COMP 251: Data Structures and Algorithms Winter 2006

Dynamic Programming #2

Lecturer/TA: Ethan Kim

1 All-pairs shortest path

In a previous lecture, we discussed a version of shortest path problem: given a graph $G = (V, E)$, a source vertex s, and the distance function $l : (u, v) \in E \rightarrow Z^+ \cup 0$, find the distance from s to all other vertices. In this lecture, we look at a different version of shortest path problem.

- Given: a graph $G = (V, E)$, a distance function $l : (u, v) \in E \rightarrow Z^+ \cup 0$
- Find: the shortest path distance function $d : (u, v) \rightarrow Z^+ \cup 0$, $\forall u, v \in V$ such that $u \neq v$.

In short, we want to calculate the length of shortest path between *each pair of vertices*. Notice that the range of our distance function is again restricted to non-negative integers. In general, It is unknown that the problem can be solved efficiently if we allow negative distances on our edges.

By the usual convention, denote n the number of vertices ($|V|$), and denote m the number of edges ($|E|$) in graph G. We will express our distance function using an $n \times n$ matrix L as follows:

$$L[i][j] = \begin{cases} 0 & \text{if } i = j \\ +\infty & \text{if } (i, j) \notin E \\ l(i, j) & \text{otherwise} \end{cases}$$

Now, we see if we can construct a recursive definition of our problem, using the *principle of optimality*. Suppose k is a vertex on the shortest path from i to j. Then, it must be the case that the shortest path from i to k and the shortest path from k to j together form a shortest path from i to j. We will exploit this observation to design our algorithm.

We construct a matrix D that gives the distance of the shortest path between each pair of vertices. We first initialize D to L, which are the direct distances between vertices. Then, we do n iterations. The loop invariant for the algorithm is as follows:

- After k^{th} iteration, D contains the distance of the shortest paths *using only* $\{v_1, v_2, \ldots, v_k\}$.

Thus, after n iterations, D would contain the distance of the shortest paths using any of the vertices in V, which is what we want. Now, we apply the observation we made. At iteration k, the algorithm must check for each pair of vertices (i, j) whether or not there exists a path from i to j passing through the vertex k. If this path passing through k is better(shorter) than what we had before with only $\{1, \ldots, k-1\}$, we simply update our information. Let D_k denote the matrix D after k^{th} iteration. Then the check can be written as:

$$D_k[i, j] = min(\ D_{k-1}[i, j]\ ,\ D_{k-1}[i, k] + D_{k-1}[k, j]\ )$$

Here, notice two things: first, the above expression makes use of the principle of optimality we mentioned. Secondly, above expression completely disregards the case where k is visited twice. This is a valid assumption, since we know all our edge lengths are non-negative, and thus any cycle around k will only increase the total length of the tour.

But, the above expression implies that we have to store all previous values of D_{k-1}'s. Since we are iterating n times as k goes from 1 to n, does this mean we need to store all n matrices $D_1, D_2, \ldots, D_n$?

... not really. Consider the value $D_{k-1}[i,k]$ in the above expression. This value tells us that the length of shortest path from i to k, using only $1, 2, \ldots, k-1$ as intermediate vertices. Would this value be different if we do allow k as intermediate vertex? No, since $D[k,k]$ would be zero. Thus, we have that $D_{k-1}[i,k] = D_k[i,k]$, and also $D_{k-1}[k,j] = D_k[k,j]$. And this holds for all values $1 < k \leq n$, so we only need to store one D matrix and keep updating the entries as we loop through the algorithm. Thus, we have the following algorithm:

Floyd's Algorithm

1. $D \leftarrow L$
2. For $k = 1$ To n
   For $i = 1$ To n
   For $j = 1$ To n
   - $D[i,j] \leftarrow min(D[i,j], D[i,k] + D[k,j])$
3. return D

The running time of this algorithm is clearly $O(n^3)$. However, this routine only tells us the length of shortest paths between every pair of vertices. To find the actual shortest path between i and j, we use an auxiliary matrix P of size $n \times n$. First, we initialize P to 0 for all its entries. Then, the innermost loop of the algorithm becomes

- If $D[i,k] + D[k,j] < D[i,j]$ Then
  $D[i,j] \leftarrow D[i,k] + D[k,j]$
  $P[i,j] \leftarrow k$

By adding the last line, the matrix P will contain, for each i and j, the last iteration that $D[i,j]$ was updated. Since we start by initializing the matrix P to be all-zero, if $P[i,j] = 0$, there was no intermediate vertex between i and j, so they must be traversed directly via the edge (i,j). If, on the other hand, $P[i,j] = k$ for some k, then k is an intermediate vertex between i and j. This allows us to solve the problem recursively. For example, suppose the matrix P contains the following entries after Floyd's algorithm is run:

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Now, let's find the shortest path from vertex 1 to vertex 3. (Notice that the matrix may not be symmetric, since it only stores the "last time" D was updated.) Since $P[1,3] = 4$, the shortest path from 1 to 3 passes through 4. Recursively looking now at $P[1,4]$ and $P[4,3]$, we discover that the shortest path from 1 to 4 passes through 2, and 4 and 3 are directly connected. Then, $P[1,2] = P[2,4] = 0$. Hence, we now have the shortest path $1-2-4-3$. (Running time anyone?)

1.1 Hard Problems

shortest path with negative edges: Suppose our graph G contains negative edge weights. Then there are two cases: 1) there may be a cycle C in G, whose total weight is negative. 2) there may be negative edges, but there are no negative cycles.

The shortest path problem for first case is somewhat absurd, since we can always find a shorter path by traversing the negative cycle C again and again. Even if the problem specifically asks for shortest *simple* paths, the problem is not easy.(No efficient algorithm is known.)

For the second case, it is suprsing that the problem admits an efficent algorithm. A similar algorithm (using dynamic programming) solves this problem. Look at *Bellman-Ford Algorithm* in Chapter 24.1 of CLRS.

longest path problem: In this problem, we ask the opposite of the shortest path problem. Again, we assume that we are only interested in *simple* paths. (Otherwise, routing around a cycle yields a longer path.) Can this problem be solved using dynamic programming? Let's look at the following two statements:

1. *shortest path*: The shortest path from u to v contains a shortest path from u to w, where w is a neighbour of v.
2. *longest path*: The longest path from u to v contains a longest path from u to w, where w is a neighbour of v.

Notice that the first statement is indeed true(simple proof by contradiction - try it!), and is the principle of optimality we used for Floyd's Algorithm.

On the other hand, the second statement is not true. Consinder a 3-cycle containing vertices u, w, v, and edges $(u,v),(u,w),(w,v)$. Now, the longest path from u to v is clearly $u-w-v$. But the longest path from u to w is $u-v-w$, which is a counter example. To see this more generally, let P be a longest path from u to v. Now, let Q be the part of the path P, except v. (So, $P = Qv$.) Then, Q is not necessarily the longest path from u to w, since it cannot use v as intermediate vertex.

Thus, the longest-path problem does not have the structure we can exploit using dynamic programming. An efficient(polynomial time) algorithm for this problem is unknown.

2 Matrix Multiplication

In this problem, we are given n matrices, each of which has its own dimension. Recall that the product C of a $p \times q$ matrix A and a $q \times r$ matrix B is the $p \times r$ matrix given by

$$c_{i,j} = \sum_{k=1}^{q} a_{ik} b_{kj} \quad 1 \leq i \leq p, 1 \leq j \leq r.$$

Based on the above expression, we know that there are pqr sclar multiplications are required to compute C. (There are algorithms to speed up this process, but we will only use this direct computation for this lecture.) However, if you have a series of matrices to multiply, things are trickier. Because matrix multiplications are associative, we can choose any of the following methods to compute $ABCD$:

- $(AB)(CD)$
- $A(BC)D$
- $A(B(CD))$
- ...

Thus, we are interested in the *order* in which we multiply these matrices, so that we minimize the scalar multiplications in total. For notation, suppose we are given a sequence of matrices $M_1, M_2, \ldots, M_n$, and we are to compute the matrix $M = M_1 \cdot M_2 \ldots M_n$. In the spirit of dynamic programming, we make the following observation: Suppose the optimal solution has a parenthesis around the first i matrices, and then another parenthesis around the rest:

$$M_{OPT} = (M_1 \ldots M_i)(M_{i+1} \ldots M_n)$$

Then, the optimal solution for the sequence of $M_1 \ldots M_i$ and the sequence $M_{i+1} \ldots M_n$ must each be part of the solution to M_{OPT}, where M_{OPT} denotes the optimal parentheses structure that minimizes the scalar multiplication. This observation gives us a hint to construct a dynamic programming algorithm.

We construct an $n \times n$ table m, where m_{ij} gives the minimum number of scalar multiplications for the part $M_i M_{i+1} \cdots M_j$. Then, the value m_{1n} would give us the minimum number of scalar multiplications for the entire product.

We also need to store the dimensions of the matrices. Define a vector $d[0..n]$ such that the matrix M_i, $1 \leq i \leq n$, is of dimension $d_{i-1} \times d_i$. Then, we build the table m diagonal by diagonal, starting from the main diagonal, towards the upper-right corner. (Thus, we leave the lower half of the matrix empty). For convetional purposes, we will use s to indicate which diagonal we use throughout the algorithm. When $s = 0$, we mean the main diagonal. In general, the diagonal s contains the elements m_{ij} such that $j - i = s$.

Now, we start from $s = 0$. Since m_{ij} indicates the minimum number of scalar multiplications needed for matrices $M_i \ldots M_j$, and $i = j$ when $s = 0$, we put 0 along the main diagonal s. The diagonal $s = 1$ contains the elements $m_{i,i+1}$ corresponding products of the form $M_i M_{i+1}$. There is

no choice(since we only have 2 matrices) in putting parentheses anywhere, so we simply multiply them directly using $d_{i-1}d_id_{i+1}$ scalar multiplications.

Finally, when $s > 1$, the diagonal s contains the elements $m_{i,i+s}$ corresponding to products of the form $M_iM_{i+1}\cdots M_{i+s}$. Now we have a choice: we can make the first cut after any matrix between M_i and M_{i+s}. Suppose we make the first cut after M_k, for $i \leq k < i+s$. Then, we need $m_{i,k}$ scalar multiplications to calculate the left-hand term, and $m_{k+1,i+s}$ to calculate the right-hand term, and then finally $d_{i-1}d_kd_{i+s}$ scalar multiplications to merge the two terms together. To find the optimum, we merely choose the cut that minimizes the required number of scalar multiplications. We outline the algorithm below:

1. For $s = 0$ to $n-1$

   If $s = 0$ Then $m_{i,i} = 0 \quad \forall i = 1, 2, \ldots, n$

   Else If $s = 1$ Then $m_{i,i+1} = d_{i-1}d_id_{i+1} \quad \forall i = 1, 2, \ldots, n-1$

   Else $m_{i,i+s} = \min_{i \leq k < i+s}(m_{ik} + m_{k+1,i+s} + d_{i-1}d_kd_{i+s}) \quad \forall i = 1, 2, \ldots, n-s$

Notice that the order in which we fill up the table lets us make sure that we have all the required values are already computed, at each iteration. For instance, when we are scanning through all possible values of k to find the minimum, the values needed from the table are m_{ik}, and $m_{k+1,i+s}$. Since $k - i$ and $(i+s) - (k+1)$ are strictly less than s, these values have been computed already, in previous iterations.

Again, we usually want to know not only the number of scalar multiplications needed, but also how exactly to perform this computation efficiently. As in the Floyd's algorithm, we keep an auxiliary matrix to keep track of the choices we make. Let this new array be *bestk*. Now, when we compute m_{ij}, we save in $bestk[i,j]$ the value of k that corresponds to the cut with the minimum scalar multiplications. In general, the entry $bestk[i,j]$ will contain after which matrix we should make the cut, to compute $M_i \cdots M_j$. Thus, after the algorithm above terminates, $bestk[1,n]$ will tell us where to make the first cut. Then, we can recursively find the cuts on both terms, just like we did when we constructed the shortest path after Floyd's algorithm.

3 Exercises

1. In Floyd's algorithm, we computed all-pairs shortest paths. In this problem, we have a *directed* graph. Now, we want to compute if there is a path from i to j, for all pairs $i, j \in V$. Modify the Floyd's algorithm to solve this problem. *Hint*: the output is an $n \times n$ matrix P, where

$$P[i][j] = \begin{cases} 1 & \text{if } j \text{ is reachable from } i \text{ by a path in } G \\ 0 & \text{otherwise.} \end{cases}$$

2. This is not a problem per se, but something to think about: Recall that the Floyd's algorithm runs in time $O(n^3)$. Using the Dijkstra's algorithm(single-source shortest path), we can achieve the same task by running n iterations to choose different source each time. The running time of Dijkstra's Algorithm is $\Theta(n^2)$ when using distance matrices, and $\Theta((n+m)\log n)$ when using a heap. For the first case, the running time for solving all-pairs shortest

path is $n \times \Theta(n^2) = \Theta(n^3)$. For the secnd case, it is $n \times \Theta((n+m)\log n) = \Theta((n^2+nm)\log n)$. But m can be as large as n^2 (e.g. complete graphs), so the running time could be as bad as $O(n^3 \log n)$. Despite the simplicity of the Floyd's algorithm, we can see it can be really good, especially the graph is quite dense.