网络流初步

一个网络$G=(V,E)$是一张有向图，图中每条有向边$(x,y)\in E$都有一个给定的权值$c(x,y)$，称为边的**容量**。图中还有两个节点S和T，源点和汇点。
网络的流函数：$f(x,y)$具有一下特性：
1、容量限制，$f(x,y)\le c(x,y)$
2、斜对称，$f(x,y)=-f(y,x)$
3、流量守恒，$x\ne S \,\&\, x\ne T, \sum_{(u,x)\in E} f(u,x)=\sum_{(x,v)\in E} f(x,v)$
称$f(x,y)$为边的流量，则$c(x,y)-f(x,y)$为边的剩余流量，对于每条边，都有一个反向边，且反向边的流量是负流量。

最大流

使得整张网络的$\sum_{(S,v)\in E} f(S,v$最大的流函数被称为网络的最大流，此时流量被称为网络流的最大流量。

利用最大流求二分图匹配数量

可以新增一个S节点和一个T节点，从S出发连接每个左部节点，原来的每条边看做从左部节点连接到右部节点的有向边，从每个右部节点出发连T，所有边的容量都为1。求出的最大流量就是二分图最大匹配数。
在允许多重匹配的情况下，可以将从S到左部节点的边容量设为匹配上限，右部节点到T的边容量设为匹配上限。

Edmonds-Karp 增广路算法

若一条从源点到汇点的路径上，各边剩余容量都大于0，则这条路是一条增广路。那么可以利用这条增广路使网络流增大。
增广路就是每次BFS寻找增广路，找到流量为e的增广路之后，就更新路径上每条正向边剩余流量$-e$，反向边流量$+e$。直到找不到增广路，算法结束。
时间复杂度为$O(nm^2)$，实际使用远远达不到这个数值，可以处理10^3~10^4规模的图。
P3376 【模板】网络最大流

```
#include<bits/stdc++.h>
using namespace std;
const int MAXN = 205;
const int MAXM = 5e5 + 10;
#define int long long
const int INF = INT_MAX;
int head[MAXN], ver[MAXM << 1], nxt[MAXM << 1], edge[MAXM << 1], tot = 1;
inline void add(const int &x, const int &y, const int &z)
{
	ver[++tot] = y;
	edge[tot] = z;
	nxt[tot] = head[x];
	head[x] = tot;
}
int n, m, s, t, maxflow;
int v[MAXN], incf[MAXN], pre[MAXN];
bool bfs()
{
	memset(v, 0, sizeof(v));
	queue<int> q;
	q.push(s);
	v[s] = 1;
	incf[s] = INF;//增广路上边的最小容量
	while(q.size())
	{
		int x = q.front();
		q.pop();
		for(int i = head[x]; i; i = nxt[i])
		{
			if(edge[i])
			{
				int y = ver[i];
				if(v[y])
					continue;
				incf[y] = min(incf[x], edge[i]);//更新最小容量
				pre[y] = i;//记录前驱，用于更新
				q.push(y);
				v[y] = 1;
				if(y == t)
					return true;
			}
		}
	}
	return false;
}
void update()
/*更新增广路上边和反向边的容量*/
{
	int x = t;
    //利用前驱遍历增广路
	while(x != s)
	{
		int i = pre[x];
		edge[i] -= incf[t];//正向边-e
		edge[i ^ 1] += incf[t];//反向边+e
		x = ver[i ^ 1];//利用了成对存储的技巧
	}
	maxflow += incf[t];
}
signed main()
{
	cin >> n >> m;
	cin >> s >> t;
	for(int i = 1; i <= m; ++i)
	{
		int x, y, z;
		cin >> x >> y >> z;
		add(x, y, z);
		add(y, x, 0);
	}
	while(bfs())
		update();
	cout << maxflow << endl;
}
```

Dinic算法

残量网络：在任意时刻，网络中所有节点以及剩余容量大于0的边构成的子图被称为**残量网络**，Edmonds-Karp每一次bfs遍历了整个残量网络，但是只找一个增广路，不优。
分层图：$d[y] = d[x] + 1$的边(x, y)构成的子图就是一张分层图
Dinic算法过程如下：
1、bfs残量网络，构造分层图
2、dfs分层图，寻找增广路并更新剩余容量。优点在于dfs的时候可以同时寻找多条增广路，回溯时可以更新剩余容量。（dfs的时候有一些剪枝）
算法复杂度$O(n^2m)$，实际处理10^4~10^5的数据，并且在求解二分图最大匹配的时候复杂度为$O(m\sqrt{n})$

```
#include<bits/stdc++.h>
using namespace std;
#define int long long
const int MAXN = 205;
const int MAXM = 5e3 + 10;
int head[MAXN], nxt[MAXM << 1], ver[MAXM << 1], edge[MAXM << 1], tot = 1;
int d[MAXN];
int n, m, s, t, maxflow;
inline void add(const int &x, const int &y, const int &z)
{
	ver[++tot] = y;
	edge[tot] = z;
	nxt[tot] = head[x];
	head[x] = tot;
}
bool bfs()
/*bfs构造分层图*/
{
	queue<int> q;
	memset(d, 0, sizeof(d));
	q.push(s);
	d[s] = 1;
	while(q.size())
	{
		int x = q.front();
		q.pop();
		for(int i = head[x]; i; i = nxt[i])
		{
			int y = ver[i];
			if(edge[i] && !d[y])
			{
				q.push(y);
				d[y] = d[x] + 1;
				if(y == t)
					return true;
			}
		}
	}
	return false;
}
int dinic(int x, int flow)
/*dfs寻找多条增广路*/
{
	if(x == t)
		return flow;
	int rest = flow, k;
	for(int i = head[x]; i && rest; i = nxt[i])
	{
		int y = ver[i];
		if(edge[i] && d[y] == d[x] + 1)
		{
			k = dinic(y, min(rest, edge[i]));
			if(!k)//后续没有增广路，直接从分层图中去掉该节点，剪枝
				d[ver[i]] = 0;
			edge[i] -= k;
			edge[i ^ 1] += k;
			rest -= k;
		}
	}
	return flow - rest;
}
signed main()
{
	cin >> n >> m >> s >> t;
	for(int i = 1; i <= m; ++i)
	{
		int x, y, z;
		cin >> x >> y >> z;
		add(x, y, z);
		add(y, x, 0);
	}
	int flow = 0;
	while(bfs())
		while(flow = dinic(s, LONG_LONG_MAX))
			maxflow += flow;
	cout << maxflow << endl;
}
```

其他例题:

P2740 [USACO4.2]草地排水Drainage Ditches

还有一些其它算法等待学习:

Ford-Fulkerson（FF）：以为是很高级的其实是不分层图的Dinic（或者dfs的EK）

ISAP(Improved Shortest Augmenting Path，更优最短增广路径算法)：优化了求分层图的过程

HLPP算法：不依赖增广路，是一种预流推进算法