ASSIGNMENT 4: Solutions CMSC 451 (Spring 2016)

1. *Comparing strings.*

   In this problem you will explore some of the many possible measures of similarity between pairs of strings. Give an efficient algorithm for computing each of the following measures, with a proof of correctness and analysis of the running time in each case.

   (a) *Longest common subsequence.* A *subsequence* of a string is obtained by taking a subset of its characters without changing their order. (For example, ART is a subsequence of ALGORITHM.) A *common subsequence* of a pair of strings x and y is a string that is a subsequence of both x and y. Consider the length of a longest common subsequence of x and y.

   Solution: Let $x = x_1x_2 \ldots x_m$ and $y = y_1y_2 \ldots y_n$ denote the two input strings. For $1 \leq i \leq m$ and $1 \leq j \leq n$, let $\text{Opt}(i, j)$ denote the length of the longest common subsequence of the strings $x_1 \ldots x_i$ and $y_1 \ldots y_j$.

   Now there are two possibilities: either (i) the last characters match, i.e., $x_i = y_j$ or (ii) the last characters do not match. In the former case, the optimal longest common subsequence of $x_1 \ldots x_i$ and $y_1 \ldots y_j$ is to append x_i (or y_j) to the longest common subsequence of the shorter strings $x_1 \ldots x_{i-1}$ and $y_1 \ldots y_{j-1}$. In the latter case, the longest common subsequence can either use the character x_i (so we can safely drop y_j) or it may not (so we can safely drop x_i). From this discussion, we obtain the following recurrence:

$$\text{Opt}(i, j) = \begin{cases} 1 + \text{Opt}(i-1, j-1) & \text{if } x_i = y_j \\ \max\{\text{Opt}(i-1, j), \text{Opt}(i, j-1)\} & \text{if } x_i \neq y_j. \end{cases}$$

   The base cases can be computed easily as $\text{Opt}(0, j) = \text{Opt}(i, 0) = 0$.

   Running Time: Since computing $\text{Opt}(i, j)$ for each $1 \leq i \leq n$ and $1 \leq j \leq m$ takes only constant time by using previously computed values, the total time complexity is $O(mn)$.

   (b) *Edit distance.* The *edit distance* between x and y is the smallest number of single-character insertions, deletions, or substitutions that suffice to change x into y.

   Solution: Let $x = x_1x_2 \ldots x_m$ and $y = y_1y_2 \ldots y_n$ denote the two input strings. For $1 \leq i \leq m$ and $1 \leq j \leq n$, let $\text{Opt}(i, j)$ denote the optimal edit distance between strings $x_1 \ldots x_i$ and $y_1 \ldots y_j$.

   We have the following possibilities. If $x_i \neq y_j$, then we must either (i) delete x_i, or (ii) insert y_j, or (iii) substitute x_i into y_j. If $x_i = y_j$, then we have the same three options, but we no longer need to substitute x_i by y_j as they are identical. Thus, we have the following recurrence:

$$\text{Opt}(i, j) = \min \begin{cases} \text{Opt}(i-1, j) + 1 & \text{// Deleting } x_i \\ \text{Opt}(i, j-1) + 1 & \text{// Inserting } y_j \\ \text{Opt}(i-1, j-1) + (x_i = y_j ?\ 0 : 1) & \text{// Substituting} \end{cases}$$

   where $(x_i = y_j ?\ 0 : 1)$ equals 0 if $x_i = y_j$ and equals 1 otherwise.

   The base cases can be computed easily as $\text{Opt}(i, 0) = i$ and $\text{Opt}(0, j) = j$.

   Running Time: Since computing $\text{Opt}(i, j)$ for each $1 \leq i \leq n$ and $1 \leq j \leq m$ takes only constant time by using previously computed values, the total time complexity is $O(mn)$.

(c) *Edit distance with transpositions.* The *edit distance with transpositions* between x and y is the smallest number of single-character insertions, deletions, or substitutions, or transpositions of two adjacent characters, that suffice to change x into y. (For example, the edit distance with transpositions between NOSE and ONCE is 2, whereas their edit distance is 3.)

Solution: As above, let $\text{Opt}(i, j)$ denote the optimal edit distance with transpositions between strings $x_1 \dots x_i$ and $y_1 \dots y_j$.

In addition to the cases considered above, if $x_i = y_{j-1}$ and $x_{i-1} = y_j$, then now we have an additional option of swapping adjacent characters. Incorporating this change into the dynamic program above, we get the following recurrence.

$$\text{Opt}(i,j) = \min \begin{cases} \text{Opt}(i-1,j)+1 & \text{// Deleting } x_i \\ \text{Opt}(i,j-1)+1 & \text{// Inserting } y_j \\ \text{Opt}(i-1,j-1)+(x_i = y_j ?\ 0 : 1) & \text{// Substituting} \\ \text{Opt}(i-2,j-2)+((x_i = y_{j-1} \,\&\, x_{i-1} = y_j)?\ 1 : \infty) & \text{// Transposition} \end{cases}$$

where $((x_i = y_{j-1} \,\&\, x_{i-1} = y_j)?\ 1 : \infty)$ equals 1 if the swap is valid and equals infinity otherwise.

Running Time: Since computing $\text{Opt}(i, j)$ for each $1 \le i \le n$ and $1 \le j \le m$ takes only constant time by using previously computed values, the total time complexity is $O(mn)$.

2. *Processing sheet metal.*

You run a company that processes sheet metal. You have a price list indicating that a rectangular piece of sheet metal of dimensions $x_i \times y_i$ can be sold for v_i dollars, where $i \in \{1, 2, \dots, n\}$. Starting from a raw piece of sheet metal of dimensions $A \times B$, you would like to make a sequence of horizontal or vertical cuts, each of which cuts a given piece into two smaller pieces, to produce pieces from your price list (and possibly some additional worthless scrap pieces). Design an algorithm that maximizes the total value of the pieces obtained by some sequence of cuts. Your algorithm should return both the maximum value that can be obtained and a description of the cuts that should be made to achieve it. Assume that the numbers A, B, and x_i, y_i, v_i for $i \in \{1, 2, \dots, n\}$ are all positive integers. Demand for everything on your price list is high, so you can produce any number of copies of any of the pieces. Prove that your algorithm is correct and analyze its running time.

Solution: For $1 \le a \le A$ and $1 \le b \le B$, let $\text{Opt}(a, b)$ be the maximum value that can be obtained from a rectangular piece of sheet metal of dimension $a \times b$.

We now have three possibilities:

(a) The dimensions of the sheet $a \times b$ matches one of the sheets from the price list, i.e., $\exists i, x_i = a \ \&\ y_i = b$. In this case, we can obtain a value of v_i.

(b) We can make a horizontal cut at some integer index $1 \le p \le b - 1$ to obtain two pieces of sheet metal of dimensions $a \times p$ and $a \times (b - p)$.

(c) We can make a vertical cut at some integer index $1 \le q \le a - 1$ to obtain two pieces of sheet metal of dimensions $q \times b$ and $(a - q) \times b$.

Thus, we have the following recurrence:

$$\text{Opt}(a,b) = \max \begin{cases} v_i & \text{if } x_i = a \text{ and } y_i = b \\ \max_{1 \le p \le b-1} \left[\text{Opt}(a,p) + \text{Opt}(a,b-p)\right] & \\ \max_{1 \le q \le a-1} \left[\text{Opt}(q,b) + \text{Opt}(a-q,b)\right] & \end{cases}$$

Algorithm: The algorithm can be modified easily to obtain the actual sequence of cuts that yields the maximum value by maintaining back pointers. Algorithm 1 shows the complete pseudocode.

```
// Initialization
for a = 0, ..., A do
    Opt(a, 0) = 0
for b = 0, ..., B do
    Opt(0, b) = 0
for a = 1, ..., A do
    for b = 1, ..., B do
        best_hor_cut = max_{1≤p≤b-1}[Opt(a, p) + Opt(a, b − p)];
        best_ver_cut = max_{1≤q≤a-1}[Opt(q, b) + Opt(a − q, b)];
        value = 0;
        for i = 1, ..., n do
            if x_i = a and y_i = b then
                value = v_i
        Opt(a, b) = max{value, best_hor_cut, best_ver_cut};
        // Store the action taken at this step
        if Opt(a, b) = value then
            action(a, b) = "Sell piece a × b";
            prev(a, b) = [(0,0), (0,0)]
        else if Opt(a, b) = best_hor_cut then
            action(a, b) = "Cut a × b horizontally at arg max_{1≤p≤b-1}[Opt(a, p) + Opt(a, b − p)]";
            prev(a, b) = [(a, p), (a, b − p)]
        else if Opt(a, b) = best_ver_cut then
            action(a, b) = "Cut a × b vertically at arg max_{1≤q≤a-1}[Opt(q, b) + Opt(a − q, b)]";
            prev(a, b) = [(q, b), (a − q, b)]
GetDescription(A, B);


// Get a description of cuts
Function GetDescription(a, b):
if a > 0 and b > 0 then
    print action(a, b);
    GetDescription(prev(a, b)[0]);
    GetDescription(prev(a, b)[1]);
```

Algorithm 1: Algorithm to find optimal cuts of sheet metal

Running Time: Since computing $\text{Opt}(a,b)$ for each $1 \le a \le A$ and $1 \le b \le B$ takes

$O(\max(a, b, n))$ time, the total time complexity is $O(AB \max(A, B, n))$.

3. *Network flows with vertex capacities.*

   (a) Let $G = (V, E)$ be a directed graph with source vertex $s \in V$ and sink vertex $t \in V$. Whereas the standard network flow problem involves capacities for the edges, suppose that instead every *vertex* $v \in V$ has a capacity $c_v \geq 0$. A *vertex-capacitated flow* in G is a function $f : E \to \mathbb{R}^+$ such that

   i. *(Capacity conditions)* For each vertex $v \in V$, we have

   $$\sum_{e \text{ into } v} f(e) \leq c_v \quad \text{and} \quad \sum_{e \text{ out of } v} f(e) \leq c_v.$$

   ii. *(Conservation conditions)* For each vertex $v \in V \setminus \{s, t\}$, we have

   $$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e).$$

   As usual, the value of a flow is $\sum_{e \text{ out of } s} f(e)$. Give an efficient algorithm to find a maximum vertex-capacitated flow in G from s to t. Establish its correctness and analyze its running time.

   Solution: We reduce the maximum flow problem in vertex-capacitated graphs to the standard network flow problem with edge capacities. For every vertex $v \in V$, we create two vertices v_{in} and v_{out} and add an edge (v_{in}, v_{out}) whose capacity is equal to the capacity of the vertex v. Then for every edge $(u, v) \in E$, we add an edge (u_{out}, v_{in}) of infinite capacity. Let $H = (V', E')$ denote the new edge-capacitated directed graph constructed above. The source of H is s_{in} and the sink of H is t_{out}.

   Now we claim that G admits a vertex-capacitated flow of value F if and only if H admits a flow of value F.

   $\Rightarrow$ Let $f' : E' \to \mathbb{R}^+$ be a feasible flow in H. We can define a flow $f : E \to \mathbb{R}^+$ that is feasible for G as follows: $f(u, v) = f'(u_{out}, v_{in})$. The flow f satisfies capacity conditions at every vertex $v \in V$ as

   $$\sum_{e=(u,v)} f(e) = \sum_{e'=(u_{out}, v_{in})} f'(e') = f'(v_{in}, v_{out}) \leq c_v.$$

   Similarly, f satisfies the conservation conditions as

   $$\sum_{e=(u,v)} f(e) = \sum_{e'=(u_{out}, v_{in})} f'(e') = f'(v_{in}, v_{out}) = \sum_{e''=(v_{out}, w_{in})} f'(e'') = \sum_{e=(v,w)} f(e)$$

   $\Leftarrow$ Similarly, let $f : E \to \mathbb{R}^+$ be a feasible vertex-capacitated flow in G. Then $f'(u_{out}, v_{in}) = f(u, v)$ and $f'(v_{in}, v_{out}) = \sum_{e \text{ into } v} f(e)$ is a feasible flow for H of the same value.

   Thus, to find a maximum vertex-capacitated flow in G, it suffices to find a maximum (edge-capacitated) flow in H. Note that if G has n vertices and m edges, then H has $2n$ vertices and $n + m$ edges. Furthermore, the total capacity C of the edges out of s_{in} in H is simply the capacity of s in G. Thus the Ford-Fulkerson algorithm finds a maximum flow in G in time $O((m + n)C)$. Clearly H can be constructed from G in time $O(m + n)$, and a maximum vertex-capacitated flow in G can be reconstructed from a maximum flow in H in time $O(m + n)$. Overall, the running time of the algorithm for finding a maximum vertex-capacitated flow in G is $O((m + n)C)$.

(b) Define a *cut* in a vertex-capacitated network to be a vertex subset $U \subseteq V$ such that every path from s to t goes through at least one vertex of U. The *capacity* of the cut U is $\sum_{u \in U} c_u$. Using these definitions, state and prove an analog of the Max-Flow/Min-Cut Theorem for vertex-capacitated networks.

Solution: Vertex-Capacitated Max-Flow/Min-Cut Theorem: In a vertex-capacitated network, the maximum value of a vertex-capacitated s-t flow is equal to the minimum capacity of a vertex cut.

Proof: Given a vertex-capacitated directed graph $G = (V, E)$, we construct an edge-capacitated directed graph $H = (V', E')$ as described above. We claim that the capacity of the minimum-capacity vertex cut in G is equal to that of the minimum-capacity s-t cut in H.

$\Rightarrow$ Let (A, B) denote the minimum-capacity s-t cut in H. Since only edges of the form $e' = (v_{in}, v_{out})$ have finite capacity in H, let $U = \{v \in V \mid (v_{in}, v_{out}) \text{ is an edge in cut } (A, B)\}$. By definition, U is a vertex cut in G and capacity of U = capacity of cut (A, B).

$\Leftarrow$ Let $U \subseteq V$ be the minimum-capacity vertex cut in G. Then by construction, the set of edges (v_{in}, v_{out}) for all $v \in U$ is a cut in H with capacity equal to the capacity of U.

Now from part (a) and the standard Max-Flow/Min-Cut theorem, we have that

$$\text{max-flow}(G) = \text{max-flow}(H) = \text{min-cut}(H) = \text{min-cut}(G)$$

and hence we have the theorem.

4. *Can you hear me now?*

Suppose you have deployed a cellular phone network with n cell towers at points (x_i, y_i) for $i \in \{1, \ldots, n\}$. Also suppose there are m cell phones in use at points (p_i, q_i) for $i \in \{1, \ldots, m\}$. Each cell tower has a range of r, meaning it can connect only to phones within a distance r of the tower. However, the capacity of the towers is limited: each tower can connect to at most k phones at a time. Given this input data, your goal is to determine the largest number of phones that can be connected to the network, and to specify which phone should be connected to which tower to achieve this optimum. Design an algorithm for this problem, prove its correctness, and analyze its running time as a function of n and m.

Solution: We construct a flow network as shown in Figure 1: create a vertex for each cell tower and each mobile phone. Connect a source vertex to each cell tower with an edge of capacity of k and connect all the mobile phones to a sink vertex with an edge of capacity 1. We add an edge from cell tower i to mobile phone j of capacity 1 if and only if the distance between (x_i, y_i) and (p_j, q_j) is at most the range r. For a given pair (i, j), we can tell whether phone j is in range of tower i in constant time, so the graph can be constructed in time $O(mn)$.

We then find a maximum flow in the constructed flow network. Since all the capacities are integers, the Ford-Fulkerson algorithm finds an integral maximum flow. An integer flow naturally defines an assignment of mobile phones to cell towers as follows: assign phone j to tower i if and only if there is unit flow through the edge (i, j). By construction, any feasible assignment defines a feasible flow in the network and vice versa.

Since the constructed flow network has $|V| = O(m+n)$ and $|E| = O(mn)$, and the total capacity of edges leaving the source is kn, the Ford-Fulkerson algorithm finds the maximum flow in time $O(kmn^2)$. (Also note that the total capacity of edges entering the sink is m, so another upper bound on the running time is $O(m^2n)$.) The time to construct the graph is asymptotically

negligible, and it is also straightforward to extract the pairing of phones and towers from the resulting flow in time $O(mn)$, simply by checking which of the edges carry flow.

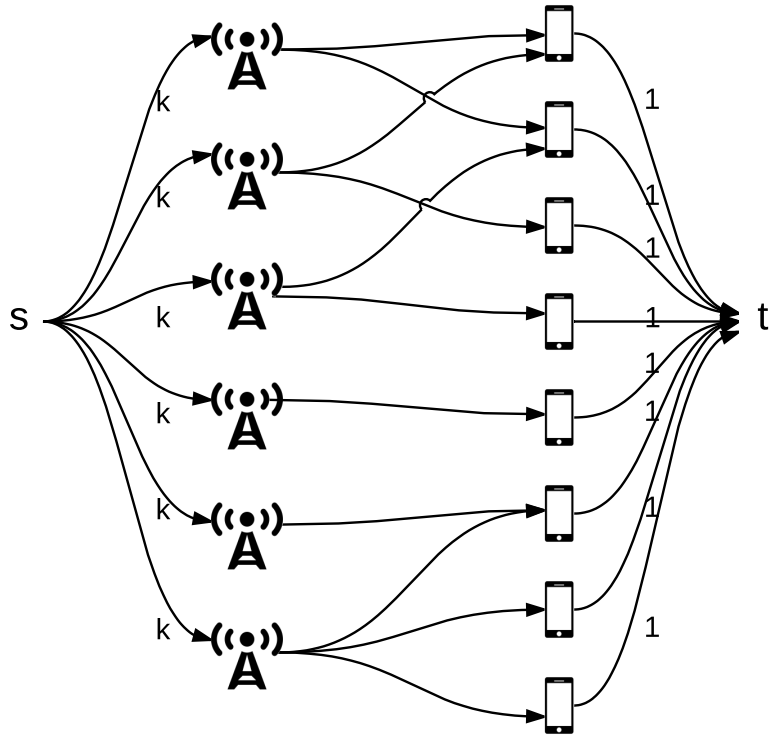

Figure 1: Flow Network Construction for Cellular Phone Network