Worksheet 20. Dynamic Programming III

Multiplying lots of matrices Suppose that we want to multiply four matrices, $A \times B \times C \times D$, with the following dimensions.

matrix	dimensions
A	50×20
B	20×1
C	1×10
D	10×100

This will involve iteratively multiplying two matrices at a time. Matrix multiplication is noncommutative: generally $A \times B \neq B \times A$; but it is associative: $A \times (B \times C) = (A \times B) \times C$. Thus we can compute our product of four matrices in many different ways,[1] depending on how we parenthesize it. Are some of these better than others?

Let's assume that multiplying an $m \times n$ matrix by an $n \times p$ matrix requires mnp multiplications. Fill in the rest of this table to compare the costs of several different ways of multiplying $ABCD$:

parenthesization	cost computation	total cost
$A \times ((B \times C) \times D)$	$20 \cdot 1 \cdot 10 + 20 \cdot 10 \cdot 100 + 50 \cdot 20 \cdot 100$	$120,200$
$(A \times (B \times C)) \times D$	$\boxed{}$	$\boxed{}$
$(A \times B) \times (C \times D)$	$\boxed{}$	$\boxed{}$
$A \times (B \times (C \times D))$	$\boxed{}$	$\boxed{}$

Problem 1. There is a natural greedy approach: multiply the "cheapest pair" available to reduce the problem's size by 1, and repeat. Does this work?

(*Hint:* Look at the table you just made.)

Suppose more generally that we want to compute the matrix product $A_1 \times A_2 \times \cdots \times A_n$ where the A_is are matrices with dimensions $m_0 \times m_1$, $m_1 \times m_2$, …, $m_{n-1} \times m_n$, respectively.

Problem 2. A parenthesization can naturally be represented by a binary tree. Comment, and draw the trees for the parenthesizations in the table.

Problem 3. For a tree to be optimal, its subtrees must also be optimal! This suggests a dynamic programming approach. Which subproblems correspond to subtrees?

Problem 4. Let $C(i,j) =$ the minimum cost of multiplying $A_i \times A_{i+1} \times \cdots \times A_j$. Relate $C(i,j)$ to the minimum cost of multiplying smaller subproducts.

(*Hint:* The optimal parenthesization splits the product $A_i \times A_{i+1} \times \cdots \times A_j$ into two pieces, $A_i \times \cdots \times A_{k^*}$ and $A_{k^*+1} \times \cdots \times A_j$ for some k^* between i and j.)

Problem 5. We have the crucial relation between subproblems, so we are nearly ready to write the algorithm. But what is the base case? That is, which values of $C(i,j)$ are filled in first, and in which order are the remaining entries filled in? Contrast with the edit-distance problem.

Problem 6. Now write the algorithm in pseudocode. It should run in time $O(n^3)$, where n is the number of matrices to be multiplied.

[1] How many?

Min-weight paths again Let's examine some shortcomings of Dijkstra's algorithm. As you saw on the problem set, Dijkstra's algorithm may not produce the correct answer on a graph with negative edge weights, even if minimal-weight paths exist.

Dijkstra's algorithm also doesn't take into account the number of edges used, which we might care about if (for example) we are writing a Google Maps algorithm and would like to minimize the number of instructions that our users with their puny human brains must follow.

Problem 7. If there is a walk from s to t that includes a negative-weight cycle, then there is no min-weight walk from s to t. Explain why.

Problem 8. Suppose that G is a graph with n vertices, of which s and t are two. Prove that if G has no negative-weight cycles, then the shortest *walk* between any two vertices s and t is a *path* (i.e., doesn't repeat vertices) and hence has at most $n-1$ edges.

Suppose now that G has no negative-weight cycles. We define, for i an integer and v a vertex in G,

$$\mathrm{dist}(i,v) = \text{the minimum weight of a path from } v \text{ to } t \text{ using } \le i \text{ edges.}$$

In light of Problem 8, we want to compute $\mathrm{dist}(n-1,s)$. (This is the *single-target* approach; it is also reasonable (and looks more like Dijkstra) to analyze the min weight of a path from s to v using $\le i$ edges.)

We need the crucial relation between subproblems.

Given an optimal path P from v to t, consider two cases:

if P uses $\le i-1$ edges then: $\mathrm{dist}(i,v) = \boxed{}$.

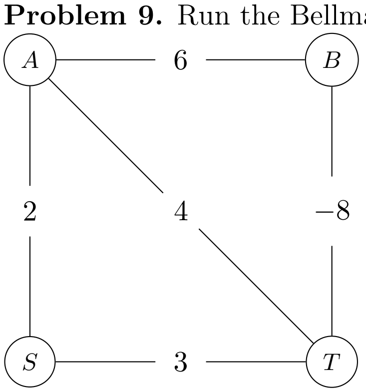
if P uses i edges and the first edge is (v,w), then:

$$\mathrm{dist}(i,v) = \boxed{}.$$

In other words,

$$\mathrm{dist}(i,v) = \min\left(\boxed{}, \boxed{}\right).$$

Problem 9. Run the Bellman–Ford algorithm on this example, with target vertex T.



	0	1	2	3
S	∞			
A	∞			
B	∞			
T	0			

Algorithm 1: Bellman–Ford

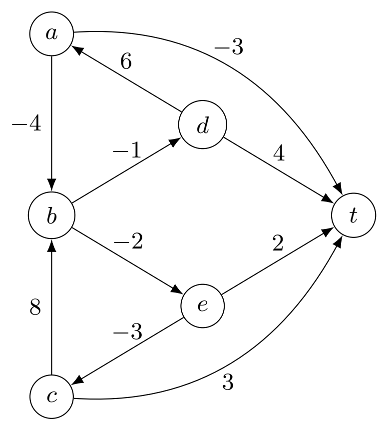
create an array M ;
set $M[0,t]=0$, $M[0,v]=\infty$ for all other vertices v ;
foreach $i = 1,\ldots,|V|-1$ **do**
  foreach $v \in V$ **do**
    set $M[i,v] = \min(M[i-1,v], \min_u(M[i-1,u] + \mathrm{wt}(u,v)))$;
return $M[n-1,t]$;

Problem 10.

(a) What happens if you run the Bellman–Ford algorithm on a graph with a negative cycle? Give a simple example.

(b) Using your previous answer, describe how the ideas from the Bellman–Ford algorithm could be used to write an algorithm that determines whether a weighted digraph has a negative cycle.

(c) Like Dijkstra's algorithm, the Bellman–Ford algorithm works just as well on weighted digraphs. Convince yourself that this is true by executing the algorithm starting at s on the weighted digraph pictured here.



	0	1	2	3	4	5
t	0					
a	∞					
b	∞					
c	∞					
d	∞					
e	∞					

Problem 11. Assuming that all edge-weights are nonnegative, compare and contrast Dijkstra's algorithm with the Bellman–Ford algorithm. Do you see any advantages to one algorithm over the other?

General principles of dynamic programming.

Optimal substructure: An optimal solution to the problem includes optimal solutions to subproblems.

Note that this also suggests a greedy solution! This is why many problems (e.g. interval-scheduling) have two variants: one that admits a greedy solution and another that admits a solution by dynamic programming. Usually DP succeeds where greedy methods fail, not vice versa.

Subproblem structure: (This one we have already discussed.) There is an underlying DAG of subproblems: $u \to v$ means that solving v requires first solving u.

Overlapping subproblems: This means that the recursion is amenable to memoization (storing solutions to subproblems).

Contrast this with Divide & Conquer, where usually subproblems are solved only once.

Common subproblem forms.

(1) **Initial segments (or final segments).** The input looks like $x_1, x_2, \ldots, x_n$, and the subproblems are $x_1, \ldots, x_j$ for $j \leq n$.

(E.g. weighted interval scheduling) Notice there are $O(n)$ subproblems.

(2) **Bi-initial segments.** The input looks like $x_1, \ldots, x_m$, $y_1, \ldots, y_n$ and the subproblems are

$$\begin{array}{ll} x_1, \ldots, x_i & i \leq m \\ y_1, \ldots, y_j & j \leq n \end{array}$$

(E.g. edit distance.) Notice that there are $O(mn)$ subproblems.

(3) **Interval segments.** The input looks like $x_1, \ldots, x_n$ and the subproblems are $x_i, \ldots, x_j$, $i \leq j \leq n$.
(E.g. chain matrix multiplication) Notice that there are $O(n^2)$ subproblems.

(4) **Subtrees.** The input is a rooted tree, and subproblems are rooted subtrees.
(E.g. chain matrix multiplication)

Problem 12. Listed below are the problems we have solved using dynamic programming. Categorize their subproblems using the four categories described above.

- weighted interval-scheduling
- min/max-weight paths in weighted DAGs
- longest increasing subsequence
- edit distance
- knapsack with repetition
- knapsack without repetition
- chain matrix multiplication
- min-weight paths with negative weights (Bellman–Ford)

Problem 13. Consider two problems: finding shortest paths in unweighted graphs and finding longest paths in unweighted graphs. Which of these two exhibits optimal substructure? Explain.

Problem 14. Consider merge-sort on an array of 16 elements. Would memoizing the recursion in merge-sort speed up the runtime? Explain.
(*Hint:* Draw the recursion tree.)