Homework 3: Dynamic Programming and More

Handed out Tue, Mar 11. **Due at the start of class (11am), Tue, Apr 1**. (electronic submission through Gradescope.)

Problem 1. (5 points) Work through the chain-matrix multiplication algorithm on a sequence of matrices $A_1 \cdot A_2 \cdot A_3 \cdot A_4$, where matrices are of dimensions 2×2, 2×5, 5×3, and 3×1, respectively. Following the algorithm from Lecture 10. Present both the M and H matrices and show the final multiplication order.

You may present your matrices either in traditional form, or in the rotated form that is used in Figure 5 (page 5) of Lecture 10. You may present the final multiplication order as a tree or by adding parentheses to "$A_1 \cdot A_2 \cdot A_3 \cdot A_4$".

To simplify your life, we have provided the first two diagonals of the M matrix in Fig. 1 below. (Hint: Be careful with your math. If you miscalculate an entry, it will ruin the entire result.)

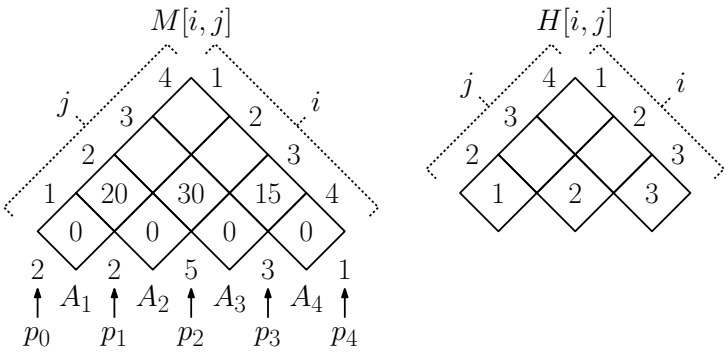


Figure 1: Chain-matrix multiplication.

Problem 2. (12 points) Let's return to the typesetting problem from Homework 3. Recall that we are given a line of length L and a paragraph consisting of a sequence of words whose lengths are $W = \langle w_1, \ldots, w_n \rangle$. (We assume that $w_i \leq L$ for all i.) We are to place words in order along each line subject to the condition that the sum of word lengths on any line does not exceed L. The *penalty* for each line is defined to be the difference between the sum of word lengths on this line and L. The objective is to place the words to minimize the *maximum penalty* over all the lines (see Fig. 2(a)).

In an earlier homework assignment, we showed that a greedy strategy is not optimal. In this problem we will show that this problem can be solved optimally by dynamic programming.

(a) (6 points) Derive a (recursive) dynamic programming formulation, which given L and the sequence of word lengths $W = \langle w_1, \ldots, w_n \rangle$, determines the segmentation of words to lines (without reordering) that minimizes the maximum penalty (see Fig. 2(a)). As is typical with DP problems, your formulation will compute the maximum penalty, not the actual segmentation. Briefly justify the correctness of your formulation.

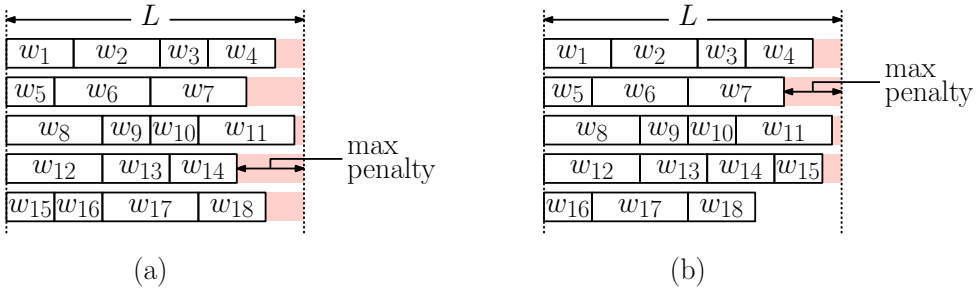


Figure 2: Optimal typesetting of words to minimize the maximum penalty.

(b) (4 points) Present an implementation of your DP formulation. What is its running time? It may help to imagine that you have access to a function $\text{len}(i, j)$ that returns the sum of word lengths from w_i up to w_j (assuming that $1 \leq i \leq j \leq n$) that runs in constant time.

(c) (2 points) In practice, when laying out a paragraph we do not care whether the last line is "ragged." Modify your solution from part (a) to compute the layout that minimizes the maximum layout *excluding* the last line. (For example, by this metric the layout shown in Fig. 2(b) has a lower cost than the layout from Fig. 2(a).)

As in part (a), briefly justify the correctness of your formulation.

Problem 3. (13 points) In your new job for a major chip manufacturer, you are tasked to help processing defects in the fabrication process. A fabricated chip is a square of some given side length L. After fabrication, it is tested for defects. Let us assume that the defects take the form of a set of points $P = \{p_1, \ldots, p_n\}$ (see Fig. 3(a)).

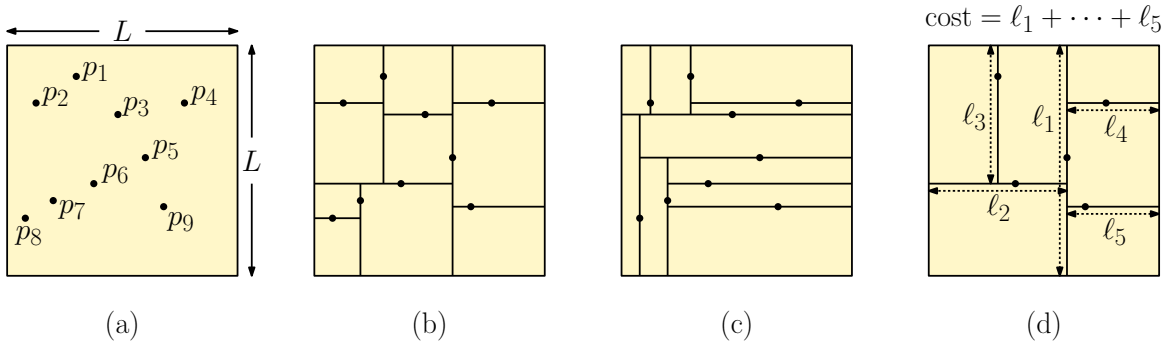


Figure 3: Cutting away defects in a fabricated chip.

Since we cannot sell chips with defects, we need to cut them out. The laser cutting tool makes a horizontal or vertical cut through a defect point that runs the entire length of chip. This splits the chip into two smaller chips. The cutting process is then repeated on each of these smaller chips. This is repeated until all the defects are cut out. The resulting set of defect-free rectangles are called *subchips* (see Fig. 3(b)). This is called a *hierarchical cutting*.

There are many different hierarchical cuttings for a given set of defects. To maximize profits, we want the sum of lengths of laser cuts to be as small as possible. (For example, Fig. 3(c)

shows another valid cutting, but uses longer cuts than (b).) Define the *total cost* of a hierarchical cutting to be the sum of lengths of all cuts (see Fig. 3(d)).

The objective of this problem is to devise an efficient DP algorithm, which given an $L \times L$ chip and set P of defect points, computes a hierarchical cutting that minimizes the total cost.

(a) (3 points) Throughout, let us assume that, among the defect points, there are no duplicate x-coordinates nor duplicate y-coordinates. Prove that if the number of defects is n, then the number of subchips in any hierarchical cutting is $n + 1$.

(b) (5 points) Derive a DP formulation, which given a set $P = \{p_1, \ldots, p_n\}$ of defects, determines the best (that is, minimum) total cost of any hierarchical cutting. Your formulation should be expressed as a recursive function. Be sure to include the basis case(s) for the recursive function, and indicate what initial function call provides the global answer. Justify the correctness of your formulation.

(Hint: The subproblems are associated with rectangles of the original image, which can arise through any valid hierarchical cutting process. It may be helpful to presort the x- and y-coordinates of the defect points.)

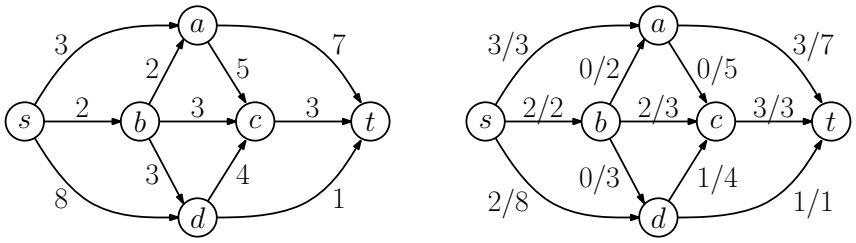
(c) (3 points) Present an implementation of your formulation from part (a). You may assume that you have been given access to functions to answer any geometric queries about the defects. For example, the following function may be useful. Given $x < x'$ and $y < y'$, the function $\mathsf{defectCount}(x, x', y, y')$, returns the number of defect points that lie within the *interior* of the rectangle $[x, x'] \times [y, y']$.

(Hint: Memoization is probably simpler than a bottom-up computation, but either is acceptable. Given that each subproblem is bounded by four sides, I would expect a running time of at least $O(n^4)$. It may be larger, however, depending on the amount of time it takes to compute each table entry.)

(d) (2 points) Derive the running time of your algorithm from (c).

Problem 4. (10 points) In this problem, we will trace the partial execution of the Ford-Fulkerson algorithm on a sample network.

(a) (2 points) Consider the s-t network G shown in Fig. 4(a), and consider the initial flow f in Fig. 4(b). Show the residual network G_f for this flow.



(a) Initial network G (b): Initial flow f

Figure 4: Tracing the Ford-Fulkerson algorithm.

(b) (2 points) Find any s-t path in G_f. How much flow can you push along this path? Show the updated flow (in the same manner as Fig. 4(b)).

(c) (2 points) Show the new residual network that results for your flow from (b).

(d) (1 points) Is this the maximum flow in this network? (If not, keep running Ford-Fulkerson until you get the maximum flow, and show the final flow.) What is the value of the maximum flow?

(e) (1 points) Show the residual network for your maximum flow from (d). (If the flow from (c) was already maximum, then state this.)

(f) (2 points) Show the cut that results by partitioning the network into two subsets of vertices, the vertices X that are reachable from s and the remaining vertices $Y = V \setminus X$. What is the capacity of this cut? (It should match your flow value, if you did everything correctly.)

Problem 5. (10 points) In this problem we will consider network flows involving networks with vertex capacities, rather than edge capacities. You are given a directed s-t network $G = (V, E)$, in which each vertex $u \in V \setminus \{s, t\}$ (that is, every vertex excluding the source and sink) is associated with a nonnegative *capacity*, denoted $c(u)$ (see Fig. 5(a)). We call this a *vertex-capacitated network*. A *flow* in G is defined the same as for a standard network except the capacity constraint applies to the flow into of each vertex (excluding s and t), that is,

$$\forall u \in V \setminus \{s, t\}, \quad \sum_{(w,u)\in E} f(w, u) \leq c(u)$$

As with standard flows, excluding s and t, the flow into the vertex must equal the flow out of the vertex (see Fig. 5(b)).

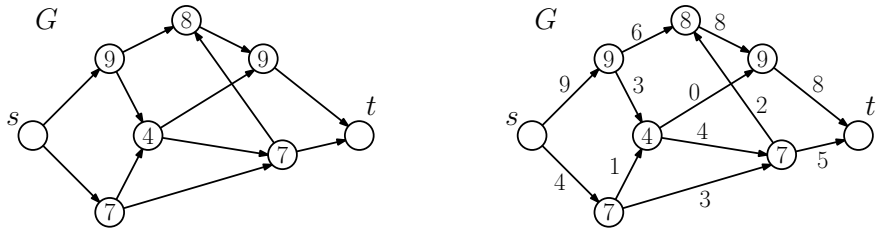


Figure 5: (a) A network with capacities on the vertices and (b) a valid flow.

We assert that having vertex capacities is essentially the same as having edge capacities. To show this, answer the following two questions.

(a) (5 points) Explain how to modify any vertex-capacitated network G into an equivalent edge-capacitated network G' so that, for any flow in G there exists a flow of equivalent value in G', and vice versa.

(b) (5 points) Explain how to modify any edge-capacitated network G into an equivalent vertex-capacitated network G' so that, for any flow in G there exists a flow of equivalent value in G', and vice versa.

Your answers should first explain how to convert G into G'. (Hint: The conversions are both very simple and are implementable in $O(m+n)$ time. They do *not* involve computing flows, residual networks, or cuts.) Next, show for any flow f in G, there exists a flow f' in G' of equal value, and vice versa. If done carefully, these proofs can be quite long. Instead, it is fine to give a brief explanation of your construction, and then provide a few figures illustrating your conversion and how flows in G correspond to flows in G'.

(Note: Challenge problems count for extra credit points. These additional points are factored in only *after* the final cutoffs have been set and can only increase your final grade.)

Challenge Problem 1. In Problem 2, we suggested using a function $\text{len}(i, j)$ that return the sum of word lengths w_i through w_j in constant time. Show that, given the word lengths $\langle w_1, \ldots, w_n \rangle$, after $O(n)$ preprocessing time it is possible to build a data structure from which $\text{len}(i, j)$ can be computed in $O(1)$ time. (If you do not see how to do this, you might try for a solution in which the preprocessing time is increased to $O(n^2)$ and/or the access time is increased to $O(\log n)$.)

Challenge Problem 2. In Problem 3, we suggested using a function to count the number of defects in a given rectangular region. The following problem is closely related. Suppose that we have a matrix $M[1..n][1..n]$ of integers. Given indices (i, i', j, j'), where $1 \leq i \leq i' \leq n$ and $1 \leq j \leq j' \leq n$, we want to compute the sum of all the entries of M lying within between rows i and i' and between columns j and j'. That is, define

$$\text{blockSum}(i, i', j, j') \;=\; \sum_{i''=i}^{i'} \sum_{j''=j}^{j'} M[i'', j''].$$

Show that, given M, after $O(n^2)$ preprocessing time, it is possible to build a data structure with $O(n^2)$ space from which this function can be computed in $O(1)$ time. (If you do not see how to do this, you might try for a solution in which the preprocessing time is increased to $O(n^4)$ and/or the access time is increased to $O(\log n)$.)