Solutions to Practice Problems 8

Solution 1:

(a) We claim that given n defects, there will be $n+1$ subchips. The proof by (strong) induction on n. The basis ($n = 0$) is trivial (zero defects and one chip). The first cut eliminates one defect. Suppose there are n_1 and n_2 defects remaining in the interiors of the two resulting subrectangles. We have $n_1 + n_2 = n - 1$. By the induction hypothesis, cutting these results in $n_1 + 1$ and $n_2 + 1$ subchips, respectively. Combining these, we have a total of

$$(n_1 + 1) + (n_2 + 1) \;=\; (n_1 + n_2) + 2 \;=\; (n - 1) + 2 \;=\; n + 1$$

subchips, as desired.

(b) The subproblems are the possible rectangles within the original chip. Let's assume that we have sorted the x-coordinates of the points in ascending order $x_1 \leq \cdots \leq x_n$ and the same for the y-coordinates $y_1 \leq \cdots \leq y_n$. Let's add two additional coordinates to each list to cover the sides of the enclosing square by defining $x_0 = y_0 = 0$ and $x_{n+1} = y_{n+1} = L$ (see Fig. 1(a)). As a convenience, let us assume we have access to a geometry query defects(i, i', j, j'), which returns a count of defects in the interior of the rectangle $[x_i, x_{i'}] \times [y_j, y_{j'}]$.

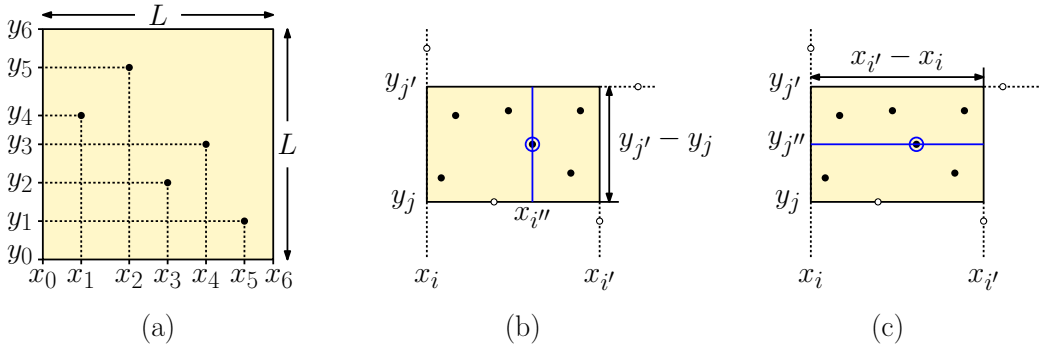

Figure 1: Chip cutting for $n = 5$.

Given $0 \leq i < i' \leq n+1$, and $0 \leq j < j' \leq n+1$, define $C(i, i', j, j')$ to be the minimum total cost of any hierarchical cutting of the rectangle $[x_i, x_{i'}] \times [y_j, y_{j'}]$. Ultimately, our objective is to cut the entire chip, that is, $C(0, n+1, 0, n+1)$.

In order to compute $C(i, i', j, j')$ we first observe the basis case that if this subrectangle has no defects in its interior, that is, defects$(i, i', j, j') = 0$, then it does not need to be cut, and hence its cost is zero. (Another possible basis case would be when $i' = i + 1$ or $j' = j'$, but we assert that before we get to this point, we will encounter the defect-count basis case, since such a rectangle cannot contain any defects in its interior.)

Otherwise, we will need to apply either a vertical or horizontal cut. There are two cases:

- We make a vertical cut between x_i and $x_{i'}$, by considering all cuts through $x_{i''}$, where $i < i'' < i'$. This generates a cutting cost of $y_{j'} - y_j$ and yields two subrectangles whose total cost we compute recursively as $C(i, i'', j, j') + C(i'', i', j, j')$ (see Fig. 1(b)). Thus, we have

$$C(i, i', j, j') \;=\; (y_{j'} - y_j) + \min_{i<i''<i'} C(i, i'', j, j') + C(i'', i', j, j').$$

- Otherwise, we make a horizontal cut between y_j and $y_{j'}$, by considering all cuts through $y_{j''}$, where $j < j'' < j'$. This generates a cutting cost of $x_{i''} - x_i$ and yields two subrectangles whose total cost we compute recursively as $C(i, i', j, j'') + C(i, i', j'', j')$ (see Fig. 1(c)).

$$C(i, i', j, j') \;=\; (x_{i'} - x_i) + \min_{j<j''<j'} C(i, i', j, j'') + C(i, i', j'', j').$$

Observe that the principle of optimality holds, which means that when cutting the two subrectangles, we should do so in a manner that minimizes the total cutting costs. Combining these, we have the following recursive rule:[1]

$$C(i, i', j, j') \;=\; \begin{cases} 0 & \text{if defects}(i, i', j, j') = 0 \\ \min \begin{pmatrix} (y_{j'} - y_j) + \min_{i<i''<i'} C(i, i'', j, j') + C(i'', i', j, j') \\ (x_{i'} - x_i) + \min_{j<j''<j'} C(i, i', j, j'') + C(i, i', j'', j') \end{pmatrix} & \text{otherwise.} \end{cases}$$

The overall cost is $C(0, n+1, 0, n+1)$, which covers the entire rectangle $[x_0, x_{n+1}] \times [y_0, y_{n+1}] = [0, L] \times [0, L]$.

Solution 2: For $1 \le i, j \le n$, let $P_{i,j}$ denote the maximum probability of any path from i to j. Our objective is to compute $P_{i,j}$ for all i and j.

(a) For $0 \le k \le n$, let $P_{i,j}^{(k)}$ denote the maximum probability of any path from i to j, subject to the restriction that the intermediate vertices (that is, the vertices between i and j along the path) are taken from $\{1, \ldots, k\}$.

To start things off, define $p_{i,j} = p(i, j)$ if $i \neq j$ and $p_{i,i} = 1$. For the basis case, we observe that $P_{i,j}^{(0)} = p_{i,j}$, since such a path consists of either a single edge (if $i \neq j$) or an empty path (if $i = j$). The probability of an empty path is 1 (since we do not need to traverse any edges).

For $k \ge 1$, observe that any path from i to j that passes through intermediate vertices $\{1, 2, \ldots, k\}$ either passes through k or it does not. If not, then the maximum probability path has already been computed as $P_{i,j}^{(k-1)}$. If so, observe that there is no benefit to repeatedly

[1] There is something obviously wasteful about this formulation. When making the vertical cut at $x_{i''}$, we did not bother to check that the point having this x-coordinate even lies within the rectangle $[x_i, x_{i'}] \times [y_j, y_{j'}]$! (The same applies for $y_{j''}$.) It turns out the sloppiness does not adversely affect the correctness nor the asymptotic running time. To see why, observe first that any such cut will not be helpful in forming the optimum solution, since the cut does not pass through a defect, and hence it does not reduce the number of defects. It can also be shown that, even if we consider only points lying within the rectangle, the running time will be $O(n^5)$, only with a smaller constant. Thus, while it is wasteful from a practical perspective, if we are wearing our "theoretician pants", we just don't care.

cycle through k, since this could only decrease the path probability. Thus, the path first goes from i to k, using intermediate vertices 1 through $k-1$, and then goes from k to j, using intermediate vertices 1 through $k-1$. Thus, the probability is the product of these two, which is $P_{i,k}^{(k-1)} \cdot P_{k,j}^{(k-1)}$. Since we wish the maximize the probability, we take the maximum of these two alternatives. This yields the following formulation:

$$P_{i,j}^{(k)} = \begin{cases} 1 & \text{if } k = 0 \text{ and } i = j, \\ p(i,j) & \text{if } k = 0 \text{ and } i \neq j, \\ \max\left(P_{i,j}^{(k-1)}, P_{i,k}^{(k-1)} \cdot P_{k,j}^{(k-1)}\right) & \text{otherwise.} \end{cases}$$

The final max probability is $P_{i,j} = P_{i,j}^{n}$. This is clearly has the same form as the Floyd-Warshall algorithm, and hence the running time is $O(n^3)$.

(b) We note that the above algorithm is essentially equivalent to the Floyd-Warshall algorithm where we have replaced min with max and changed addition into multiplication. This suggests that the transformation is based on taking the logarithms of the weights.

We transform G by replacing each edge probability $p(i,j)$ with the weight $w(i,j) \leftarrow -\log p(i,j)$. (The base of the logarithms does not matter.) Observe that if $p(i,j) = 0$ (that is, the edge does not exist), then this effectively treats the weight as $-\log 0 = +\infty$. This corresponds to the fact that in the shortest-path problem, a non-existent edge is treated as having weight $+\infty$. Observe that smaller probabilities correspond to larger edge weights. Let G' denote the digraph with modified weights.

Given any path $\pi = \langle v_0, \ldots, v_k \rangle$, let $P(\pi)$ denote the probability of this path in G and let $d(\pi)$ denote the length of the path in G'. The shortest path problem for G' is equivalent to finding the path π that minmizes $d(\pi)$. Using the fact that the sum of logs is the log of the product, for any path, it follows that the path length is

$$d(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i) = \sum_{i=1}^{k} -\log p(v_{i-1}, v_i) = -\log \prod_{i=1}^{k} p(v_{i-1}, v_i) = -\log P(\pi).$$

Since the logarithm function is monotone, it follows that minimizing the path length in G' is equivalent to maximizing the path probability in G. Thus, after running the Floyd-Warshall algorithm on G', we can obtain the max probability path by transforming the distance by negating and exponentiating it.

Solution 3: (This is structurally equivalent to part (a) of the previous problem, but the basis case needs to be handled carefully.) For $1 \leq i, j \leq n$, let $C_{i,j}$ denote the number of distinct paths from i to j. For $0 \leq k \leq n$, let $C_{i,j}^{(k)}$ denote the number of distinct paths from i to j, subject to the restriction that the intermediate vertices (that is, the vertices between i and j along the path) are taken from $\{1, \ldots, k\}$. It will be convenient to ignore empty paths, and set $C_{i,i} = 0$. (Formally, there is an empty path each vertex to itself, but if we were to count this path, we'll see below that it results in double-counting some paths due to the way we set up our DP formulation.)

To start things off, define $a_{i,j} = \text{adj}[i,j]$ if $i \neq j$ and $a_{i,i} = 0$. For the basis case, we have $C_{i,j}^{(0)} = a_{i,j}$ if $i \neq j$.

For $k \geq 1$, observe that any path from i to j that passes through intermediate vertices $\{1, \ldots, k\}$ either passes through k or it does not. If not, then the number of paths has already been computed as $C_{i,j}^{(k-1)}$. If so, then we can concatenate any path from i to k with any path from k to j. (We need not consider paths from k to k, since G is a DAG.) Thus, the total number of paths is the product $C_{i,k}^{(k-1)} \cdot C_{k,j}^{(k-1)}$. This yields the following formulation:

$$C_{i,j}^{(k)} = \begin{cases} a_{i,j} & \text{if } k = 0, \\ C_{i,j}^{(k-1)} + \left(C_{i,k}^{(k-1)} \cdot C_{k,j}^{(k-1)}\right) & \text{otherwise.} \end{cases}$$

The final count is $C_{i,j} = C_{i,j}^{n}$. This is clearly has the same form as the Floyd-Warshall algorithm, and hence the running time is $O(n^3)$.

As mentioned above, this does not count the empty path from each vertex to itself. If this bothers you, we can follow this up by setting $C_{i,i} \leftarrow 1$, for all i. You might wonder why we don't fix this by setting $a_{i,i} = 1$. Unfortunately, this would result in double counting, since if $i = k$, the above formulation gives

$$C_{i,j}^{(i)} = C_{i,j}^{(i-1)} + \left(C_{i,i}^{(i-1)} \cdot C_{i,j}^{(i-1)}\right) = C_{i,j}^{(i-1)} + \left(a_{i,i} \cdot C_{i,j}^{(i-1)}\right) = 2C_{i,j}^{(i-1)},$$

which is not correct.

Solution 4:

(a) We convert a vertex-capacitated network $G = (V, E)$ into an equivalent edge-capacitated network $G' = (V', E')$ as follows. First, we split each $u \in V$ vertex other than s into t vertex into a pair of vertices $u', u'' \in V'$, which are connected by a "mini-edge". This mini-edge has a capacity equal to the vertex capacity. Let s'' and t' denote the source and sink vertices in G'. Next, for each edge (u, v) in the original graph, create an edge (u'', v') of infinite capacity in your new network (see Fig. 2(b)).

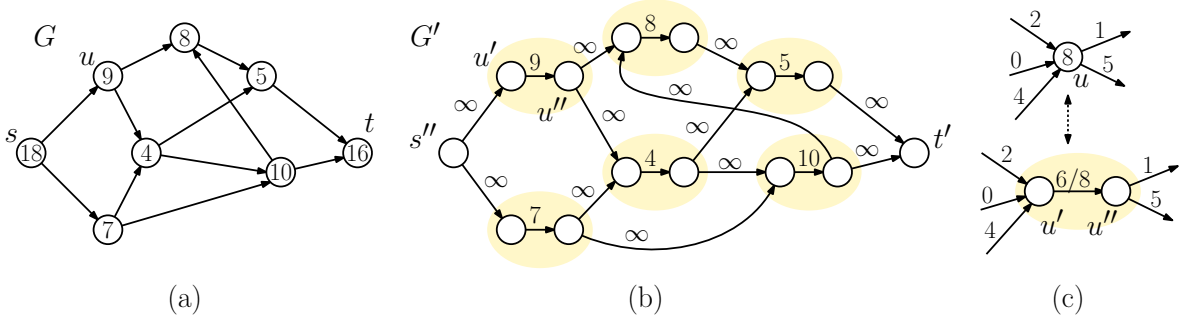

Figure 2: Transforming a vertex-capacitated network to an edge-capacitated network.

To establish correctness, we show that given any flow f in G there exists a flow of equal value in G'. To go from G to G', we just copy the flow values from each edge (u, v) to the corresponding edge (u'', v'). We set the flow on the mini-edge to the sum of flows on the incoming edges. By flow conservation this must match the sum of flows on the outgoing edges (see Fig. 2(c)). The capacity on the mini-edge enforces the vertex capacity. To go in

the other direction, we copy the flows on the edges (u'', v') to the original edge (u, v). Because of the capacity constraint on the mini-edges, the flow through each vertex satisfies the vertex capacities.

(b) We convert an edge-capacitated network $G = (V, E)$ into an equivalent vertex-capacitated network $G' = (V', E')$ as follows. We split each edge into two edges by adding a "mini-vertex" in the middle. We set the capacity of the mini-vertex to the capacity of the edge. We set the capacities of the original vertices to ∞ (see Fig. 3(b)).

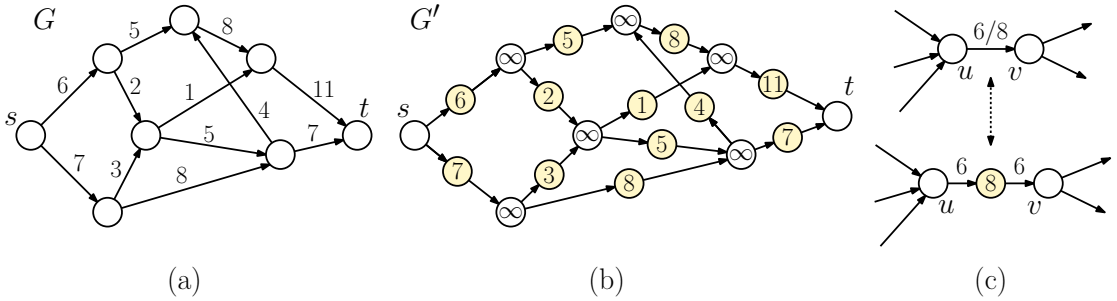

Figure 3: Transforming an edge-capacitated network to a vertex-capacitated network.

To establish correctness, we show that given any flow f in G there exists a flow of equal value in G'. To go from G to G', we just copy the flow value on each edge (u, v) to the two halves of the split edge. We set the flow on the two edges to be the same as the flow in the original edge (see Fig. 3(c)). The capacity on the mini-vertex enforces the edge capacity constraint. To go in the other direction, observe that the flows on the two copies of each split edge are equal, and we copy the flow to the original edge (u, v).