网络流

最大流

Dinic 解

使用 Dinic 算法，理论最坏复杂度为 $\mathcal{O}(N^2M)$ ，例题范围：$N = 1200,\ m = 5 \times 10^3$ 。一般步骤：BFS 建立分层图，无回溯 DFS 寻找所有可行的增广路径。封装：求从点 S 到点 T 的最大流。

```
template<typename T> struct Flow_ {
    const int n;
    const T inf = numeric_limits<T>::max();
    struct Edge {
        int to;
        T w;
        Edge(int to, T w) : to(to), w(w) {}
    };
    vector<Edge> ver;
    vector<vector<int>> h;
    vector<int> cur, d;

    Flow_(int n) : n(n + 1), h(n + 1) {}
    void add(int u, int v, T c) {
        h[u].push_back(ver.size());
        ver.emplace_back(v, c);
        h[v].push_back(ver.size());
        ver.emplace_back(u, 0);
    }
    bool bfs(int s, int t) {
        d.assign(n, -1);
        d[s] = 0;
        queue<int> q;
        q.push(s);
        while (!q.empty()) {
            auto x = q.front();
            q.pop();
            for (auto it : h[x]) {
                auto [y, w] = ver[it];
                if (w && d[y] == -1) {
                    d[y] = d[x] + 1;
                    if (y == t) return true;
                    q.push(y);
                }
            }
        }
        return false;
    }
    T dfs(int u, int t, T f) {
        if (u == t) return f;
        auto r = f;
        for (int &i = cur[u]; i < h[u].size(); i++) {
            auto j = h[u][i];
```

```
            auto &[v, c] = ver[j];
            auto &[u, rc] = ver[j ^ 1];
            if (c && d[v] == d[u] + 1) {
                auto a = dfs(v, t, std::min(r, c));
                c -= a;
                rc += a;
                r -= a;
                if (!r) return f;
            }
        }
        return f - r;
    }
    T work(int s, int t) {
        T ans = 0;
        while (bfs(s, t)) {
            cur.assign(n, 0);
            ans += dfs(s, t, inf);
        }
        return ans;
    }
};
using Flow = Flow_<int>;
```

预流推进 HLPP

理论最坏复杂度为 $\mathcal{O}(N^2\sqrt{M})$ ，例题范围：$N = 1200,\ m = 1.2 \times 10^5$ 。

```
template <typename T> struct PushRelabel {
    const int inf = 0x3f3f3f3f;
    const T INF = 0x3f3f3f3f3f3f3f3f;
    struct Edge {
        int to, cap, flow, anti;
        Edge(int v = 0, int w = 0, int id = 0) : to(v), cap(w), flow(0), anti(id) {}
    };
    vector<vector<Edge>> e;
    vector<vector<int>> gap;
    vector<T> ex; // 超额流
    vector<bool> ingap;
    vector<int> h;
    int n, gobalcnt, maxH = 0;
    T maxflow = 0;

    PushRelabel(int n) : n(n), e(n + 1), ex(n + 1), gap(n + 1) {}
    void addedge(int u, int v, int w) {
        e[u].push_back({v, w, (int)e[v].size()});
        e[v].push_back({u, 0, (int)e[u].size() - 1});
    }
    void PushEdge(int u, Edge &edge) {
        int v = edge.to, d = min(ex[u], 1LL * edge.cap - edge.flow);
        ex[u] -= d;
        ex[v] += d;
        edge.flow += d;
        e[v][edge.anti].flow -= d;
```

```
        if (h[v] != inf && d > 0 && ex[v] == d && !ingap[v]) {
            ++gobalcnt;
            gap[h[v]].push_back(v);
            ingap[v] = 1;
        }
    }
    void PushPoint(int u) {
        for (auto k = e[u].begin(); k != e[u].end(); k++) {
            if (h[k->to] + 1 == h[u] && k->cap > k->flow) {
                PushEdge(u, *k);
                if (!ex[u]) break;
            }
        }
        if (!ex[u]) return;
        if (gap[h[u]].empty()) {
            for (int i = h[u] + 1; i <= min(maxH, n); i++) {
                for (auto v : gap[i]) {
                    ingap[v] = 0;
                }
                gap[i].clear();
            }
        }
        h[u] = inf;
        for (auto [to, cap, flow, anti] : e[u]) {
            if (cap > flow) {
                h[u] = min(h[u], h[to] + 1);
            }
        }
        if (h[u] >= n) return;
        maxH = max(maxH, h[u]);
        if (!ingap[u]) {
            gap[h[u]].push_back(u);
            ingap[u] = 1;
        }
    }
    void init(int t, bool f = 1) {
        ingap.assign(n + 1, 0);
        for (int i = 1; i <= maxH; i++) {
            gap[i].clear();
        }
        gobalcnt = 0, maxH = 0;
        queue<int> q;
        h.assign(n + 1, inf);
        h[t] = 0, q.push(t);
        while (q.size()) {
            int u = q.front();
            q.pop(), maxH = h[u];
            for (auto &[v, cap, flow, anti] : e[u]) {
                if (h[v] == inf && e[v][anti].cap > e[v][anti].flow) {
                    h[v] = h[u] + 1;
                    q.push(v);
                    if (f) {
                        gap[h[v]].push_back(v);
```

```
                        ingap[v] = 1;
                    }
                }
            }
        }
    }
    T work(int s, int t) {
        init(t, 0);
        if (h[s] == inf) return maxflow;
        h[s] = n;
        ex[s] = INF;
        ex[t] = -INF;
        for (auto k = e[s].begin(); k != e[s].end(); k++) {
            PushEdge(s, *k);
        }
        while (maxH > 0) {
            if (gap[maxH].empty()) {
                maxH--;
                continue;
            }
            int u = gap[maxH].back();
            gap[maxH].pop_back();
            ingap[u] = 0;
            PushPoint(u);
            if (gobalcnt >= 10 * n) {
                init(t);
            }
        }
        ex[s] -= INF;
        ex[t] += INF;
        return maxflow = ex[t];
    }
};
```

最小割

基础模型：构筑二分图，左半部 n 个点代表盈利项目，右半部 m 个点代表材料成本，收益为盈利之和减去成本之和，求最大收益。

建图：建立源点 S 向左半部连边，建立汇点 T 向右半部连边，如果某个项目需要某个材料，则新增一条容量 $+\infty$ 的跨部边。

割边：放弃某个项目则断开 S 至该项目的边，购买某个原料则断开该原料至 T 的边，最终的图一定不存在从 S 到 T 的路径，此时我们得到二分图的一个 $S-T$ 割。此时最小割即为求解最大流，边权之和减去最大流即为最大收益。

```
signed main() {
    int n, m;
    cin >> n >> m;

    int S = n + m + 1, T = n + m + 2;
    Flow flow(T);
    for (int i = 1; i <= n; i++) {
        int w;
```

```
        cin >> w;
        flow.add(S, i, w);
    }

    int sum = 0;
    for (int i = 1; i <= m; i++) {
        int x, y, w;
        cin >> x >> y >> w;
        flow.add(x, n + i, 1E18);
        flow.add(y, n + i, 1E18);
        flow.add(n + i, T, w);
        sum += w;
    }
    cout << sum - flow.work(S, T) << endl;
}
```

最小割树 Gomory-Hu Tree

无向连通图抽象出的一棵树，满足任意两点间的距离是他们的最小割。一共需要跑 n 轮最小割，总复杂度 $\mathcal{O}(N^3M)$ ，预处理最小割树上任意两点的距离 $\mathcal{O}(N^2)$ 。

过程：分治 n 轮，每一轮在图上随机选点，跑一轮最小割后连接树边；这一网络的残留网络会将剩余的点分为两组，根据分组分治。

```
void reset() { // struct需要额外封装退流
    for (int i = 0; i < ver.size(); i += 2) {
        ver[i].w += ver[i ^ 1].w;
        ver[i ^ 1].w = 0;
    }
}

signed main() { // Gomory-Hu Tree
    int n, m;
    cin >> n >> m;

    Flow<int> flow(n);
    for (int i = 1; i <= m; i++) {
        int u, v, w;
        cin >> u >> v >> w;
        flow.add(u, v, w);
        flow.add(v, u, w);
    }

    vector<int> vis(n + 1), fa(n + 1);
    vector ans(n + 1, vector<int>(n + 1, 1E9)); // N^2 枚举出全部答案
    vector<vector<pair<int, int>>> adj(n + 1);
    for (int i = 1; i <= n; i++) { // 分治 n 轮
        int s = 0; // 本质是在树上随机选点、跑最小割后连边
        for (; s <= n; s++) {
            if (fa[s] != s) break;
        }
        int t = fa[s];
```

```cpp

        int ans = flow.work(s, t); // 残留网络将点集分为两组，分治
        adj[s].push_back({t, ans});
        adj[t].push_back({s, ans});

        vis.assign(n + 1, 0);
        auto dfs = [&](auto dfs, int u) -> void {
            vis[u] = 1;
            for (auto it : flow.h[u]) {
                auto [v, c] = flow.ver[it];
                if (c && !vis[v]) {
                    dfs(dfs, v);
                }
            }
        };
        dfs(dfs, s);
        for (int j = 0; j <= n; j++) {
            if (vis[j] && fa[j] == t) {
                fa[j] = s;
            }
        }
    }

    for (int i = 0; i <= n; i++) {
        auto dfs = [&](auto dfs, int u, int fa, int c) -> void {
            ans[i][u] = c;
            for (auto [v, w] : adj[u]) {
                if (v == fa) continue;
                dfs(dfs, v, u, min(c, w));
            }
        };
        dfs(dfs, i, -1, 1E9);
    }

    int q;
    cin >> q;
    while (q--) {
        int u, v;
        cin >> u >> v;
        cout << ans[u][v] << "\n"; // 预处理答数组
    }
}
```

费用流

给定一个带费用的网络，规定 (u, v) 间的费用为 $f(u, v) \times w(u, v)$ ，求解该网络中总花费最小的最大流称之为**最小费用最大流**。总时间复杂度为 $\mathcal{O}(NMf)$ ，其中 f 代表最大流。

```cpp
struct MinCostFlow {
    using LL = long long;
    using PII = pair<LL, int>;
    const LL INF = numeric_limits<LL>::max();
```

```
    struct Edge {
        int v, c, f;
        Edge(int v, int c, int f) : v(v), c(c), f(f) {}
    };
    const int n;
    vector<Edge> e;
    vector<vector<int>> g;
    vector<LL> h, dis;
    vector<int> pre;

    MinCostFlow(int n) : n(n), g(n) {}
    void add(int u, int v, int c, int f) { // c 流量, f 费用
        // if (f < 0) {
        //     g[u].push_back(e.size());
        //     e.emplace_back(v, 0, f);
        //     g[v].push_back(e.size());
        //     e.emplace_back(u, c, -f);
        // } else {
            g[u].push_back(e.size());
            e.emplace_back(v, c, f);
            g[v].push_back(e.size());
            e.emplace_back(u, 0, -f);
        // }
    }
    bool dijkstra(int s, int t) {
        dis.assign(n, INF);
        pre.assign(n, -1);
        priority_queue<PII, vector<PII>, greater<PII>> que;
        dis[s] = 0;
        que.emplace(0, s);
        while (!que.empty()) {
            auto [d, u] = que.top();
            que.pop();
            if (dis[u] < d) continue;
            for (int i : g[u]) {
                auto [v, c, f] = e[i];
                if (c > 0 && dis[v] > d + h[u] - h[v] + f) {
                    dis[v] = d + h[u] - h[v] + f;
                    pre[v] = i;
                    que.emplace(dis[v], v);
                }
            }
        }
        return dis[t] != INF;
    }
    pair<int, LL> flow(int s, int t) {
        int flow = 0;
        LL cost = 0;
        h.assign(n, 0);
        while (dijkstra(s, t)) {
            for (int i = 0; i < n; ++i) h[i] += dis[i];
            int aug = numeric_limits<int>::max();
            for (int i = t; i != s; i = e[pre[i] ^ 1].v) aug = min(aug, e[pre[i]].c);
```

```
            for (int i = t; i != s; i = e[pre[i] ^ 1].v) {
                e[pre[i]].c -= aug;
                e[pre[i] ^ 1].c += aug;
            }
            flow += aug;
            cost += LL(aug) * h[t];
        }
        return {flow, cost};
    }
};
```

/END/