CSE 3500 Algorithms and Complexity – Fall 2016
Lecture 19: November 1, 2016

Dynamic Programming Continued: Single Source Shortest Paths (SSSP) Problem - General Weights

- Recall that Dijkstra's algorithm for the SSSP problem assumes that there are no negative edges in the graph. When there are negative edges we can employ dynamic programming.

  1. For any node $u \in V - \{s\}$, define $dist^l[u]$ to be the weight of the shortest s to u path from among all the s to u paths that have no more than l edges. Here s is the source node.

     Note that the input for this problem is $dist^1[u]$ for each $u \in V - \{s\}$. We are interested in computing $dist^{n-1}[u]$ for every $u \in V - \{s\}$.

  2. A recurrence relation for $dist^l[u]$ can be derived as follows. If we consider any s to u path that has at most l edges, there are two possibilities: either this path has $\leq (l-1)$ edges or it has l edges. If there are $\leq (l-1)$ edges, then the shortest such path has a weight of $dist^{l-1}[u]$. If this path has l edges, then the weight of the shortest such path is $\min_{w:(w,u)\in E} dist^{l-1}[w] + cost(w,u)$ (where $cost(w,u)$ is the weight of the edge (w,u)). Put together we arrive at:

$$dist^l[u] = \min\left[dist^{l-1}[u], \min_{w:(w,u)\in E} dist^{l-1}[w] + cost(w,u)\right] \quad (1)$$

  3. We can use equation 1 to compute $dist^{n-1}[u]$ starting from $dist^1[u]$ for $u \in V - \{s\}$. The idea is to compute $dist^2[u], dist^3[u], \ldots, dist^{n-1}[u]$. A pseudocode is given below.

```
1) for  u ∈ V − {s} do dist[u] = cost(s, u);
2) for l = 2 to n − 1 do
3)       for u ∈ V − {s} do
4)             for each w such that (w, u) ∈ E do
5)                   dist[u] = min{dist[u], dist[w] + cost(w, u)};
```

     Run Time Analysis: Step 1 takes $O(|V|)$ time. Step 5 takes $O(1)$ time. If d_u is the in-degree of the node u, then the **for** loop of line 4 takes $O(d_u)$ time. As a result, the **for** loop of line 3 takes $O(\sum_{u\in V-\{s\}} d_u) = O(|E|)$ time. The **for** loop of line 2 is iterated $|V| - 2$ times. Thus the run time of the **for** loop of line 2 is $O(|V|\ |E|)$.

     In summary, the total run time of the algorithm is $O(|V|\ |E|)$. This algorithm is known as the Bellman-Ford algorithm.

String Editing Problem (SEP)

- Input for this problem are two strings $X = x_1x_2\cdots x_n$ and $Y = y_1y_2\cdots y_m$ from some alphabet Σ. The goal is to convert X into Y using a sequence of three operations: Insert, Delete, and Exchange. Each operation has an associated cost. The output should be a sequence (known as an *edit sequence*) of these operations that will convert X into Y such that the total cost of all the operations in the sequence is as small as possible. This minimum cost can be thought of as a *distance* between X and Y.

- This problem has numerous applications. For example, in the bioinformatics domain, X and Y could be either DNA sequences or protein sequences. The distance between X and Y as defined above can be used to determine how similar X and Y are.

- **An Example:** Consider the following strings from the alphabet $\Sigma = \{a, b\}$: $X = abbbabaaba$ and $Y = abbbbab$. Assume that each operation has a unit cost.

  One sequence of operations that converts X into Y is: Delete x_1, Delete x_2, ..., Delete x_{10}, Insert y_1, Insert y_2, ..., Insert y_6, and Insert y_7. The total cost is 17.

  Another possible edit sequence is: Delete x_5, Delete x_7, and Delete x_{10}. The total cost here is only 3!

A Dynamic Programming Solution for SEP

1. Define $cost(i, j)$ to be the minimum cost of any edit sequence that converts $x_1x_2\cdots x_i$ into $y_1y_2\cdots y_j$, for any $i = 0, 1, 2, \ldots, n$ and $j = 0, 1, 2, \ldots, m$. Note that we are only interested in the value of this function for $i = n$ and $j = m$.

   We can enumerate some base cases as follows: $cost(0, 0) = 0$. $cost(i, 0) = cost(i - 1, 0) + D(x_i)$, for any $1 \leq i \leq n$, where $D(x_i)$ is the cost for deleting x_i. Also, $cost(0, j) = cost(0, j - 1) + I(y_j)$, for any $1 \leq j \leq m$, where $I(y_j)$ is the cost for inserting y_j.

2. A recurrence relation for $cost(i, j)$ can be obtained by looking at the last operation performed to convert X into Y. The last operation has to be Insert, Delete, or Exchange.

   If the last operation is Insert, it means that we are converting $x_1x_2\cdots x_i$ into $y_1y_2\cdots y_{j-1}$ and inserting y_j at the end. The minimum cost of any such edit sequence will be $cost(i, j - 1) + I(y_j)$.

   If the last operation is Delete, it means that we are converting $x_1x_2\cdots x_{i-1}$ into $y_1y_2\cdots y_j$ and finally deleting x_i. The minimum cost of any such edit sequence will be $cost(i-1, j) + D(x_i)$.

   Finally, if the last operation is Exchange, we are converting $x_1x_2\cdots x_{i-1}$ into $y_1y_2\ldots y_{j-1}$, and exchanging x_i with y_j. The minimum cost of any such edit sequence will be $cost(i - 1, j - 1) + E(x_i, y_j)$, where $E(x_i, y_j)$ is the cost associated with exchanging x_i with y_j.

Putting these observations together, we get:

$$cost(i,j) = \min\{cost(i,j-1)+I(y_j),\ cost(i-1,j)+D(x_i),\ cost(i-1,j-1)+E(x_i,y_j)\} \quad (2)$$

3. Starting from the base cases we can enumerate $cost(i,j)$ in as many points as needed before we can compute $cost(n,m)$. Think of a matrix $M[0:n, 0:m]$ such that $M[i,j] = cost(i,j)$, for $0 \leq i \leq n$ and $0 \leq j \leq m$.

   We can compute the entries of M in a row major order. Note that row 0 and column 0 of M correspond to base cases. $M[i,j]$ depends on $M[i-1,j]$, $M[i,j-1]$ and $M[i-1,j-1]$ (for any $1 \leq i \leq n$ and $1 \leq j \leq m$). If we proceed with the computations in a row major order, when we are ready to compute $M[i,j]$, the three values of M that $M[i,j]$ depends on would already have been computed. As a result, each entry $M[i,j]$ can be computed in $O(1)$ time.

   Therefore, it follows that the run time of the algorithm is $O(mn)$.

Tree Traversal and Graph Search

- Tree traveral refers to visiting the nodes of a tree systematically and possibly performing some operations at each node. Similary, graph search refers to visiting the nodes of an arbitrary graph. Traversals and search are basic tasks performed on trees and graphs, respectively, with many applications.

- There are several ways to traverse a tree. Three popular schemes are in-order, pre-order, and post-order traversals.

- For instance in in-order traversal, we start from the root, recursively traverse the left subtree, visit the root (i.e., perform the required operations at the root), and recursively traverse the right subtree.

- Pseudocodes for the three traversals follow. In these algorithms, if t is a node, then, $t \rightarrow lchild$ and $t \rightarrow rchild$ refer to the left child and the right child of the node t, respectively.

```
In-Order(t)
    In-Order(t → lchild);
    Visit(t);
    In-Order(t → rchild);

Pre-Order(t)
    Visit(t);
    Pre-Order(t → lchild);
```

Pre-Order($t \rightarrow rchild$);

Post-Order(t)
Post-Order($t \rightarrow lchild$);
Post-Order($t \rightarrow rchild$);
Visit(t);

- All of the above algorithms take $O(n)$ time each, where n is the number of nodes in the tree. This can be seen as follows. Each node is looked at at most three times: Once when the algorithm is called on this node, once when the recursive call returns from the left child, and once when the recursive call returns from the right child. Thus the time spent on each node is $O(1)$ and the claim follows.