最大流

Maximum Flow

Ford-Fulkerson 方法

Concepts:

- 剩余容量 Residual Capacity：一条边的容量与流量之差，$c_f(u,v)=c(u,v)-f(u,v)$
- 残量网络 Residual Network：所有剩余容量大于 0 的边的生成子图
- 增广路 Augmenting Path：原图 G 中，一条从源点到汇点的由剩余容量都大于 0 的边构成的路径

Idea：不断寻找增广路直到找不到为止。

Edmonds-Karp

Idea：bfs 寻找增广路。

Complexity：$O(VE^2)$

ATT：链式前向星存储时，`edgeNum` 初始化为1；建图时建流为 0 的反向边。

Code:

```cpp
int pre[N], minFlow[N];
int bfs(){
    queue<int> q;
    for(int i = 1; i <= n; i++){
        pre[i] = 0;
        minFlow[i] = INF;
    }
    q.push(src);
    while(!q.empty()){
        int cur = q.front(); q.pop();
        for(int i = head[cur]; i; i = edge[i].nxt){
            if(edge[i].flow && !pre[edge[i].to]){
                pre[edge[i].to] = i;
                minFlow[edge[i].to] = min(minFlow[cur], edge[i].flow);
                q.push(edge[i].to);
            }
        }
    }
    if(pre[dst] == 0)   return -1;
    return minFlow[dst];
}

int EK(){
    int flow = 0, maxflow = 0;
    while((flow = bfs()) != -1){
        int t = dst;
        while(t != src){
            edge[pre[t]].flow -= flow;
            edge[pre[t]^1].flow += flow;
            t = edge[pre[t]^1].to;
        }
        maxflow += flow;
    }
    return maxflow;
}
```

Dinic

Idea：bfs 将图分层，dfs 按分层图寻找增广路。

Optimization：当前弧优化。

Complexity：$O(V^2E)$

ATT：链式前向星存储时，`edgeNum` 初始化为1；建图时建流为 0 的反向边。

Code（当前弧优化）：

```
namespace FLOW{

    int n, S, T;
    struct Edge{
        int nxt, to;
        LL flow;
    }edge[M<<1];
    int head[N], edgeNum = 1;
    void addEdge(int from, int to, LL flow){
        edge[++edgeNum] = (Edge){head[from], to, flow};
        head[from] = edgeNum;
    }

    bool inq[N];
    int dep[N], curArc[N];
    inline bool bfs(){
        for(int i = 1; i <= n; i++)
            dep[i] = 1e9, inq[i] = 0, curArc[i] = head[i];
        queue<int> q;
        q.push(S);
        inq[S] = 1;
        dep[S] = 0;
        while(!q.empty()){
            int cur = q.front(); q.pop();
            inq[cur] = 0;
            for(int i = head[cur]; i; i = edge[i].nxt){
                if(dep[edge[i].to] > dep[cur] + 1 && edge[i].flow){
                    dep[edge[i].to] = dep[cur] + 1;
                    if(!inq[edge[i].to]){
                        q.push(edge[i].to);
                        inq[edge[i].to] = 1;
                    }
                }
            }
        }
        if(dep[T] != 1e9)   return 1;
        return 0;
    }
    LL dfs(int x, LL minFlow){
        LL flow = 0;
        if(x == T)  return minFlow;
        for(int i = curArc[x]; i; i = edge[i].nxt){
            curArc[x] = i;
            if(dep[edge[i].to] == dep[x] + 1 && edge[i].flow){
                flow = dfs(edge[i].to, min(minFlow, edge[i].flow));
                if(flow){
                    edge[i].flow -= flow;
                    edge[i^1].flow += flow;
                    return flow;
                }
            }
        }
        return 0;
    }
    inline LL Dinic(){
        LL maxFlow = 0, flow = 0;
        while(bfs()){
            while(flow = dfs(S, INF))
                maxFlow += flow;
        }
        return maxFlow;
    }
}
```

ISAP

Idea：依旧是分层，不过只从汇点到源点 bfs 一次，在后续 dfs 的过程中不断更新每个点的层数，从而避免了多次 bfs 来分层。如何更新层数呢？如果 dfs 时能到某点的流量大于它能往下一层流出的流量，说明它要把剩余的流量流出去，就必须增加层数。

Optimization：当前弧优化；GAP 优化。

Complexity：$O(V^2E)$ （但是表现优于 **Dinic**）

ATT：链式前向星存储时，`edgeNum` 初始化为1；建图时建流为 `0` 的反向边。

Code（当前弧优化+GAP优化）：

```
namespace FLOW{

    int n, S, T;
    struct Edge{
        int nxt, to;
        LL flow;
    }edge[M<<1];
    int head[N], edgeNum = 1;
    void addEdge(int from, int to, LL flow){
        edge[++edgeNum] = (Edge){head[from], to, flow};
        head[from] = edgeNum;
    }

    int dep[N], gap[N], curArc[N];
    void bfs(){
        for(int i = 1; i <= n; i++) dep[i] = -1, gap[i] = 0;
        dep[T] = 0, gap[0] = 1;
        queue<int> q;
        q.push(T);
        while(!q.empty()){
            int cur = q.front(); q.pop();
            for(int i = head[cur]; i; i = edge[i].nxt){
                if(dep[edge[i].to] != -1)   continue;
                dep[edge[i].to] = dep[cur] + 1;
                gap[dep[edge[i].to]]++;
                q.push(edge[i].to);
            }
        }
    }
    LL dfs(int x, LL minFlow){
        if(x == T)  return minFlow;
        LL outFlow = 0;
        for(int i = curArc[x]; i; i = edge[i].nxt){
            curArc[x] = i;
            if(dep[edge[i].to] + 1 == dep[x] && edge[i].flow){
                LL flow = dfs(edge[i].to, min(minFlow - outFlow, edge[i].flow));
                if(flow){
                    edge[i].flow -= flow;
                    edge[i^1].flow += flow;
                    outFlow += flow;
                }
                if(outFlow == minFlow)  return minFlow;
            }
        }
        if(--gap[dep[x]] == 0)  dep[S] = n + 1;
        gap[++dep[x]]++;
        return outFlow;
    }
    inline LL ISAP(){
        LL maxFlow = 0;
        bfs();
        while(dep[S] < n){
            for(int i = 1; i <= n; i++) curArc[i] = head[i];
            maxFlow += dfs(S, INF);
        }
        return maxFlow;
    }
}
```

预流推进 Push-Relable

Idea：预流推进算法弱化了流守恒性质，流入节点的流可以大于流出节点的流，多余的流被称作“超额流”。同时每一个节点都有一个高度，我们只能从高处向低处推送流。

定义高度函数，满足对于残存网络中的边 (u,v)，有 $h(u)\leqslant h(v)+1$，且 $h(S)=|V|,h(T)=0$.

我们有两种操作：

- 推送（push）：若节点 u 溢出，且边 $(u,v)\in E_f$ 满足容量 >0，$h(u)=h(v)+1$，则可从 u 推送 $\min(\text{extra}(u),c(u,v))$ 到 v.
- 重贴标签（relabel）：若节点 u 溢出，且对于所有边 $(u,v)\in E_f$，都有 $h(u)\leqslant h(v)$，则可以对 u 重贴标签，即设其高为 $1+\min\limits_{(u,v)\in E_f}h(v)$.

初始化：S 的高度设为 $|V|$，其余为 0；随后将从 S 发出的所有边都充满流，其他边上都没有流。

步骤：选取能进行 push 或 relabel 的节点进行相应操作，当没有能够 push 或 relabel 的节点时算法结束，此时 T 的超额流就是答案。

最高标号预流推进 HLPP

Idea：每次选择高度最高的超额流不为 0 的节点进行推送或重贴标签。使用优先队列维护即可。

Optimization：bfs 优化；GAP 优化。

Complexity：$O(V^2\sqrt{E})$

ATT：链式前向星存储时，`edgeNum` 初始化为1；建图时建流为 0 的反向边。

Code（bfs优化+GAP优化）：

```cpp
namespace FLOW{

    int n, S, T;
    struct Edge{
        int nxt, to;
        LL flow;
    }edge[M<<1];
    int head[N], edgeNum = 1;
    void addEdge(int from, int to, LL flow){
        edge[++edgeNum] = (Edge){head[from], to, flow};
        head[from] = edgeNum;
    }

    LL extra[N], h[N], gap[N];
    inline bool bfs(){
        for(int i = 1; i <= n; i++) h[i] = INF;
        h[T] = 0, gap[0] = 1;
        queue<int> q;
        q.push(T);
        while(!q.empty()){
            int cur = q.front(); q.pop();
            for(int i = head[cur]; i; i = edge[i].nxt){
                if(edge[i^1].flow && h[edge[i].to] == INF){
                    h[edge[i].to] = h[cur] + 1;
                    gap[h[edge[i].to]]++;
                    q.push(edge[i].to);
                }
            }
        }
        return h[S] != INF;
    }
    struct cmp{ bool operator ()(int a, int b){ return h[a] < h[b]; } };
    priority_queue<int, vector<int>, cmp> pq;
    bool inq[N];
    inline void Push(int x){
        for(int i = head[x]; i; i = edge[i].nxt){
            if(h[x] == h[edge[i].to] + 1 && edge[i].flow){
                LL mn = min(edge[i].flow, extra[x]);
                edge[i].flow -= mn, edge[i^1].flow += mn;
                extra[x] -= mn, extra[edge[i].to] += mn;
                if(!inq[edge[i].to] && edge[i].to != T && edge[i].to != S)
                    pq.push(edge[i].to), inq[edge[i].to] = true;
                if(!extra[x])   break;
            }
```

```
        }
    }
    inline void Relabel(int x){
        h[x] = INF;
        for(int i = head[x]; i; i = edge[i].nxt)
            if(edge[i].flow)
                h[x] = min(h[x], h[edge[i].to] + 1);
    }
    inline LL HLPP(){
        if(!bfs())  return 0;
        h[S] = n;
        for(int i = head[S]; i; i = edge[i].nxt){
            if(edge[i].flow){
                extra[S] -= edge[i].flow, extra[edge[i].to] += edge[i].flow;
                edge[i^1].flow += edge[i].flow, edge[i].flow = 0;
                if(!inq[edge[i].to] && edge[i].to != T && edge[i].to != S)
                    pq.push(edge[i].to), inq[edge[i].to] = true;
            }
        }
        while(!pq.empty()){
            int cur = pq.top(); pq.pop();
            inq[cur] = false;
            Push(cur);
            if(extra[cur]){
                if(--gap[h[cur]] == 0)
                    for(int i = 1; i <= n; i++)
                        if(i != S && i != T && h[i] > h[cur] && h[i] < n + 1)
                            h[i] = n + 1;
                Relabel(cur);
                gap[h[cur]]++;
                pq.push(cur), inq[cur] = true;
            }
        }
        return extra[T];
    }
}
```