Maximum flow - MPM algorithm

Table of Contents

MPM (Malhotra, Pramodh-Kumar and Maheshwari) algorithm solves the maximum flow problem in $O(V^3)$. This algorithm is similar to Dinic's algorithm.

Algorithm

Like Dinic's algorithm, MPM runs in phases, during each phase we find the blocking flow in the layered network of the residual network of G. The main difference from Dinic's is how we find the blocking flow. Consider the layered network L. For each node we define its' *inner potential* and *outer potential* as:

$$p_{in}(v) = \sum_{(u,v) \in L} (c(u,v) - f(u,v))$$

$$p_{out}(v) = \sum_{(v,u) \in L} (c(v,u) - f(v,u))$$

Also we set $p_{in}(s) = p_{out}(t) = \infty$. Given p_{in} and p_{out} we define the *potential* as $p(v) = min(p_{in}(v), p_{out}(v))$. We call a node r a *reference node* if $p(r) = min\{p(v)\}$. Consider a reference node r. We claim that the flow can be increased by $p(r)$ in such a way that $p(r)$ becomes 0. It is true because L is acyclic, so we can push the flow out of r by outgoing edges and it will reach t because each node has enough outer potential to push the flow out when it reaches it. Similarly, we can pull the flow from s. The construction of the blocked flow is based on this fact. On each iteration we find a reference node and push the flow from s to t through r. This process can be simulated by BFS. All completely saturated arcs can be deleted from L as they won't be used later in this phase anyway. Likewise, all the nodes different from s and t without outgoing or incoming arcs can be deleted.

Each phase works in $O(V^2)$ because there are at most V iterations (because at least the chosen reference node is deleted), and on each iteration we delete all the edges we passed through except at most V. Summing, we get $O(V^2 + E) = O(V^2)$. Since there are less than V phases (see the proof here), MPM works in $O(V^3)$ total.

Implementation

```
struct MPM{
    struct FlowEdge{
```

```
    int v, u;
    long long cap, flow;
    FlowEdge(){}
    FlowEdge(int _v, int _u, long long _ca
        : v(_v), u(_u), cap(_cap), flow(_f
    FlowEdge(int _v, int _u, long long _ca
        : v(_v), u(_u), cap(_cap), flow(0l
};
const long long flow_inf = 1e18;
vector<FlowEdge> edges;
vector<char> alive;
vector<long long> pin, pout;
vector<list<int> > in, out;
vector<vector<int> > adj;
vector<long long> ex;
int n, m = 0;
int s, t;
vector<int> level;
vector<int> q;
int qh, qt;
void resize(int _n){
    n = _n;
    ex.resize(n);
    q.resize(n);
    pin.resize(n);
    pout.resize(n);
    adj.resize(n);
    level.resize(n);
    in.resize(n);
    out.resize(n);
}
MPM(){}
MPM(int _n, int _s, int _t){resize(_n); s
```

```
void add_edge(int v, int u, long long cap)
    edges.push_back(FlowEdge(v, u, cap));
    edges.push_back(FlowEdge(u, v, 0));
    adj[v].push_back(m);
    adj[u].push_back(m + 1);
    m += 2;
}
bool bfs(){
    while(qh < qt){
        int v = q[qh++];
        for(int id : adj[v]){
            if(edges[id].cap - edges[id].f
            if(level[edges[id].u] != -1)co
            level[edges[id].u] = level[v]
            q[qt++] = edges[id].u;
        }
    }
    return level[t] != -1;
}
long long pot(int v){
    return min(pin[v], pout[v]);
}
void remove_node(int v){
    for(int i : in[v]){
        int u = edges[i].v;
        auto it = find(out[u].begin(), out
        out[u].erase(it);
        pout[u] -= edges[i].cap - edges[i]
    }
    for(int i : out[v]){
        int u = edges[i].u;
        auto it = find(in[u].begin(), in[u
        in[u].erase(it);
```

```
            pin[u] -= edges[i].cap - edges[i].
        }
    }
    void push(int from, int to, long long f, b
        qh = qt = 0;
        ex.assign(n, 0);
        ex[from] = f;
        q[qt++] = from;
        while(qh < qt){
            int v = q[qh++];
            if(v == to)
                break;
            long long must = ex[v];
            auto it = forw ? out[v].begin() :
            while(true){
                int u = forw ? edges[*it].u :
                long long pushed = min(must, e
                if(pushed == 0)break;
                if(forw){
                    pout[v] -= pushed;
                    pin[u] -= pushed;
                }
                else{
                    pin[v] -= pushed;
                    pout[u] -= pushed;
                }
                if(ex[u] == 0)
                    q[qt++] = u;
                ex[u] += pushed;
                edges[*it].flow += pushed;
                edges[(*it)^1].flow -= pushed;
                must -= pushed;
                if(edges[*it].cap - edges[*it]
```

```
                auto jt = it;
                ++jt;
                if(forw){
                    in[u].erase(find(in[u]
                    out[v].erase(it);
                }
                else{
                    out[u].erase(find(out[
                    in[v].erase(it);
                }
                it = jt;
            }
            else break;
            if(!must)break;
        }
    }
}
long long flow(){
    long long ans = 0;
    while(true){
        pin.assign(n, 0);
        pout.assign(n, 0);
        level.assign(n, -1);
        alive.assign(n, true);
        level[s] = 0;
        qh = 0; qt = 1;
        q[0] = s;
        if(!bfs())
            break;
        for(int i = 0; i < n; i++){
            out[i].clear();
            in[i].clear();
        }
```

```
        for(int i = 0; i < m; i++){
            if(edges[i].cap - edges[i].flo
                continue;
            int v = edges[i].v, u = edges[
            if(level[v] + 1 == level[u] &&
                in[u].push_back(i);
                out[v].push_back(i);
                pin[u] += edges[i].cap - e
                pout[v] += edges[i].cap -
            }
        }
        pin[s] = pout[t] = flow_inf;
        while(true){
            int v = -1;
            for(int i = 0; i < n; i++){
                if(!alive[i])continue;
                if(v == -1 || pot(i) < pot
                    v = i;
            }
            if(v == -1)
                break;
            if(pot(v) == 0){
                alive[v] = false;
                remove_node(v);
                continue;
            }
            long long f = pot(v);
            ans += f;
            push(v, s, f, false);
            push(v, t, f, true);
            alive[v] = false;
            remove_node(v);
        }
```

```
            }
            return ans;
        }
    };
```