Floyd-Warshall Algorithm

Table of Contents

Given an undirected weighted graph G with n vertices. The task is to find the length of the shortest path d_{ij} between each each pair of vertices i and j.

The graph may have negative weight edges, but no negative weight cycles (for then the shortest path is undefined).

This algorithm can also be used to detect the presence of negative cycles. The graph has a negative cycle if at the end of the algorithm, the distance from a vertex v to itself is negative.

This algorithm has been simultaneously published in articles by Robert Floyd and Stephen Warshall in 1962. However, in 1959, Bernard Roy published essentially the same algorithm, but its publication went unnoticed.

Description of the algorithm

The key idea of the algorithm is to partition the process of finding the shortest path between any two vertices to several incremental phases.

Let us number the vertices starting from 1 to n. The matrix of distances is $d[][]$.

Before k-th phase ($k = 1 \dots n$), $d[i][j]$ for any vertices i and j stores the length of the shortest path between the vertex i and vertex j, which contains only the vertices $\{1, 2, \dots, k-1\}$ as internal vertices in the path.

In other words, before k-th phase the value of $d[i][j]$ is equal to the length of the shortest path from vertex i to the vertex j, if this path is allowed to enter only the vertex with numbers smaller than k (the beginning and end of the path are not restricted by this property).

It is easy to make sure that this property holds for the first phase. For $k = 0$, we can fill matrix with $d[i][j] = w_{ij}$ if there exists an edge between i and j with weight w_{ij} and $d[i][j] = \infty$ if there doesn't exist an edge. In practice ∞ will be some high value. As we shall see later, this is a requirement for the algorithm.

Suppose now that we are in the k-th phase, and we want to compute the matrix $d[][]$ so that it meets the requirements for the $(k+1)$-th phase. We have to fix the distances for some vertices pairs (i, j). There are two fundamentally different cases:

- The shortest way from the vertex i to the vertex j with internal vertices from the set $\{1, 2, \dots, k\}$ coincides with the shortest path with internal vertices from the set $\{1, 2, \dots, k-1\}$.

  In this case, $d[i][j]$ will not change during the transition.

- The shortest path with internal vertices from $\{1, 2, \dots, k\}$ is shorter.

  This means that the new, shorter path passes through the vertex k. This means that we can split the shortest

path between i and j into two paths: the path between i and k, and the path between k and j. It is clear that both this paths only use internal vertices of $\{1, 2, \dots, k-1\}$ and are the shortest such paths in that respect. Therefore we already have computed the lengths of those paths before, and we can compute the length of the shortest path between i and j as $d[i][k] + d[k][j]$.

Combining these two cases we find that we can recalculate the length of all pairs (i, j) in the k-th phase in the following way:

$$d_{\text{new}}[i][j] = min(d[i][j], d[i][k] + d[k][j])$$

Thus, all the work that is required in the k-th phase is to iterate over all pairs of vertices and recalculate the length of the shortest path between them. As a result, after the n-th phase, the value $d[i][j]$ in the distance matrix is the length of the shortest path between i and j, or is ∞ if the path between the vertices i and j does not exist.

A last remark - we don't need to create a separate distance matrix $d_{\text{new}}[][]$ for temporarily storing the shortest paths of the k-th phase, i.e. all changes can be

made directly in the matrix $d[][]$ at any phase. In fact at any k-th phase we are at most improving the distance of any path in the distance matrix, hence we cannot worsen the length of the shortest path for any pair of the vertices that are to be processed in the $(k+1)$-th phase or later.

The time complexity of this algorithm is obviously $O(n^3)$.

Implementation

Let $d[][]$ is a 2D array of size $n \times n$, which is filled according to the 0-th phase as explained earlier. Also we will set $d[i][i] = 0$ for any i at the 0-th phase.

Then the algorithm is implemented as follows:

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            d[i][j] = min(d[i][j], d[i][k] + d
        }
    }
}
```

It is assumed that if there is no edge between any two vertices i and j, then the matrix at $d[i][j]$ contains a large number (large enough so that it is greater than the length of any path in this graph). Then this edge will always be unprofitable to take, and the algorithm will work correctly.

However if there are negative weight edges in the graph, special measures have to be taken. Otherwise the resulting values in matrix may be of the form $\infty - 1$, $\infty - 2$, etc., which, of course, still indicates that between the respective vertices doesn't exist a path. Therefore, if the graph has negative weight edges, it is better to write the Floyd-Warshall algorithm in the following way, so that it does not perform transitions using paths that don't exist.

```
for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            if (d[i][k] < INF && d[k][j] < INF
                d[i][j] = min(d[i][j], d[i][k]
        }
    }
}
```

Retrieving the sequence of vertices in the shortest path

It is easy to maintain additional information with which it will be possible to retrieve the shortest path between any two given vertices in the form of a sequence of vertices.

For this, in addition to the distance matrix $d[][]$, a matrix of ancestors $p[][]$ must be maintained, which will contain the number of the phase where the shortest distance between two vertices was last modified. It is clear that the number of the phase is nothing more than a vertex in the middle of the desired shortest path. Now we just need to find the shortest path between vertices i and $p[i][j]$, and between $p[i][j]$ and j. This leads to a simple recursive reconstruction algorithm of the shortest path.

The case of real weights

If the weights of the edges are not integer but real, it is necessary to take the errors, which occur when working with float types, into account.

The Floyd-Warshall algorithm has the unpleasant effect, that the errors accumulate very quickly. In fact if there is an error in the first phase of δ, this error may propagate to the second iteration as 2δ, to the third iteration as 4δ, and so on.

To avoid this the algorithm can be modified to take the error (EPS = δ) into account by using following comparison:

```
if (d[i][k] + d[k][j] < d[i][j] - EPS)
    d[i][j] = d[i][k] + d[k][j];
```

The case of negative cycles

Formally the Floyd-Warshall algorithm does not apply to graphs containing negative weight cycle(s). But for all pairs of vertices i and j for which there doesn't exist a path starting at i, visiting a negative cycle, and end at j, the algorithm will still work correctly.

For the pair of vertices for which the answer does not exist (due to the presence of a negative cycle in the path between them), the Floyd algorithm will store any

number (perhaps highly negative, but not necessarily) in the distance matrix. However it is possible to improve the Floyd-Warshall algorithm, so that it carefully treats such pairs of vertices, and outputs them, for example as $-\text{INF}$.

This can be done in the following way: let us run the usual Floyd-Warshall algorithm for a given graph. Then a shortest path between vertices i and j does not exist, if and only if, there is a vertex t that is reachable from i and also from j, for which $d[t][t] < 0$.

In addition, when using the Floyd-Warshall algorithm for graphs with negative cycles, we should keep in mind that situations may arise in which distances can get exponentially fast into the negative. Therefore integer overflow must be handled by limiting the minimal distance by some value (e.g. $-\text{INF}$).

To learn more about finding negative cycles in a graph, see the separate article Finding a negative cycle in the graph.

Practice Problems

- UVA: Page Hopping
- SPOJ: Possible Friends
- CODEFORCES: Greg and Graph
- SPOJ: CHICAGO - 106 miles to Chicago
- UVA 10724 - Road Construction
- UVA 117 - The Postal Worker Rings Once
- Codeforces - Traveling Graph
- UVA - 1198 - The Geodetic Set Problem
- UVA - 10048 - Audiophobia
- UVA - 125 - Numbering Paths
- LOJ - Travel Company
- UVA 423 - MPI Maelstrom
- UVA 1416 - Warfare And Logistics
- UVA 1233 - USHER
- UVA 10793 - The Orc Attack
- UVA 10099 The Tourist Guide
- UVA 869 - Airline Comparison
- UVA 13211 - Geonosis
- SPOJ - Defend the Rohan
- Codeforces - Roads in Berland
- Codeforces - String Problem
- GYM - Manic Moving (C)
- SPOJ - Arbitrage
- UVA - 12179 - Randomly-priced Tickets
- LOJ - 1086 - Jogging Trails

- SPOJ - Ingredients