网络流

> By DaDaMr_X

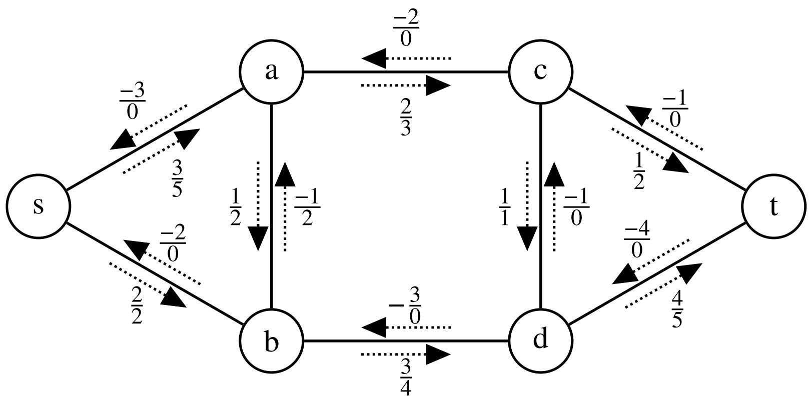
一、最大流

1. 流网络

1. **概述**：在图论中，一个**流网络**是指一个有向图，其中每条边都有一个**容量**限制并可以接受**流**，满足每一条边的流量不会超过它的容量。一道流必须符合一个结点的进出的流量相同的限制，除非这是一个**源点——**只有向外的流，或是一个**汇点——**只有向内的流。这种流网络可以用来建模很多实际问题，如液体在管道中的流动、装配线上部件的流动、电

网中电流的流动和通信网络中信息的流动等。

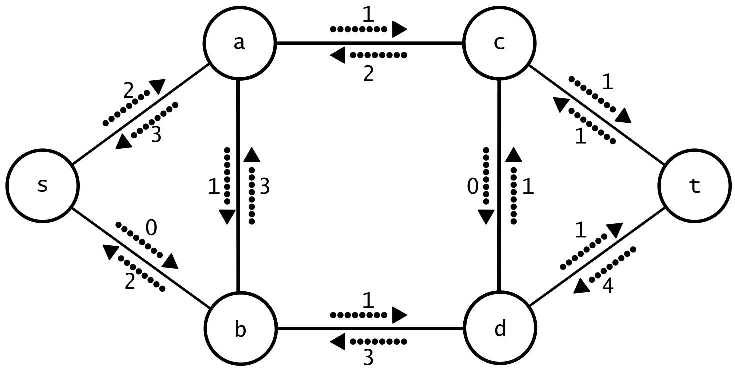
2. **流网络(Flow Network)**：一个有向图$G=(V,E)$，图中每条边$(u,b)\in E$有一个非负的容量值$c(u,v)\geqslant 0$，称为容量函数。规定两个特殊的结点，源点s和汇点t。(G,c,s,t)就称为一个**流网络**。
3. **流(Flow)**：一个流网络中的流是一个实值函数$f:V\times V\to\mathbb{R}$，满足以下三条性质：
    - **容量限制(Capacity Constraints)**：$f(u,v)\leqslant c(u,v)$，即一条边上的流量不能超过这条边上的容量。
    - **斜对称(Skew Symmetry)**：$f(u,v)=-f(v,u)$，由u到v的净流必须是由v到u的净流的相反数。
    - **流守恒(Flow Conservation)**：$\forall v\in V/\{s,t\}$，$\sum_{(u,v)\in E}f(u,v)=\sum_{(v,z)\in E}f(v,z)$，即对于除源点和汇点外的每个结点，流入的流量等于流出的流量。
4. 一个标注了每条边的容量和流量的流网络（横线下为容量，横线上为流量）。



2. 最大流 Dinic算法

1. **最大流问题**：我们希望在不违反任何容量限制的情况下，使从源点出发的流的总量最大或者流入汇点的流的总量最大。我们想知道这个最大值是多少，这就是最大流问题。
2. **Ford-Fulkerson方法**：Ford-Fulkerson方法可以用来解决最大流问题。之所以称其为“方法”而不是“算法”，是因为它包含了几种运行时间各不相同的具体实现，Dinic算法就是其中的一种实现。Ford-Fulkerson方法的基本思想是：开始时网络中初始的流值为0，在每一次迭代中，在“残余网络”中寻找一条“增广路径”，沿增广路径增加流量，直到残余网络中不存在增广路径为止。最大流最小割定理保证了在算法结束时，该算法将获得一个最大流。
3. **残余网络(Residual Network)**：一条边的剩余容量定义为$c_f(u,v)=c(u,v)-f(u,v)$，由此可以构造出剩余网络$G_f(V,E_f)$，它显示了网络中剩余的可用容量。形式化地定义，对于一个流网络(G,c,s,t)，它的残余网络为(G_f,c_f,s,t)，其中$G_f=(V,E_f)$，$E_f=\{(u,v)\in V\times V:c_f(u,v)>0\}$，$c_f(u,v)=c(u,v)-f(u,v)$。

4. 上文的流网络的残余网络，图中标注出了每条边的剩余容量。（注意一些原来没有容量的边在残余网络中有了容量）



5. **增广路径(augmenting path)**：增广路径是一条路径$(u_1, u_2, \cdots, u_k)$，其中$u_1 = s$，$u_k = t$，$c_f(u_i, u_{i+1}) > 0, (1 \leqslant 1 < k)$。增广路径表示沿这条路径传送更多的流量是可能的。当残余网络中没有增广路径时得到最大流。
6. **割(Cut)**：一个流网络$(G, c, s, t), G = (V, E)$的割$C = (S, T)$是顶点集合V的一个划分，满足$s \in S$，$t \in T$。割(S, T)的容量是$c(S, T) = \sum_{u \in S, v \in T} c(u, v)$。
7. **最大流最小割定理**：最大流 = 最小割。(限制流量的瓶颈)
8. **Dinic算法**：每次对残余网络BFS标注层次图，即用数组$level[i]$记录i点在BFS树中的深度。然后严格按照层次顺序不断DFS寻找增广路，即增广路上结点的层次编号严格递增。如果残余网络中已经不存在到增广路了就重新标BFS注层次图，再次不断DFS增广，直到源点与汇点不再连通。累计每次增广得到的流量就得到最大流。

Code

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <queue>
using namespace std;
typedef long long ll;
const int INF = 0x3f3f3f3f;
const int N = 1e3 + 10;
const int M = 2e3 + 10;

struct Edge
{
    int to, c, next;
    Edge() {}
```

```
    Edge(int to, int c, int next) : to(to), c(c), next(next) {}
} edge[M];
int adj[N], tot;

void init()
{
    memset(adj, -1, sizeof(adj));
    tot = 0;
}

void add(int u, int v, int c)
{
    edge[tot] = Edge(v, c, adj[u]);
    adj[u] = tot++;
    edge[tot] = Edge(u, 0, adj[v]);
    adj[v] = tot++;
}

int level[N];
queue<int> q;
bool bfs(int s, int t)
{
    while (!q.empty()) q.pop();
    memset(level, -1, sizeof(level));
    level[s] = 0; q.push(s);
    while (!q.empty())
    {
        int u = q.front(); q.pop();
        for (int i = adj[u]; i != -1; i = edge[i].next)
        {
            Edge &e = edge[i];
            if (e.c && level[e.to] == -1)
            {
                level[e.to] = level[u] + 1;
                if (e.to == t) return true;
                q.push(e.to);
            }
        }
    }
```

```
    return false;
}

int cur[N];
int dfs(int u, int t, int flow)
{
    if (u == t) return flow;
    for (int &i = cur[u]; i != -1; i = edge[i].next)
    {
        Edge &e = edge[i];
        if (e.c && level[e.to] > level[u])
        {
            int f = dfs(e.to, t, min(flow, e.c));
            if (f)
            {
                e.c -= f;
                edge[i ^ 1].c += f;
                return f;
            }
        }
    }
    return 0;
}

int dinic(int s, int t)
{
    int flow = 0;
    while (bfs(s, t))
    {
        memcpy(cur, adj, sizeof(adj));
       int f;
        while (f = dfs(s, t, INF)) flow += f;
    }
    return flow;
}

int main()
{
    int n, m;
```

```
    scanf("%d%d", &n, &m);
    init();
    while (m--)
    {
        int u, v, c;
        scanf("%d%d%d", &u, &v, &c);
        add(u, v, c);
    }
     int s, t;
    scanf("%d%d", &s, &t);
    int ans = dinic(s, t);
    printf("%d\n", ans);
     return 0;
}
```

Input

第一行给出结点数n和边数m，接下来的m行，每行给出两个三个整数u，v和c，表示从u到v有一条容量为c的边。最后一行给出源点s和汇点t。

Output

输出一个整数，表示最大流。

Sample Input

```
6 9
0 1 10
0 2 10
1 2 2
1 3 4
1 4 8
2 4 9
3 5 10
4 3 6
4 5 10
0 5
```

Sample Output

```
19
```

3. 最大流变式

3.1 多源汇的情况

添加一个超级源点和超级汇点，超级源点到每一个源点建一条容量为无穷大的边，每个汇点到超级汇点建一条容量为无穷大的边，从超级源点到超级汇点跑最大流。

3.2 点上有容量限制的情况

如果限制条件为i点的流量不能超过c，就把i点拆为i_{in}和i_{out}两个点并建一条从i_{in}到i_{out}的容量为c的边，原图中进入i点的边都连到i_{in}点，从i出发的边都从i_{out}出发。

例题 POJ 3281 Dining

Description

Farmer John有N头牛，F种食物和D种饮料，每头牛都有自己喜欢的若干种食物和若干种饮料，已知一头牛最多能吃一种食物和一种饮料，每种饮料或食物最多能被一头牛吃，求以上条件下，最多能有多少头牛能吃到他所喜爱的食物和饮料。

Solution

建三排点，中间N个点代表N头牛，左边F个点代表F种食物，右边D个点代表D种饮料。每种食物向喜爱它的牛建一条容量为1的边，每头再向它喜爱的每种饮料建一条容量为1的边。又因为每头牛最多只计算一次，故将表示每头牛的点拆为两个，建容量为1的边。建立源点s，与每种食物建容量为1的边，建立汇点t，每种饮料与它建容量为1的边，跑最大流即为最终答案。

3.3 最小费用最大流

Description

给出一个N个点M条边的有向图，每条边上有一个容量限制cap和单位流量的花费$cost$。给出源点s和汇点t，求从源点s到汇点t的花费最小的最大流。输出最小花费和最大流的值。

Solution

初始时流量为0，花费也为0，此时是当前流量下的最小花费。每次增广，我们不再找任意增广路径，而是找花费最小的路径，这样就保证了每次迭代得到的都是当前流量下的最小花费。如果已经不能再增广了，说明已经找到了最大流量下的最小花费。寻找花费最小的路径就是在以花费为边权的图上找最短路。

Code

```
#include <cstdio>
#include <cstring>
#include <algorithm>
#include <queue>
using namespace std;
typedef long long ll;
const int INF = 0x3f3f3f3f;
const int N = 1e3 + 10;
const int M = 2e3 + 10;

struct Edge
{
    int from, to, next, cap, cost;
    Edge() {}
    Edge(int from, int to, int next, int cap, int cost) :
        from(from), to(to), next(next), cap(cap), cost(cost) {}
} edge[M];
int head[N], tot;

void add(int from, int to, int cap, int cost)
{
    edge[tot] = Edge(from, to, head[from], cap, cost);
    head[from] = tot++;
    edge[tot] = Edge(to, from, head[to], 0, -cost);
    head[to] = tot++;
}

void init()
{
    memset(head, -1, sizeof(head));
    tot = 0;
}

int dis[N], pre[N];
bool vis[N];
queue<int> q;
```

```
bool spfa(int s, int t)
{
    while (!q.empty()) q.pop();
    memset(dis, 0x3f, sizeof(dis));
    memset(vis, false, sizeof(vis));
    memset(pre, -1, sizeof(pre));
    q.push(s); vis[s] = true; dis[s] = 0;
    while (!q.empty())
    {
        int u = q.front(); q.pop(); vis[u] = false;
        for (int i = head[u]; i != -1; i = edge[i].next)
            if (edge[i].cap && dis[u] + edge[i].cost < dis[edge[i].to])
            {
                int v = edge[i].to;
                dis[v] = dis[u] + edge[i].cost;
                pre[v] = i;
                if (!vis[v]) q.push(v), vis[v] = true;
            }
    }
    return dis[t] < INF;
}

int mcmf(int s, int t, int& maxflow)
{
    int mincost = 0;
    maxflow = 0;
    while (spfa(s, t))
    {
        int flow = INF;
        for (int i = pre[t]; i != -1; i = pre[edge[i].from])
            flow = min(flow, edge[i].cap);
        for (int i = pre[t]; i != -1; i = pre[edge[i].from])
        {
            edge[i].cap -= flow;
            edge[i ^ 1].cap += flow;
            mincost += edge[i].cost * flow;
        }
        maxflow += flow;
    }
```

```
    return mincost;
}

int main()
{
    int n, m;
    scanf("%d%d", &n, &m);
    init();
    while (m--)
    {
        int u, v, cap, cost;
        scanf("%d%d%d%d", &u, &v, &cap, &cost);
        add(u, v, cap, cost);
    }
    int s, t, maxflow;
    scanf("%d%d", &s, &t);
    int mincost = mcmf(s, t, maxflow);
    printf("%d %d\n", mincost, maxflow);
    return 0;
}
```

Input

第一行给出两个整数n和m，表示结点数和边数。 接下来m行，每行给出4个整数u，v，w，c，表示从u到v有一条容量为w，单位流量花费为c的边。 最后一行给出两个整数s和t，表示源点和汇点。

Output

输出两个整数，分别表示最小花费和最大流。

Sample Input

```
5 5
1 2 2 2
1 3 3 2
2 4 3 1
3 4 2 3
4 5 2 1
1 5
```

Sample Output

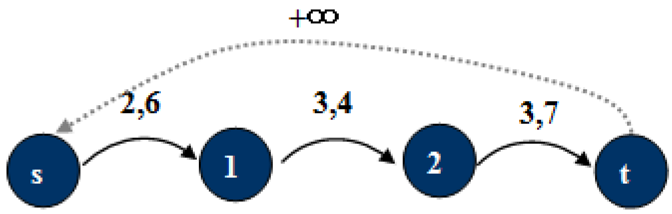
```
8 2
```

3.4 有上下界的网络流

无源汇有上下界的可行流

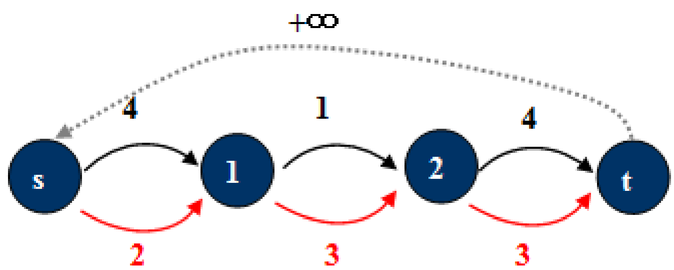
Description

给出一个N个点M条边的有向图，每条边都有一个容量下限l和容量上限r，即每条边的流量必须在$[l,r]$的区间内，没有源点和汇点，问是否存在满足限制的一道流。

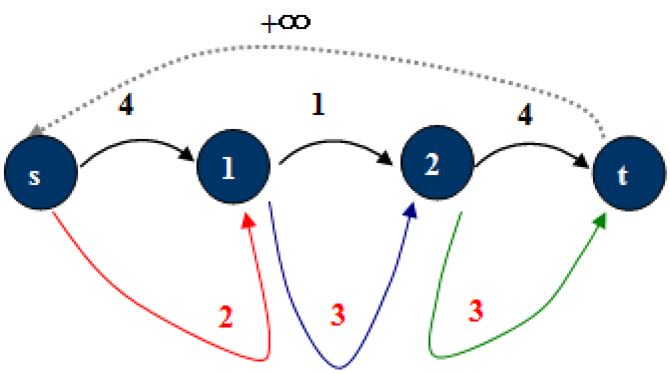


Solution

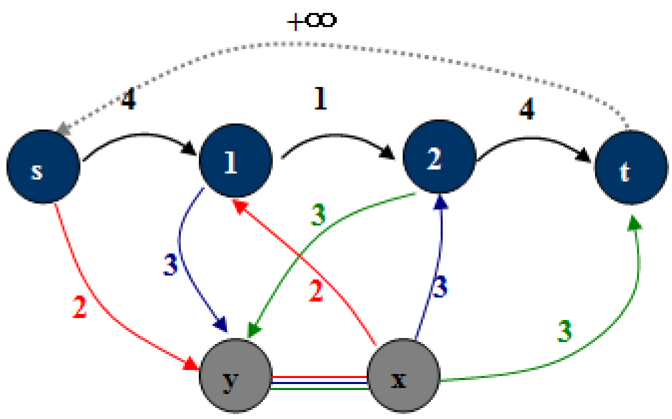
思路就是将有上下界的网络转化为没有下界限制的网络。把原来的容量为$[l,r]$的边分为两条没有下限的边，一条容量为l，是必须要流满的，称为“必要弧”，另一条边的容量为$r-l$。



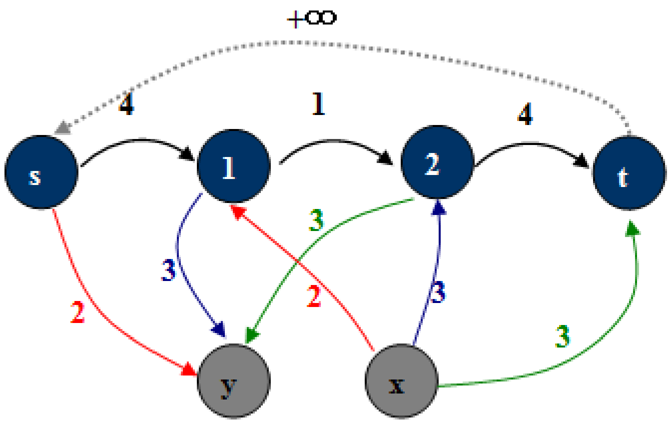
由于必要弧的有一定要满的限制，将必要弧“拉”出来集中考虑。



添加两个点x和y，并且y到x有一条容量无限大的边，把每一条必要弧改为先统一进入y点，流到x点后统一流出。因为边(y, x)的容量为无限大，所以原网络中有可行流当且仅当新网络中有可行流。



因为每条必要流都必须流过(y, x)边，于是去掉这条边，从x到y跑最大流，如果满流，则说明原图存在可行流。



实现上，令$in[i]$为i点必须要流入的流量，$out[i]$为i点必须要流出的流量，x到每个点i连一条容量为$in[i]$的边，i到y连一条容量为$out[i]$的边，从x到y跑最大流，如果满流，则说明所有的必要弧可以流满，存在可行流。否则不存在可行流。

另法：另$du[i] = in[i] - out[i]$表示i点净的要流入的流量，遍历每个顶点，如果$du[i] > 0$，就从x到i建一条容量为$du[i]$的边，如果$du[i] < 0$，就从i到y建一条容量为$-du[i]$的边。从x到y跑最大流，如果满流，说明原图存在可行流。

有源汇有上下界的可行流

从汇点到源点建一条容量为无穷大的边，成为一个无源汇的网络，另外添加超级源点ss和超级汇点tt，按上文方法建图。新网络中的每一道可行流对应原网络中的一道可行流，新网络中的最大流对应原网络中的最大流。

有源汇有上下界的最大流

从汇点t到源点s建边成为无源汇的网络，二分最大流的值，作为从t到s的边的容量，判断如果存在可行流，说明合法，反之不合法。

令法：从汇点t到源点s建一条容量为无穷大的边转化为无源汇的网络，添加超级源点ss和超级汇点tt，按上文给的第二种方法建边，判断存在可行流。删除超级源点和超级汇点，从原来的源点s到原来的汇点t跑最大流，此时就是原网络的最大流。

因为第一次从超级源点到超级汇点跑最大流时，流量存储在了从t到s的无穷容量的边的反向弧中。再从s到t跑最大流时，无穷大的边的反向弧中流过了必要流，原网络中流过了自由流，合起来就是原网络的最大流。

例题 ZOJ 3229 Shoot the Bullet

Description

一位摄影师要在n天给m个女孩拍照，第x个女孩至少要拍G_x张，在第k天，摄影师有C_k个目标，T_{k1}，T_{k2}，$\cdots$，T_{kCk}，给目标T_{ki}拍的照片的数量要在$[L_{ki}, R_{ki}]$的范围内，第k天摄影师最多只能拍D_k张照片。问在不违反这些限制条件的情况下，摄影师最多可以拍的照片数量。

Solution

左侧n个点代表n天，右侧m个点代表m个女孩，若第i天要给第j个女孩拍照片，照片数量要在$[l, r]$范围内，则代表第i天的点到第j天的点有一条容量范围位$[l, r]$的边。添加源点s和汇点t，源点s与代表n天的点之间建容量范围为$[0, d]$的边，代表m个女孩的点与汇点t之间有容量范围为$[g, \infty]$的边。这样一个有源汇带上下界的流网络就建好了，求最大流即可。

二、二分图匹配

1. 二分图

1. **二分图(Bipartite Graph)**：简单来说，如果图中点可以被分为两组，并且使得所有边都跨越组的边界，则这就是一个二分图。准确地说，把一个图的顶点划分为两个不相交集U和V，使得每一条边都分别连接U、V中的顶点。如果存在这样的划分，则此图为一个二分图。下图中图2是一个二分图，图1也是一个二分图，仔细观察会发现，这两个图其实是完全一样的。

Fig.1

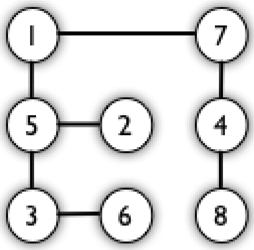
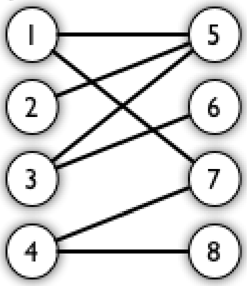


Fig.2



2. **匹配(Matching)**：在图论中，一个“匹配”是一个边的集合，其中任意两条边都没有公共顶点。例如，图3、图4中红色的边就是图2的匹配。

Fig.3

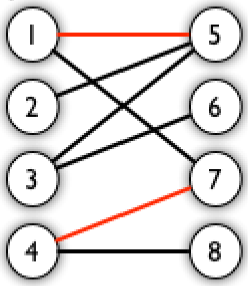
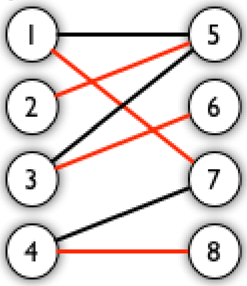


Fig.4



3. 我们定义**匹配点**、**匹配边**、**未匹配点**、非匹配边，它们的含义非常显然。例如图 3 中 1、4、5、7 为匹配点，其他顶点为未匹配点；(1,5)、(4,7)为匹配边，其他边为非匹配边。
4. **最大匹配(Maximum Matching)**：一个图所有匹配中，所含匹配边数最多的匹配，称为这个图的最大匹配。图4是一个最大匹配，它包含4条匹配边。
5. **完美匹配(Perfect Matching)**：如果一个图的某个匹配中，所有的顶点都是匹配点，那么它就是一个完美匹配。图4是一个完美匹配。显然，完美匹配一定是最大匹配，但并非每个图都存在完美匹配。

2. 二分图最大匹配 Hungary算法

5. **交错路**：从一个未匹配点出发，依次经过非匹配边、匹配边、非匹配边...形成的路径叫交错路。
6. **增广路**：从一个未匹配点出发，走交替路，终点为另一个未匹配点的路径。图5中的一条增广路如图6所示，匹配边和匹配点用红色标出。

Fig.5

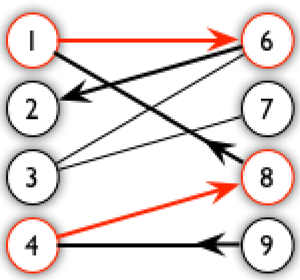
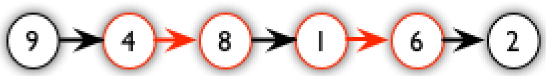


Fig.6



7. **性质**：非匹配边比匹配边多一条。因此，研究增广路的意义是改进匹配。只要把增广路中的匹配边和非匹配边的身份交换即可。由于中间的匹配节点不存在其他相连的匹配边，所以这样做不会破坏匹配的性质。交换后，图中的匹配边数目比原来多了1条。
8. **定理**：我们可以通过不停地找增广路来增加匹配中的匹配边和匹配点。找不到增广路时，达到最大匹配。匈牙利算法正是这么做的。
9. **匈牙利算法**：依次从左边集合的每个点出发DFS寻找增广路，一旦找到，就反转这条路径上的匹配边和未匹配边，并且计数器加一。在对左边集合每个点都处理过一遍后，保证图中不再有增广路。

Code

```
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
typedef long long ll;
const int INF = 0x3f3f3f3f;
const int N = 1e3 + 10;
const int M = 1e3 + 10;

struct Edge
{
    int to, next;
} edge[M];
int adj[N], no;
int n, m;

void init()
{
    memset(adj, -1, sizeof(adj));
    no = 0;
}

void add(int u, int v)
{
    edge[no].to = v;
    edge[no].next = adj[u];
    adj[u] = no++;
}
```

```
int left, right;
int match[N];
bool vis[N];
bool dfs(int u)
{
    for (int i = adj[u]; i != -1; i = edge[i].next)
    {
        int v = edge[i].to;
        if (vis[v]) continue;
        vis[v] = true;
        if (match[v] == -1 || dfs(match[v]))
        {
            match[v] = u;
            return true;
        }
    }
    return false;
}

int hungary(int x, int y)
{
    left = x; right = y;
    int ans = 0;
    memset(match, -1, sizeof(match));
    for (int u = 1; u <= left; u++)
    {
        memset(vis, false, sizeof(vis));
        if (dfs(u)) ans++;
    }
    return ans;
}

int main()
{
    int n, m, e;
    scanf("%d%d%d", &n, &m, &e);
    init();
    while (e--)
```

```
    {
        int u, v;
        scanf("%d%d", &u, &v);
        add(u, v);
    }
    int ans = hungary(n, m);
    printf("%d\n", ans);
    return 0;
}
```

Input

第一行给出三个整数n，m和e，分别表示左右集合的大小和边数。 接下来的e行，每行给出两个整数u和v，表示左边集合中的u点与右边集合中的v点之间有一条边相连。左边集合结点编号从1到n，右边集合结点编号从1到m。

Output

输出一个整数，表示最大匹配数。

Sample Input

```
5 4 8
1 1
2 1
2 2
3 3
3 4
4 2
5 1
5 4
```

Sample Output

```
4
```

3. 二分图匹配最大相关问题

3.1 二分图最大点独立集

点独立集：图的顶点集的一个子集，其中任意两点之间没有边相连。

二分图最大点独立集 = 顶点总数 - 最大匹配数

证明：设一个二分图的顶点集合为V，最大匹配为M，匹配的顶点集合为V_M，则有$|V_M| = 2|M|$。设最大独立集为U，证明$|U| = |V| - |M|$。

由二分图和最大匹配的定义可知，

$$|U| \geqslant |V| - |V_M|$$

现依次从最大匹配M中匹配的每一对点中取一个放入独立集U中。设u，v是匹配的两个点，$(u, v) \in M$，则u和v不可能同时与未匹配的点有边相连，否则还存在一条交错路，与最大匹配的相矛盾。于是把没有与未匹配点相连的点加入独立集中。重复下去，直到每一对匹配点中都有一个点加入了独立集中。此时有，

$$|U| \geqslant |V| - |V|$$

此时再向U中加入任何一个点都会与已有的点有匹配边，亦有

$$|U| \leqslant |V| - |M|$$

因此$|U| = |V| - |M|$，得征。

例题 HDU 1068 Girls and Boys

Description

n个同学，一些男女同学会有缘分成为情侣，已知每一位同学有缘分成为情侣的对象，求集合中不存在有缘成为情侣的同学的最大同学数。

Solution

二分图最大点独立集

3.2 有向无环图最小路径覆盖

路径覆盖：一个有向图中的一些路径，它们覆盖了图中所有的点，且任何一个顶点只有一条路径与之关联。

有向无环图最小路径覆盖 = 顶点总数 - 对应二分图的最大匹配数

证明：初始时n个点每个点自己就是一条路径覆盖，每添加一条有向边，路径覆盖数就减一。因此路径覆盖数=顶点总数-添加的有向边数。最多的有向边就对应最小的路径覆盖。下面通过建图将每一条有向边转化为二分图中的一条边。

建图：输入一个有n个点的有向无环图，把图中的每个顶点u拆分为两个顶点u'和u''，对于原图中的每条边(u, v)，就建一条从u'到v''的边，这样就转化为了一个总共有$2n$个结点的二分图。二分图中的每条匹配边，对应有向图中选出的一条有向边，路径覆盖中每个点只能属于一个路径覆盖，对应在匹配中每个点只能使用一次。

例题 HDU 1151 Air Raid

Description

在一个城镇，有n个路口和m条路，这些路都是单向的，而且路不会形成环.现在要弄一些伞兵去巡查这个城镇，伞兵只能沿着路的方向走，问最少需要多少伞兵才能把所有的路口城镇搜一遍。

Solution

有向无环图最小路径覆盖

扩展

1. 动态Dinic
2. 最大权不相交路径
3. 最长k可重区间
4. 最大权闭合图