All–Points Shortest Path Problem

Zak Smith

0.0.1 Description of Problem

Given a directed graph, find the shortest path from each vertex to every other vertex in the graph.

0.0.2 Relation to Dynamic Programming

Dynamic programming finds optimal solutions to problems by recursively figuring out which sub–solution, combined with the choices at the current level, will produce the best result.

For example, if we are solving problem A, which can be broken up and solved 4 ways a_1, a_2, a_3, a_4, we do the following:

1. figure out cost of $a_1, \ldots, a_4$: $COST(a_i)$
2. pick best of $a_1, \ldots, a_4$ by choosing the minimum of $COMBINE\_COST(a_i) + COST(a_i)$: the cost to combine a_i at the current level plus the cost of a_i at the lower levels
3. we then return this solution and cost to the caller

Another feature of dynamic programming is the caching of solutions to sub–problems. Once we find the best solution for a sub–solution, we can remember this value and we need not recalculate it as we take other branches down the recursion tree.

A simple example is if we are trying to calculate $x!$ for $x = 1, 2, 3, 4, 5$. Once we calculate $4!$, for $x = 4$, we need not recalculate $4!$ when we find $5!$, we can simple use the value we already calculated.

Often these cached values are kept in a matrix, and often the dependencies between entries have geometric significance (like in the matrix–multiply problem). In the example above ($5!$), an element on depends only on the element directly to the left of it.

0.0.3 Solution to Problem

The input to this algorithm is a sparsely populated matrix $M(i, j)$, whose (i, j) element is the distance from vertex v_i to vertex v_j. The input matrix only contains values for cities which are directly connected. As the algorithm runs, it fills in the remainder of the entries.

$$M(i,j) = \text{MIN}\left\{M(i,k) + M(k,j)\right\}$$

... for all choices of k such that v_i is connected to v_k, ie: we pick the best sub–solution for all the vertices directly connected to the current vertex, based on the cost to combine that vertex v_k with the current vertex v_i. The "destination" vertex is v_j.

After $M(i,j)$ is computed, the value is stored in the matrix position in M so that subsequent queries do not have to recompute $M(i,j)$.

I have just described how to find a singular shortest–path. To find all the shortest–paths, we merely look up the next (i,j) pair using the same matrix. As more and more paths are known, we quickly build a "library" of cached sub–solutions so instead of lots of computation, we merely look up the answers to the sub–solutions in our matrix. $M(i,j)$ is called with all possible vertex pairs, n^2. The first computed pairs take generally longer to compute, and the later pairs take generally less time to compute because of the already–computed values in the matrix.

0.0.4 Comments

This algorithm does not fill the matrix in a geometric order because the graph is, by nature, not geometrically regular. Nevertheless, this algorithm does use the ideas of Dynamic Programming to find the best solution in the least ammount of time.

An alternative algorithm would be to start at each vertex, $1 \ldots n$, and find the shortest path to each other vertex, using a method similar to the one specified above. Thus, when starting at the first vertex, we would visit $n-1$ vertices. At the end, we would only need to go to one other vertex. At the first vertex, it must visit $O(n^2)$ nodes, so it is equivalent in complexity to the solution I have proposed above.