Homework 6

CS41, Spring 2023 — Alina Palacios and Nina Zhuo

1. We see that a path with the probability p^ℓ must have ℓ number of edges such that ℓ is the distance from the start vertex s. Let $L = floor\{log_p q\}$ such that L is the length of the longest possible $s - t$ path and p^L is the lowest possible probability greater than q.

   We create a two-dimensional array $A_1[0...L][1...n]$ where n is number of vertices. A_1 will keep track of the minimum cost of traveling from s to some vertex v with probability p^ℓ for $\ell \in [0...L]$.

   Base Cases $\begin{cases} A_1[0][s] = 0 \text{ because there is no cost at the start vertex} \\ A_1[0][n] = \infty \text{ because there is no cost if no edges were traveled} \\ A_1[1...L][1...n] = \infty \text{ acting as placeholders} \end{cases}$

   Then, we have another two-dimensional array $A_2[0...L][1..m]$ where m is the number of edges. A_2 will keep track of the maximum probability of traveling from s to some vertex v such that the end probability of that path will be p^ℓ for $\ell \in [0...L]$.

   Base Case: $A_2[0][m] = \infty$ because there is no cost if no edges were traveled.

   For all possible ℓ from $[0...L]$, we check each edge m such that:

   i. For each edge e in $[1...m]$, $A_2[\ell][e] = A_1[e-1][e] + c_v$ where e an edge in the set of edges $[1...m]$, and v a vertex in the set of vertices $[1...n]$. The minimum cost of this path will the be minimum cost to travel from s to the second to last vertex $v - 1$ plus the cost to travel to the current vertex v.

   ii. We fill out A_2 using a for loop that will iterate over m starting from 1 such that $A_1[\ell][e] = min(A_1[e][n], A_2[e][m])$. We search through all the edges in $D[\ell]$ to see which one leads to n_2 with the minimum cost. If there is no edge in $D[\ell]$ that leads to a vertex n, then $A_1[e][n]$ will be ∞, meaning that it is unreachable for a certain distance ℓ away from s.

   We return $min(A_1[\ell][t])$ for all $0 \leq \ell \leq L$ giving us the lowest possible cost to reach vertex t from s with a path length of at most L. This will give us running time $O(m \cdot log_p q)$ because $L = log_p q)$ and our for loop will iterate over $[1..m]$ for all ℓ in $[0...L]$.

2. To write this as a DP algorithm, we must take into consideration three cases where potential barrier can be placed relative to the current seat at (i, j):

   Base Case: If $j = 0$, then the cost is whatever the cost is of P[i][j].

   i. A barrier can be placed at $(i - 1, j)$ directly to the left of the current seat .

   ii. A barrier can be placed at $(i, j + 1)$ directly under the current seat.

   iii. A barrier can be placed at $(i + 1, j)$ directly to the right of the current seat.

   We can generalize the algorithm to determine the minimum cost of completing the barrier using the three cases:

$$F(i, j) = min\{F(i - 1, j - 1), F(i, j - 1), F(i + 1, j)\} + F(i, j)$$

This equation represents the three different potential places where seats can be placed to complete the barrier. We add the cost of the current seat P[i][j] to the equation because it is included in the overall cost of the barrier. The algorithm operates as follows: We will apply the recursive characterization stated above in a DP algorithm using nested for loops to iterate over the current seat of the barrier (i, j). This operates in $O(mn)$ time because we are iterating over the rows m and columns n of the grid.

3. Let G be a graph $G = \{V, E\}$. We set $V = (i, j, t)$ where i is the index of the nucleobase string $x[1...n]$, j is the index of the nucleobase string $y[1...n]$, and t is the number of possible consecutive insertions and deletions up to i that is needed so that $x == y$ is true.

   We draw each vertex using a triple nested for-loop representing all possible combinations of i, j, t and draw three edges out from each vertex to represent substitution, insertion, and deletion.

$$E = \begin{cases} (i+1, j+1) & \text{substitution} \\ (i, j+1) & \text{insertion} \\ (i+1, j) & \text{deletion} \end{cases}$$

   We know that $p + kq$ is the cost of starting and continuing a chain of insertions and deletions. Thus, a single insertion or deletion would cost $p + q$ and a continuation of an existing chain of insertions of deletions would cost an additional q. We imagine each edge as a single operation weighted on $D(i, j, t)$ defined as

$$D(i,j,t) = min \begin{cases} D(i-1, j-1, t) + c(x[i], y[j]), \\ D(i-1, j, t) + \begin{cases} p+q \text{ if } t = 0, \\ q \text{ otherwise} \end{cases} \\ D(i, j-1, t) + \begin{cases} p+q \text{ if } t = 0, \\ q \text{ otherwise} \end{cases} \end{cases}$$

   We set the start vertex $s \in V$ as $s = (0, 0)$ and the terminal vertex $t \in V$ as $t = (n, n)$ and use DAG shortest path to find the minimum total weight of an $s - t$ path on G. This is possible in $O(n^3)$ because creating G using a triple nested for-loop takes $O(n^3)$ for $n \cdot n \cdot n$ or n^3 vertices with an 3 outgoing edges from each vertex. When traversing the graph, we only take on of the three possible edges, giving us $O(n^3)$ traversal time for $O(n^3)$ overall.