ISAP

初始化、网络流加流量边

edge_flow同时存储了流量和容量 node_flow是边表的表头，仅存储出边、入边的编号 在E中编号最低位为0代表原边，为1代表虚边,使用reverse根据原边寻址虚边

```
#define reverse(e) ((e) ^ 1)

struct edge_flow{
    edge_flow(int a, int b, llint cap,llint flow) {
        this->u = a;
        this->v = b;
        this->cap = cap;
        this->flow = flow;
    }
    int u, v;
    llint cap;
    llint flow;
};
vector<edge_flow> E;
struct node_flow {
    vector<int> e;
    node_flow() { e = *(new vector<int>);}
};

node_flow nod[MAXN];

void addedg_flow(int a, int b, llint cap)
{
    E.emplace_back(edge_flow(a, b, cap, 0));
    E.emplace_back(edge_flow(b, a, 0, 0));
    int ptr = E.size();
    nod[a].e.emplace_back(ptr - 2);
    nod[b].e.emplace_back(ptr - 1);
}
```

level是节点分层信息

higher_link是直连高一层节点的编号，在非递归寻找增广路回溯时用

gap用于GAP优化，cur用于当前弧优化

```
int level[MAXN]={0};
int higher_link[MAXN]={0};
bool vis[MAXN]={0};
```

```
int gap[MAXN]={0};
int cur[MAXN]={0};
```

复杂度nm^2

```
llint ISAP(int Source, int Target)
{
    s = Source; t = Target;
    llint max_flow = 0;
    //分层，统计gap数组
    BFS_markDepth();
    for (int i = 1; i <= n; i++)    gap[level[i]]++;
    //增广
    int ptr = s;
    while (level[s]<n)
    {
        //抵达源点，结算并修改增广路上的流量
        if (ptr == t)
        {
            max_flow += augment();
            ptr = s;
        }
        //非递归寻找增广路
        //遍历连接当前节点高节点的边
        //cur[ptr]用于当前弧优化
        bool success = false;
        for (int i = cur[ptr]; i < nod[ptr].e.size(); i++)
        {
            edge_flow& e = E[nod[ptr].e[i]];
            if (e.cap > e.flow && level[ptr] - 1 == level[e.v])
            {
                success = true;
                higher_link[e.v] = nod[ptr].e[i];
                cur[ptr] = i;
                ptr = e.v;
                break;
            }
        }
        //当前节点增广结束，维护当前节点的深度信息
        if (!success)
        {
            int new_level = n - 1;//若没有出边层次为n-1
            for (auto i : nod[ptr].e)
            {
                edge_flow& e = E[i];
                if (e.cap > e.flow) new_level = min(new_level, level[e.v]);
            }
            if (--gap[level[ptr]] == 0) break;//出现gap
            level[ptr] = new_level + 1;
            gap[level[ptr]]++;
            cur[ptr] = 0;//相当于dinic重新分层，所有边再次可用
                if (ptr != s)   ptr = E[higher_link[ptr]].u;
```

```
        }
    }
    return max_flow;
}

//bfs分层
bool BFS_markDepth()
{
    queue<int>q;
    q.push(t);  vis[t] = 1; level[t] = 0;
    while (!q.empty())
    {
        int p = q.front();  q.pop();
        for (auto x : nod[p].e)
        {
            edge_flow& e = E[reverse(x)];
            if (!vis[e.u] && e.cap > e.flow)
            {
                vis[e.u] = true;
                level[e.u] = level[e.v] + 1;
                q.push(e.u);
            }
        }
    }
    return vis[s];
}

//在找到增广路后增广（路径信息保存在higher中）
llint augment()
{
    int ptr = t;
    llint remain_flow = 9999999999999999999;
    //沿分层向前访问，寻找增广路及增广路流量
    while (ptr != s)
    {
        edge_flow& e = E[higher_link[ptr]];
        remain_flow = min(remain_flow, e.cap - e.flow);
        ptr = e.u;
    }
    //再次遍历该路径，减掉增加流量并维护层数
    ptr = t;
    while (ptr!=s)
    {
        E[higher_link[ptr]].flow += remain_flow;//反向边流量+
        E[reverse(higher_link[ptr])].flow -= remain_flow;//反向边流量-
        ptr =  E[higher_link[ptr]].u;
    }
    return remain_flow;
}
```