Name: Suhail Basalama | Howework: #10 Barter System | Date: 04/30/2019

Report

Data Structures

1. Vertix object to hold a vertex id and parent pointer.
2. Graph object to represent a graph and it contains the following members and methods:
   Members:
   a. Adjacency matrix (2-D Array of double type) called adj which holds the weights of the edges or infinity if no edge exist between to vertices.
   b. Array of Vertix objects called V
   c. Integer to hold the number of vertices in the graph called v_count.
   Methods:
   a. setEdge(u,v,w) to add an edge to with weight w to the adjacency matrix.
   b. FloyedMarshall algorithm to detect negative weight cycles in the graph which has the following data structures:
      - D (distance) Matrix which is a 2-D matrix of double type to hold the ongoing shortest paths between vertices.
      - P (parent) Matrix which is a 2-D matrix of int type to hold the ongoing parent IDs of each vertix.
3. Line object to hold the data of an edge from the input files with u, v, a, b members, like (u=1, v=2, a=1, b=0.9) for the first line of input1, and a parameterized constructor.
4. In/Out objects to read data from the input files and write data to the output files.
5. ArrayLists and Stacks to help retrieving the path of a cycle.

Algorithms

First, I tried to understand the problem by working on the first and second input files by hand. I noticed that multiplying 'b/a' of the edges of each simple cycle in the graph can determine whether a system is efficient or not. I set the weight of each edge as 'b/a'. If the multiplication of the weights of any cycle gives a result greater than one, then the system is inefficient. However, this approach would be costly and hard to fit for the algorithms that tend to sum up the weight rather than multiply them. Therefore, I figured out that I can use the logarithmic function of 'a/b' as the weight for every edge.

$$if\ \frac{b_1}{a_1}.\frac{b_2}{a_2}.\frac{b_3}{a_3} > 1\ then \log\left(\frac{a_1}{b_1}\right) + \log\left(\frac{a_2}{b_2}\right) + \log\left(\frac{a_3}{b_3}\right) < 0$$

$$\log\left(\frac{b_1}{a_1}.\frac{b_2}{a_2}.\frac{b_3}{a_3}\right) > \log(1) \rightarrow \log\left(\frac{a_1}{b_1}.\frac{a_2}{b_2}.\frac{a_3}{b_3}\right) < 0 \rightarrow \log\left(\frac{a_1}{b_1}\right) + \log\left(\frac{a_2}{b_2}\right) + \log\left(\frac{a_3}{b_3}\right) < 0$$

Using $\log(\frac{a}{b})$ as the weights let us use summation rather than multiplication to find an inefficient cycle. If the sum of the weights of any cycle is negative, then the system is inefficient. This, as a result, turns the problem into a problem of finding a negative-weight cycle in the graph. I started with Bellman Ford algorithm which can detect negative cycles, but my implementation caused my computer to run out of memory, so I decided to use Floyed Warshall algorithm to detect the negative cycles of the graph. The algorithm uses the substructure that a shortest path between two vertices i and j $d(i \text{ to } j)$ is the minimum

of two paths: a path passing through an intermediate vertex $k\ where\ d(i \text{ to } j)$ becomes $d(i \text{ to } k) + d(k \text{ to } j)$ Or a path not passing through vertex k. Using this observation, we can construct a recurrence relationship to get the shortest distance $d_{ij}^{(k)}$ between two vertices i and j, and k as an intermediate vertex.

$$d_{ij}^{(k)} = \begin{cases} w_{ij}, & k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}), & k > 0 \end{cases}$$

In plain English, the algorithm starts by assigning distances to edges that are directly connected where $k = 0$. The next iterations, the algorithm compares the current distances between each pair of vertices, with the distances between those vertices but with vertex k as an intermediate vertex. By the end of the iterations, the algorithm will have compared the distances passing through all vertices and chosen the shortest distances.

The algorithm initializes the D (distance) matrix, where $k = 0$, with the adjacency matrix because the shortest path passing through vertex 0 is simply the weights of all edges. The next iteration represents the distances between each pair of vertices passing through vertex $< 0,1 >$, next iteration, distances passing through vertices $< 0,1,2 >$, at the end, distances passing through vertices $< 0,1,2, \dots, |V| >$. In our problem, the distances initially, where $k = 0$, between each vertex to itself is 0. However, once we start using the recursion relationship, we might find a distance from vertex i to itself passing through some vertex k that is smaller than 0. Once this happens, we know that we found a negative-weight cycle, and we could keep looping to decrease the distance between that vertex with itself infinitely. Therefore, a negative weight cycle can be detected once any value in the diagonal of the D matrix gets below zero. More formally, when $d_{ij}^{(k)} < 0, where\ i == j$. To keep track of the path of the cycle, I used the parent matrix P populated with -1 instead of NIL and initialized for each ($P_{ij}^{(0)} = i, where\ i \neq j$)((u, v) as $v.p = u$). The parent of any vertex gets updated to k if we found that going through vertex k is a shorter path, otherwise, the parent (predecessor) retains its old parent. The formula to populate the P matrix is:

$$P_{ij}^{(k)} = \begin{cases} P_{ij}^{(k-1)}, & if\ d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ P_{kj}^{(k-1)}, & if\ d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

In case no negative-weight cycles are detected, the parent matrix diagonal remains NIL (-1), otherwise, it will hold the predecessor in the cycle. The worst run time of my algorithm is $O(V^3)$ because we have three nested loops running $|V|$ times each. My algorithm may perform better because I added an early stopping condition, which checks for every iteration if $d_{ij}^{(k)} < 0, where\ i == j$, then a negative cycle is detected, and the vertex is returned with the parents assigned properly to get the cycle. The space complexity is $O(V^2)$ because I used the same matrix to update for each k.

Results

The algorithm produces the same results as the expected outputs, with a little variation in the order of edges due to implementation, but that does not affect the correctness of the algorithm since a cycle does not have a starting or ending point. I also tried to modify input5.txt file to check the early stopping property and it does reduce the running time significantly.