算法基础（十五）： 图 - 二分图问题

二分图：

设`G=（V,E）`是一个无向图，如果顶点`V`可分割为两个互不相交的子集`（A,B）`，并且图中的每条边`（i，j）`所关联的两个顶点`i`和`j`分别属于这两个不同的顶点集`（i in A,j in B）`，则称图`G`为一个二分图，子集`A`，`B`的内部没有边

二分图的重要性质：

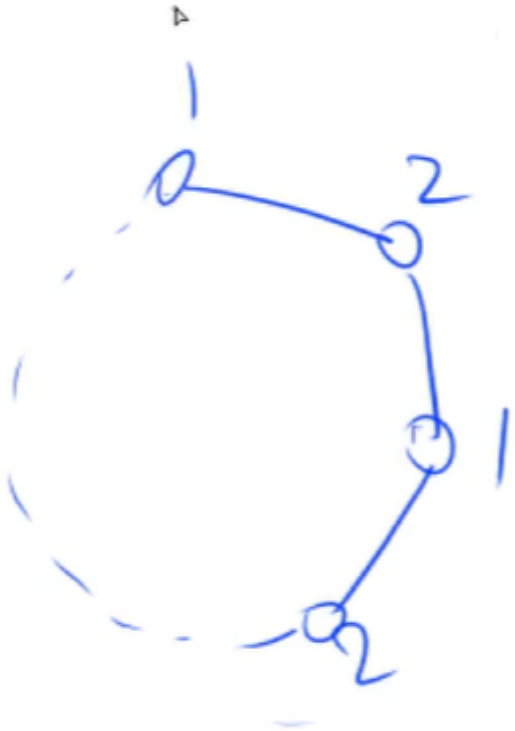
一个图是二分图 当且仅当 图中不含奇数环，奇数环是指，图中环的边的个数是奇数

证明：

必要性：一个图是二分图则图中不含奇数环

可证必要性的逆否命题：**如果有一个奇数环则一定不是一个二分图**

假设有以下这个图：

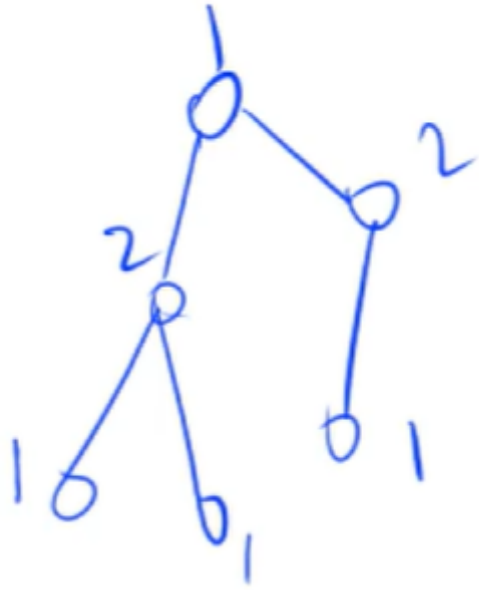


起点是`1`集合的点然后这个环是奇数环，那么沿着这个环，由于不同子集内部没有边，所以一条边的两个点必然不属于同一个集合中，又由于是奇数环，所以最后得到起点是属于两个不同的集合，矛盾，所以必要性成立

充分性：如果没有奇数环，则一定是一个二分图

我们需要一种方法来构造一个图，图中的两个点分别属于两个不同的集合，每个集合的内部没有边，即一个二分图

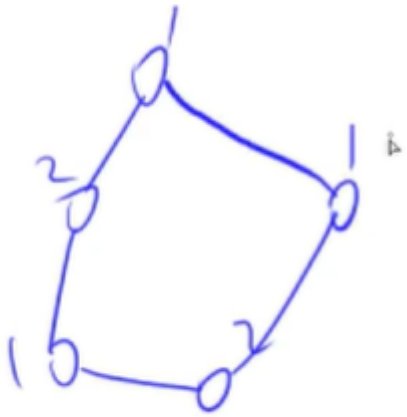
我们遍历所有的点，当这个点没有分组的时候，我们将他归为第一个集合（染成白色），与他相连的边的点归为第二个集合（染成黑色），于是所有与这个点连通的点的归属也就确定了：



并且由于图中不含奇数环所以，我们这种方法（染色法）没有矛盾构成了一个二分图，也就一定可以得到一个二分图

下面来用反证法证明为什么没有矛盾：

假设不含奇数环但是出现了矛盾，则一定是下面这种情况，之前已经染过颜色的点与当前的点的颜色是相同的：



有矛盾，说明这个环的点的个数一定是奇数，因为没有矛盾的话，属于集合 1 与属于集合 2 的点的个数一定是相同的，所以这个环一定是一个奇数环，跟没有奇数环冲突，所以染色法构造二分图一定没有矛盾

染色法判断二分图

基本思想

同上用染色法来判断是不是二分图

只要一个图在染色过程中**出现了矛盾**，构造二分图失败，就说明是存在奇数环的（充分性的逆否命题）

所以不是一个二分图（必要性的逆否命题）

只要一个图在染色过程中**没有矛盾**那就是一个二分图（这个方法本身就是一个构造二分图的方法，构造成功也就构造了一个二分图）

也就没有了奇数环（必要性）

伪码：

```
for(i = 1; i <= n; i ++)
    if(i没有染色)
        dfs(i, 1);//将i的连通块的所有的点染色
```

代码实现

给定一个 n 个点 m 条边的无向图，图中可能存在重边和自环。

请你判断这个图是否是二分图。

输入格式

第一行包含两个整数 n 和 m。

接下来 m 行，每行包含两个整数 u 和 v，表示点 u 和点 v 之间存在一条边。

输出格式

如果给定图是二分图，则输出 `Yes`，否则输出 `No`。

数据范围

$1 \leq n, m \leq 10^5$

输入样例：

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

输出样例：

```
Yes
```

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;

//由于是无向图，要向连接表中存两次，所以M开到题中的数据的两倍
const int N = 100010, M = 200020;

int n, m;
```

```cpp
//邻接表的方式来存储图
//h[]用来存每个表的表头，个数最大是N
//每个链表的元素包括值与下一个元素的地址，用下标关联起来
//每个元素的值就是当前点对应的出边的点也就是e， 每个元素中的下一个元素的地址就是ne
//每个边对应一个元素
//所以e, ne的个数得开到M
//将idx理解为内存中的地址，e[], ne[]理解为内存中的空间
//地址相同的空间存放值与下一块元素内存的地址
int h[N], e[M], ne[M], idx;

int color[N];

//插入一条边到对应链表的头部
void add(int a, int b)
{
    //给空闲空间赋值
    e[idx] = b;
    //空闲空间的指针域指向头指针指向的地址
    ne[idx] = h[a];
    //头指针指向分配的空间地址
    h[a] = idx;
    //指向下一个待分配的空间
    idx ++;
}

bool dfs(int u, int c)
{
    //将当前点的颜色记录为c
    color[u] = c;

    //从前往后遍历当前点的所有的临点
    for(int i = h[u]; i != -1; i = ne[i])
    {
        //记录当前点的某一个临点
        int j = e[i];
        //如果当前这个点的一个临点没有被染色
        if(!color[j])
        {
            //我们将其染成与当前点不同的颜色
            if(!dfs(j, 3 - c))
            {
                //如果染色失败，返回false
                return false;
            }
        }
        //如果当前点的临点已经被染色了，而且染的是跟当前点一样的颜色
        else if(color[j] == c)
        {
            return false;
        }
    }
    //如果成功染色，返回true
```

```
    return true;
}


int main()
{
    scanf("%d%d", &n, &m);

    memset(h, -1, sizeof h);

    while(m --)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        //加入边，由于是无向边所以得添加两次边
        add(a, b);
        add(b, a);
    }

    bool flag = true;//用来表示是否有矛盾发生

    for(int i = 1; i <= n; i ++)
    {
        //如果当前这个点没有被染色
        if(!color[i])
        {
            //染成第一种颜色，如果返回false，认为有矛盾发生
            if(!dfs(i, 1))
            {
                //染色失败，退出
                flag = false;
                break;
            }
        }
    }

    if(flag == true) puts("Yes");
    else puts("No");


    return 0;
}
```

1. 分析下面这两段核心代码

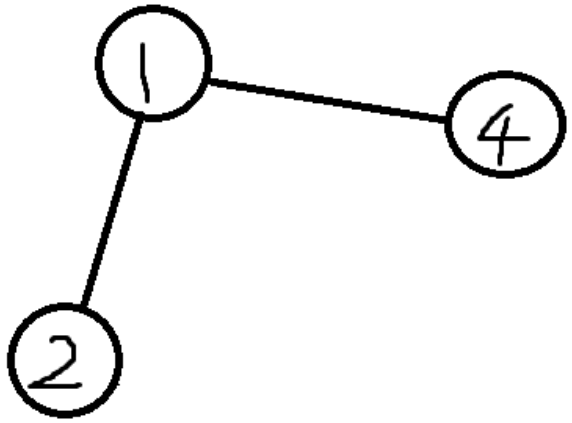
```
bool dfs(int u,int c)
{
    color[u]=c;
    for(int i=h[u];i!=-1;i=ne[i])
    {
        int j=e[i];
        if(!color[j])
        {
          if(!dfs(j,3-c))return false;
        }
        else if(color[j]==c)return false;

    }
    return true;
}
```

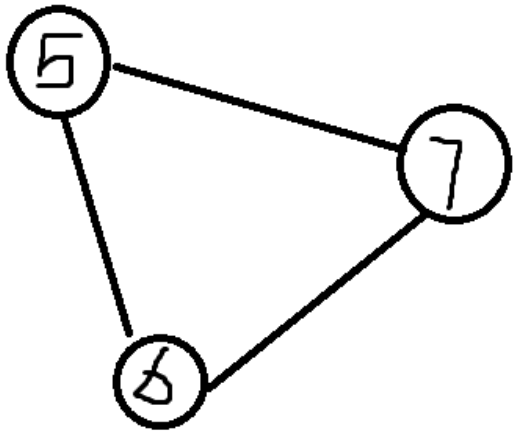
```
bool flag=true;
for(int i=1;i<=n;i++)
{
    if(!color[i])
    if(!dfs(i,1))
    {
        flag=false;
        break;
    }
}
```

要弄清楚上面的递归程序，我们需要知道两件事情

1. 递归过程
2. 结束条件

要理解上面两件事，最好的方法是举一个最简单的例子，我们可以从树的叶子结点开始一步步向上理解，我们以下面的这个图为例：



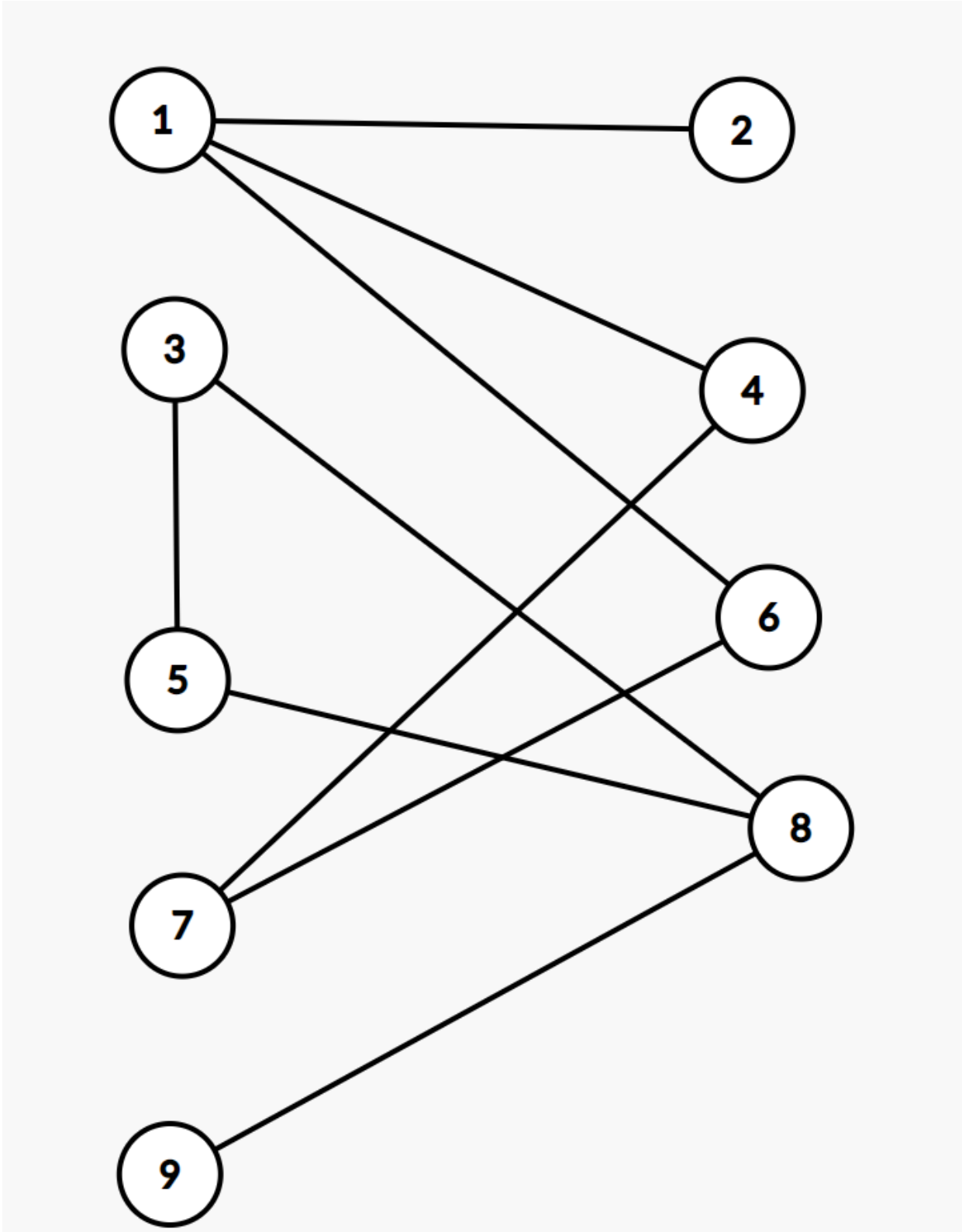


1. 进入主函数的`for`循环，从点`1`开始
    1. `1`没有染色，进入`dfs(1, 1)`
        1. 将其染成第一种颜色，遍历`1`的所有邻接点`2`，`4`
        2. `2`没有染色，进入`dfs(2, 2)`
            1. 将其染成第二种颜色，遍历`2`的所有邻接点`1`
            2. `1`被染色但是是与`2`不同的颜色，没有矛盾
            3. 跳出`2`的`for`循环，染色完成，返回`true`
        3. `true`返回到语句`if(!dfs(2, 2))`，进入下一次`for`循环，遍历点`4`
        4. `4`没有被染色，进入`dfs(4, 2)`
            1. 将其染成第二种颜色，遍历`4`的所有邻接点`1`
            2. `1`被染色但是与`4`的颜色不同，没有矛盾
            3. 跳出`4`的`for`循环，染色完成，返回`true`
        5. `true`返回到语句`if(!dfs(4, 2))`，跳出`for`循环

6. `1`的染色完成，返回`true`

2. `true`返回到主函数语句`if(!dfs(1, 1))`，进入下一次`for`循环，遍历下一个没有被染色的点
    1. `5`没有被染色，进入`dfs(5,1)`
        1. 将其染成第一种颜色，遍历`5`的所有邻接点`6`， `7`
        2. `6`没有染色，进入`dfs(6, 2)`
            1. 将其染成第二种颜色，遍历`6`的所有邻接点`5`，`7`
            2. `5`被染色但是是与`6`不同的颜色，没有矛盾
            3. `7`没有被染色，进入`dfs(7, 1)`
                1. 将`7`染成第一种颜色
                2. 遍历`7`的邻接点`5`，`6`
                3. `5`被染色，但是染的颜色与`7`相同，出现矛盾，说明染色失败返回`false`
        3. `7`的`dfs(7, 1)`返回的`false`返回到`6`的语句`!dfs(7, 1)`
        4. 于是`6`这里也染色失败，返回`false`
    2. `6`的`false`返回到`5`的语句`!dfs(6, 2)`，于是`5`也染色失败，返回`false`
3. `5`的`false`返回到主函数的语句`if(!dfs(5, 1))`，整个染色失败，`flag = false`退出循环

引用`acwing`用户`风雨zzm`的题解过程：


1
2
3
4
6
5
8
7
9

----->从主函数遍历，先进入1号点，将1号点染为颜色1，枚举1号点的所有邻点2，4，6号点

--->进入2号点，将2号点染为颜色2，枚举2号点的所有邻点1号点，1号点已经染过色，且与2号点颜色不同
<---离开2号点

--->进入4号点，将4号点染为颜色2，枚举4号点的所有邻点1，7号点，1号点已经染过色，且与4号点颜色不同
-->进入7号点，将7号点染为颜色1，枚举7号点的所有邻点4，6号点，4号点已经染过色，且与7号点颜色不同

->进入6号点，将6号点染为颜色2，枚举6号点的所有邻点1，7号点，1，7号点已经染过色，且与6号点颜色不同
<-离开6号点

<--离开7号点
<---离开4号点

<----6号点已经染过色，离开1号点

2号点已经染色，跳过

----->进入3号点，将3号点的颜色染为1，枚举3号点所有邻点5，8号点

--->进入5号点，将5号点的颜色染为2，枚举5号点的所有邻点3，8号点，3号点已经染过色，且与5号点颜色不同
-->进入8号点，将8号点的颜色染为1，枚举8号点的所有邻点3，5号点，发现3号点的颜色与8号点颜色相同，都为1，说明存在奇环
<--return false
<---return false

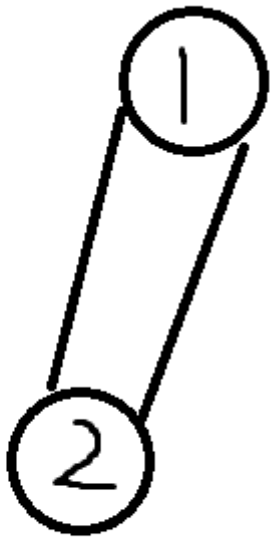
<----return false

flag=false，停止循环
输出No

我们可以看到这种递归程序与一般的简单的`dfs`相比他将递归函数本身嵌套进了`if`判断语句中，这颗子树染色成功，则进行下一棵子树的染色，这颗子树染色失败，则一步步返回到主函数

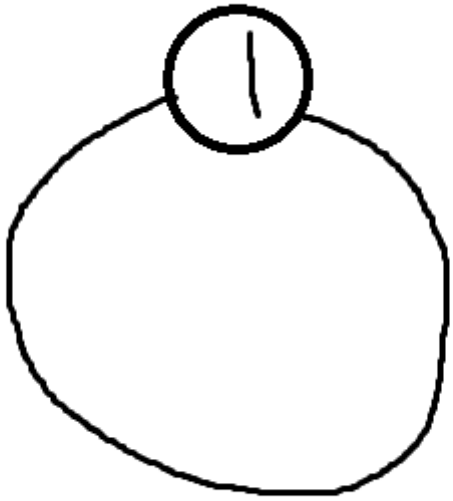
2. 对重边和自环的处理：

当出现重边的时候：



其实没有影响，将`1`染成一种颜色后开始遍历`1`的所有邻接点`2`，`2`没有被染色，然后染成另一种颜色，然后`2`，染色成功返回到`1`，然后继续遍历`2`（我们存边的时候存了重复的边进去），但此时`2`已经染色了，跳过`1`的这次`for`循环，`1`染色成功，返回。

当出现自环的时候：



很显然这是一个奇数环，遍历`1`的所有邻接点，就是`1`自己，自己已经被染色，而且染的颜色与自己相同，染色失败，接着一步步返回到主函数

匈牙利算法（NTR算法）寻找二分图的最大匹配

基本思想

几个基本概念：

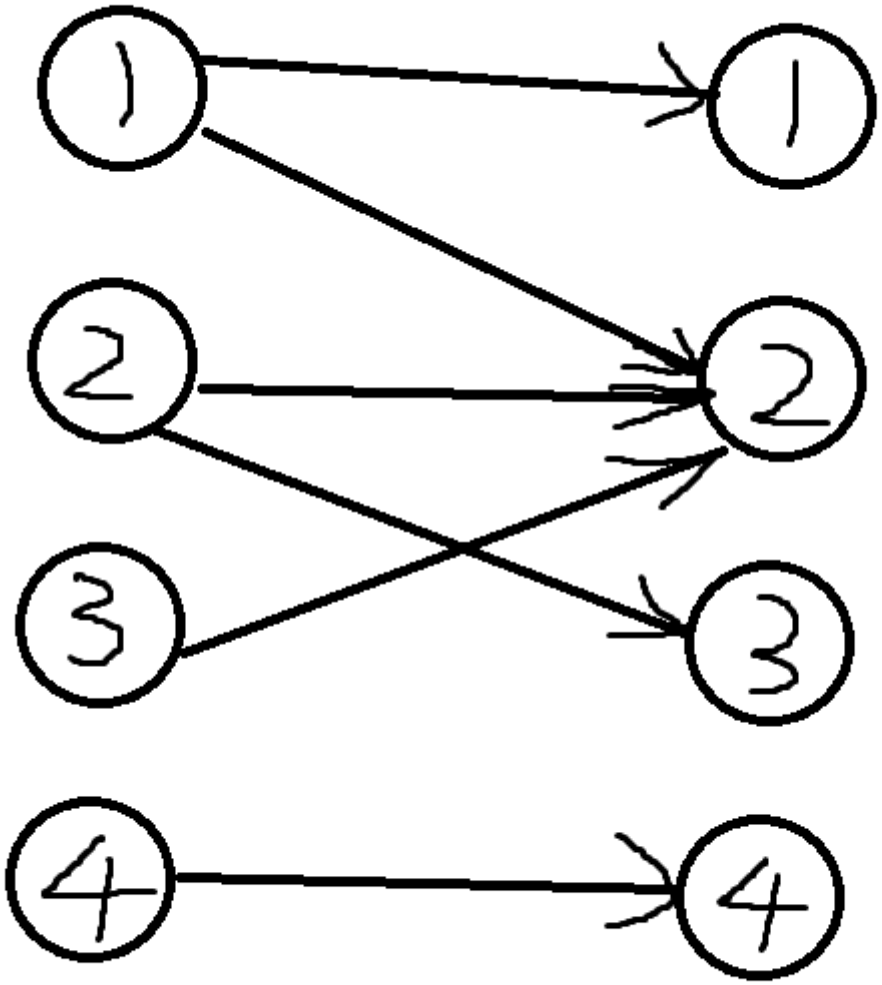
二分图的匹配：给定一个二分图`G`，在`G`的一个子图`M`中，`M`的边集`{E}`中的任意两条边都不依附于同一个顶点，则称`M`是一个匹配。

二分图的最大匹配：所有匹配中包含边数最多的一组匹配被称为二分图的最大匹配，其边数即为最大匹配数。

这里的匹配指的是二分图的一个子图，不过我们在下面的文字中，匹配指的是两个点之间连一条边

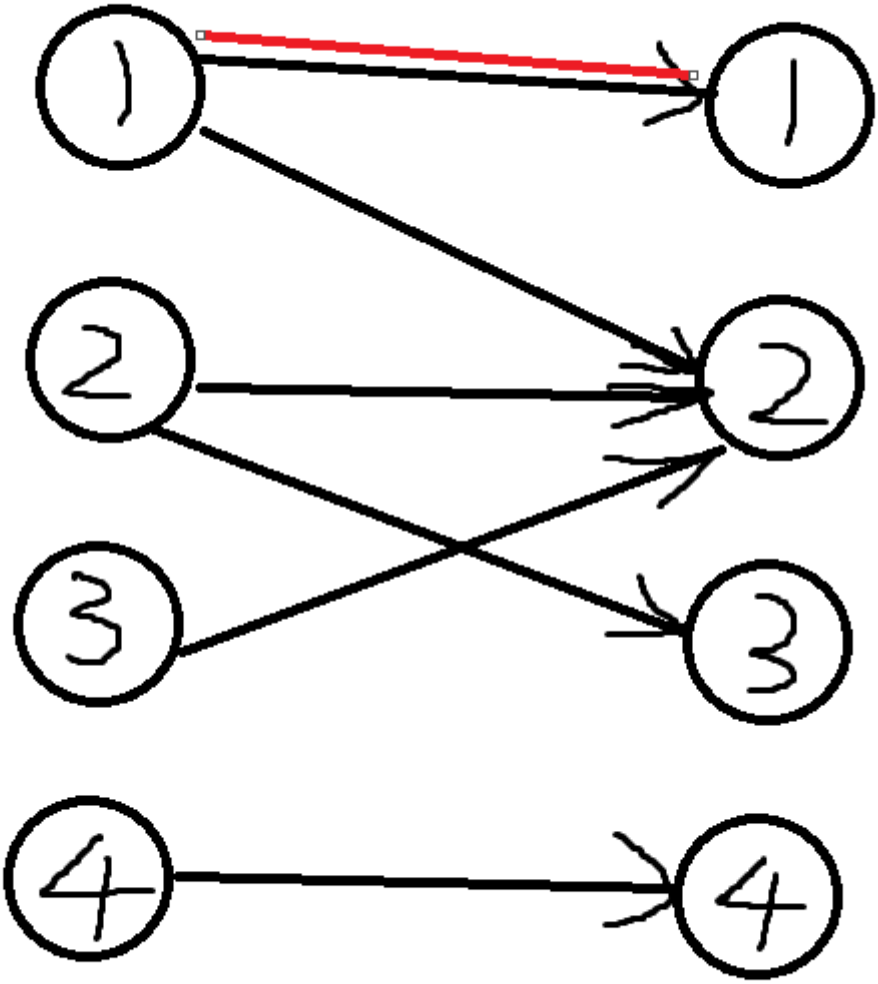
我们来将这个算法形象一下，将左边的点称为男生，将右边的点称为女生，原图的两者连线成为男生对女生有好感，匹配成功的那条线，称为配对成功

如下图：

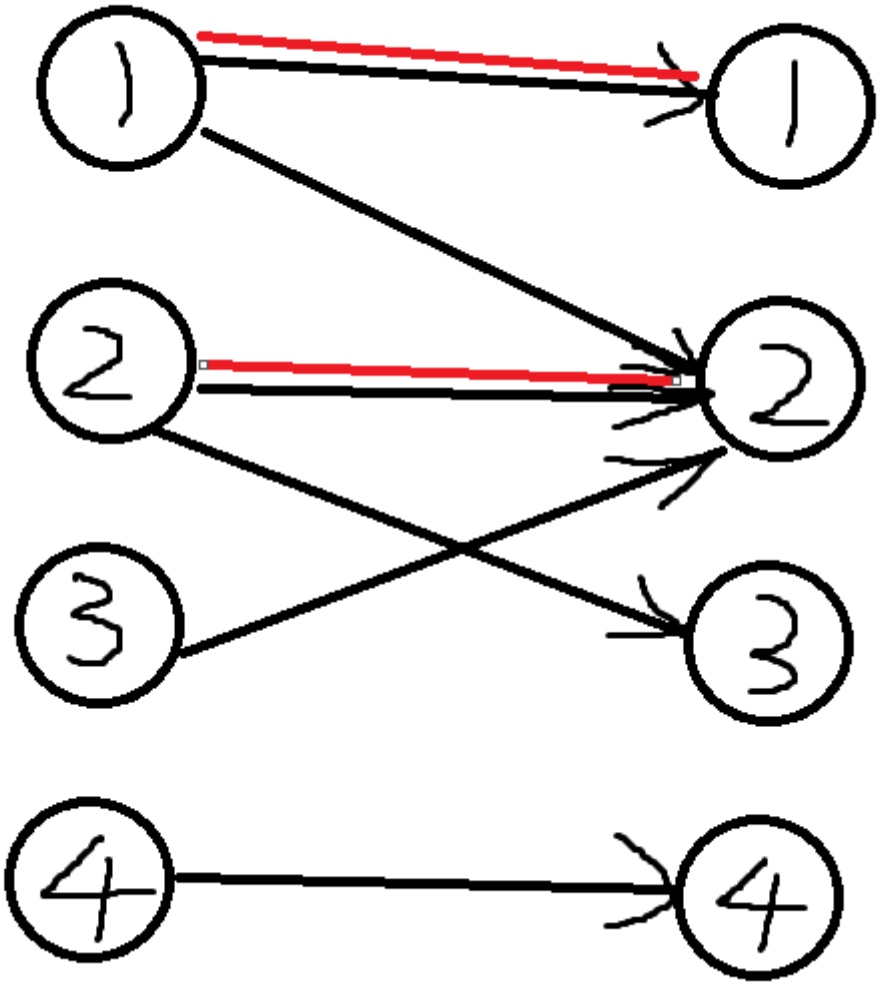


为什么这个算法叫做 NTR 算法呢，我们来看过程

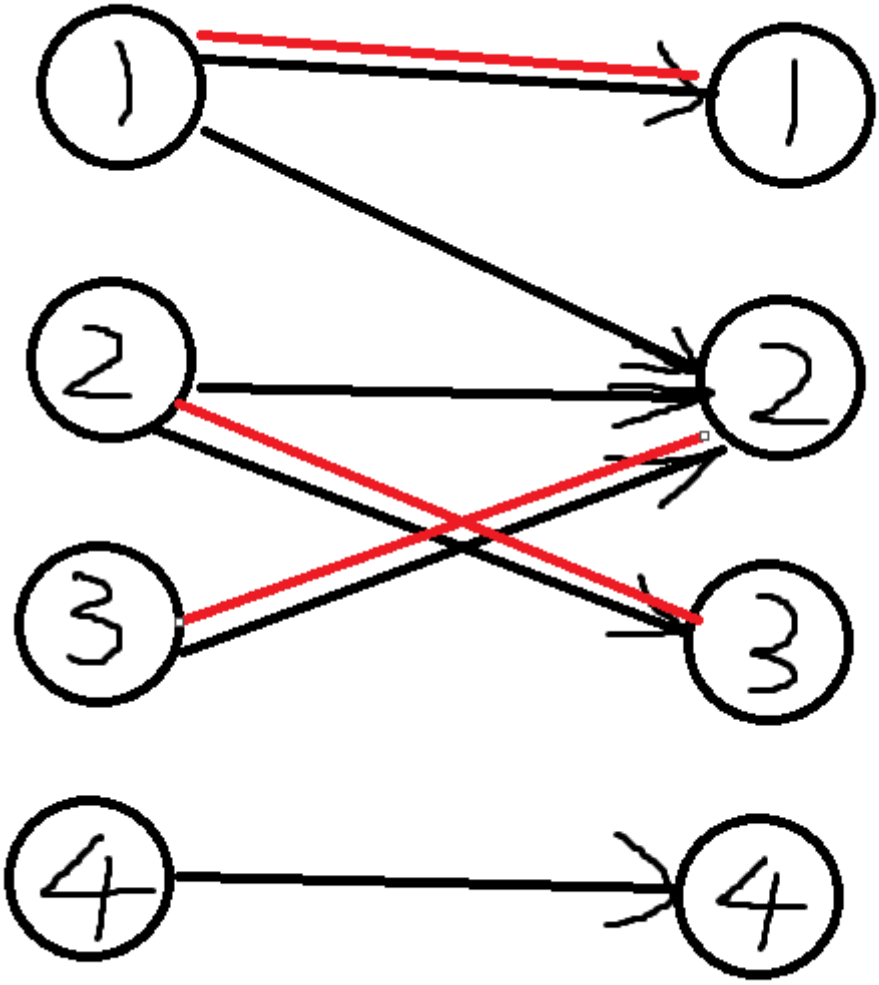
首先，从第一个男生开始配对，遍历右边他有好感的女生，找到一个还没有配对的女生，与第一个配对成功：



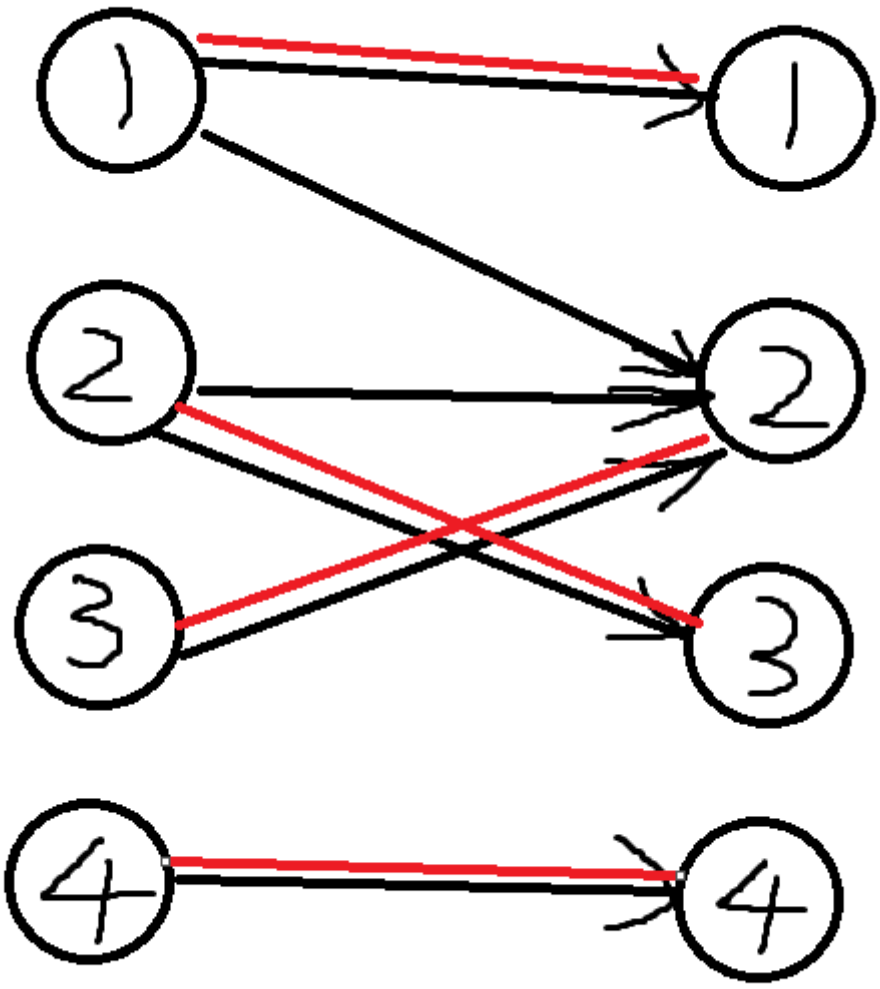
然后是第二个男生，遍历右边他又好感的女生，找到了第二个女生还没有配对于是配对成功：



第三个男生，遍历右边他有好感的女生，注意看是第二个女生但是第二个女生已经被配对了，但是三号男生不会放弃，他会一直尝试，这时二号女生之前配对的二号男生会寻找下一个他又好感的女生也就是三号女生，二号男生开始与三号女生配对，三号男生跟二号女生配对，`NTR`成功：



最后是四号男生遍历他有好感的女生，也就是四号，配对成功，得到原图的最大匹配：



代码实现

给定一个二分图，其中左半部包含 n_1 个点（编号 $1 \sim n_1$），右半部包含 n_2 个点（编号 $1 \sim n_2$），二分图共包含 m 条边。

数据保证任意一条边的两个端点都不可能在同一部分中。

请你求出二分图的最大匹配数。

> 二分图的匹配：给定一个二分图 G，在 G 的一个子图 M 中，M 的边集 $\{E\}$ 中的任意两条边都不依附于同一个顶点，则称 M 是一个匹配。
>
> 二分图的最大匹配：所有匹配中包含边数最多的一组匹配被称为二分图的最大匹配，其边数即为最大匹配数。

输入格式

第一行包含三个整数 n_1、 n_2 和 m。

接下来 m 行，每行包含两个整数 u 和 v，表示左半部点集中的点 u 和右半部点集中的点 v 之间存在一条边。

输出格式

输出一个整数，表示二分图的最大匹配数。

数据范围

$1 \leq n_1, n_2 \leq 500,$
$1 \leq u \leq n_1,$
$1 \leq v \leq n_2,$
$1 \leq m \leq 10^5$

输入样例：

```
2 2 4
1 1
1 2
2 1
2 2
```

输出样例：

```
2
```

```cpp
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;
const int N = 510, M = 100010;

int n1, n2, m;

int h[N], e[M], ne[M], idx;
//match存右边的点对应的点
//match[i]表示右边的点i匹配的左边的点
int match[N];

//用来判重，表示每次左边的点匹配的时候不要重复搜索右边的一个点
bool st[N];

void add(int a, int b)
```

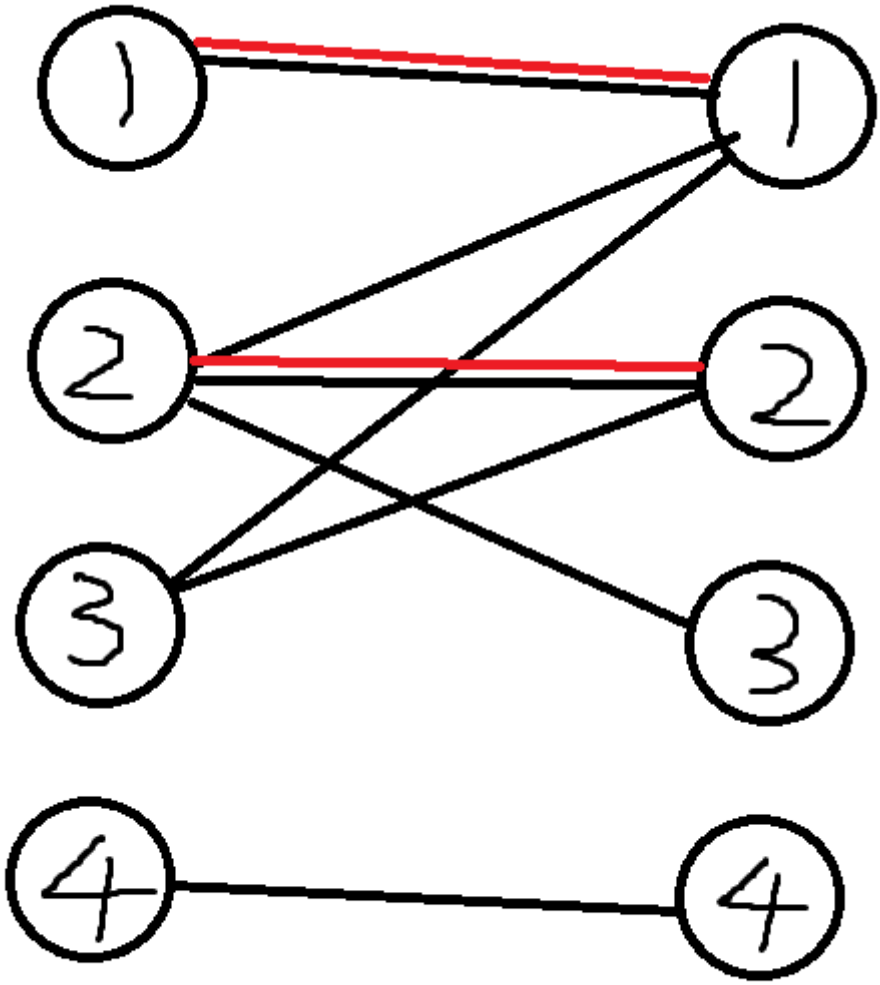
```
{
    e[idx] = b;
    ne[idx] = h[a];
    h[a] = idx;
    idx ++;
}

bool find(int x)
{
    //遍历左边这个点可能的边
    //也就是遍历左边这个点连向右边的边
    for(int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];

        if(!st[j])//如果右边这个点没有尝试过
        {
            st[j] = true;//进行尝试
            //如果右边这个点没有匹配对象
            //或者右边这个点之前匹配的左边的点可以找到下一个匹配对象
            if(match[j] == 0 || find(match[j]))
            {
                //将右边这个点的匹配对象设置为左边这个点x
                match[j] = x;
                //表示匹配成功，返回true
                //只要在递归中配对成功就会一直返回true到主函数，表示路径上的所有男生更换女朋友成功
                return true;
            }
        }
    }
    //遍历完男生所有有好感的女生都无法配对到对象，表示配对失败
    //return返回到主函数表示主函数配对的这个男生配对失败
    //return返回到调用的地方表示调用这个函数的男生ntr别人失败，调用这个函数的男生继续尝试配对下一个有好感女生
    return false;
}


int main()
{
    scanf("%d%d%d", &n1, &n2, &m);
    memset(h, -1, sizeof h);
    while(m --)
    {
        int a, b;
        scanf("%d%d", &a, &b);
        //加边的话只用加左边的点到右边的边
        //因为我们遍历的时候总是遍历男生有好感的女生
        add(a, b);
    }
    //res存的是当前匹配的数量
```

```
    int res = 0;
    //依次看左边的点看与右边哪个点进行匹配
    for(int i = 1; i <= n1; i ++)
    {
        //每次左边的点开始匹配的时候先将右边的点的所有状态设置为0
        //表示没有进行尝试
        memset(st, -1, sizeof st);
        //看这个左边的点能否找到右边的一个匹配的点
        //如果可以找到，就将匹配的的边数加一
        if(find(i)) res ++;
    }
    printf("%d", res);
    return 0;
}
```

需要注意的几个地方：

1. `st[]` **数组的作用是什么？**

`st[]` 数组是用来避免重复的尝试的，比如下面这个图，在配对到三号男生的时候是这样一种情况：



三号一开始将所有的`st[] = 0`表示自己没有进行尝试，首先从一号女生开始，表示自己尝试过了`st[1] = true`，这时由于一号男生只能与一号女生配对，所以三号男生不能去抢人家女朋友，三号男生跟一号女生配对失败，然后三号男生开始跟二号女生进行尝试`st[2] = true`，这时递归到二号男生，二号男生开始对自己有好感的女生进行尝试，那么显然，因为三号男生抢一号女生失败，二号男生也必然抢一号女生失败，于是直接跳过了一号(`if(!st[1])`)，而二号正在被别人抢，如果不跳过会进行无穷递归，所以也跳过，于是直接尝试三号女生

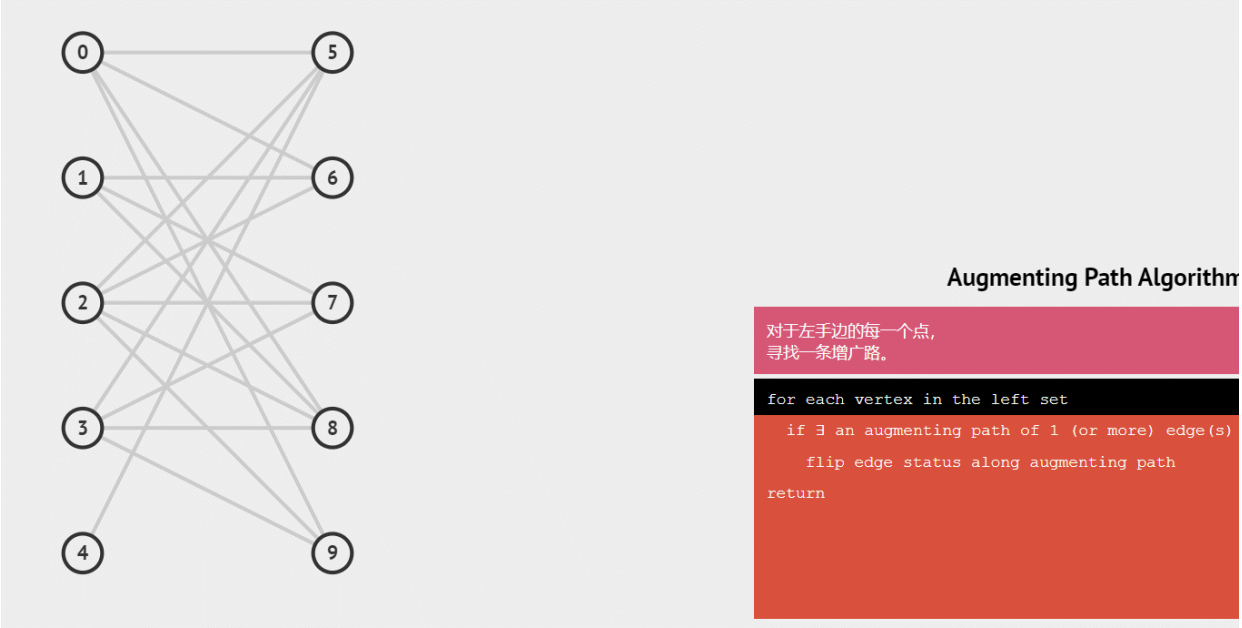
所以`st[]`数组的作用就是在一轮男生配对中跳过后面递归过程注定不可能的配对

那么为什么`st[]`数组需要重置呢? 我们注意到，二号男生与二号女生配对后二号女生对二号男生是尝试过的，但是在三号男生配对中，二号女生不一定是注定配对失败的，三号男生仍然可以把二号女生抢过来，所以在新一轮的配对过程中将`st[]`数组重置，表示所有的女生三号男生都有可能配对成功。

2. **递归过程分析，以三号男生尝试二号女生为例：**

1. 三号男生开始尝试二号女生，即自己第二个有好感的女生
2. `st[2] = true`表示自己尝试过了
3. 二号女生已经被配对过了且`match[2] = 2`，开始递归重新尝试二号男生`find(2)`
    1. 一号，二号女生都被尝试过了，三号男生直接尝试三号女生，`st[3] = true`
    2. 三号女生没有被匹配过，三号男生匹配成功，返回`true`
4. 返回到二号男生的匹配过程，`if(match[2] == 0 || find(match[2]))`为真，表示二号男生更换配对成功
5. 将`match[2]`改为`3`，表示二号女生与三号男生配对
6. 三号男生配对成功，返回`true`到主函数
7. `res ++`表示最大匹配的边数`++`

见下面的动图：



这个递归过程与`dfs`有点类似，如果感兴趣与`dfs`的关系可以看这篇题解