Last update: July 10, 2023 Translated From: e-maxx.ru

Maximum flow - Dinic's algorithm

Dinic's algorithm solves the maximum flow problem in $O(V^2E)$. The maximum flow problem is defined in this article Maximum flow - Ford-Fulkerson and Edmonds-Karp. This algorithm was discovered by Yefim Dinitz in 1970.

Definitions

A **residual network** G^R of network G is a network which contains two edges for each edge $(v, u) \in G$:

- (v, u) with capacity $c_{vu}^R = c_{vu} - f_{vu}$
- (u, v) with capacity $c_{uv}^R = f_{vu}$

A **blocking flow** of some network is such a flow that every path from s to t contains at least one edge which is saturated by this flow. Note that a blocking flow is not necessarily maximal.

A **layered network** of a network G is a network built in the following way. Firstly, for each vertex v we calculate $level[v]$ - the shortest path (unweighted) from s to this vertex using only edges with positive capacity. Then we keep only those edges (v, u) for which $level[v] + 1 = level[u]$. Obviously, this network is acyclic.

Algorithm

The algorithm consists of several phases. On each phase we construct the layered network of the residual network of G. Then we find an arbitrary blocking flow in the layered network and add it to the current flow.

Proof of correctness

Let's show that if the algorithm terminates, it finds the maximum flow.

If the algorithm terminated, it couldn't find a blocking flow in the layered network. It means that the layered network doesn't have any path from s to t. It means that the residual network doesn't have any path from s to t. It means that the flow is maximum.

Number of phases

The algorithm terminates in less than V phases. To prove this, we must firstly prove two lemmas.

Lemma 1. The distances from s to each vertex don't decrease after each iteration, i. e. $level_{i+1}[v] \geq level_i[v]$.

Proof. Fix a phase i and a vertex v. Consider any shortest path P from s to v in G_{i+1}^R. The length of P equals $level_{i+1}[v]$. Note that G_{i+1}^R can only contain edges from G_i^R and back edges for edges from G_i^R. If P has no back edges for G_i^R, then $level_{i+1}[v] \geq level_i[v]$ because P is also a path in G_i^R. Now, suppose that P has at least one back edge. Let the first such edge be (u, w).Then $level_{i+1}[u] \geq level_i[u]$ (because of the first case). The edge (u, w) doesn't belong to G_i^R, so the edge (w, u) was affected by the blocking flow on the previous iteration. It means that $level_i[u] = level_i[w] + 1$. Also, $level_{i+1}[w] = level_{i+1}[u] + 1$. From these two equations and $level_{i+1}[u] \geq level_i[u]$ we obtain $level_{i+1}[w] \geq level_i[w] + 2$. Now we can use the same idea for the rest of the path.

Lemma 2. $level_{i+1}[t] > level_i[t]$

Proof. From the previous lemma, $level_{i+1}[t] \geq level_i[t]$. Suppose that $level_{i+1}[t] = level_i[t]$. Note that G_{i+1}^R can only contain edges from G_i^R and back edges for edges from G_i^R. It means that there is a shortest path in G_i^R which wasn't blocked by the blocking flow. It's a contradiction.

From these two lemmas we conclude that there are less than V phases because $level[t]$ increases, but it can't be greater than $V - 1$.

Finding blocking flow

In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS from s to t in the layered network while it can be pushed. In order to do it more quickly, we must remove the edges which can't be used to push anymore. To do this we can keep a pointer in each vertex which points to the next edge which can be used.

A single DFS run takes $O(k + V)$ time, where k is the number of pointer advances on this run. Summed up over all runs, number of pointer advances can not exceed E. On the other hand, total number of runs won't exceed E, as every run saturates at least one edge. In this way, total running time of finding a blocking flow is $O(VE)$.

Complexity

There are less than V phases, so the total complexity is $O(V^2E)$.

Unit networks

A **unit network** is a network in which for any vertex except s and t **either incoming or outgoing edge is unique and has unit capacity**. That's exactly the case with the network we build to solve the maximum matching problem with flows.

On unit networks Dinic's algorithm works in $O(E\sqrt{V})$. Let's prove this.

Firstly, each phase now works in $O(E)$ because each edge will be considered at most once.

Secondly, suppose there have already been $\sqrt{V}$ phases. Then all the augmenting paths with the length $\le \sqrt{V}$ have been found. Let f be the current flow, f' be the maximum flow. Consider their difference $f' - f$. It is a flow in G^R of value $|f'| - |f|$ and on each edge it is either 0 or 1. It can be decomposed into $|f'| - |f|$ paths from s to t and possibly cycles. As the network is unit, they can't have common vertices, so the total number of vertices is $\ge (|f'| - |f|)\sqrt{V}$, but it is also $\le V$, so in another $\sqrt{V}$ iterations we will definitely find the maximum flow.

Unit capacities networks

In a more generic settings when all edges have unit capacities, *but the number of incoming and outgoing edges is unbounded*, the paths can't have common edges rather than common vertices. In a similar way it allows to prove the bound of $\sqrt{E}$ on the number of iterations, hence the running time of Dinic algorithm on such networks is at most $O(E\sqrt{E})$.

Finally, it is also possible to prove that the number of phases on unit capacity networks doesn't exceed $O(V^{2/3})$, providing an alternative estimate of $O(EV^{2/3})$ on the networks with particularly large number of edges.

Implementation

```
struct FlowEdge {
    int v, u;
    long long cap, flow = 0;
    FlowEdge(int v, int u, long long cap) : v(v), u(u), cap(cap) {}
};

struct Dinic {
```

```
    const long long flow_inf = 1e18;
    vector<FlowEdge> edges;
    vector<vector<int>> adj;
    int n, m = 0;
    int s, t;
    vector<int> level, ptr;
    queue<int> q;

    Dinic(int n, int s, int t) : n(n), s(s), t(t) {
        adj.resize(n);
        level.resize(n);
        ptr.resize(n);
    }

    void add_edge(int v, int u, long long cap) {
        edges.emplace_back(v, u, cap);
        edges.emplace_back(u, v, 0);
        adj[v].push_back(m);
        adj[u].push_back(m + 1);
        m += 2;
    }

    bool bfs() {
        while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (int id : adj[v]) {
                if (edges[id].cap - edges[id].flow < 1)
                    continue;
                if (level[edges[id].u] != -1)
                    continue;
                level[edges[id].u] = level[v] + 1;
                q.push(edges[id].u);
            }
        }
        return level[t] != -1;
    }

    long long dfs(int v, long long pushed) {
        if (pushed == 0)
            return 0;
        if (v == t)
            return pushed;
        for (int& cid = ptr[v]; cid < (int)adj[v].size(); cid++) {
            int id = adj[v][cid];
            int u = edges[id].u;
            if (level[v] + 1 != level[u] || edges[id].cap - edges[id].flow < 1)
                continue;
            long long tr = dfs(u, min(pushed, edges[id].cap - edges[id].flow));
            if (tr == 0)
                continue;
            edges[id].flow += tr;
            edges[id ^ 1].flow -= tr;
```

```
            return tr;
        }
        return 0;
    }

    long long flow() {
        long long f = 0;
        while (true) {
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            q.push(s);
            if (!bfs())
                break;
            fill(ptr.begin(), ptr.end(), 0);
            while (long long pushed = dfs(s, flow_inf)) {
                f += pushed;
            }
        }
        return f;
    }
};
```

Contributors:

SYury (87.18%) adamant-pwn (9.62%) jakobkogler (1.28%) Naman-Bhalla (1.28%) MiniCube11 (0.64%)