网络流

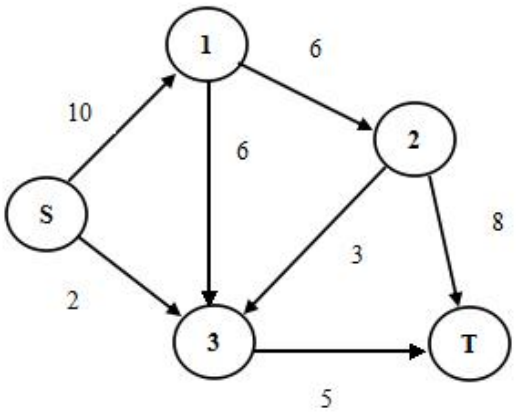
一、模型引入

流模型

如同我们可以把一个实际的道路地图抽象成一个有向图来计算两点之间的最短路径，我们也可以将一个有向图看作一个网流络来解决另一类型的问题。网络流比较适合用来模拟液体流经管道、电流在电路网络中的运动、信息网络中信息的传递等等类似的过程。

我们可以将网络流类比成水流，如下图给出一个有向图 $G=(V,E)$.图中的边可以想象成管道，顶点则是管道的连接点。边上的权值我们称之为这条边的**流量上限**，可以把它想象成这条管道能通过的水流的多少。对于图中每条边 $e\in E$ 我们记它的流量上限为 $c(e)$.

（这里还并没有引入源点汇点等概念，尽管图中有 s,t 点。）



记每条边 $e\in E$ 流过的**流量**为 $f(e)$.

没有源汇，水流就不会凭空产生也不会凭空消失。因此若在这个水流图中出现了一条循环流动的水流，则这个水流必定满足下列条件：

(1) **流量约束条件**：每条边流过的流量小于它的流量上限，即

$$\forall e, 0\le f(e)\le c(e)$$

(2) **流量平衡条件**：流入每个顶点的流量与从这个顶点流出的流量相等，即

$$\forall v, \sum_{e=(u,v)\in E} f(e)=\sum_{e=(v,w)\in E} f(e)$$

我们称满足这两个条件的一个流方案，即使得每条边都满足流量约束，每个顶点满足流量平衡的方案为一条**无源汇网络的可行(循环)流**。

最大流模型

如上图中所示，图中有两个特殊的结点为**源点** s 和**汇点** t，这样原图就变为了有源汇的网络流图。在这样的图中，源点入度为 0 且可以产生无穷大的流量，汇点出度为 0 且可以吸收无穷大的流量。

称一条从源点到汇点的，并使除源汇之外所有结点满足流量平衡，所有边满足流量约束且满足

$$\sum_{e=(s,v)\in E} f(e) = \sum_{e=(v,t)\in E} f(e)$$

的流方案为**有源汇网络的可行流**。(这里的流方案并不一定只沿一条路流，也可能在多个结点上出现支路，但要保证满足流量约束与流量平衡条件。)

我们称使得从 s 出发流出的流量：

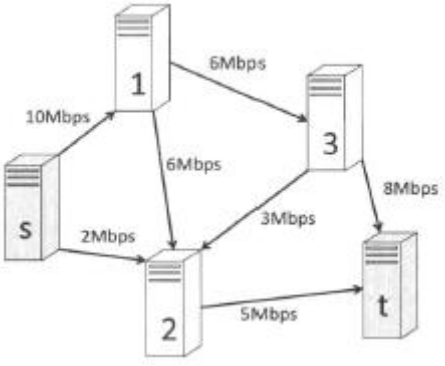
$$\sum_{e=(s,v)\in E} f(e)$$

最大的可行流方案为**最大流**。求解最大流的问题称为最大流问题。

给定一个有向图 $G=(V,E)$，把图中的边看作管道，每条边上有一个权值，表示该管道的流量上限(单位时间内通过的水流量上限)。给定源点 s 和汇点 t，现在假设在 s 处有一个水源，t 处有一个蓄水池，问从 s 到 t 的(单位时间内)最大水流量是多少，类似于这类的问题都可归结为最大流问题。

例如上图的问题可以归结为：

网络中有两台计算机 s 和 t，现在想从 s 传输数据到 t。该网络中一共有 N 台计算机，其中一些计算机之间连有一条单向的通信电缆，每条通信电缆都有对应的 1 秒钟内所能传输的最大数据量。求在 1 秒钟内 s 最多可以传送多少数据到 t?



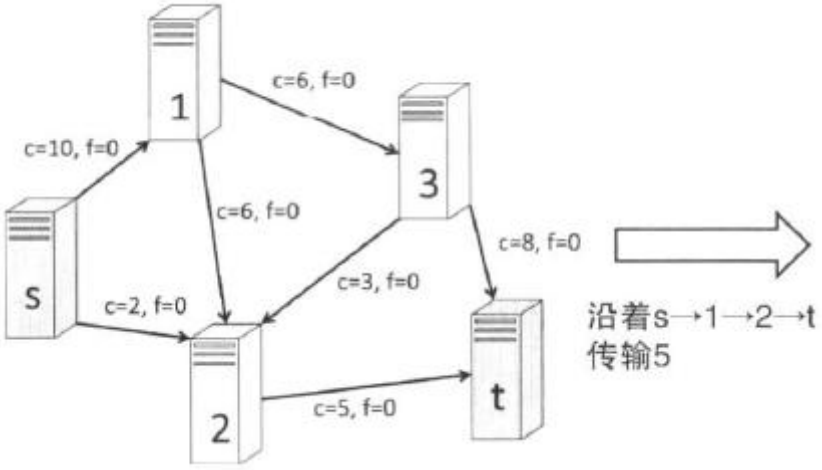
二、求解最大流

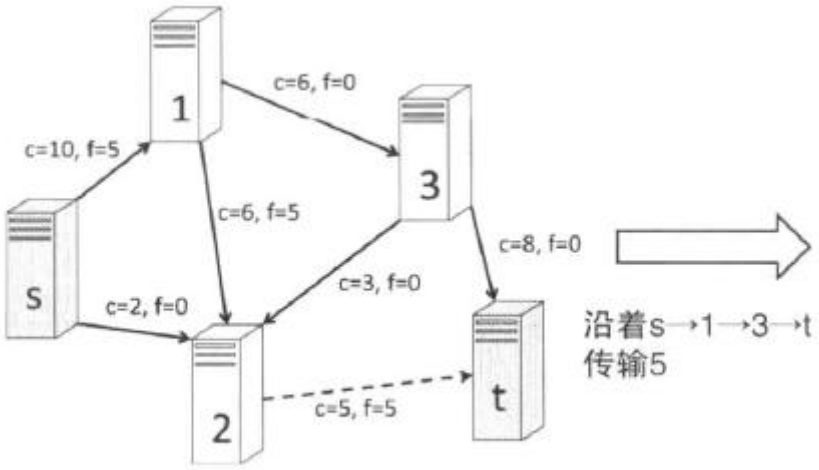
我们首先考虑下面的这个贪心算法：

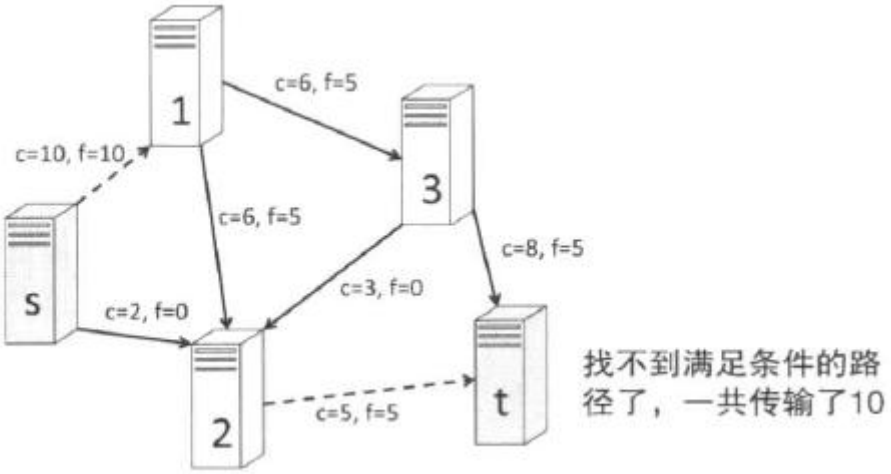
(1) 找一条 s 到 t 的只经过 $f(e)<c(e)$ 的边的路径。

(2) 如果不存在满足条件的路径，则算法结束。否则，沿着该路径尽可能地增加 $f(e)$，返回第(1)步。这一步骤称作增广。

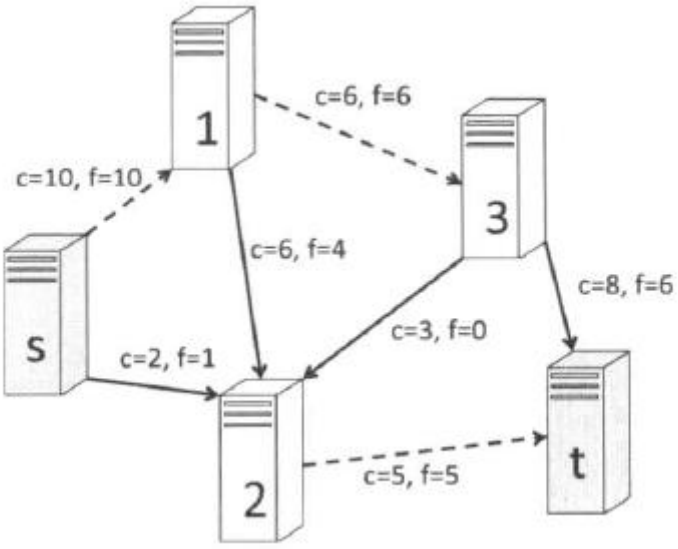
该算法用于上图，得到如下结果：



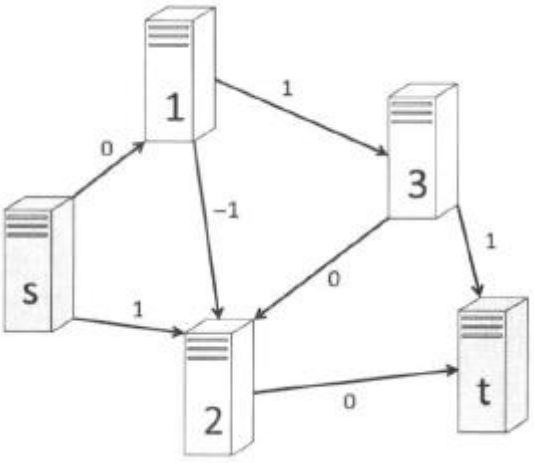




那么这样得到的是最大流吗？事实上采用下图的方案更优，可知这个贪心是不正确的。



为了找出二者的区别，不妨来看看它们的流量差。



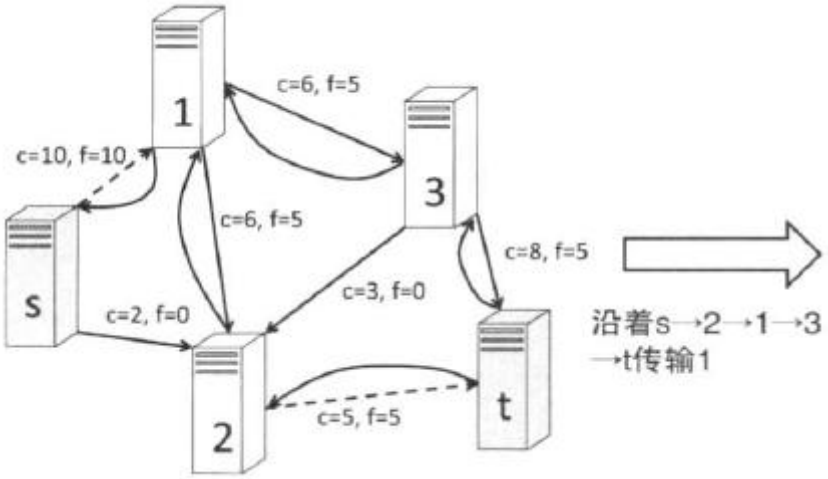
通过对流量差的观察可以发现，我们通过将原先得到的流给退回去(图中的-1)，而得到

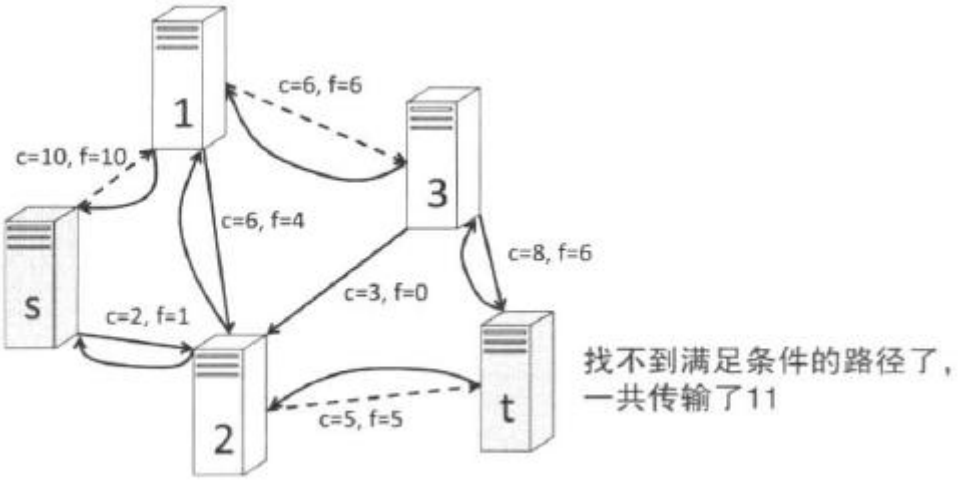
新的流。即退流操作。因此可以试着在之前的贪心算法中加上这一步骤，算法改进如下：

(1) 只利用满足 $f(e)<c(e)$ 的 e 或 $f(e)>0$ 的 e 的反向边 $rev(e)$，寻找一条 s 到 t 的路径。

(2) 如果不存在满足条件的路径，则算法结束。否则，沿着该路径尽可能地增加流，返回(1)。

将改进后的贪心算法用于样例(原算法增广两次后)：





这个算法总能求得最大流，我们将会在后面证明。

我们称在(1)中所考虑的 $f(e)<c(e)$ 的 e 和满足 $f(e)>0$ 的 e 的反向边 $rev(e)$ 所组成的图为残余(残量)网络，并称残余网络上的 $s-t$ 路径为增广路。

不断在残余网络上找到增广路并进行增广直至不存在增广路，每次增广所产生的新流量之和即为最大流。

三、最小割

为了证明上述贪心算法所求得的确实是最大流，我们首先介绍割这一概念。

所谓图的割，指的是对于某个顶点集合 $P \subset V$ ，从 P 出发指向 P 外部的那些原图中的边的集合，记为割 $(P, V \setminus P)$.这些边的容量之和被称为割的容量。如果有 $s \in P$ ，而 $t \in V \setminus P$ ，那么此时的割又称为 $s-t$ 割。

如果将网络中 $s-t$ 割所包含的边都删去，也就不再有从 s 到 t 的路径了。因此可以考虑一下如下问题：对于给定网络，为了保证没有从 s 到 t 的路径，需要删去的边的总容量的最小值是多少？ 该问题即为最小割问题。

首先我们考虑一下任意的 $s-t$ 流 F 和任意的 $s-t$ 割 $(P, V \setminus P)$，因为有

$$F\text{的流量} = \text{源点}s\text{的出边的总流量}$$

而对 $v \in P \setminus \{s\}$ ，又有

$$v\text{的出边的总流量} = v\text{的入边的总流量}$$

所以有

$$F\text{的流量} = (s+v)\text{出边总流量} - (s+v)\text{入边总流量} = P\text{出边总流量} - P\text{入边总流量}$$

由此可知

$$F\text{的流量} \le \text{割的容量}.$$

接下来我们考虑通过上述贪心算法所求得的流 F 。记流 F 对应的残余网络中从 s 可达的顶点 v 组成的集合为 S ，因为 F 对应的残余网络中不存在 $s-t$ 路径，因此 $(S, V \setminus S)$ 就是一个 $s-t$ 割。此外，根据 S 的定义，对包含在割中的边 e 应该有 $F(e) = c(e)$ ，而对 $V \setminus S$ 到 S 的边 e 应该有 $F(e) = 0$ 。因此

$$F\text{的流量} = S\text{出边总流量} - S\text{入边总流量} = \text{割的容量}$$

再由之前的不等式可以知道， F 即是最大流。同时 $(S, V \setminus S)$ 这个割是最小割。(若它不是最小割，则 F 流量等于当前割并大于最小割，那么与上面的不等式矛盾).

同时这也是一个重要的性质：**最大流等于最小割**。

从上面的算法也可以看出，若边的容量都是整数，则最大流与最小割也会是整数。

四、Dinic 算法

朴素的实现上面所述的求解最大流的算法叫做 Ford-Fulkerson 算法。设求解的最大流的流量为 F，则它最多会进行 F 次增广，其复杂度为 O(F|E|).不过这个上界很松，所以实际效率还是比较快的。

事实上，还有许许多多不同的求解最大流问题的算法，它们主要有两类：增广路算法和预流推进算法。最大流算法有很多，它们有不同的复杂度，不同的优缺点和对不同的图不同的实际运行效率。竞赛中常用 SAP 与 Dinic 两种最大流算法。下面来介绍一下 Dinic 算法。

Dinic 算法的主要思想就是每次寻找最短的增广路，并沿着它增广。因为最短增广路的长度在增广过程中始终不会变短，所以无需每次都通过广搜来寻找最短增广路。我们可以先进行一次广搜，然后考虑由近距离顶点指向远距离顶点的边所组成的分层图，在上面进行深搜寻找最短增广路(这里一次深搜就可以完成多次增广的工作)。如果在分层图上找不到新的增广路了(此时我们得到了分层图所对应的阻塞流)，则说明最短增广路的长度变长了或不存在增广路了，于是重新构造新的分层图。

设网络流图中顶点个数为 n 边数为 m，每一步构造分层图的复杂度为 $O(m)$，而每一步完成之后最短增广路长度至少增加 1，由于增广路长度不会超过 $n-1$，因此构造分层图次数最多为 $O(n)$ 次。每次对分层图进行深搜寻找增广路时，若避免对无用边进行多次检查(这步优化称为当前弧优化)，则可以保证复杂度为 $O(nm)$，总时间复杂度即为 $O(n^2m)$.不过在实际应用中算法复杂度远达不到这个上界，很多时候即便图的规模很大也没有问题(常常能用来跑 10^4 甚至 10^5 级别的图).当图是一张二分图时(s->X 部分点->Y 部分点->t)，Dinic 算法的效率能达到 $O(\sqrt{n}m)$.

模板代码：

```
// N 为点数，M 为边数，INF 代表一个极大值
const int N = 1000;
const int M = 10000;
const int INF = 0x3f3f3f3f;

// src 为源点,des 为汇点,lev 为分层后点的距离标号,cur 为当前弧优化所用的临时数组
int src, des, lev[N], cur[N];
// ecnt 从 2 开始计数，这样 e 的反向边为 e^1 方便处理
// cap 表示这条边剩余的容量
int ecnt = 1, adj[N], nxt[M], go[M], cap[M];

// 连一条 u->v 容量为 w 的边
```

```cpp
inline void addEdge(const int &u, const int &v, const int &w) {
  nxt[++ecnt] = adj[u], adj[u] = ecnt, go[ecnt] = v, cap[ecnt] = w;
  nxt[++ecnt] = adj[v], adj[v] = ecnt, go[ecnt] = u, cap[ecnt] = 0;
}

// 将图分层
inline bool Bfs() {
  static int qN, que[N];
  int u, v, e;
  // 存储的时候源点编号最小，汇点编号最大，因此初始化可以这样枚举进行
  // 这不是必须的，可以根据个人习惯更改
  for (int i = src; i <= des; ++i) lev[i] = -1, cur[i] = adj[i];
  que[qN = 1] = src, lev[src] = 0;
  for (int ql = 1; ql <= qN; ++ql) {
    u = que[ql];
    for (e = adj[u]; e; e = nxt[e]) {
      if (cap[e] > 0 && lev[v = go[e]] == -1) {
        lev[v] = lev[u] + 1, que[++qN] = v;
        if (v == des) return true; // 当前还存在增广路
      }
    }
  }
  return false;
}

// 从 u 点往外尝试流出 flow 的流量，返回的是最后成功流出的流量
inline int Dinic(const int &u, const int &flow) {
  if (u == des) return flow;
  int res = 0, v, delta;
  for (int &e = cur[u]; e; e = nxt[e]) {
    // 这里 e 是一个引用，这样随着 e=nxt[e],cur[u]也会等于 nxt[e],那么原来 cur[u]那条边就不会再被重复检查
    if (cap[e] > 0 && lev[u] < lev[v = go[e]]) {
      delta = Dinic(v, std::min(cap[e], flow - res));
      if (delta) {
        cap[e] -= delta; cap[e ^ 1] += delta;
        res += delta; if (res == flow) break;
        // 这里没有直接返回，它实际上体现了一种多路增广的思想
      }
    }
  }
  if (res != flow) lev[u] = -1;
  // 当前有流无法增广完，那么当前分层图上这个点永远都不能进行增广了，所以将距离标号设成-1 以后不再会访问它
```

```
  return res;
}

inline int maxFlow() {
  int ans = 0;
  while (Bfs()) ans += Dinic(src, INF);
  // 不停增广直至没有增广路
  return ans;
}
```

五、例题

Drainage Ditches (POJ 1273)

题目大意：

给定一张 n 个点 m 条边的网络流图，求从 1 到 n 的最大流。

n,m <= 200

分析：

模板题。

代码：

```
#include <cstdio>
#include <algorithm>

const int N = 205;
const int M = 405;
const int INF = 0x3f3f3f3f;

int n, m;
int src, des, lev[N], cur[N];
int ecnt, adj[N], nxt[M], go[M], cap[M];

inline void addEdge(const int &u, const int &v, const int &w) {
  nxt[++ecnt] = adj[u], adj[u] = ecnt, go[ecnt] = v, cap[ecnt] = w;
  nxt[++ecnt] = adj[v], adj[v] = ecnt, go[ecnt] = u, cap[ecnt] = 0;
}

inline bool Bfs() {
  static int qN, que[N];
  int u, v, e;
  for (int i = src; i <= des; ++i) lev[i] = -1, cur[i] = adj[i];
  que[qN = 1] = src, lev[src] = 0;
```

```
  for (int ql = 1; ql <= qN; ++ql) {
    u = que[ql];
    for (e = adj[u]; e; e = nxt[e]) {
      if (cap[e] > 0 && lev[v = go[e]] == -1) {
        lev[v] = lev[u] + 1, que[++qN] = v;
        if (v == des) return true;
      }
    }
  }
  return false;
}

inline int Dinic(const int &u, const int &flow) {
  if (u == des) return flow;
  int res = 0, v, delta;
  for (int &e = cur[u]; e; e = nxt[e]) {
    if (cap[e] > 0 && lev[u] < lev[v = go[e]]) {
      delta = Dinic(v, std::min(cap[e], flow - res));
      if (delta) {
        cap[e] -= delta; cap[e ^ 1] += delta;
        res += delta; if (res == flow) break;
      }
    }
  }
  if (res != flow) lev[u] = -1;
  return res;
}

inline int maxFlow() {
  int ans = 0;
  while (Bfs()) ans += Dinic(src, INF);
  return ans;
}

int main() {
  while (scanf("%d%d", &m, &n) != EOF) {
    ecnt = 1;
    for (int i = 1; i <= n; ++i) adj[i] = 0;
    src = 1, des = n;
    for (int i = 1; i <= m; ++i) {
      int u, v, w;
      scanf("%d%d%d", &u, &v, &w);
      addEdge(u, v, w);
    }
```

```
66.     printf("%d\n", maxFlow());
67.   }
68.   return 0;
69. }
```

练习题

POJ 3041 ; POJ 3057 ; POJ 1149 ; POJ 3281 ; POJ 3469 ; POJ 2987 ; POJ 3155

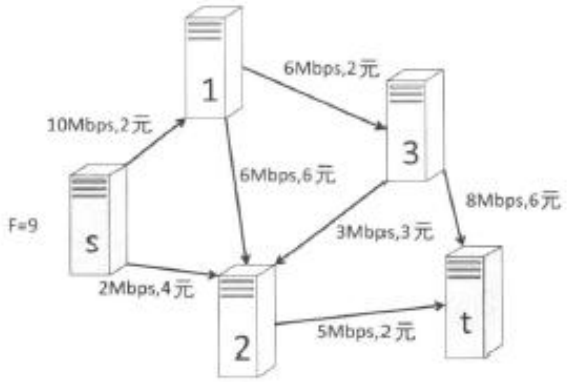
费用流

一、模型引入

费用流与网络流类似，也是解决类似信息传输，水流电流运动等问题的算法。最大流可以解决单位时间内最大数据传输量、最大流量等问题；而有时传输数据、流体流经管道需要费用，并且要求的是流量为某个特定值时最小（或是最大）的费用值，此时就需要使用费用流，我们也称这样的问题为最小（最大）费用流问题。

最大流中数据传输问题的例子稍加修改就可以变为一个最小费用流问题：

网络中有两台计算机 s 和 t，现在每秒钟要从 s 传输大小为 F 的数据到 t。该网络中一共有 N 台计算机，其中一些计算机之间连有一条单向的通信电缆，每条通信电缆都有对应的 1 秒钟内所能传输的最大数据量。此外，每条通信电缆还有对应的传输费用，单位传输费用为 d 的通信电缆每秒传输大小为 x 的数据，需要花费的费用为 dx。求传输数据所需的最小费用。



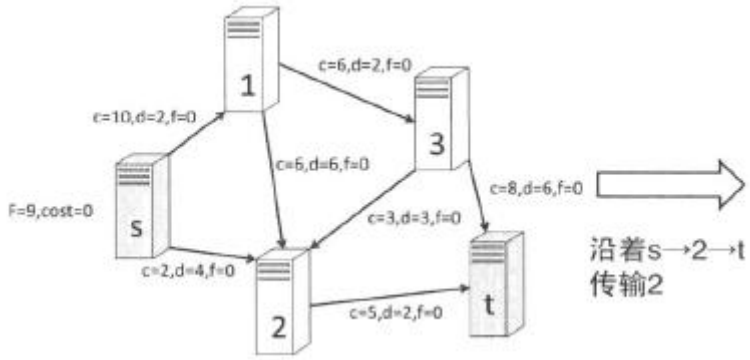
将问题转化为图，则可以将计算机当做顶点，通信电缆当做边，从而能得到一个有向图。每条边 e 都有容量 $c(e)$ 与费用 $d(e)$ 。而题目所求是在 s 到 t 的流量为 F 的情况下，使得费用 $\sum_e (f(e)\times d(e))$ 最小，其中 $f(e)$ 是边的实际流量。

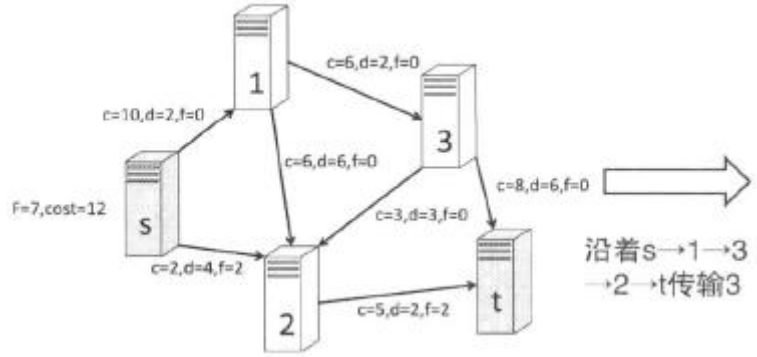
可以看到，费用流在网络流的基础上，每条边多了一个属性，即单位流量的费用 $d(e)$ 。而费用流的可行方案中，仍然需要满足流量约束与流量平衡条件。

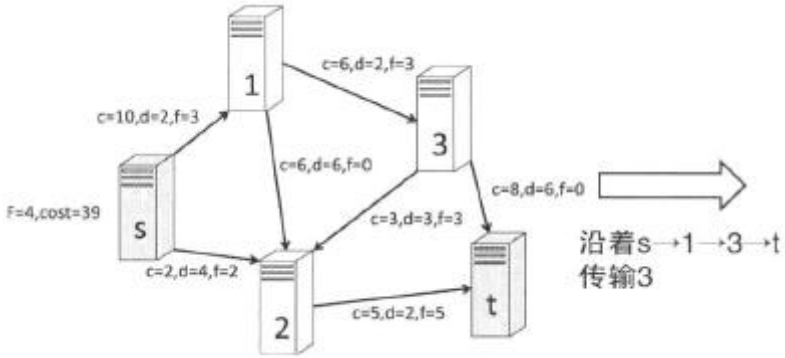
二、求解费用流

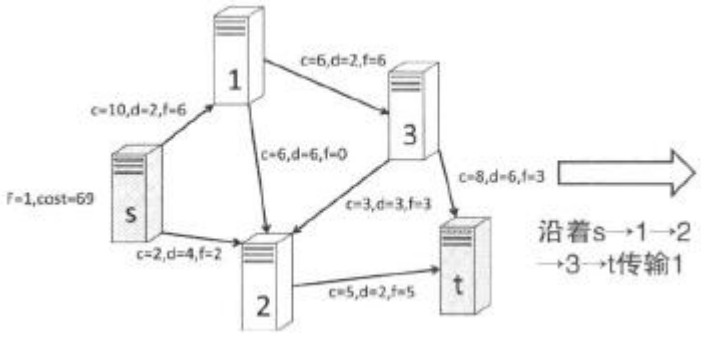
在求解最大流时，我们在残量网络上不断贪心地增广而得到了最大流。现在边上多了费用，为了使最后总费用最小，我们自然能想到每次贪心地在残量网络上，沿着费用最小的那

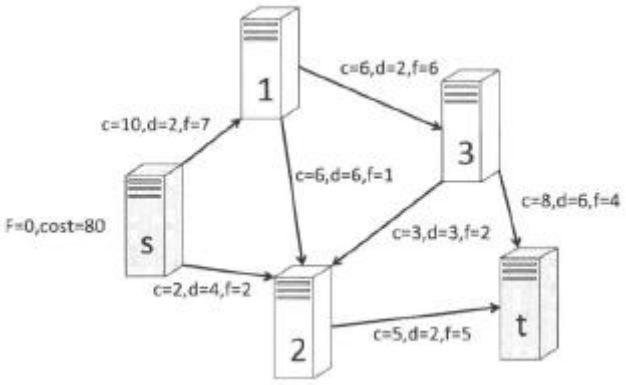
条路进行增广。此时残量网络中的反向边费用应该是原边费用的相反数，以保证退流操作是可逆、正确的。而此时图中就有可能出现负边权，所以不能使用 Dijkstra 求解最短路，而只能使用队列优化的 Bellman-Ford 算法。对上面的样例使用这个算法：











三、正确性证明

下面我们来证明（一般情况下）这个算法所求得的流 f 确实是最小费用流。

假设还有同样流量而费用比 f 更小的流 f'，则我们观察流 $f'-f$ 。在 f 中，除 s 和 t 以外的顶点的流入量等于流出量，在 f' 中亦然。并且，由 f 和 f' 的流量相同可知，流 $f'-f$ 中所有顶点的流入量都等于流出量，即它是由若干圈组成的。因为流 f' 费用更小，所以流 $f'-f$ 的费用是负的，因此在这些圈中，至少存在一个负圈。也就是说：

$$f\text{是最小费用流} \Leftrightarrow \text{残量网络中没有负圈}$$

利用这个结论，我们可以归纳证明上面的贪心算法的正确性。

令算法所求得的流量为 i 的流为 f_i。首先对于流量为 0 的流 f_0，其残量网络就是原图，只要原图中没有负圈，那么 f_0 就是流量 0 的最小费用流。

若我们目前已证流量为 i 的流是最小费用流，并且下一步我们求得了流量为 $i+1$ 的流 f_{i+1}。此时，$f_{i+1}-f_i$ 就是 f_i 对应的残量网络中 s 到 t 的最短路。

假设 f_{i+1} 不是最小费用流，即存在更小的流 f'_{i+1}。那么此时 $f'_{i+1}-f_i$ 中除 s 和 t 以外的顶点的流入量等于流出量，因而是由一条从 s 到 t 的路径和若干圈组成的。而 $f_{i+1}-f_i$ 是一条从 s 到 t 的最短路，而 f'_{i+1} 的费用还要小，因此 $f'_{i+1}-f_i$ 中至少含有一个负圈，这与 f_i 是最小费用流矛盾。所以 f_{i+1} 也是最小费用流。根据归纳法，对任意的 i 都有 f_i 是最小费用流。

另外，由最大流算法正确性可知若原图存在大小为 F 的流，则这个费用流算法也能够

得到流量为 F 的流。

该算法最多执行 F 次 Bellman-Ford 算法，所以其复杂度为 $O(F \cdot nm)$。

四、算法优化

实际上我们算法的核心仍然是交替的进行最短路与最大流操作。

我们可以考虑在这两种操作上分别优化：

我们可以使用类似 Dinic 的思想，对图做最短路然后分层，之后在图中只走那些在最短路上的边进行增广，这样的好处是我们可以进行多路增广。也就是在最大流操作上优化。

我们还可以通过导入势的概念，对最短路过程进行优化：

势指的是给每个顶点赋予一个新的标号 $h(v)$，在这个势的基础上，将边 $e=(u,v)$ 的长度变为 $d'(e)=d(e)+h(u)-h(v)$。于是从 d' 中的 $s-t$ 路径的长度中减去常数 $h(s)-h(t)$ 就得到了 d 中对应路径的长度，因此 d' 中的最短路也就是 d 中的最短路。

因此，合理的选取势，使得对所有的 e 都有 $d'(e)\geq 0$ 的话，我们就可以在 d' 中用 Dijkstra 算法求最短路，从而得到 d 的最短路。

对于不含负圈的图，我们可以通过选取 $h(v)=(s到v的最短距离)$ 来做到 $d'(e)\geq 0$。因为对于边 $e=(u,v)$ 有：

$$(s到v的最短距离)\leq(s到u的最短距离)+d(e)$$

因此 $d'(e)=d(e)+h(u)-h(v)\geq 0$

下面来考虑如何依次更新流量为 i 时的最小费用流 f_i 以及其所对应的势 h_i。首先我们定义以下变量：

$$f_i(e):流量为i的最小费用流中边e的流量$$
$$h_i(v):f_i的残量网络中s到v的最短距离$$
$$d_i(e):考虑势h_i后边e的长度$$

若原图不含负圈，则我们可以将 $f_i(e)$ 初始化为 0，若原图也不存在负权边，那么我们

还可以直接用 Dijkstra 计算 h_0（否则使用 Bellman-Ford）。

求得 f_i 与 h_i 后，我们沿着 f_i 的残量网络中 s 到 t 的最短路增广，我们就得到了 f_{i+1}，而这步可以借助 h_i，找到那些只经过 $d_i(e)=0$ 的边的路径进行增广（这些边一定在最短路上）。

而为了求 h_{i+1}，我们需要求 f_{i+1} 的残量网络上的最短路。考虑 f_{i+1} 的残量网络中的边 $e=(u,v)$。如果 e 也是 f_i 的残量网络中的边的话，那么根据 h 的定义有 $d_i(e)\geq 0$。如果 e 不是 f_i 的残量网络中的边的话，那么 $rev(e)$ 一定是 f_i 的残量网络中 s 到 t 的最短路中的边，所以有 $d_i(e)=-d_i(rev(e))=0$。综上，f_{i+1} 的残量网络中的所有边 e 满足 $d_i(e)\geq 0$，因而可以用 Dijkstra 算法求最短路。

如上所述，我们依次更新 f_i 与 h_i 就能在 $O(Fm\log n)$ 时间内求出最小费用流。

若原图中有负圈存在，则需要使用消圈算法。由于竞赛中较少遇到这种情况，因此这里不再详细介绍，大家可以自行查阅资料。

五、算法模板

普通的队列优化 Bellman-Ford+多路增广：

```
namespace MCMF {

const int N = 1000 + 5;
const int M = 50000;
const int INF = 0x3f3f3f3f;

int src, des, ans, dis[N];
int m = 1, adj[N], nxt[M], go[M], cap[M], cost[M];
bool vis[N], walk[N];
std::queue<int> que;

inline void addEdge(int u, int v, int f, int w) {
  nxt[++m] = adj[u], adj[u] = m, go[m] = v, cap[m] = f, cost[m] = w;
  nxt[++m] = adj[v], adj[v] = m, go[m] = u, cap[m] = 0, cost[m] = -w;
}

bool buildGraph() {
```

```
  memset(dis, INF, sizeof(dis));
  memset(walk, false, sizeof(walk));
  que.push(src), dis[src] = 0;
  int u, v, e;
  while (!que.empty()) {
    u = que.front(), que.pop();
    vis[u] = false;
    for (e = adj[u]; e; e = nxt[e]) {
      if (cap[e] > 0 && dis[u] + cost[e] < dis[v = go[e]]) {
        dis[v] = dis[u] + cost[e];
        if (!vis[v]) {
          vis[v] = true;
          que.push(v);
        }
      }
    }
  }
  return dis[des] < INF;
}

int dfs(int u, int flow) {
  if (u == des) {
    ans += flow * dis[des];
    return flow;
  }
  walk[u] = true;
  int v, e, res = 0, delta;
  for (e = adj[u]; e; e = nxt[e]) {
    if (!walk[v = go[e]] && cap[e] > 0 && dis[u] + cost[e] == dis[v]) {
      delta = dfs(v, std::min(cap[e], flow - res));
      if (delta) {
        cap[e] -= delta, cap[e ^ 1] += delta;
        res += delta; if (res == flow) break;
      }
    }
  }
  return res;
}

int solve() {
  ans = 0;
  while (buildGraph()) dfs(src, INF);
  return ans;
}
```

```
62.
63. }
```

利用势：

```
1.  class MCMF {
2.    int arc[N], adj[N], to[E], next[E], cap[E], cost[E], cnt;
3.    int s, t, cur;
4.    int dist[N];
5.    std::pair<int, int> heap[E];
6.    bool dijkstra() {
7.      std::fill(dist, dist + t + 1, INF);
8.      int top = 1;
9.      for (heap[0] = std::make_pair(dist[s] = 0, s); top;) {
10.       std::pop_heap(heap, heap + top);
11.       std::pair<int, int> info = heap[--top];
12.       int a = info.second;
13.       if (-info.first > dist[a]) continue;
14.       for (int i = adj[a]; i; i = next[i]) {
15.         int b = to[i], c = cost[i];
16.         if (cap[i] && dist[a] + c < dist[b]) {
17.           heap[top++] = std::make_pair(-(dist[b] = dist[a] + c), b);
18.           std::push_heap(heap, heap + top);
19.         }
20.       }
21.     }
22.     if (dist[t] == INF) return false;
23.     for (int a = 0; a <= t; ++a)
24.       for (int i = adj[a]; i; i = next[i])
25.         cost[i] -= dist[to[i]] - dist[a];
26.     // 还有一种写法是显式的维护 h(v)，那么更新时即为 h(v) += dist(v)
27.     // 若显式维护 h(v)则要将边权的表达式稍加修改
28.     cur += dist[t];
29.     return true;
30.   }
31.   int tag[N], tot;
32.   int dfs(int a, int df) {
33.     if (a == t) return df;
34.     tag[a] = tot;
35.     int res = 0;
36.     for (int &i = arc[a]; i; i = next[i]) {
37.       int b = to[i];
38.       if (cap[i] && !cost[i] && tag[b] != tot) {
39.         int f = dfs(b, std::min(df - res, cap[i]));
```

```
        cap[i] -= f;
        cap[i ^ 1] += f;
        res += f;
      }
      if (res == df) break;
    }
    return res;
  }
 public:
  inline void clear(int _s, int _t) {
    cnt = 2, s = _s, t = _t;
    memset(adj, 0, sizeof adj);
  }
  inline void link(int a, int b, int c, int d) {
    to[cnt] = b, next[cnt] = adj[a], cap[cnt] = c, cost[cnt] = d, adj[a] = cnt++;
    to[cnt] = a, next[cnt] = adj[b], cap[cnt] = 0, cost[cnt] = -d, adj[b] = cnt++;
  }
  std::pair<int, int> flow() {
    std::pair<int, int> res(0, 0);
    for (cur = 0; dijkstra();) {
      std::copy(adj, adj + t + 1, arc);
      do {
        ++tot;
        int f = dfs(s, INF);
        if (!f) break;
        res.first += f;
        res.second += cur * f;
      } while (1);
    }
    return res;
  }
};
```

六、例题

Football Game (POJ 2135)

题目大意：

一个图上有 N 个顶点，从 1 到 N 标号，顶点之间存在一些无向边，边有长度，要求从顶点 1 走到顶点 N，再从顶点 N 走回顶点 1，其中不必要经过每个顶点，但是要求走的路径

上的边只能经过一次。求出这样 1->N->1 的路径的长度最小值。

分析：

题目相当于是从 1 到 N 找两条不相交的路径并使得总长度最小。由于每条边不能重复经过，所以我们将每条边容量视为 1，费用为边的长度，新建源点 s 向 1 连一条容量 2 费用 0 的边，新建汇点 t，N 向 t 连一条容量为 2 费用为 0 的边，则题目就转化为求从 s 到 t 的最小费用最大流问题。

代码：

```
#include <queue>
#include <cstdio>
#include <cstring>

const int INF = 0x3f3f3f3f;
int n, m;

namespace MCMF {

const int N = 1000 + 5;
const int M = 50000;

int src, des, ans, dis[N];
int m = 1, adj[N], nxt[M], go[M], cap[M], cost[M];
bool vis[N], walk[N];
std::queue<int> que;

inline void addEdge(int u, int v, int f, int w) {
  nxt[++m] = adj[u], adj[u] = m, go[m] = v, cap[m] = f, cost[m] = w;
  nxt[++m] = adj[v], adj[v] = m, go[m] = u, cap[m] = 0, cost[m] = -w;
}

void init(int _n, int _m) {
  src = 0, des = _n + 1;
  for (int i = 1; i <= _m; ++i) {
    int u, v, w;
    scanf("%d%d%d", &u, &v, &w);
    addEdge(u, v, 1, w);
    addEdge(v, u, 1, w);
  }
  addEdge(src, 1, 2, 0);
  addEdge(n, des, 2, 0);
}

```

```
bool buildGraph() {
  memset(dis, INF, sizeof(dis));
  memset(walk, false, sizeof(walk));
  que.push(src), dis[src] = 0;
  int u, v, e;
  while (!que.empty()) {
    u = que.front(), que.pop();
    vis[u] = false;
    for (e = adj[u]; e; e = nxt[e]) {
      if (cap[e] > 0 && dis[u] + cost[e] < dis[v = go[e]]) {
        dis[v] = dis[u] + cost[e];
        if (!vis[v]) {
          vis[v] = true;
          que.push(v);
        }
      }
    }
  }
  return dis[des] < INF;
}

int dfs(int u, int flow) {
  if (u == des) {
    ans += flow * dis[des];
    return flow;
  }
  walk[u] = true;
  int v, e, res = 0, delta;
  for (e = adj[u]; e; e = nxt[e]) {
    if (!walk[v = go[e]] && cap[e] > 0 && dis[u] + cost[e] == dis[v]) {
      delta = dfs(v, std::min(cap[e], flow - res));
      if (delta) {
        cap[e] -= delta, cap[e ^ 1] += delta;
        res += delta; if (res == flow) break;
      }
    }
  }
  return res;
}

int solve() {
  ans = 0;
  while (buildGraph()) dfs(src, INF);
  return ans;
```

```
}

}

int main() {
  scanf("%d%d", &n, &m);
  MCMF::init(n, m);
  printf("%d\n", MCMF::solve());
  return 0;
}
```

练习题

POJ 3686 ; POJ 3068 ; POJ 2195 ; POJ 3422 ; POJ 2175

上述内容参考：

1.挑战程序设计竞赛（第 2 版）