CSCI 406 Dynamic Programming Review Problem

The Floyd-Warshall Algorithm

1 Problem Description

The Floyd–Warshall algorithm is a dynamic programming method used to compute the shortest-path distances between *every pair of vertices* in a weighted graph. It works for directed or undirected graphs, and it allows positive or negative edge weights (as long as the graph contains no negative cycles).

The key idea is to consider the vertices one by one as possible *intermediate* points on a path. Initially, the algorithm assumes that the shortest known distance from vertex i to vertex j is simply the weight of the direct edge $i \to j$, if such an edge exists. The algorithm then repeatedly improves these distances.

For each vertex k, the algorithm asks:

> Is the shortest path from i to j improved by allowing the path to go through vertex k?

This leads to the fundamental update rule:

$$d[i][j]_k = \min\big(d[i][j]_{k-1},\ d[i][k]_{k-1} + d[k][j]_{k-1}\big).$$

By applying this update for every pair (i, j) and every possible intermediate vertex k, the algorithm systematically considers all possible routes. After all vertices have been considered as intermediates, the value $d[i][j]$ holds the length of the shortest path from i to j.

This results in a time complexity of $O(n^3)$ due to considering every combination of each vertex for i, j, and k. You may recall that other shortest-path algorithms like Djikstra's have a complexity of $O(m + n \log n)$, which when run for each vertex has a complexity of $O(nm + n^2 \log n)$. In sparse graphs where $m \ll n^2$ this takes less time than Floyd-Warshall, but in dense graphs where $m \sim n^2$, the complexity looks more like $O(n^3 + n^2 \log n)$. Thus this implementation of finding the shortest path using Floyd-Warshall can be faster than other algorithms in certain cases.

2 Recurrence Relation

2.1 Notation

- Vertices are numbered $1, 2, ..., n$.
- $w(i, j)$ is the edge weight from a vertex i to a vertex j. In this implementation $w(i, i) = 0$, or in other words, a vertex has an implicit edge weight of 0 with itself. If an edge does not exist from i to j then $w(i, j) = \infty$.
- Let $D(i, j, k)$ represent the shortest distance from i to j considering vertices $\{1, 2, ..., k\}$ as intermediate steps. To solve the full problem, you need to find $D(i, j, n)$ for every combination of $1 \leq i \leq n$ and $1 \leq j \leq n$.

2.2 Recurrence

$$D(i,j,k) = \left\{ \begin{array}{ll} w(i,j) & \text{for } k = 0 \\ \min[D(i,j,k-1), D(i,k,k-1) + D(k,j,k-1)] & \text{for } k > 0 \end{array} \right\}$$

3 Deliverable

Write a dynamic programming algorithm to solve the problem based on the information below. It can be either a top-down or bottom-up implementation depending on your preference or skills:

Input Format:

- The first line contains two space-separated integers: n the number of vertices and m the number of edges.
- m additional lines containing three space-separated integers: u the starting vertex, v the ending vertex, and w the weight of the directed edge from u to v.

Output Format: The output consists of n lines of n space-separated values representing a matrix of the shortest path length from i to j with lengths corresponding to the jth entry of the ith row, and an additional line consisting of `NEGATIVE WEIGHT CYCLE` if a negative weight cycle exists or `NO NEGATIVE WEIGHT CYCLE` if one does not exist. Values of ∞ should be outputted as `INF`.

Example

Input:

```
4 4
1 2 1
2 3 1
3 1 -1
3 4 1
```

Output:

```
0 1 2 3
0 0 1 2
-1 0 0 1
INF INF INF 0
NO NEGATIVE WEIGHT CYCLE
```

4 Programming Hints/Tips

Because of the limited time to complete the assignment, here are some tips that may help you think about how to implement your algorithm:

- When establishing base-cases review the notation and meaning for $w(i, j)$. When $k = 0$, no intermediate vertices are considered, so the only values should be edge weights and ∞. You can use an arbitrarily big value for ∞, though path lengths will never be larger than the sum of all edge weights.
- Depending on how you choose to implement your solution you will need to store all combinations of i, j, and k. One way to do this would be to create an $n \times n$ table for each value of $0 \leq k \leq n$ (for a total of $n + 1$ tables), though there may be more space-efficient options.
- If you choose a bottom-up solution, k should be iterated as the outer loop since the recurrence relation depends upon values of $k - 1$.
- If you choose a top-down solution, remember that you must find all of the values for the combinations of i and j, which will require multiple (n^2) first calls of the recursive function. Make sure to use memoization so you do not have to recalculate values.
- To determine if there is a negative weight cycle think about what happens when updating shortest paths. What does it mean when the shortest path between a vertex and itself is less than 0?