P3376 【模板】网络最大流

2025年11月12日 19:55

P3376 【模板】网络最大流

提交答案 | 加入题单 | 复制题目

提交	通过	时间限制	内存限制
239.77k	104.89k	1.00s	512.00MB

题目描述

复制 Markdown 折叠 进入 IDE 模式

如题，给出一个网络图，以及其源点和汇点，求出其网络最大流。

输入格式

第一行包含四个正整数 n, m, s, t，分别表示点的个数、有向边的个数、源点序号、汇点序号。

接下来 m 行每行包含三个正整数 u_i, v_i, w_i，表示第 i 条有向边从 u_i 出发，到达 v_i，边权为 w_i（即该边最大流量为 w_i）。

输出格式

一行，包含一个正整数，即为该网络的最大流。

输入输出样例

输入 #1 复制

```
4 5 4 3
4 2 30
4 3 20
2 3 20
2 1 30
1 3 30
```

输出 #1 复制

```
50
```

【Ford_Fulkerson算法】

时间复杂度O(Fn)，其中F表示最大流，n表示顶点个数，因为每找到一条增广路径，使得最大流至少增加1，最多增加F次，每次dfs时间复杂度为O(n)。在本题中，F的最大值会到大int的最大值，会超时 。

dfs+邻接表

【代码实现】

```
#include<bits/stdc++.h>
using namespace std;
// soruce:源点  sink:终点  capacity:容量  residual:残余 flow:流量  naive:朴素的(幼稚的，原始的)
const int oo = (1ll<<31)-1;
const int M = 5e3;
const int N = 2e2;
typedef long long ll;
int n,m,s,t;
struct node{
	int to,residual,next;
	int rev;//反向边的编号
}e[2*M+5];
int head[N+5],cnt;
void add(int u,int v,int cap){
	e[++cnt] = { v,cap,head[u],cnt+1 }; head[u] = cnt;
	e[++cnt] = { u,0,head[v],cnt-1 }; head[v] = cnt;
}
bool vis[N];
//通过dfs寻找一条从源点到终点的增广路径，并返回路径中最小边权minflow
int dfs(int u,int flow){
	vis[u] = 1;
	if( u==t ){
		return flow;
	}
	for(int i=head[u];i;i=e[i].next){
		int pre = i;
		if( !vis[ e[i].to ] ){
			if( e[i].residual > 0  ){
```

```
                        int minflow = dfs( e[i].to , min( flow , e[i].residual ) );
                        if( minflow>0 ){
                                e[i].residual -= minflow;
                                e[e[i].rev].residual += minflow;
                                return minflow;
                        }
                }
                else{
                        e[pre].next = e[i].next;//删边
                }
        }
    }
    return 0;
}
ll ford_fulkerson(){
    ll maxflow = 0;
    while( 1 ){
        memset( vis , 0 ,sizeof vis);
        int flow = dfs(s,oo);
        if( !flow ) break;
        maxflow += flow;
    }
    return maxflow;
}
int main(){
    cin>>n>>m>>s>>t;
    for(int i=1;i<=m;i++){
        int u,v,cap;
        cin>>u>>v>>cap;
        add(u,v,cap);
    }
    cout << ford_fulkson();
    return 0;
}
```

【Edmonds_Karp算法】

Edmonds-Karp算法，其常规时间复杂度为O(VE^2)，其中V是图中顶点数，E是边数。
该算法是Ford_Fulkson算法的优化，通过广度优先搜索（BFS）寻找最短（即最少边数）增广路径，每次增广操作的时间复杂度为O(E)，而总增广次数上限为O(VE)。

bfs+邻接表

【代码实现】

```
#include<bits/stdc++.h>
using namespace std;
// soruce:源点  sink:终点  capacity:容量  residual:残余 flow:流量  naive:朴素的(幼稚的，原始的)
const int oo = (1ll<<31)-1;
const int M = 5e3;
const int N = 2e2;
typedef long long ll;
int n,m,s,t;
struct node{
    int to,residual,next;
}e[2*M+5];
int head[N+5];
void add(int u,int v,int cap,int id){
    e[id] = { v,cap,head[u] }; head[u] = id;
//      当i为偶数时，i^1 = i+1 ，(i+1)^1 = i. 方便找相反边
    e[id^1] = { u,0,head[v] }; head[v] = id^1;
}
int pre[N]; //pre[i]:记录pre[i]号边到达i号点，同时可以找到pre[i]的反向边 pre[i]^1
int flow[N];//flow[i]:记录每轮到达点i的最大流
int bfs(){
    memset(pre,-1,sizeof pre);
    flow[s] = oo;
    queue<int> q;
    q.push(s);
    while( q.size() ){
        int u = q.front();q.pop();
        if( u == t ){
            return flow[t];
        }
        for(int i=head[u];i!=-1;i=e[i].next){
            int v = e[i].to ;
            if( v!=s && pre[v]==-1 && e[i].residual>0 ){ //注意起点pre[s]恒为-1，需要特别判断v是否为起点s
                q.push( v );
                pre[v] = i;
                flow[v] = min( flow[u],e[i].residual );
            }
```

```
            }
        }
        return 0;
}
ll edmonds_karp(){
        ll maxflow = 0;
        while( 1 ){

                int flow = bfs();
                if( !flow ) break;
                maxflow += flow;

                //对当前增广路径的边进行调整
                int cur = pre[t];
                while( cur!=-1 ){
                        e[cur].residual -= flow;
                        e[cur^1].residual += flow;
                        cur = pre[e[cur^1].to];
                }
        }
        return maxflow;
}
int main(){
        cin>>n>>m>>s>>t;
        memset( head,-1,sizeof head );
        for(int i=0;i<m;i++){
                int u,v,cap;
                cin>>u>>v>>cap;
                add(u,v,cap,i<<1);
        }
        cout << edmonds_karp();
        return 0;
}
```

bfs+邻接矩阵

【代码实现】

```
#include<iostream>
#include<queue>
#include<cstring>
using namespace std;
typedef long long ll;
const int INF = 0x7fffffff;
const int maxn = 205;
int n,m,s,t;// n: 点的个数;  m: 有向边的个数; s:源点, t:终点
ll graph[maxn][maxn],pre[maxn];
ll bfs(int s,int t){
        ll flow[maxn];
        memset(pre,-1,sizeof(pre));
        flow[s] = INF;pre[s] = 0;
        queue<int> q;
        q.push(s);
        while(!q.empty()){
                int u = q.front();
                q.pop();
                if( u==t ) break;
                for( int i=1;i<=n;i++ ){
                        if( i!=s && graph[u][i]>0 && pre[i]==-1 ){
                                pre[i] = u;
                                q.push(i);
                                flow[i] = min( flow[u],graph[u][i] );
                        }
                }
        }
        if( pre[t]==-1 ) return -1;
        return flow[t];
}
ll maxflow(int s,int t){
        ll Maxflow=0;
        while(true){
                ll flow = bfs(s,t);
                if( flow==-1 ) break;
                int cur = t;
                while( cur!=s ){ // 将增广路径中的每条边容量减去最小值flow
                        int father = pre[cur];
                        graph[father][cur] -= flow;
                        graph[cur][father] += flow;
```

```
                cur = father;
            }
            Maxflow+=flow;
        }
        return Maxflow;
}
int main(){
        cin>>n>>m>>s>>t;
        for(int i=1;i<=m;i++){
                int ui,vi,wi;
                cin>>ui>>vi>>wi;
                graph[ui][vi]+=wi;//可能存在重边
        }
        cout<< maxflow(s,t) <<endl;
        return 0;
}
```

【Dinic算法】

Dinic算法（又称Dinitz算法）是一个在网络流中计算最大流的强多项式复杂度的算法，设想由以色列（前苏联）的计算机科学家Yefim (Chaim) A. Dinitz在1970年提出。

时间复杂度： （ ^{2} ）。

EK 算法找增广路径是基于 bfs 来进行的， bfs 会把周围能够扩展的点全部扩展进来，直到找到汇点为止，相当于每找一次增广路径都要搜索大量的点。 Dinic 算法实际上是对 EK 的优化。

Dinic 算法利用 bfs 建立分层网络（按照每个点到源点的距离进行分层）。然后基于这个分层网络，利用 dfs 找到当前分层网络下的所有增广路径，并做好相应的修改。之后不断重复这两个过程，直到无法分层为止，这样做只需要一次 bfs 就可以找到一簇增广路径。

所谓分层网络，就是利用 bfs 特性，以源点为起点，一直向外扩散。每经过一个点都打一个标记，标记到源点的路径所经过的最少的边的数量，假设用 dist[i] 表示 i 到源点 s 的所经过的边的数量，这样就可以将整个网络分层。基于这个分层网络，dfs 在找增广路时，就可以找到最短的增广路径。如果当前点是 x，dfs 搜到下一个点 y，一定要满足 dist[y] = dist[x] + 1，这样才是最短的增广路径，效率才是最高的。

过程：

- 采用邻接矩阵（或邻接表）存储图。
- 利用 bfs 建立分层网络（记录每个节点的深度）。
- 按照当前分层网络进行 dfs （一层一层找），找到所有该分层网络下的增广路径，并更新残余网络）。
- 重复 1，2 直到无法建立分层网络为止。

【Dinic算法代码实现】

```
#include<bits/stdc++.h>
using namespace std;
// soruce:源点  sink:终点  capacity:容量  residual:残余 flow:流量  naive:朴素的(幼稚的, 原始的)
const int INF = (1ll<<31)-1;
const int M = 5e3;
const int N = 2e2;
typedef long long ll;
int n,m,s,t;
struct node{
        int to,residual,next;
}e[2*M+5];
int head[N+5];
int cnt;
void add(int u,int v,int cap){
        e[cnt] = { v,cap,head[u] }; head[u] = cnt; ++cnt;
        e[cnt] = { u,0,head[v] }; head[v] = cnt; ++cnt; // 当i为偶数时, i^1 = i+1 , (i+1)^1 = i. 方便找相反边
}
//初始化建图
inline void _init(){
        ios::sync_with_stdio(0);
        cin>>n>>m>>s>>t;
        memset( head,-1,sizeof head );
        for(int i=0;i<m;i++){
                int u,v,cap;
                cin>>u>>v>>cap;
                add(u,v,cap);
        }
}
int dep[N];
int bfs(){
        for(int i=1;i<=n;i++) dep[i] = INF;
        queue<int> q;
```

```
        q.push(s);
        dep[s] = 0;
        while( q.size() ){
                int u = q.front();q.pop();
                if( u == t ){
                        return 1;
                }
                for(int i=head[u];i!=-1;i=e[i].next){
                        int v = e[i].to ;
                        if( dep[v]==INF && e[i].residual>0 ){
                                q.push( v );
                                dep[v] = dep[u]+1;
                        }
                }
        }
        return 0;
}
//在对图进行bfs分层后，可以找到一簇最短路径的增广路径，
ll dfs(int u,int flow){
        if( u==t ){
                return flow;
        }
        ll sum_flow = 0 , min_flow;
        for(int i=head[u];i!=-1&&flow>0 ;i=e[i].next){
                int v = e[i].to ;
                if( dep[v] == dep[u]+1 && e[i].residual>0 ){ // 第63行
                        min_flow = dfs( v,min(flow,e[i].residual) );
                        if( min_flow==0 ) dep[v] = INF; //剪枝优化，63行条件dep[v]==dep[u]+1将不再成立
                        else{
                                e[i].residual -= min_flow;
                                e[i^1].residual += min_flow;
                                sum_flow += min_flow;
                                flow -= min_flow;
                        }
                }
        }
        return sum_flow;
}
ll Dinic(){
        ll maxflow = 0;
        while( bfs() ) maxflow += dfs(s,INF);
        return maxflow;
}
int main(){
        _init();
        cout << Dinic();
        return 0;
}
```