Network Flow

Chenqi Wang

2021.9.2

1 Problem

https://www.luogu.com.cn/problem/P3376

2 Definitions

1. Flow Network: A directed graph G(V, E) with 2 special vertices source $s \in V$ and sink $t \in V$, equipped with a capacity function c: $V \times V \rightarrow R_+$(including 0)

2. Flow: $f: V \times V \rightarrow R_+$, s.t.

- Capacity Constraint: $\forall u, v \in V,\ 0 \leq f(u,v) \leq c(u,v)\ (f(u,v) = c(u,v) = 0\ if\ (u,v) \notin E)$
- Flow Conservation: $\forall u \in V - \{s,t\},\ \sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$ (i.e. inflow == outflow)

3. The Value of a Flow: $|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$ (i.e. the net outflow of source s)

4. Residual Capacity:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & if\ (u,v) \in E \\ f(v,u) & if\ (v,u) \in E \\ 0 & otherwise \end{cases}$$

assuming $\forall u, v \in V$, at most one of (u,v) and (v,u) exists.

5. Residual Network: $G_f = (V, E_f)$, where $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$, equipped with residual capacity c_f. (s, t remains the same of course)

6. Augmenting Path: a simple path p in G_f from s to t

7. Residual Capacity of p: $c_f(p) = min\{c_f(u,v) : (u,v) \in p\}$

8. s-t Cut: $C = (S,T)$ where $S \subset V$, $T = V - S$, $s \in S, t \in T$. The number of s-t Cuts is 2^{V-2}

9. Net Flow cross the Cut (S, T): $f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u)$

10. Capacity of Cut (S, T): $c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v)$

11. **Max-Flow Min-Cut Theorem**: Given a flow network G=(V,E), a flow f on it, a source s and a sink t, the following statements are equivalent:

a. f is a max flow of G.

b. there is no augmenting path in the residual network G_f

c. $\exists C = (S,T)$, $|f| = c(S,T)$

Max Flow: a flow f s.t. $\forall flow\ f', |f'| \leq |f|$

Min Cut: a cut C=(S,T) s.t. $\forall cut\ C' = (S', T'), c(S,T) \leq c(S', T')$ (the cut in c is actually a min cut)

Lemma: $\forall C = (S,T), |f| = f(S,T) \leq c(S,T)$

3 Ford-Fulkerson Method

Ford-Fulkerson Method:

1. Initialize flow $f(u,v) = 0$, for all $u, v \in V$, and G_f accordingly ($c_f = c$)
2. While there exists augmenting path p, update f along p: for $(u,v) \in p$, if $(u,v) \in E, f(u,v) += c_f(p)$; if $(u,v) \notin E, f(v,u) -= c_f(p)$
3. When the loop in 2 ends, we get a max flow f

Traditional FF algorithm uses DFS to search for augmenting path, with a time complexity of $O(E|f^*|)$, where $|f^*|$ is the value of a max flow f^*, because each DFS costs $O(V+E) = O(E)$ (assume every $u \in V$ can be reached from s, then $E \geq V-1, V=O(E)$), and assuming c is a integer-valued function (rational-valued can be scaled to be integer-valued), each loop increases $|f|$ at least by 1, and loop ends when $|f|$ reachs $|f^*|$, $O(|f^*|)$ loops in total.

4 Edmonds-Karp

We use BFS to search for augmenting path.

Time Complexity $O(VE^2)$, because each edge can become a critical edge (the edge with minimum residual capacity in an augmenting path) at most $O(V/2)$ times, so $O(VE)$ critical edges in total, and each loop decreases the number of critical edges by at least 1, $O(VE)$ loops, and each loop $O(E)$ for BFS.

5 Dinic

1. Initialize $f=0$, $c_f=c$, $ans=0$

2. Use BFS to calculate the distance of any node u from s in G_f, $dis[s]=0$, $dis[u]:=$ the length of a shortest path from s to u in G_f. If $dis[t]=+\infty$, the algorithm ends and we return ans as the value of max flow.

3. Use DFS to augment flow f: Starting from u = s (and mark vis[s] = 1), we can go to any v s.t. $dis[v]=dis[u]+1, cf[u][v]>0, vis[v]=0$. In the meanwhile, maintain a Min to record the maximal amount of flow we can augment (i.e. the minimal cf in the path from s to u).

The return value of dfs(u) is the sum of the augmented amount of flow on all outedges of u, and the return value of dfs(t) is actually the Min up till t.

Every time we return from dfs(v) (let sub = dfs(v)), we update the value of f and cf: if $c[u][v]>0$, $f[u][v]+=sub, cf[u][v]-=sub, cf[v][u]+=sub$; if $c[v][u]>0$, $f[v][u]-=sub, cf[v][u]-=sub, cf[u][v]+=sub$. And update Min -= sub (this Min is corresponding to the path from s to u).

Update ans : ans += dfs(s)

4. Go to 2

Time Complexity $O(V^2E)$. (don't know why)

6 Solutions

6.1 Edmonds-Karp

Listing 1: Edmonds-Karp

```
#include <iostream>
#include <vector>
#include <climits>
#include <queue>
#define ll long long
#define INF LLONG_MAX
using namespace std;

// EdmondsKarp: use bfs to search for augmenting paths
inline bool bfs(vector<vector<ll>>& f, vector<vector<ll>>& c, vector<
    vector<ll>>& cf, int s, int t) {
    int V = cf.size() - 1;

    queue<int> q;
    q.push(s);
    vector<int> vis(V + 1);
    vis[s] = 1;
    vector<int> pre(V + 1, -1);

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        if (u == t) {
            ll aug = INF;

            while (pre[u] != -1) {
                aug = min(aug, cf[pre[u]][u]);
                u = pre[u];
            }

            u = t;
            while (pre[u] != -1) {
                if (c[pre[u]][u] > 0) {
```

```
                    f[pre[u]][u] += aug;
                    cf[pre[u]][u] = c[pre[u]][u] - f[pre[u]][u];
                    cf[u][pre[u]] = f[pre[u]][u];
                }
                else if (c[u][pre[u]] > 0) {
                    f[u][pre[u]] -= aug;
                    cf[u][pre[u]] = c[u][pre[u]] - f[u][pre[u]];
                    cf[pre[u]][u] = f[u][pre[u]];
                }

                u = pre[u];
            }

            while (!q.empty()) q.pop();
            return true;
        }

        for (int v = 1; v <= V; ++v) {
            if (vis[v] == 0 && cf[u][v] > 0) {
                q.push(v);
                vis[v] = 1;
                pre[v] = u;
            }
        }
    }

    return false;
}

inline ll EdmondsKarp(vector<vector<ll>>& c, int s, int t) {
    int V = c.size() - 1;

    vector<vector<ll>> f(V + 1, vector<ll>(V + 1)); // flow
    vector<vector<ll>> cf(V + 1, vector<ll>(V + 1)); // residual network
    for (int u = 1; u <= V; ++u) {
        for (int v = 1; v <= V; ++v) {
            cf[u][v] = c[u][v];
        }
    }

    while (bfs(f, c, cf, s, t));

    ll maxFlow = 0;
    for (int u = 1; u <= V; ++u) maxFlow += (ll)f[s][u] + (ll)f[u][s];
    return maxFlow;
}
```

```

inline void eliminateAntiparallel(vector<vector<ll>>& c) {
    int V = c.size() - 1;

    for (int u = 1; u <= V; ++u) {
        for (int v = u + 1; v <= V; ++v) {
            if (c[u][v] > 0 && c[v][u] > 0) {
                int cur = c.size();
                c.push_back(vector<ll>(cur + 1));
                for (int i = 1; i < cur; ++i) {
                    c[i].push_back(0);
                }

                c[u][cur] = c[u][v];
                c[cur][v] = c[u][v];
                c[u][v] = 0;
            }
        }
    }
}

int main() {
    int V, E, s, t;
    cin >> V >> E >> s >> t;

    // flow network
    vector<vector<ll>> c(V + 1, vector<ll>(V + 1, 0));

    for (int i = 0; i < E; ++i) {
        ll u, v, w;
        cin >> u >> v >> w;
        if (u == v) continue;
        c[u][v] += w;
    }

    // eliminateAntiparallel(c);
    cout << EdmondsKarp(c, s, t) << endl;

    return 0;
}
```

6.2 Dinic

Listing 2: Dinic

```
#include <iostream>
#include <vector>
#include <climits>
#include <queue>
#define ll long long
#define INF LLONG_MAX
using namespace std;

inline bool bfs(vector<vector<ll>>& cf, vector<ll>& dis, int s, int t) {
    int V = cf.size() - 1;
    dis = vector<ll>(V + 1, INF);
    dis[s] = 0;

    vector<int> vis(V + 1);

    queue<int> q;
    q.push(s);
    vis[s] = 1;

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (int v = 1; v <= V; ++v) {
            if (cf[u][v] > 0 && vis[v] == 0) {
                dis[v] = dis[u] + 1;
                q.push(v);
                vis[v] = 1;
            }
        }
    }

    return dis[t] != INF;
}

ll dfs(int u, int t, ll Min, vector<vector<ll>>& f, vector<vector<ll>>&
    cf, vector<ll>& dis, vector<vector<ll>>& c, vector<int>& vis) {
    if (u == t) {
        return Min;
    }

    int V = cf.size() - 1;
    ll ret = 0;

```

```
    for (int v = 1; v <= V; ++v) {
        if (vis[v] == 0 && dis[v] == dis[u] + 1 && cf[u][v] > 0) {
            vis[v] = 1;
            ll sub = dfs(v, t, min(Min, cf[u][v]), f, cf, dis, c, vis);
            ret += sub;
            if (c[u][v] > 0) {
                f[u][v] += sub;
                cf[u][v] -= sub;
                cf[v][u] += sub;
            }
            else if (c[v][u] > 0) {
                f[v][u] -= sub;
                cf[v][u] += sub;
                cf[u][v] -= sub;
            }

            Min -= sub;
        }
    }

    return ret;
}

inline ll Dinic(vector<vector<ll>>& c, int s, int t) {
    int V = c.size() - 1;
    vector<vector<ll>> f(V + 1, vector<ll>(V + 1));
    vector<vector<ll>> cf(c);
    vector<ll> dis(V + 1, INF);
    ll ans = 0;

    while (bfs(cf, dis, s, t)) {
        vector<int> vis(V + 1);
        vis[s] = 1;
        ans += dfs(s, t, INF, f, cf, dis, c, vis);
    }

    /*
    ll ans2 = 0;
    for (int v = 1; v <= V; ++v) {
        ans2 += f[s][v] + f[v][s];
    }
    */

    // cout << ans << endl << ans2 << endl;
    return ans;
}
```

```

int main() {
    int V, E, s, t;
    cin >> V >> E >> s >> t;

    // flow network
    vector<vector<ll>> c(V + 1, vector<ll>(V + 1, 0));

    for (int i = 0; i < E; ++i) {
        ll u, v, w;
        cin >> u >> v >> w;
        if (u == v) continue;
        c[u][v] += w;
    }

    cout << Dinic(c, s, t) << endl;

    return 0;
}
```

source:

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. Introduction to Algorithms, Third Edition (3rd. ed.). The MIT Press.