Advanced Algorithms, Fall 2012

Prof. Bernard Moret

Homework Assignment #9: Solutions

Question 1.

Perhaps the simplest dynamic program is one that finds all pairwise shortest paths between the vertices of an undirected graph. You are given a graph by its adjacency matrix, in which entry (i, j) is either ∞ (say the maximum integer representable in your machine) if there is no edge between i and i, or the length of the edge $\{i, j\}$ (a positive integer) otherwise. The output is a pairwise distance matrix giving the length of the shortest path between any two vertices. Show how to formulate this algorithm with three nested loops and verify that it runs in cubic time.

Solution:

This algorithm is know as Floyd's algorithm. Basically, the dynamic programming computes all (i, j) entries of a distance matrix n times: during the kth computation, paths from i to j are allowed to use intermediate vertices with index no larger than k. The matrix is initialized with ∞ for pairs not connected by an edge, with the length of the connecting edge otherwise. In the iteration, the new distance (shortest path) between i and j is simply the shorter of the old one and of a new one composed of an existing shortest path from i to k and of an existing shortest path from k to j. Hence the program reads as follows:

```
/* initialize matrix M from graph */
...
/* triple loop: the outer loop is the bound in the index of vertices */
for k=1 to n do
    for i=1 to n do
        for j=i+1 to n do
            new = M(i,k) + M(k,j)
            if new < M(i,j) then M(i.j) := new
```

Clearly the program runs in $\Theta(n^3)$ time: the one line in the inner loop performs on addition, one variable assignment, one arithmetic comparison, and possibly one more variable assignment, and so take (very low) constant time, while each of the three loops runs $\Theta(n)$ times.

Question 2.

You are given a free tree T with n nodes. (A free tree is not rooted, its edges are undirected, and the degree of a node is not bounded—in other words, it is just a connected acyclic undirected graph.) Each edge is assigned a weight, which may be positive, zero, or negative. Devise a linear-time dynamic program to find a path in T such that the sum of the edge weights on this path is minimized.

Solution:

We first root the whole tree by arbitrarily choosing a node r as root. Now we process the tree in post-order. For each subtree T_v rooted at node v, we calculate and maintain two values, $F(v)$ and $G(v)$. $F(v)$ is the minimum total weight of the best path in T_v that has v as one endpoint, while

$G(v)$ is just the minimum total weight of the best path in T_v without restriction. When node v is a leaf, we have $F(v) = G(v) = 0$. Suppose node v is an internal node and it has k children, v_1, v_2, ..., v_k. We can write

$$F(v) = \min_{i=1}^{k} (F(v_i) + w(v, v_i))$$

where $w(v, v_i)$ is the weight of the edge connecting v and v_i. This is because the path with v as one endpoint must consist of one edge (v, v_i), plus the best path in T_{v_i} with v_i as one endpoint. The path with minimum weight in T_v can be within a single subtree of v, or it can have v as an endpoint, or it can pass through v; formally,

$$G(v) = \min \left(\min_{i=1}^{k} G(v_i), F(v), \min_{i,j} (F(v_i) + w(v, v_i) + F(v_j) + w(v, v_j)) \right)$$

In order to calculate $\min_{i,j} (F(v_i) + w(v, v_i) + F(v_j) + w(v, v_j))$, we need only find out the minimum value and the second minimum value of the array formed by $F(v_i) + w(v, v_i)$, $i = 1, 2, \cdots, k$, which costs $O(k)$ time.

For each edge, the recursion costs constant time. Thus, the total running time of the algorithm is $O(n)$.

Question 3.

You are given a convex polygon represented by the list of its vertices, say in counterclockwise order along the perimeter, p_1, p_2, ..., p_n. Devise an $\Theta(n^3)$ dynamic program to find a triangulation of this polygon such that the sum of the (Euclidean) lengths of the $(n-3)$ added edges is minimized. (A triangulation is a collection of chords*, that is, edges between polygon vertices that cross the interior of the polygon, that partition the interior of the polygon into triangles. A simple polygon of n vertices always needs exactly $n-3$ chords for a triangulation.)*

Solution:

We define subproblems by considering only portions of the perimeter of the polygon, from i to j, $i < j$, and "closing" the perimeter by adding an edge directly from j back to i. Notice that the resulting smaller polygon is still convex. Let $F(i, j)$ be the minimum total length of the added edges of the triangulation for the polygon formed by points $\{p_i, p_{i+1}, \cdots, p_j\}$; in particular, the optimal value for the whole polygon is $F(1, n)$.

Consider the triangle containing edge (p_j, p_i); its third point is some point p_k, for $i < k < j$. This triangle divides the remaining part of the polygon (the part not included in the $p_i, \ldots, p_j$ piece) into two small polygons, $\{p_i, p_{i+1}, \cdots, p_k\}$ and $\{p_k, p_{k+1}, \cdots, p_j\}$, one of both of which may be degenerate. (With $k = i+1$ or $k = j-1$, one of the small polygons degenerates into a segment.) Thus, we have the following recursion

$$F(i, j) = \min_{k=i+1}^{j-1} (d(p_i, p_k) + d(p_j, p_k) + F(i, k) + F(k, j))$$

where $d(p_i, p_k)$ is the Euclidean distance between p_i and p_k, unless (p_i, p_k) is an edge of the original polygon, in which case $d(p_i, p_k)$ is set to 0. Using this recursion, we can fill the matrix $F(-, -)$. Each entry costs $\Theta(n)$ time to fill in, so that the total running time of the algorithm is $\Theta(n^3)$.

This solution is a classic example of DP used in lieu of D& C: the edge $\{i, j\}$ added for closing the smaller polygon is in fact the boundary of a D& C cut, breaking the original polygon

into two subpolygons at the cost of edge $\{i, j\}$; but we do not know how to select i and j, so we simply compute every possible (i, j) subproblem and pick the specific (i, j) pair that gets us the best solution.

Question 4.

We saw how to use divide-and-conquer to reduce the cost of multiplying two square matrices. Now consider using dynamic programming to reduce the cost of multiplying a chain of rectangular matrices. You are given matrices A_i, $I = 1, \ldots, n$; these matrices are rectangular (i.e., not necessarily square), but the number of columns of matrix A_i always equals the number of rows of matrix A_{i+1}, for all i, $1 \leq i < n$. You want to compute the product $\Pi_{i=1}^{n} A_i$ as efficiently as possible, using plain matrix multiplication. That is, multiplying an $k \times l$ matrix by a $l \times m$ matrix to produce a $k \times m$ matrix will cost $\Theta(klm)$ time. Since matrix multiplication is associative, you can parenthesize the product in any properly nested way that you want—you can choose any pair of adjacent matrices to multiply first, replacing them by their product, and so on. Different parenthesizations will give different amounts of work, so you are to devise a dynamic programming algorithm to find the optimal parenthesization.

Solution:

This application is to a somewhat different domain, but the fact that these are matrices and we are doing matrix multiplication is actually irrelevant. Rather, we have a sequence of ordered pairs, (m_1, n_1), (m_2, n_2), $\ldots$, (m_k, n_k), where we always have $n_j = m_{j+1}$, and we are to find a parenthesization of the sequence such, using the reduction

$$\big((m, l)(l, n)\big) = (m, n)$$

each of which costs $m \cdot l \cdot n$ units, the cost of reducing the entire sequence to the single pair (m_1, n_k) is minimized.

For simplicity of notation, we will refer to the ith pair as P_i or as (m_i, n_i), as suits best. A legal parenthesization of the sequence $P_1 P_2 \ldots P_k$ is really just a binary tree with the P_is at the leaves and a reduction (product) at each internal node. Thus it has a natural decomposition into subproblems defined by the sequence $P_1 P_2 \ldots P_a$ and the sequence $P_{a+1} P_{a+2} \ldots P_k$, for $1 \leq i \leq k$. This gives us the recurrence relation defining the dynamic program:

$$c(i, j) = \min_{r=i,\ldots,j} c(i, r) + c(r + 1, j) + m_i \cdot n_r \cdot n_j$$

where we have $1 \leq i < j \leq n$. Our answer is the computed value of $c(1, k)$ and it takes us $O(k^3)$ to compute it, since we must fill in the values $c(i, j)$ (a quadratic number of values) and each requires minimization over all choices of k.

This problem is identical to the triangulation of a convex polygon in terms of structure: in each case, a top-level choice (a dividing edge, the root multiplication) divides the problem into two subproblems, but that choice cannot be made on the fly, so results are computed for every possible choice. Also in each case the cost of computing the value of an optimal local solution is linear, since in both cases a minimization is conducted over all possible divisions.