Notes 11-13

Cameron Chalk

1 Longest Common Subsequence Cont.

Recall the longest common subsequence (LCS) problem: given two strings, find the longest string which is a subsequence of both strings.

E.g.,

- $X =$ a **b** c b **d a b**
- $Y =$ **b d** c **a b** a

An LCS of X and Y is, for example, b d a b. The length of the LCS is four.

Let's try to define subproblems of the LCS problem. First consider two strings shorter than X and Y: X_m, the first m characters of x, and Y_n, the first n characters of Y. We will try to write the length of the LCS is the following way in terms of m and n: $\mathrm{OPT}(m, n)$.

Let Z_k be the LCS of X_m and Y_n.

- If the last character of X_m is the same as the last character of Y_n, then Z_k (the LCS of the two) must end on that character—if it didn't end on that character, then Z_k could be extended by adding that final character, contradicting that Z_k is the LCS). Then we know that Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
- If the last characters of X_m and Y_n are not equal, then Z_k *does not* end on at least one of the two characters. Then Z_k is an LCS of X_{m-1} and Y_n or an LCS of X_m and Y_{n-1}.

With these facts, we can write the length of the LCS of X_m and Y_n recursively as:

$$\mathbf{OPT}(m,n) = \begin{cases} 0 & \text{if } m = 0 \text{ or } n = 0 \\ \\ \mathrm{OPT}(m-1, n-1) + 1 & \text{if last char of } X_m = \text{ last char of } Y_n \\ \\ \max\{\mathrm{OPT}(m-1, n), & \text{otherwise} \\ \quad\quad \mathrm{OPT}(m, n-1)\} & \end{cases}$$

and implementing this algorithm using a table has running time $\mathcal{O}(|X| \cdot |Y|)$.

2 Bellman-Ford Algorithm

We want to compute shortest paths allowing negative edge weights. Previously we saw graphs with negative edge weights are not suitable for Dijkstra's algorithm. Here are some motivations for finding shortest paths in graphs with negative edge weights:

- If looking for longest paths, multiplying all edge weights by -1 (and thus introducing negative weights if there were none initially present) will allow shortest paths algorithms to find the longest path.
- If you have a graph where vertices are currencies, and edges between currencies are directed and have weight equal to the conversion rate for those currencies, then we want to find a path which maximizes the product of the weights along that path. This gives the optimal way to exchange a currency from one to another. To solve this, take the negative log of each edge weight, and then find the shortest path. Note that after taking the negative log, some values may be negative, so we are interested in shortest path on graphs with possibly negative weights.

With the Bellman-Ford algorithm, we want to find the shortest s-t path in a graph which may have negative edge weights. To do this, we must consider negative weight cycles. We can consider the net weight of a cycle as the sum of the weights of the edges in the cycle. If a net weight of a cycle is negative, then the shortest path can be made arbitrarily small by taking that negative weight cycle indefinitely. Thus, if there exists a s-t path with a negative weight cycle, then we define the shortest path length as $-\infty$. Otherwise, if there are no negative weight cycles, then the shortest path is a simple path which does not repeat any nodes.

For Bellman-Ford, we want to define the subproblems of computing a length of a shortest s-t path. There are two ways we can define a subproblem: first, restrict the number of edges in a path. E.g., what is the shortest path using three edges? What is the shortest path using two edges? Second, we can consider the shortest path from a vertex closer to the target; e.g., instead of finding a shortest path from s to t, finding a shortest path from v to t where we know the shortest path from s to t may include v. Then we can write the optimal solution as $\text{OPT}(i, v)$ (the length of the shortest v-t path using $\leq i$ edges).

Since we are looking for a v-t path to find $\text{OPT}(i, v)$, we know that it must include some edge starting from v. However, the shortest path may not have i edges; so the optimal path from v to t may use $i-1$ edges. So to find $\text{OPT}(i, v)$, we will take the min of two options: assuming there are i edges and taking an edge from v which minimizes the shortest path, or assuming there are less than i edges and not taking an edge and "staying" at v. Thus we have the recurrence:

$$\text{OPT}(i, v) = \min\{\text{OPT}(i-1, v), \min_{u \text{ s.t. there is an edge } (v,u)} \{\text{OPT}(i-1, u) + w(v, u)\}\}$$

And including our base cases, we have the full recurrence:

$$\mathrm{OPT}(i,v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = t \\ -\infty & \text{if } i = 0 \text{ and } v \neq t \\ \min\{\mathrm{OPT}(i-1,v), & \text{otherwise} \\ \qquad \min_{u \text{ s.t. there is an edge } (v,u)}\{\mathrm{OPT}(i-1,u) + w(v,u)\}\} & \end{cases}$$

We can construct a table T iterating over values of i and v, letting values of i be the columns and v be the rows. We must compute the column corresponding $i-1$ to compute the column corresponding to i. We can assume i won't exceed n (the number of vertices), since there exists a shortest path which is simple or there exists a negative weight cycle. Thus, T is an $n \times n$ table. When computing an entry $T[i,v]$ in the table, we must look at all edges leaving v. So one may argue in the worst case an entry requires n time to compute, since v may have up to n edges, and there are n^2 entries, so one may think the runtime is $\mathcal{O}(n^3)$.

However, a closer analysis can give a better runtime. Since each row corresponds to a different vertex, the sum of the edges viewed in a column is m, the number of edges in the graph ($2m$ if the graph is undirected, since we look at the edge from either side). Thus one can argue each column requires $\mathcal{O}(m)$ work, and there are n columns total, so the runtime is $\mathcal{O}(n \cdot m)$.