D´Esopo-Pape algorithm

Table of Contents

Given a graph with n vertices and m edges with weights w_i and a starting vertex v_0. The task is to find the shortest path from the vertex v_0 to every other vertex.

The algorithm from D´Esopo-Pape will work faster than Dijkstra's algorithm and the Bellman-Ford algorithm in most cases, and will also work for negative edges. However not for negative cycles.

Description

Let the array d contain the shortest path lengths, i.e. d_i is the current length of the shortest path from the vertex v_0 to the vertex i. Initially this array is filled with infinity for every vertex, except $d_{v_0} = 0$. After the algorithm finishes, this array will contain the shortest distances.

Let the array p contain the current ancestors, i.e. p_i is the direct anchestor of the vertex i on the current shortest path from v_0 to i. Just like the array d, the array

p changes gradually during the algorithm and at the end takes its final values.

Now to the algorithm. At each step three sets of vertices are maintained:

- M_0 - vertices, for which the distance has already been calculated (although it might not be the final distance)
- M_1 - vertices, for which the distance currently is calculated
- M_2 - vertices, for which the distance has not yet been calculated

The vertices in the set M_1 are stored in a bidirectional queue (deque).

At each step of the algorithm we take a vertex from the set M_1 (from the front of the queue). Let u be the selected vertex. We put this vertex u into the set M_0. Then we iterate over all edges coming out of this vertex. Let v be the second end of the current edge, and w its weight.

- If v belongs to M_2, then v is inserted into the set M_1 by inserting it at the back of the queue. d_v is set to $d_u + w$.
- If v belongs to M_1, then we try to improve the value of d_v: $d_v = \min(d_v, d_u + w)$. Since v is already in M_1, we don't need to insert it into M_1 and the queue.
- If v belongs to M_1, and if d_v can be improved $d_v > d_u + w$, then we improve d_v and insert the

vertex v back to the set M_1, placing it at the beginning of the queue.

And of course, with each update in the array d we also have to update the corresponding element in the array p.

Implementation

We will use an array m to store in which set each vertex is currently.

```
struct Edge {
    int to, w;
};

int n;
vector<vector<Edge>> adj;

const int INF = 1e9;

void shortest_paths(int v0, vector<int>& d, ve
    d.assign(n, INF);
    d[v0] = 0;
    vector<int> m(n, 2);
    deque<int> q;
    q.push_back(v0);
    p.assign(n, -1);

    while (!q.empty()) {
        int u = q.front();
        q.pop_front();
        m[u] = 0;
```

```
        for (Edge e : adj[u]) {
            if (d[e.to] > d[u] + e.w) {
                d[e.to] = d[u] + e.w;
                p[e.to] = u;
                if (m[e.to] == 2) {
                    m[e.to] = 1;
                    q.push_back(e.to);
                } else if (m[e.to] == 0) {
                    m[e.to] = 1;
                    q.push_front(e.to);
                }
            }
        }
    }
}
```

Complexity

The algorithm performs usually quite fast. In most cases even faster than Dijkstra's algorithm. However there exist cases for which the algorithm takes exponential time.