Leviticus algorithm finding the shortest paths from a given vertex to all other vertices

Let a graph with N vertices and M edges, each of which lists its weight L_I. Also given starting vertex V_0. Required to find the shortest path from vertex V_0 to all other vertices.

Leviticus algorithm solves this problem very efficiently (about asymptotics and speed, see below).

Description

Let array D [1 .. N] will contain the current shortest path lengths, ie D_I - this is the current length of the shortest path from vertex V_0 to vertex i. Initially D array is filled with values "infinity", except $D_{V0} = 0$. Upon completion of the algorithm, this array will contain the final shortest distance.

Let array P [1 .. N] contains the current ancestors, ie P_I - is the peak preceding the vertex i in the shortest path from vertex V_0 to i. As well as an array D, array P is gradually changed in the course of the algorithm, and it receives the end of the final values.

Now, actually, the algorithm Leviticus. At each step, supported by three sets of vertices:

- M_0 - peaks, the distance to which has already been calculated (but perhaps not completely);
- M_1 - vertex distance are calculated;
- M_2 - vertex distance are not yet calculated.

Vertices in the set M_1 is stored as a bidirectional queue (deque).

Initially all vertices are placed in the set M_2, apart from the vertex V_0, which is placed in the set M_1.

At each step of the algorithm, we take the top of the set M_1 (pull out the top element of the queue). Let V - is the selected vertex. Translate this vertex in the set M_0. Then review all edges emanating from that vertex. Let T - is the second end of the current edge (ie not equal to V), and L - is the length of the current edge.

- If T belongs to M_2, then T is transferred to a set of M_1 to the end of the queue. D_T D is set equal to $_V$ + L.

- If T belongs to M_1, we try to improve the value of D_T: $D_T = \min(D_T, D_V + L)$. The very top of T does not move in the queue.
- If T belongs to M_0, and if D_T can be improved ($D_T > D_V + L$), then improving D_T, and T return to the top of the set M_1, placing it in the top of the queue.

Of course, whenever you update the array D must be updated and the value in the array P.

Implementation Details

Create an array ID [1 .. N], in which each vertex will be stored, what set it belongs: 0 - if M_2 (ie, the distance is equal to infinity), 1 - if M_1 (ie, a vertex is queue), and 2 - when M_0 (a path has been found, the distance is less than infinity).

Queue processing can implement standard data structure deque. However, there is a more efficient way. First, obviously, in the queue at any one time will be a maximum of N elements stored. But, secondly, we can add elements in the beginning and the end of the queue. Consequently, we can arrange a place on the array size N, but we need to loop it. Ie do array Q [1 .. N], pointers (int) to the first element QH and the element after the last QT. The queue is empty when QH == QT. Adding to the end - just a record in Q [QT] QT and increase by 1, if the QT then went beyond the line (QT == N), then do the QT = 0. Adding the queue - reduce QH-1, if it has moved beyond the stage of (QH == -1), then do the QH = N-1.

The algorithm implementing exactly the description above.

Asymptotics

I do not know more or less good asymptotic estimate of this algorithm. I have met only estimate O (NM) of the similar algorithm.

However, in practice, the algorithm has proven itself very well: while I appreciate his work as **O (Log M N)** , although, again, this is purely **experimental**evaluation.

Implementation

```
typedef pair <int,int> rib;
typedef vector <vector <rib>> graph;

const int inf = 1000 * 1000 * 1000;


int main ()
{
```

```
	int n, v1, v2;
	graph g (n);

	... Reading the graph ...

	vector <int> d (n, inf);
	d [v1] = 0;
	vector <int> id (n);
	deque <int> q;
	q.push_back (v1);
	vector <int> p (n, -1);

	while (! q.empty ())
	{
		int v = q.front (), q.pop_front ();
		id [v] = 1;
		for (size_t i = 0; i <g [v]. size (); + + i)
		{
			int to = g [v] [i]. first, len = g [v] [i]. second;
			if (d [to]> d [v] + len)
			{
				d [to] = d [v] + len;
				if (id [to] == 0)
					q.push_back (to);
				else if (id [to] == 1)
					q.push_front (to);
				p [to] = v;
				id [to] = 1;
			}
		}
	}

	... result output ...

}
```