Last update: September 24, 2023 Translated From: e-maxx.ru

Dijkstra Algorithm

You are given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s. This article discusses finding the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves.

This problem is also called **single-source shortest paths problem**.

Algorithm

Here is an algorithm described by the Dutch computer scientist Edsger W. Dijkstra in 1959.

Let's create an array $d[]$ where for each vertex v we store the current length of the shortest path from s to v in $d[v]$. Initially $d[s] = 0$, and for all other vertices this length equals infinity. In the implementation a sufficiently large number (which is guaranteed to be greater than any possible path length) is chosen as infinity.

$$d[v] = \infty,~ v \ne s$$

In addition, we maintain a Boolean array $u[]$ which stores for each vertex v whether it's marked. Initially all vertices are unmarked:

$$u[v] = {\rm false}$$

The Dijkstra's algorithm runs for n iterations. At each iteration a vertex v is chosen as unmarked vertex which has the least value $d[v]$:

Evidently, in the first iteration the starting vertex s will be selected.

The selected vertex v is marked. Next, from vertex v **relaxations** are performed: all edges of the form (v,to) are considered, and for each vertex to the algorithm tries to improve the value $d[\text{to}]$. If the length of the current edge equals len, the code for relaxation is:

$$d[\text{to}] = \min (d[\text{to}], d[v] + len)$$

After all such edges are considered, the current iteration ends. Finally, after n iterations, all vertices will be marked, and the algorithm terminates. We claim that the found values $d[v]$ are the lengths of shortest paths from s to all vertices v.

Note that if some vertices are unreachable from the starting vertex s, the values $d[v]$ for them will remain infinite. Obviously, the last few iterations of the algorithm will choose those vertices, but no useful work will be done for them. Therefore, the algorithm can be stopped as soon as the selected vertex has infinite distance to it.

Restoring Shortest Paths

Usually one needs to know not only the lengths of shortest paths but also the shortest paths themselves. Let's see how to maintain sufficient information to restore the shortest path from s to any vertex. We'll maintain an array of predecessors $p[]$ in which for each vertex $v \neq s$, $p[v]$ is the penultimate vertex in the shortest path from s to v. Here we use the fact that if we take the shortest path to some vertex v and remove v from this path, we'll get a path ending in at vertex $p[v]$, and this path will be the shortest for the vertex $p[v]$. This array of predecessors can be used to restore the shortest path to any vertex: starting with v, repeatedly take the predecessor of the current vertex until we reach the starting vertex s to get the required shortest path with vertices listed in reverse order. So, the shortest path P to the vertex v is equal to:

$$P = (s, \ldots, p[p[p[v]]], p[p[v]], p[v], v)$$

Building this array of predecessors is very simple: for each successful relaxation, i.e. when for some selected vertex v, there is an improvement in the distance to some vertex to, we update the predecessor vertex for to with vertex v:

$$p[\text{to}] = v$$

Proof

The main assertion on which Dijkstra's algorithm correctness is based is the following:

After any vertex v becomes marked, the current distance to it $d[v]$ is the shortest, and will no longer change.

The proof is done by induction. For the first iteration this statement is obvious: the only marked vertex is s, and the distance to is $d[s] = 0$ is indeed the length of the shortest path to s. Now suppose this statement is true for all previous iterations, i.e. for all already marked vertices; let's prove that it is not violated after the current iteration completes. Let v be the vertex selected in the current iteration, i.e. v is the vertex that the algorithm will mark. Now we have to prove that $d[v]$ is indeed equal to the length of the shortest path to it $l[v]$.

Consider the shortest path P to the vertex v. This path can be split into two parts: P_1 which consists of only marked nodes (at least the starting vertex s is part of P_1), and the rest of the path P_2 (it may include a marked vertex, but it always starts with an unmarked vertex). Let's denote the first vertex of the path P_2 as p, and the last vertex of the path P_1 as q.

First we prove our statement for the vertex p, i.e. let's prove that $d[p] = l[p]$. This is almost obvious: on one of the previous iterations we chose the vertex q and performed relaxation from it. Since (by virtue of the choice of vertex p) the shortest path to p is the shortest path to q plus edge (p, q), the relaxation from q set the value of $d[p]$ to the length of the shortest path $l[p]$.

Since the edges' weights are non-negative, the length of the shortest path $l[p]$ (which we just proved to be equal to $d[p]$) does not exceed the length $l[v]$ of the shortest path to the vertex v. Given that $l[v] \le d[v]$ (because Dijkstra's algorithm could not have found a shorter way than the shortest possible one), we get the inequality:

$$d[p] = l[p] \le l[v] \le d[v]$$

On the other hand, since both vertices p and v are unmarked, and the current iteration chose vertex v, not p, we get another inequality:

$$d[p] \ge d[v]$$

From these two inequalities we conclude that $d[p] = d[v]$, and then from previously found equations we get:

$$d[v] = l[v]$$

Q.E.D.

Implementation

Dijkstra's algorithm performs n iterations. On each iteration it selects an unmarked vertex v with the lowest value $d[v]$, marks it and checks all the edges (v, to) attempting to improve the value $d[\text{to}]$.

The running time of the algorithm consists of:

- n searches for a vertex with the smallest value $d[v]$ among $O(n)$ unmarked vertices
- m relaxation attempts

For the simplest implementation of these operations on each iteration vertex search requires $O(n)$ operations, and each relaxation can be performed in $O(1)$. Hence, the resulting asymptotic behavior of the algorithm is:

$$O(n^2 + m)$$

This complexity is optimal for dense graph, i.e. when $m \approx n^2$. However in sparse graphs, when m is much smaller than the maximal number of edges n^2, the problem can be solved in $O(n \log n + m)$ complexity. The algorithm and implementation can be found on the article Dijkstra on sparse graphs.

```
const int INF = 1000000000;
vector<vector<pair<int, int>>> adj;

void dijkstra(int s, vector<int> & d, vector<int> & p) {
    int n = adj.size();
    d.assign(n, INF);
    p.assign(n, -1);
    vector<bool> u(n, false);

    d[s] = 0;
    for (int i = 0; i < n; i++) {
        int v = -1;
        for (int j = 0; j < n; j++) {
            if (!u[j] && (v == -1 || d[j] < d[v]))
                v = j;
        }

        if (d[v] == INF)
            break;

        u[v] = true;
        for (auto edge : adj[v]) {
            int to = edge.first;
            int len = edge.second;

            if (d[v] + len < d[to]) {
                d[to] = d[v] + len;
                p[to] = v;
            }
        }
    }
}
```

Here the graph adj is stored as adjacency list: for each vertex v $\text{adj}[v]$ contains the list of edges going from this vertex, i.e. the list of `pair<int,int>` where the first element in the pair is the vertex at the other end of the edge, and the second element is the edge weight.

The function takes the starting vertex s and two vectors that will be used as return values.

First of all, the code initializes arrays: distances $d[]$, labels $u[]$ and predecessors $p[]$. Then it performs n iterations. At each iteration the vertex v is selected which has the smallest distance $d[v]$ among all the unmarked vertices. If the distance to selected vertex v is equal to infinity, the algorithm stops. Otherwise the vertex is marked, and all the edges going out from this vertex are checked. If relaxation along the edge is possible (i.e. distance $d[\text{to}]$ can be improved), the distance $d[\text{to}]$ and predecessor $p[\text{to}]$ are updated.

After performing all the iterations array $d[]$ stores the lengths of the shortest paths to all vertices, and array $p[]$ stores the predecessors of all vertices (except starting vertex s). The path to any vertex t can be restored in the following way:

```
vector<int> restore_path(int s, int t, vector<int> const& p) {
    vector<int> path;

    for (int v = t; v != s; v = p[v])
        path.push_back(v);
    path.push_back(s);

    reverse(path.begin(), path.end());
    return path;
}
```

References

- Edsger Dijkstra. A note on two problems in connexion with graphs [1959]
- Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein. Introduction to Algorithms [2005]

Practice Problems

- Timus - Ivan's Car [Difficulty:Medium]
- Timus - Sightseeing Trip
- SPOJ - SHPATH [Difficulty:Easy]
- Codeforces - Dijkstra? [Difficulty:Easy]
- Codeforces - Shortest Path
- Codeforces - Jzzhu and Cities
- Codeforces - The Classic Problem
- Codeforces - President and Roads
- Codeforces - Complete The Graph
- TopCoder - SkiResorts
- TopCoder - MaliciousPath
- SPOJ - Ada and Trip
- LA - 3850 - Here We Go(relians) Again
- GYM - Destination Unknown (D)

- UVA 12950 - Even Obsession
- GYM - Journey to Grece (A)
- UVA 13030 - Brain Fry
- UVA 1027 - Toll
- UVA 11377 - Airport Setup
- Codeforces - Dynamic Shortest Path
- UVA 11813 - Shopping
- UVA 11833 - Route Change
- SPOJ - Easy Dijkstra Problem
- LA - 2819 - Cave Raider
- UVA 12144 - Almost Shortest Path
- UVA 12047 - Highest Paid Toll
- UVA 11514 - Batman
- Codeforces - Team Rocket Rises Again
- UVA - 11338 - Minefield
- UVA 11374 - Airport Express
- UVA 11097 - Poor My Problem
- UVA 13172 - The music teacher
- Codeforces - Dirty Arkady's Kitchen
- SPOJ - Delivery Route
- SPOJ - Costly Chess
- CSES - Shortest Routes 1
- CSES - Flight Discount
- CSES - Flight Routes

Contributors:

bimalkant-lauhny (42.27%) jakobkogler (20.62%) tcNickolas (16.49%) Morass (11.34%) adamant-pwn (3.09%) Aryamn (1.55%) ChamanAgrawal (1.55%) RodionGork (1.55%) likecs (1.03%) PirateOfAndaman (0.52%)