Last update: February 24, 2024 Translated From: e-maxx.ru

Minimum-cost flow - Successive shortest path algorithm

Given a network G consisting of n vertices and m edges. For each edge (generally speaking, oriented edges, but see below), the capacity (a non-negative integer) and the cost per unit of flow along this edge (some integer) are given. Also the source s and the sink t are marked.

For a given value K, we have to find a flow of this quantity, and among all flows of this quantity we have to choose the flow with the lowest cost. This task is called **minimum-cost flow problem**.

Sometimes the task is given a little differently: you want to find the maximum flow, and among all maximal flows we want to find the one with the least cost. This is called the **minimum-cost maximum-flow problem**.

Both these problems can be solved effectively with the algorithm of successive shortest paths.

Algorithm

This algorithm is very similar to the Edmonds-Karp for computing the maximum flow.

Simplest case

First we only consider the simplest case, where the graph is oriented, and there is at most one edge between any pair of vertices (e.g. if (i, j) is an edge in the graph, then (j, i) cannot be part in it as well).

Let U_{ij} be the capacity of an edge (i, j) if this edge exists. And let C_{ij} be the cost per unit of flow along this edge (i, j). And finally let $F_{i,j}$ be the flow along the edge (i, j). Initially all flow values are zero.

We **modify** the network as follows: for each edge (i, j) we add the **reverse edge** (j, i) to the network with the capacity $U_{ji} = 0$ and the cost $C_{ji} = -C_{ij}$. Since, according to our restrictions, the edge (j, i) was not in the network before, we still have a network that is not a multigraph (graph with multiple edges). In addition we will always keep the condition $F_{ji} = -F_{ij}$ true during the steps of the algorithm.

We define the **residual network** for some fixed flow F as follow (just like in the Ford-Fulkerson algorithm): the residual network contains only unsaturated edges (i.e. edges in which $F_{ij} < U_{ij}$), and the residual capacity of each such edge is $R_{ij} = U_{ij} - F_{ij}$.

Now we can talk about the **algorithms** to compute the minimum-cost flow. At each iteration of the algorithm we find the shortest path in the residual graph from s to t. In contrast to Edmonds-Karp, we look for the shortest path in terms of the cost of the path instead of the number of edges. If there doesn't exists a path anymore, then the algorithm terminates, and the stream F is the desired one. If a path was found, we increase the flow along it as much as possible (i.e. we find the minimal residual capacity R of the path, and increase the flow by it, and reduce the back edges by the same amount). If at some point the flow reaches the value K, then we stop the algorithm (note that in the last iteration of the algorithm it is necessary to increase the flow by only such an amount so that the final flow value doesn't surpass K).

It is not difficult to see, that if we set K to infinity, then the algorithm will find the minimum-cost maximum-flow. So both variations of the problem can be solved by the same algorithm.

Undirected graphs / multigraphs

The case of an undirected graph or a multigraph doesn't differ conceptually from the algorithm above. The algorithm will also work on these graphs. However it becomes a little more difficult to implement it.

An **undirected edge** (i, j) is actually the same as two oriented edges (i, j) and (j, i) with the same capacity and values. Since the above-described minimum-cost flow algorithm generates a back edge for each directed edge, so it splits the undirected edge into 4 directed edges, and we actually get a **multigraph**.

How do we deal with **multiple edges**? First the flow for each of the multiple edges must be kept separately. Secondly, when searching for the shortest path, it is necessary to take into account that it is important which of the multiple edges is used in the path. Thus instead of the usual ancestor array we additionally must store the edge number from which we came from along with the ancestor. Thirdly, as the flow increases along a certain edge, it is necessary to reduce the flow along the back edge. Since we have multiple edges, we have to store the edge number for the reversed edge for each edge.

There are no other obstructions with undirected graphs or multigraphs.

Complexity

The algorithm here is generally exponential in the size of the input. To be more specific, in the worst case it may push only as much as 1 unit of flow on each iteration, taking $O(F)$ iterations to find a minimum-cost flow of size F, making a total runtime to be $O(F \cdot T)$, where T is the time required to find the shortest path from source to sink.

If Bellman-Ford algorithm is used for this, it makes the running time $O(Fmn)$. It is also possible to modify Dijkstra's algorithm, so that it needs $O(nm)$ pre-processing as an initial step and then works in $O(m \log n)$ per iteration, making the overall running time to be $O(mn + Fm \log n)$. Here is a generator of a graph, on which such algorithm would require $O(2^{n/2} n^2 \log n)$ time.

The modified Dijkstra's algorithm uses so-called potentials from Johnson's algorithm. It is possible to combine the ideas of this algorithm and Dinic's algorithm to reduce the number of iterations from F to $\min(F, nC)$, where C is the maximum cost found among edges. You may read further about potentials and their combination with Dinic algorithm here.

Implementation

Here is an implementation using the SPFA algorithm for the simplest case.

```
struct Edge
{
    int from, to, capacity, cost;
};

vector<vector<int>> adj, cost, capacity;

const int INF = 1e9;

void shortest_paths(int n, int v0, vector<int>& d, vector<int>& p) {
    d.assign(n, INF);
    d[v0] = 0;
    vector<bool> inq(n, false);
    queue<int> q;
    q.push(v0);
    p.assign(n, -1);
```

```cpp
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        inq[u] = false;
        for (int v : adj[u]) {
            if (capacity[u][v] > 0 && d[v] > d[u] + cost[u][v]) {
                d[v] = d[u] + cost[u][v];
                p[v] = u;
                if (!inq[v]) {
                    inq[v] = true;
                    q.push(v);
                }
            }
        }
    }
}

int min_cost_flow(int N, vector<Edge> edges, int K, int s, int t) {
    adj.assign(N, vector<int>());
    cost.assign(N, vector<int>(N, 0));
    capacity.assign(N, vector<int>(N, 0));
    for (Edge e : edges) {
        adj[e.from].push_back(e.to);
        adj[e.to].push_back(e.from);
        cost[e.from][e.to] = e.cost;
        cost[e.to][e.from] = -e.cost;
        capacity[e.from][e.to] = e.capacity;
    }

    int flow = 0;
    int cost = 0;
    vector<int> d, p;
    while (flow < K) {
        shortest_paths(N, s, d, p);
        if (d[t] == INF)
            break;

        // find max flow on that path
        int f = K - flow;
        int cur = t;
        while (cur != s) {
            f = min(f, capacity[p[cur]][cur]);
            cur = p[cur];
        }

        // apply flow
        flow += f;
        cost += f * d[t];
        cur = t;
        while (cur != s) {
            capacity[p[cur]][cur] -= f;
            capacity[cur][p[cur]] += f;
            cur = p[cur];
        }
    }

    if (flow < K)
        return -1;
    else
        return cost;
}
```

Practice Problems

- CSES - Task Assignment
- CSES - Grid Puzzle II
- AtCoder - Dream Team

Contributors:
jakobkogler (85.12%) adamant-pwn (5.96%) prprprpony (4.76%) fffelix-huang (2.98%) maiquang04 (0.6%) hieplpvip (0.6%)