Floyd's Algorithm-Uorshella finding shortest paths between all pairs of vertices

Given a directed or undirected weighted graph $G$with $n$vertices. Required to find the values of all variables d_{ij}- length of the shortest path from vertex $i$to vertex j.

It is assumed that the graph contains no cycles of negative weight (then answer between some pairs of vertices may simply not exist - it will be infinitely small).

This algorithm has been simultaneously published in the papers of Robert Floyd (Robert Floyd) and Stephen Uorshella (Warshall) (Stephen Warshall) in 1962, on behalf of which this algorithm is called nowadays. However, in 1959, Bernard Roy (Bernard Roy) published essentially the same algorithm, but its publication unnoticed.

Description of the algorithm

The key idea of the algorithm - the process of finding a partition of the shortest paths on the **phase** .

Before $k$the second phase ($k = 1 \dots n$) considered that in the matrix of distances $d[][]$ stored length of the shortest paths, which contain as internal vertices only vertices in the set $\{1, 2, \dots, k-1\}$(we number the vertices of the graph, starting with the unit).

In other words, before $k$the second phase of the quantity $d[i][j]$equal to the length of the shortest path from vertex $i$to vertex j, if allowed to enter this path only tops with numbers below k(beginning and end of the path are not considered).

Easy to see that this property is to perform for the first phase, it is enough to distance matrix $d[][]$to record the adjacency matrix: $d[i][j] = g[i][j]$- the cost of edges from vertex $i$to vertex j. Thus, if the edges between any vertices not, should write the value of "infinity" ∞. From the vertex to itself should always record the magnitude 0, it is critical for the algorithm.

Suppose now that we are on the $k$second phase, and we want to **recalculate the** matrix $d[][]$so that it meets the requirements is for $k+1$the second phase. Fix some vertices i and j. We there are two fundamentally different cases:

- The shortest path from vertex $i$to vertex j, which is allowed to pass through the additional peaks $\{1, 2, \dots, k\}$, **coincides** with the shortest route, which is allowed to pass through the vertices of the set $\{1, 2, \dots, k-1\}$.

- In this case, the value $d[i][j]$does not change during the transition from $k$the second to the $k + 1$-th phase.
- "New" was the shortest way **better** "old" way.
- This means that the "new" shortest path passes through the vertex k. Just note that we do not lose generality by considering only the more simple way (ie paths that do not pass on some vertex twice).
- Then we note that if we break this "new" way vertex $k$into two halves (one running $i \Rightarrow k$and the other - $k \Rightarrow j$), each of these halves is not comes to the top k. But then it turns out that the length of each of these halves was deemed a further $k - 1$second phase or even earlier, and it is sufficient to simply take the sum $d[i][k] + d[k][j]$, she will give the length of the "new" shortest path.

Combining these two cases, we find that $k$the second phase is required to recalculate the length of the shortest paths between all pairs of vertices i, and $j$as follows:

```
new_d[i][j] = min (d[i][j], d[i][k] + d[k][j]);
```

Thus, all the work that is required to produce $k$the second phase - is to iterate over all pairs of vertices and recalculate the length of the shortest path between them. As a result, after the $n$second phase in a matrix of distances $d[i][j]$will be recorded between the length of the shortest path $i$and j, either ∞if the path between the nodes exist.

The last remark should be done - something that can **not create a separate matrix** $\text{new\_d}[][]$ for temporary array of shortest paths on $k$the second phase, all changes can be made directly in the matrix $d[][]$. In fact, if we have improved (reduced) a value in the matrix of distances, we could not degrade thereby length of the shortest path for some other pairs of vertices processed later.

Asymptotics algorithm obviously is $O(n^3)$.

Implementation

Fed to the input of the program graph specified as the adjacency matrix - a two-dimensional array of $d[][]$size $n \times n$, in which each element specifies the length of the edges between the vertices.

Required to satisfy $d[i][i] = 0$for any i.

```
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[k][j]);
```

It is assumed that if some vertices between two **ribs have** , in the adjacency matrix was recorded for a large number (large enough that it is greater than the length of any path in this graph), then this edge is always disadvantageous to take and the algorithm work correctly.

However, if you do not take special measures, in the presence of edges in the graph **of negative weight** , resulting in a matrix of the form may appear $\infty - 1, \infty - 2$and so on, which, of course, still means that between the corresponding vertices in general there is no way. Therefore, in the presence of negative edges in the graph algorithm Floyd better write it so that it did not fulfill the transitions of the states that already is "no way":

```
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] + d[k][j]);
```

Recovery paths themselves

Easy to maintain additional information - so-called "ancestors", which can restore itself the shortest path between any two given vertices **in a sequence of vertices** .

It's enough apart distance matrix $d[][]$also support **matrix ancestors** $p[][]$ that for every pair of vertices will contain the number of phases, which was obtained by the shortest distance between them. It is clear that this phase number is not more than the "average" tip Seeking the shortest path, and now we just need to find the shortest path between the vertices $i$and $p[i][j]$and between $p[i][j]$and j. This yields a simple recursive algorithm for reconstructing the shortest path.

Case of real weights

If the weights of edges are not integers and real numbers, it should be borne errors that inevitably arise when dealing with floating-point types.

Floyd's algorithm applied to unpleasant special effect of these errors becomes that found distance algorithm may take much less due to **accumulated errors** . In fact, if the first phase of the error took place Δ, on the second iteration of this error can already turn into 2Δ, the third - in 4Δ, and so on.

To avoid this, the comparison algorithm Floyd should be done taking into account the error:

```
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

If the graph contains a cycle of negative weight, then formally Uorshella-Floyd algorithm is not applicable to such graph.

In fact, for those pairs of vertices $i$and $j$between which it is impossible to go into a cycle of negative weight, the algorithm will work correctly.

For those pairs of vertices, the answer to which does not exist (because of the presence of a negative cycle in the path between them), Floyd's algorithm to find an answer to a number of (possibly highly negative, but not necessarily). Nevertheless, it is possible to improve the algorithm of Floyd, he carefully processed to such a pair of vertices, and drew for them, for example $-\infty$.

This can be done, for example, the following **criterion** "is not the existence of the way." So, even for a given graph worked usual Floyd algorithm. Then between the vertices i and $j$the shortest path does not exist if and only if there is a vertex t, accessible from i, and from which is achievable $j$for which you are $d[t][t] < 0$.

In addition, when using the Floyd algorithm for graphs with negative cycles should remember that arise in the process of distance can go into much less exponentially with each phase. Therefore, measures should be taken against an integer overflow, limiting all distances below some value (eg $-\mathrm{INF}$).

More information about this task, see separate article: **"Finding a negative cycle in the graph"** .