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

Given a graph with N vertices and M edges, each of which indicated its weight L i . Also, given the starting vertex V 0 .Required to find the shortest path from vertex V 0 to all other nodes.

Leviticus algorithm solves this problem very efficiently (about the asymptotic behavior and speed of work. See below.)

**Description**

Let an 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 the vertex i. Initially D is filled with an array of values ​​"infinity" but D V 0 = 0. At the end of the algorithm, this array will contain the final shortest distance.

Let an array P [1..N] contains current ancestors, ie P i - is the pinnacle of preceding vertex i in the shortest path from vertex V 0 to i. As array D, array P is changed gradually during the end of the algorithm and its final value takes.

Now actually the algorithm Leviticus. Each step is supported by three sets of vertices:

* M 0 - vertex distance which has already been calculated (but perhaps not entirely);
* M 1 - vertex distance are calculated;
* M 2 - vertex distance are not yet computed.

The vertices in the set M 1 is stored in the form of a bi-directional queue (deque).

Initially, all placed in the top of the set M 2 , except for the top of the 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 (We reach the top element of the queue). Let V - is the selected vertex. Translate this summit in a variety of M0 . Then view all edges emanating from that vertex. Let T - this is the second end of this rib (i.e., not equal to V), and L - the length of this edge.

* If T belongs to M 2 , then T is transferred to a set of M 1 at the end of the queue. D T D is set equal to V + L.
* If T belongs to M 1 , then 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 a queue.
* If T belongs to M 0 , and if D T can be improved (D T > D V + L), improving the 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 should be updated and the value in the array P.

**Implementation Details**

Create an array of ID [1..N], in which each vertex will store, which set it belongs: 0 - if the M 2 (ie, the distance is infinite), 1 - if the M 1 (ie the vertex is queue), and 2 - when M 0 (a path has already 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, is pending at any given time will be stored a maximum of N elements. But secondly, we can add elements in the beginning and the end of the queue. Therefore, we can arrange a place on the array size N, but you have to loop it. Ie do array Q [1..N], pointers (int) to the first element QH and after the last element in the QT. The queue is empty when QH == QT. Adding to the end - a record in the Q [QT] and increased QT 1; If QT then went beyond the line (QT == N), then we do QT = 0. The addition of the queue - reduce QH 1, if it has moved beyond the stage (QH == -1), then do QH = N -1.

The algorithm realize exactly according to the description above.

**Asymptotics**

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

However, in practice the algorithm has proven itself very well: while it is running, I appreciate both **O (M log N)** , although, again, this is only **an experimental** evaluation.

**Realization**

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);

Count ... read ...

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;

}

}

}

... Derivation of the result ...

}