Bellman-Ford algorithm

Let a directed weighted graph $G$with $n$vertices and $m$edges, and contains a vertex v. Required to find **the length of the shortest paths** from vertex $v$to all other vertices.

Unlike Dijkstra's algorithm , this algorithm can also be applied to graphs containing edges of negative weight.However, if the graph contains a negative cycle, then, of course, the shortest path to some of the vertices may not exist (due to the fact that the weight of the shortest path must be equal to minus infinity), however, this algorithm can be modified to signal the presence of a negative cycle weight, or even bringing the cycle itself.

The algorithm is named after two American scientists: Richard **Bellman** (Richard Bellman) and Leicester **Ford**(Lester Ford). Ford actually invented this algorithm in 1956 in the study of other mathematical problem, which has been reduced to the subproblem of finding the shortest path in the graph, and Ford gave a sketch of the algorithm to solve this problem. Bellman in 1958 published an article devoted specifically to the problem of finding the shortest path, and in this article he articulated algorithm in the form in which we know it now.

Description of the algorithm

We believe that the graph does not contain a cycle of negative weight. Case of having a negative cycle will be discussed below in a separate section.

Head array of distances $d[0 \dots n-1]$that after working algorithm will contain the answer to the problem. In early work, we fill it as follows: $d[v] = 0$, and all other elements $d[]$equal to infinity ∞.

The algorithm of Bellman-Ford is a few phases. Each phase scans all edges, and the algorithm tries to produce **relaxation** (relax, attenuation) along each edge (a, b)cost c. Relaxation along the edge - is an attempt to improve the value of $d[b]$value $d[a] + c$. In fact, this means that we try to improve response for the top b, using the edge $(a, b)$and the current response for the top a.

Argues that enough $n-1$phase algorithm to correctly calculate the lengths of the shortest paths in the graph (again, we believe that no cycles of negative weight). For inaccessible peaks distance $d[]$remains equal to infinity ∞.

Implementation

Algorithm Bellman-Ford, unlike many other graph algorithms, more convenient to represent the graph as a list of all the edges (rather than $n$lists of edges - the edges of each vertex). In the above implementation Starts data structure $\mathrm{edge}$for an edge. The input data for the algorithm are the number n, m, a list $e$of edges and the number of the starting vertex v. All rooms are numbered vertices 0on $n - 1$.

The simplest implementation

The constant INFindicates the number of "infinity" - it must be chosen so that it is clearly superior to all of the possible path lengths.

```
struct edge {
	int a, b, cost;
};

int n, m, v;
vector<edge> e;
const int INF = 1000000000;

void solve() {
	vector<int> d (n, INF);
	d[v] = 0;
	for (int i=0; i<n-1; ++i)
		for (int j=0; j<m; ++j)
			if (d[e[j].a] < INF)
				d[e[j].b] = min (d[e[j].b], d[e[j].a] +
e[j].cost);
	// вывод d, например, на экран
}
```

Check "if (d [e [j]. A] <INF)" is needed only if the graph contains edges of negative weight: without such checks would be a relaxation of the vertices to which the path has not yet been found, and the distance would appear incorrect form $\infty - 1, \infty - 2$etc.

An improved

This algorithm can speed up a few: often the answer is already in several phases, and the remaining phases of any useful work does not happen, only wasted scans all edges. Therefore, we will keep the flag that something has changed in the current phase or not, and if at some stage, nothing happened, then the algorithm can be stopped. (This optimization does not improve the asymptotic behavior, ie, on some graphs will still need all the $n - 1$phase, but significantly accelerates the behavior of the algorithm "on average", ie random graphs.)

With this enhancement becomes generally unnecessary to manually limit the number of phases of the algorithm number $n - 1$- he stops after the desired number of phases.

```
void solve() {
	vector<int> d (n, INF);
	d[v] = 0;
	for (;;) {
		bool any = false;
		for (int j=0; j<m; ++j)
			if (d[e[j].a] < INF)
				if (d[e[j].b] > d[e[j].a] + e[j].cost) {
					d[e[j].b] = d[e[j].a] + e[j].cost;
					any = true;
				}
		if (!any)  break;
	}
	// вывод d, например, на экран
}
```

Restoration paths

Let us now consider how we can modify the algorithm of Bellman-Ford so that he not only found the length of the shortest paths, but also allows you to restore themselves shortest paths.

To do this,'s have another array $p[0 \dots n - 1]$in which each vertex will keep its "ancestor", ie penultimate vertex in the shortest path leading to it. In fact, the shortest path to some vertex $a$is the shortest path to some vertex $p[a]$, which attributed to the end of the summit a.

Note that the algorithm of Bellman-Ford is working on the same logic: it is, assuming that the shortest distance to a vertex already counted, trying to improve the shortest distance to another vertex. Consequently, at the moment we need to improve in just remember $p[]$, from what tops this improvement occurred.

We present the implementation of the Bellman-Ford with the restoration of some ways to a given node t:

```
void solve() {
	vector<int> d (n, INF);
	d[v] = 0;
	vector<int> p (n, -1);
	for (;;) {
```

```
            bool any = false;
            for (int j=0; j<m; ++j)
                if (d[e[j].a] < INF)
                    if (d[e[j].b] > d[e[j].a] + e[j].cost) {
                        d[e[j].b] = d[e[j].a] + e[j].cost;
                        p[e[j].b] = e[j].a;
                        any = true;
                    }
            if (!any)  break;
        }

        if (d[t] == INF)
            cout << "No path from " << v << " to " << t << ".";
        else {
            vector<int> path;
            for (int cur=t; cur!=-1; cur=p[cur])
                path.push_back (cur);
            reverse (path.begin(), path.end());

            cout << "Path from " << v << " to " << t << ": ";
            for (size_t i=0; i<path.size(); ++i)
                cout << path[i] << ' ';
        }
}
```

Here we first passed by the ancestors, starting from the top t, and keep all the traversed path in the list path. This list is obtained from the shortest path $v$to t, but in reverse order, so we call reversehim and then draw.

Proof of the algorithm

First, we note immediately that unattainable of $v$vertices algorithm will work correctly: they label $d[]$will remain equal to infinity (since Bellman-Ford algorithm to find some way to all reachable from the $s$vertices and relaxation at all other vertices not occur even once).

We now prove the following **statement** : After a $i$phase-Bellman Ford algorithm correctly finds all the shortest path length (number of edges) does not exceed i.

In other words, every vertex $a$is denoted by $k$the number of edges in the shortest path to it (if there are several ways you can take any). Then this statement says that after k this phase the shortest path will be found guaranteed.

Proof . Consider an arbitrary vertex $a$to which there is a path from the starting vertex v , and consider the shortest path to it: $(p_0 = v, p_1, \ldots, p_k = a)$. Before the first phase of the shortest path to the top $p_0 = v$is found correctly. During the first phase of the edge (p_0, p_1)has been viewed Bellman-Ford algorithm, so that the distance to the summit $p_1$was correctly counted after the first phase. Repeating these statements k again, we see that after $k$the second phase of the distance to the vertex $p_k = a$ counted correctly, as required.

The last thing to note - is that any shortest path can not have more $n - 1$ribs. Consequently, the algorithm is sufficient to make only $n - 1$phase. After that, no relaxation guaranteed improvement can not be completed until some vertex distance.

Case of a negative cycle

Above all, we felt that a negative cycle in the graph does not contain (more precise, we are interested in a negative cycle reachable from the starting vertex $v$and unreachable cycles anything in the above algorithm does not change). If present, there are additional complexities associated with the fact that the distances to all the peaks in the same cycle as well as the distance to the reachable vertices of this cycle is not defined - they should be equal to minus infinity.

It is easy to understand that the algorithm Ford-Bellman can **do infinitely relaxation** of all the vertices of this cycle and the vertices reachable from it. Therefore, if the number of phases is not to limit the number $n - 1$, the algorithm will operate indefinitely, continuously improving the distances to these vertices.

Hence we obtain **a criterion for the achievable cycle of negative weight** : if the $n - 1$phase we perform another phase, and it happens at least one relaxation, then the graph contains a cycle of negative weight, of attainable v, otherwise, there is no such cycle.

Moreover, if such a cycle is detected, the Bellman-Ford algorithm can be modified so that it is deduced that the cycle itself as a sequence of vertices contained in it. It is enough to remember the number of vertices $x$in which there was a relaxation on $n$the second phase. This summit will either lie on a cycle of negative weight, or it can be reached from it. To obtain the peak which lies on the ring are guaranteed sufficiently, for example, $n$just once ancestor starting from the top x.Get a room $y$vertices lying on a cycle, it is necessary to go from the top of this ancestor, until we return to the same vertex y(and it will happen, because the relaxation cycle of negative weight occur in a circle).

Implementation:

```
void solve() {
	vector<int> d (n, INF);
	d[v] = 0;
	vector<int> p (n, -1);
	int x;
	for (int i=0; i<n; ++i) {
		x = -1;
		for (int j=0; j<m; ++j)
			if (d[e[j].a] < INF)
				if (d[e[j].b] > d[e[j].a] + e[j].cost) {
					d[e[j].b] = max (-INF, d[e[j].a] +
e[j].cost);
					p[e[j].b] = e[j].a;
					x = e[j].b;
				}
	}

	if (x == -1)
		cout << "No negative cycle from " << v;
	else {
		int y = x;
		for (int i=0; i<n; ++i)
			y = p[y];

		vector<int> path;
		for (int cur=y; ; cur=p[cur]) {
			path.push_back (cur);
			if (cur == y && path.size() > 1)  break;
		}
		reverse (path.begin(), path.end());

		cout << "Negative cycle: ";
		for (size_t i=0; i<path.size(); ++i)
			cout << path[i] << ' ';
	}
}
```

Because when there is a negative cycle for $n$iterations distance could go far in the minus (apparently negative numbers to order -2^n), the code has taken additional measures against such an integer overflow:

```
d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);
```

In the above implementation is sought negative cycle reachable from some starting vertex v, but the algorithm can be modified so that it was looking for just **any negative cycle** in the graph. To do this, we must put all distances $d[i]$ equal to zero and not infinity - as if we are looking for the shortest path from all vertices simultaneously for correct detection of the negative cycle is not affected.

Extras on this task - see separate article **"Search negative cycle in the graph"** .

Problem in online judges

List of tasks that can be solved using the algorithm of Bellman-Ford:

- E-OLIMP # 1453 **"Ford-Bellman"** [Difficulty: Easy]
- UVA # 423 **"MPI Maelstrom"** [Difficulty: Easy]
- UVA # 534 **"Frogger"** [Difficulty: Medium]
- UVA # 10099 **"The Tourist Guide"** [Difficulty: Medium]
- UVA # 515 **"King"** [Difficulty: Medium]

See also the list of tasks in the article **"Search negative cycle"** .