestpath}(G - s; u_i, t).$$

This will terminate, but what about the number of distinct procedure calls? It is exponential! Why?

Describe a graph on which the above recursion will have an exponential number of recursive calls.

So we order the input. The obvious thing to try first is to order the vertices. We wish to find the shortest path between s and t which uses the vertices $u_1, \ldots, u_n$.

$$S\text{hortestpath}(u_1, u_2, \ldots, u_n; s, t).$$

What about the recursion? Well, one can go along the previous path of looking at neighbors of s, but this leads to familiar problems. We have to use the order imposed on the vertices and the recursion has to be aided by it. The key to using the order that we try and restrict calls to be on suffixes/prefixes/intervals of the ordered set.

The recursion we build uses a familiar logic: that either u_1 is used in a shortest path between s and t or it is not. If it is not then we recurse on

$$S\text{econdShortestpath}(u_2 \ldots, u_n; s, t).$$

If it is then we need a shortest path from s to u_1 and then from u_1 to t. Here is the recurrence for that case.

$$S\text{econdShortestpath}(u_2 \ldots, u_n; s, u_1) + S\text{econdShortestpath}(u_2 \ldots, u_n; u_1, t).$$

What are the distinct calls? The first argument is a prefix and the second is a pair of vertices yielding $O(n^3)$. This is fine.

Another way to deal with this is **Induction**++, or souping up the induction. We increase the number of parameters in the recursion. Shortestpath(u, t, k) will return the shortest path between u and t in G which uses at most k edges. Now the recurrence works out nicely as:

$$\min_i W(u, u_i) + S\text{hortestpath}(u_i, t, k-1).$$

This souping up of induction, or adding more variables to recurse on may even be mandated by the problem, or may help get the number of distinct recursive calls down. It is an integral part of our bag of tricks which you should be ready to use.

In this course we have stressed the need to devise recursive algorithms. During implementation, recursion has other overheads and usually runs slower. Hence it is preferable to program iterative algorithms. Once this course is over and you are comfortable with designing recursive algorithms, we encourage you to convert these dynamic programming algorithms to iterative ones. The idea is simple. Fill the table from smaller inputs to larger inputs exactly as the recursion does, only now iteratively.