IEOR E4004: Introduction to OR: Deterministic Models

1 Dynamic Programming

Following is a summary of the problems we discussed in class. (We do not include the discussion on the "container" problem or the cannibals and missionaries problem because these were mostly "philosophical" discussions.)

1.1 Matrix Multiplication

We wish to compute the matrix product

$$A_1A_2\dots A_{n-1}A_n,$$

where A_i has a_i rows and a_{i+1} columns. How can find this product while minimizing the number of multiplications used?

To study this problem using dynamic programming, we let m_{ij} be the optimal number of multiplications to find the product $A_iA_{i+1}\dots A_j$, for $i<j$. The DP recursion is easily seen to be:

$$m_{i,i+1} = a_ia_{i+1}a_{i+2},$$

and

$$m_{i,j} = \min_{k:i<k<j}\left\{m_{i,k}+m_{k,j}+a_ia_{k+1}a_{j+1}\right\}, \forall j>i+1.$$

Alternatively, one can say:

$$m_{i,i} = 0,$$

and

$$m_{i,j} = \min_{k:i\le k\le j}\left\{m_{i,k}+m_{k,j}+a_ia_{k+1}a_{j+1}\right\}, \forall j>i.$$

We first find $m(\cdot)$ for all i and j for which $j-i$ is 0, then for all i and j for which $j-i$ is 1, etc. Observe that the DP recursion is such that if $(j-i)=k$, then $m(i,j)$ depends on $m(\cdot)$ values for those (i',j') pairs for which $j'-i'<k$. (Thus our way of finding the $m(\cdot)$ will work.) We are of course interested in $m(1,n)$.

The algorithm runs in $O(n^3)$ time: we need to fill a table of size $O(n^2)$, each of which requires $O(n)$ time to fill.

1.2 Knapsack problem

We are given a knapsack with capacity W. We have n items labeled $1,2,\dots,n-1,n$; item i has size s_i and value v_i. Any subset of items with total size $\le W$ can be packed into the knapsack. Thus our goal is to pack into the knapsack a subset of maximum value, among all possible subsets whose total size is at most W. Assume all the data are non-negative integers. How does one identify a maximum value subset?

We let $f(j,w)$ be the maximum value one can achieve in the knapsack problem with items $\{1,2,\ldots,j\}$ and with knapsack capacity w. We are interested in determining $f(n,W)$. It is easy to see that

$$f(j+1,w) \;=\; \max\Big\{f(j,w), v_{j+1} + f(j, w-s_{j+1})\Big\}.$$

(The second term exists only if $w \geq s_{j+1}$; otherwise it is taken to be zero.) To justify this, consider the following argument. If we are given a knapsack problem with $j+1$ items and knapsack capacity w, an optimal packing will either exclude the last item or include it; in the former case, we are left with a knapsack problem involving the first j items and knapsack capacity w, whereas in the latter case we are left with a knapsack problem involving the first j items and knapsack capacity $w - s_{j+1}$, but we also pick up an additional value v_{j+1}. We can now solve the recursion backwards: clearly $f(1,w)$ is trivial to determine, for all $0 \leq w \leq W$; using this and the recursion described above, we can determine $f(2,w)$ for all $0 \leq w \leq W$, etc. There are $n(W+1)$ entries to find, each of which can be determined in $O(1)$ time; so the complexity of the DP algorithm we just described is $O(nW)$.

1.3 Shortest path problem

Given a graph $G = (V,E)$, and given "distances" $c_{ij} \geq 0$ on the edges, find a shortest path from a given $s \in V$ to all other nodes in V. Let $c_{i,i} = 0$, for all $i \in V$. We also let $c_{i,j} = \infty$ for all $(i,j) \notin E$, so that we may now assume $c_{i,j}$ is defined for every $i,j \in V$. We considered two versions of the shortest path problem: the *one-to-many* version in which we need to find a shortest path from s to every other node in V; and the *many-to-many* version in which we need to find a shortest path from any node to every other node.

One-to-many. Let $f_k(j)$ be the shortest path length from s to j using at most k edges for $k = 1,2,\ldots,n-1$. Observe that we are interested in determining $f_{n-1}(v)$ for each $v \in V \setminus \{s\}$. The DP recursion is:

$$f_{k+1}(j) \;=\; \min_{i\in V}\Big\{f_k(i) + c_{ij}\Big\}.$$

The justification is as follows: consider *any* path that takes you from s to j using at most $k+1$ edges. Let i be the node that this path visits *just before* it reaches j, that is, let (i,j) be the last edge of this path. In that case, it is clear that the length of the "optimal" such path is $f_k(i) + c_{ij}$. Since we do not know what i is for the optimal path from s to j using at most $k+1$ edges, we try all possible values of i. We can compute the $f_k(\cdot)$ starting from $f_1(\cdot)$ (where it is trivial), and then using the recursion to find $f_2(\cdot)$, and then $f_3(\cdot)$, etc. Since the shortest paths from s to any node j does not use more than $n-1$ edges, it is enough to find $f_{n-1}(j)$ for all $j \in V \setminus \{s\}$. (What is the complexity of this procedure?)

Many-to-Many. One could solve the many-to-many problem by solving n independent one-to-many problems, one for each element $v \in V$ as a source. Here is another way. Let $g_k(i,j)$ be the shortest path length from i to j using only a subset of $\{1, 2, \dots, k\}$ as intermediate nodes. We maintain this quantity for all pairs $i, j \in V$. The DP recursion is:

$$g_{k+1}(i,j) \;=\; \min\Big\{g_k(i,j), g_k(i,k+1) + g_k(k+1,j)\Big\}.$$

The justification is as follows: the optimal path from i to j using only a subset of $\{1, 2, \dots, k, k+1\}$ as intermediate nodes may or may not use node $k+1$ as an intermediate node. If the path *does not* use $k+1$ as an intermediate node, then the path only uses a subset of $\{1, 2, \dots, k\}$ as intermediate nodes, and in that case its length is $g_k(i,j)$ (by definition). If the path *does* use $k+1$ as an intermediate node, then the total cost of this path can be decomposed into two parts: the part that takes you from i to $k+1$, and the part that takes you from $k+1$ to j; each of these paths can use only a subset of $\{1, 2, \dots, k\}$ as intermediate nodes, so the cost of the two parts are $g_k(i, k+1)$ and $g_k(k+1, j)$ respectively. Again, $g_1(\cdot)$ is trivial to compute; starting from this we can compute $g_2(\cdot)$ using the DP recursion, then we can find $g_3(\cdot)$, etc. The solution to the problem is given by $g_n(i,j)$ for all $i, j \in V$. The complexity of this procedure can be estimated as follows: there are $O(n^3)$ entries to be determined (n possible values of k, and $O(n)$ possible values of i and j each); each entry can be found by looking up three entries that are already stored in the table, and so can be found in $O(1)$ time.

1.4 Traveling salesman problem

In the traveling salesman problem, we are again given a *complete* directed graph with V being the set of nodes of the graph, and with c_{ij} representing the cost of going from node i to node j for all $i, j \in V, i \neq j$. The goal is to determine the minimum cost tour starting at node 1, visiting each node *exactly once*, and returning to node 1. To solve this problem by dynamic programming, we maintain $f(S, k)$, for each $S \subset V \setminus \{1\}$, and for each $k \in S$. The interpretation of $f(S,k)$ is that it is the cost of an optimal *path* that starts at node 1, visits each node in S *exactly once*, and ends at node k. The DP recursion is easily seen to be:

$$f(S,k) \;=\; \min_{j \in S \setminus \{k\}} \{f(S \setminus \{k\}, j) + c_{jk}.$$

That this recursion is correct follows from making a simple observation: any path that starts at 1, ends at k, and visits each node in S exactly once must have a *last edge* that goes from some node j to node k; in that case, the length of this path is precisely $f(S \setminus \{k\}, j) + c_{jk}$. Since we do not know the identity of j, we try all possibilities. Once again we start with all subsets S with one element, in which case determining $f(\cdot)$ is trivial; we then use this to determine $f(\cdot)$ on all subsets S with 2 elements, and then all subsets with 3 elements, etc. The optimal tour-length can be determined by finding

$$\min_k f(\{2, 3, \dots, n-1\}, k) + c_{k1}.$$

This is so because if any optimal tour visits some node last (before returning to node 1); if this node is k, then its length is precisely $f(\{2, 3, \dots, n-1\}, k) + c_{k1}$. Again, since we do not know k, we try all the

$(n-1)$ possibilities. The complexity of the DP procedure here is easy to determine: the number of states is $O(n2^n)$; each $f(\cdot)$ can be determined in $O(n)$ time, so the overall complexity is $O(n^2 2^n)$. While this is expensive, it is substantially better than naively enumerating all tours (in this case we should list (n-1)! tours.

1.5 A scheduling problem

As our last problem we turn to a very simple single machine scheduling problem. We have a single machine, n jobs, with job i needing processing time p_i on the machine. The *completion time* of job i is denoted C_i and represents the epoch at which job i completes its processing on the machine. To illustrate the definitions, suppose $n = 3$, and $p_1 = 1$, $p_2 = 4$, and $p_3 = 2$. If the jobs are processed in the order 123, then $C_1 = 1$, $C_2 = 5$, and $C_3 = 7$; if the jobs are processed in the order 231, then $C_2 = 4$, $C_3 = 6$, and $C_1 = 7$. Let $f_i(x)$ be the "cost" incurred by job i if its completion time is x. Suppose $f_i(x)$ is nondecreasing in x for each i. (Note that different jobs could have different "cost" functions.) For each sequence of jobs σ, let

$$g(\sigma) \;=\; \max_j \{f_j(C_j^\sigma)\},$$

where C_j^σ is the completion time of job j in the schedule σ. The goal is to find a σ that minimizes $g(\sigma)$. To illustrate, suppose $f_1(x) = 5x^2$, $f_2(x) = 3x$, and $f_3(x) = x + 3$. Then for the sequence (123), the cost of job 1 is $f_1(1) = 5$, the cost of job 2 is $f_2(5) = 15$, and the cost of job 3 is $f_3(7) = 10$; thus,

$$g(123) \;=\; \max\{5, 15, 10\} \;=\; 15.$$

For the sequence (231), the cost of jobs 1, 2, and 3 are 245, 12, and 9 respectively, which implies $g(231) = 245$. To determine the "best" sequence, we can try the remaining four sequences, find $g(\cdot)$ for each of them and find the best. This is doable for a problem with 3 jobs because there are only 6 sequences to try. In general for n jobs, there will be $n!$ sequences to try, so this becomes computationally expensive very soon. Can we do better using dynamic programming?

Observe that some job has to be scheduled last. This job finishes at time $P := \sum_{k=1}^{n} p_k$, so if job j is scheduled last, then its cost is $f_j(P)$, regardless of how the remaining jobs are sequenced. Let σ be any sequence in which j appears as the last job. Then $\sigma = (\sigma', j)$, where σ' is some ordering of the jobs $1, 2, \ldots, j-1, j+1, \ldots, n$. It should be clear that

$$g(\sigma) \;=\; \max\{g(\sigma'), f_j(P)\}.$$

This is because the job with the largest cost in σ is some job other than j (in which case its cost is $g(\sigma')$), or it is job j (in which case its cost is $f_j(P)$).

Now, let

$$j^* \in \arg\max_j f_j(P).$$

In other words, among all jobs j, let j^* be a job that minimizes $f_j(P)$. Consider the schedule obtained by scheduling j^* last; we are left with a problem involving $(n-1)$ jobs to which we apply the same rule recursively. We claim that this schedule is optimal.

Let $g^*(S)$ be the optimal cost of the scheduling problem when only the jobs in S are to be scheduled. Then it is clear that:

$$g^*(S) \ \geq \ g^*(S \setminus \{j\}), \quad \text{forany} j \in S \tag{1}$$

and

$$g^*(S) \ \geq \ \min_{j \in S} f_j(\sum_{k \in S} p_k) \tag{2}$$

The first inequality is true because if we have one job less to schedule we can only do better in an optimal solution as all "costs" are nondecreasing in the completion times; the second inequality is true because some job in S must have completion time $\sum_{k \in S} p_k$, so its cost must be at least $\min_{j \in S} f_j(\sum_{k \in S} p_k)$.

We are now ready to prove the optimality of our scheduling rule. If the number of jobs is 1, the rule is trivially optimal. Suppose the rule is optimal whenever there are fewer than n jobs. Consider an instance with n jobs. Let j^* be scheduled last, and let σ' be the schedule computed by the rule on the remaining instance. By induction, σ' is an optimal schedule for the problem in which all jobs are present except job j^*. Also,

$$g(\sigma', j^*) \ = \ \max\{g^*(\sigma'), f_j^*(P)\}.$$

But we know from (1) and (2) that each of the terms on the RHS is a *lower bound* on $g(\cdot)$ of an optimal schedule. This completes the proof.