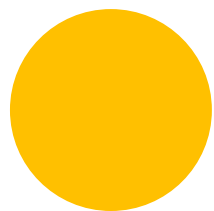
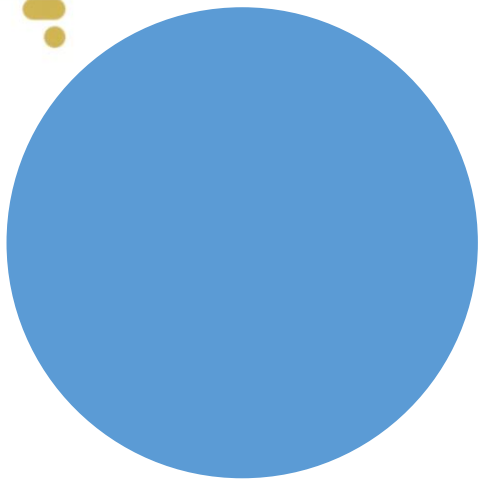


马上开始

35tang-C++竞赛系列三阶课程





《 35tang-C++竞赛系列三阶课程 》

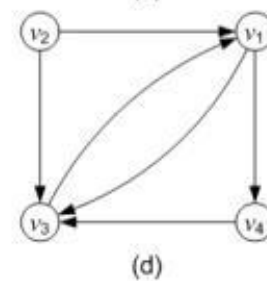
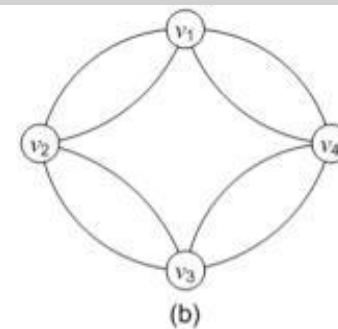
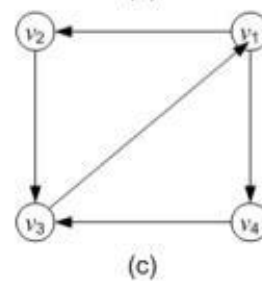
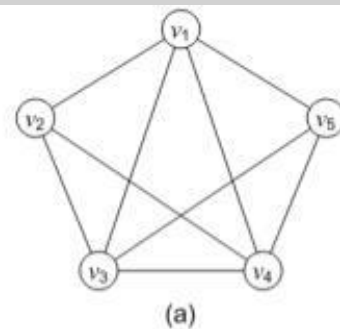


本节目标

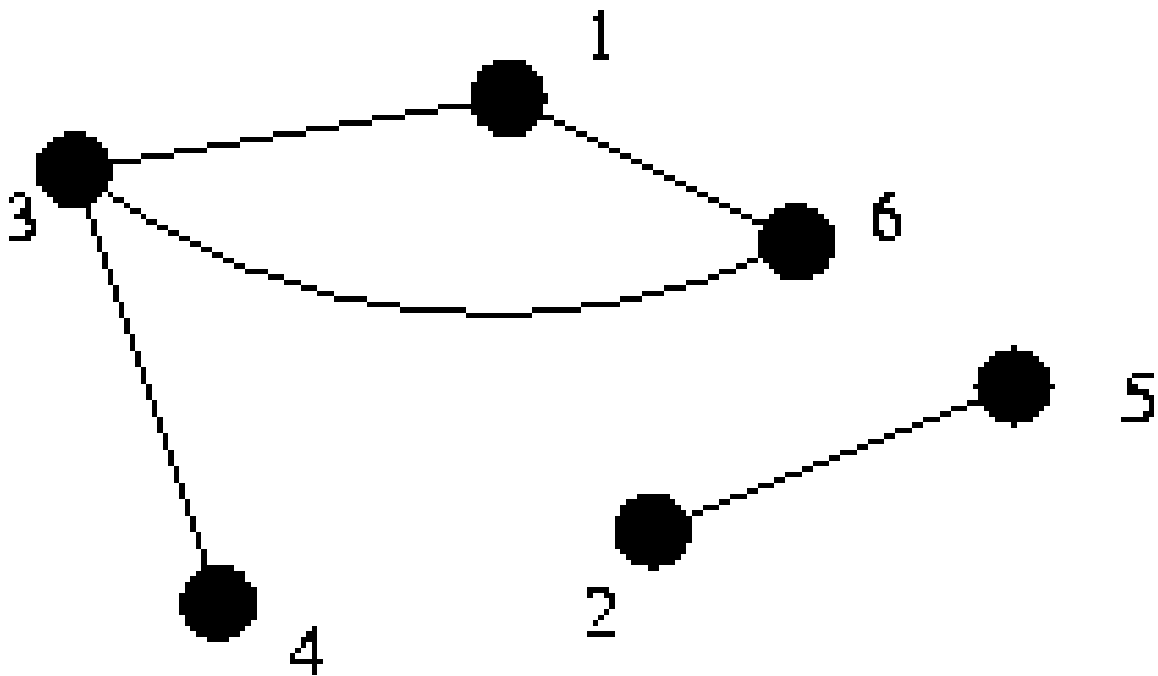
- 简单图
- 图搜索
- 寻找联通



- 度, 入度和出度
- undirected graph 无向图
- directed graph 有向图
- 连通图, 连接图, 任意两点都是连通的
- 树: 任意两点之间的简单路径有且仅有一条 (n 个点, $n-1$ 条边的连通图)
- 完全图其中边数为 $n*(n-1)/2$, 每对不同的顶点之间都恰连有一条边相连
- 欧拉图: 可以一笔画出来的图 (奇数点的点的个数为2或者为0)



图的例子：下图表示了一些点和他们的连接



- 一般节点从1开始编号，比如左图有6个节点，编号分别是1, 2, 3, 4, 5, 6
- 两个点相邻：
通过线直接连接
比如1和6, 1和3, 2和5...
- 两个点连通（连接）：
一个点可以通过他的相邻点连通到另外一个点

比如1和4连通, 1和2不连通
- 研究的问题：
 - 某两个点是否能通过这些连线到达
 - 一共多少个子连通图，比如这里2个

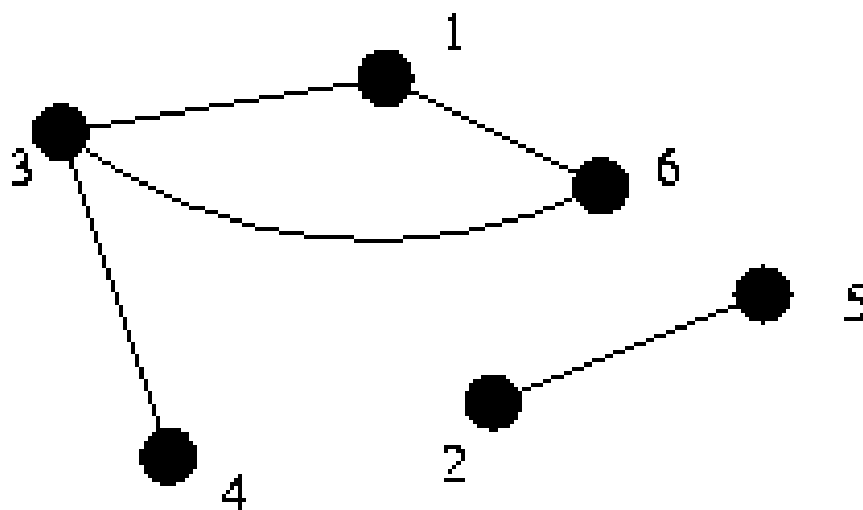
图的数据结构

- 就是要有一种数据结构，可以通过这个数据结构访问到图的信息
- 图的信息是什么？上面的例子是：这些节点如何连接（通过1个点可以找到他的相邻节点）还有的图中：有边的权值（就是从1个点到另外1个点的代价），有点的权值
- 本节课只讨论节点如何连接的问题，暂时不讨论权值



树

- 树可以说是一种特殊的图（无环，所以一个点的父亲只会有一个）。他们的节点关系可以用parent或者child来表示。比如二叉树，我们就可以用节点的left_child和right_child指向子节点，从而完成了关系的表述。
- 右图不是树，不能用这种数据结构表示，为什么？比如1和3节点，谁是谁的child？谁是谁的parent？



图的数据结构1

Edge List

结构体数组：每一个元素包含一个边的两个点

优点：边少的化比较节省存储

缺点：找一个点和哪些点相邻，需要遍历所有边

比较合适的场景：需要遍历所有的边。

	V_1	V_2
e_1	4	3
e_2	1	3
e_3	2	5
e_4	6	1
e_5	3	6

图的数据结构2

Adjacency Matrix

- `int grid[100][100]`
- 矩阵:二维数组, 比如`grid[1][2]`就表示点1和点2有连接 (有直接的边)
- 优点: 简单, 判断两个点是否相邻很容易
- 缺点: 如果边不多, 但是点很多, 二维数组就会太大, 而且找一个点的所有相邻点比较费时间。所以点不能太多。
- 比较合适的场景: 点不是很多; 每一个点的相邻边很多; 可能要判断两个点是否相邻。

	V ₁	V ₂	V ₃	V ₄	V ₅	V ₆
V ₁	0	0	1	0	0	1
V ₂	0	0	0	0	1	0
V ₃	1	0	0	1	0	1
V ₄	0	0	1	0	0	0
V ₅	0	1	0	0	0	0
V ₆	1	0	1	0	0	0

图的数据结构3

Adjacency List

`vector<vector<int>> grid(100)`

二维数组：每一个点一个数组，里面包含这个点的所有相邻点（有边直接连接的点）

当然，如果每一个点的边不多，并且知道上限，也可以用静态数组

优点：找一个点的相邻点比较容易；存储比较节省

缺点：判断两个点是否相邻不容易

比较合适的场景：点多边少；需要找到某个点所有的相邻点

Adjacent	
Vertex	Vertices
1	3, 6
2	5
3	6, 4, 1
4	3
5	2
6	3, 1

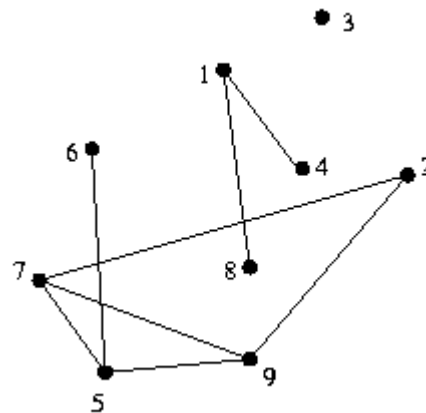
无向图和有向图

- 无向图：
 - 边都是双向的或者说无向的
 - 如果点a是点b的相邻点，那么点b也是点a的相邻点
- 有向图：
 - 边是单向的或者说带箭头的
 - 如果a连接到b，不代表b可以连接到a

图的联通 (component) 搜索: Flood Fill

BFS

DFS



右侧图10个节点构成了3个联通图，分别包含若干个节点。可以定义数组components[]来保存 component numbers（每一个联通的编号），比如components[1]表示1号节点所在的component编号，一共编号1到3。这里的flood fill就是要填写1到3给components数组。图中：

component[1]= component[4]= component[8]=1;

component[2]= component[5]= component[6]= component[7]= component[9]=2;

component[3]=3

DFS一般就是从从一个顶点开始，找和他有连接的相邻的点，然后递归找这些邻居节点，一直到找到或者找完位置，一次找到的都是一个component的。

BFS就是从从一个顶点开始，不断地把他的邻居节点加入队列，进行处理，处理后把处理过的邻居节点再加入队列，直到队列为空。一次对列不空，找到的都是一个component的。

给定若干个城市，编号1到n， ($n \leq 10^3$)，他们之间m条双向道路($m \leq 10^5$)。

输入：第1行2个整数n, m空格隔开，后面第2行到m+1行为m条边，每行两个整数a b表示节点a和节点b有双向道路。

输出：一个整数，最少还需要修建多少条道路才能保证任意两个城市之间都可以实现交通（不一定有直接的道路相连，只要互相间接通过道路可达即可）不需要输出0.

示例输入1:

9 9

1 2

3 1

2 4

5 2

3 6

4 7

8 4

4 9

5 4

示例输出1:

0

示例输入2:

9 8

1 2

3 1

2 4

5 2

3 6

4 7

8 9

5 4

示例输出2:

1

例题

一起画图

示例输入2:

9 8

1 2

3 1

2 4

5 2

3 6

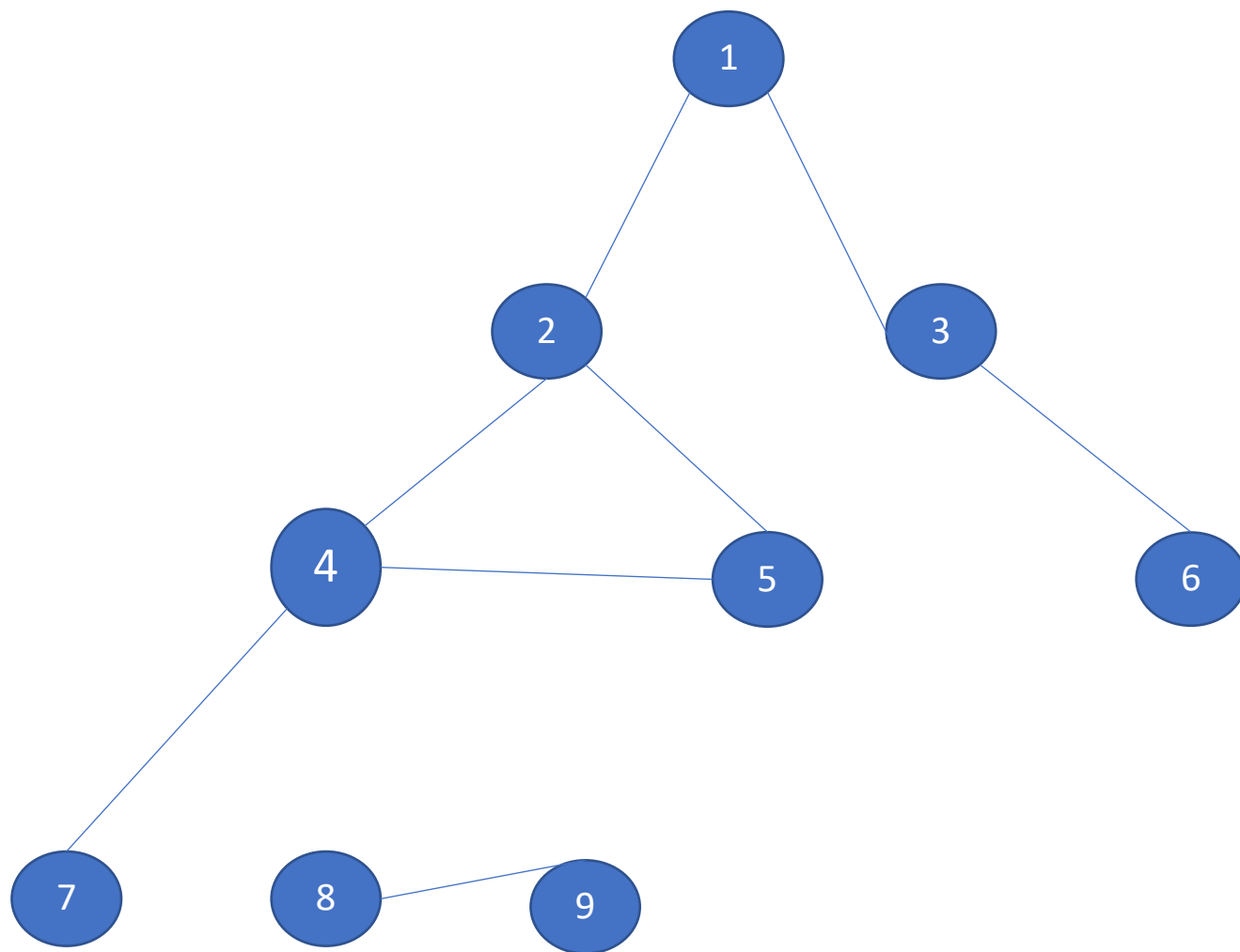
4 7

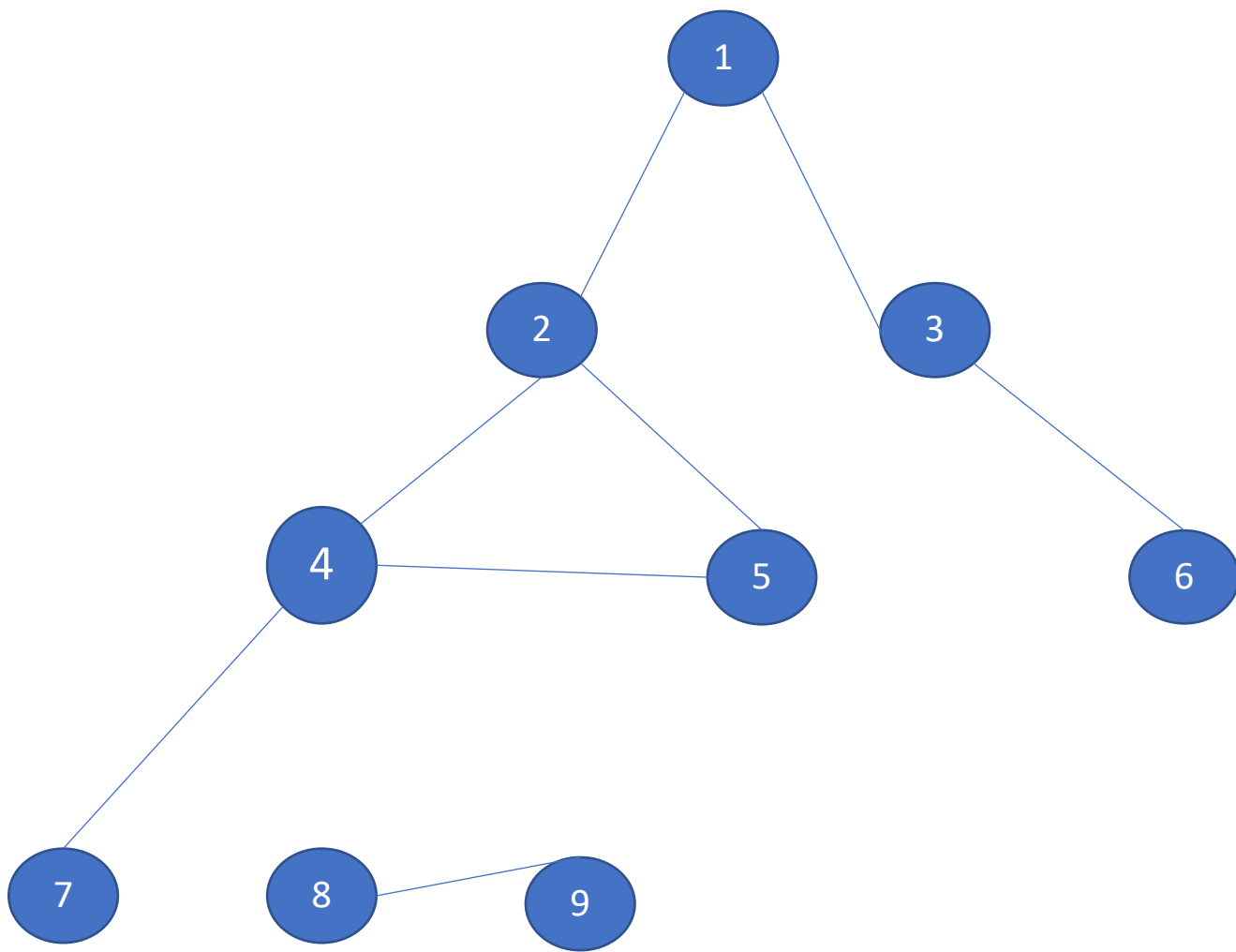
8 9

5 4

示例输出2:

1





实际上就是找有几个连通子图,有 n 个联通子图就需要 $n-1$ 个新的边(路)连接

9 8

1 2

3 1

2 4

5 2

3 6

4 7

8 9

5 4

$n=9, m=8$

n 用来遍历所有的点

m 用来遍历所有的边



数据结构

- 如何存储图的信息，就是所有点和边的信息，就是如何有一个数据结构可以让我们很容易找到某一个点的相邻节点有哪些。
- 其实用Adjacency List，就是第三种结构最好，因为我们无论DFS还是BFS找一个节点的相邻节点其实就是找一个点有边的其他点。

```
vector< vector<int> > grid(n);
```

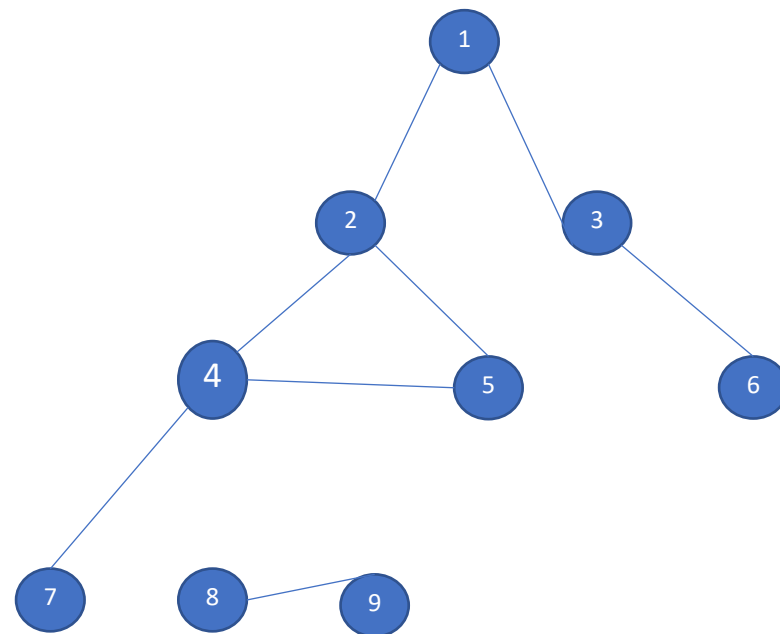
//grid[i]是一个vector数组，存放节点i所有相邻节点

//(n)是初始化有n个节点，也就是保证可以直接访问grid[0]到grid[n-1]

- 节点数不大也可以以用第二种数据结构矩阵grid[][]来表示边，操作比较简单。grid[i][j]为1就是i和j有连接。注意： grid[i][j]为1， grid[j][i]也为1，这是无向图的特点。

DFS洪水填充

- 假设第一个连通子图的编号componentnumber是1，第二个是2，.....
- 设置数组component[]来表示每一个节点属于哪一个连通子图，初始值都为0，表示还没有处理过。 componentnumber初值为0。
- 循环遍历所有节点: 找到任意一个component[i]为0的i点， componentnumber++， DFS函数从这个i节点出发，通过grid[][]寻找相邻节点的方法，找到i点出发可以到达的所有节点，他们的component[]值都为 componentnumber。
- 循环处理结束后componentnumber就是就是连通子图的个数。
- 如右图，比如先找到1号， DFS把1到7号节点都找到， component[1]... component[7]都修改为1；然后找到8号节点，把component[8]和 component[9]修改为2；



DFS程序



一起写一起看



大家课后可以在网站这节课的“课堂练习和例题”里面找到源程序。

```

int main ()
{
    cin>>n>>m;
    for (int i=1;i<=m;i++) //读取并插入grid二维数组
    {
        int a,b;
        cin>>a>>b; //读入两个节点, 比如1, 3
        grid[a][b]=1;//ab连接
        grid[b][a]=1;//ba连接
    }

    //循环处理: componentnumber++;
    //找到任意一个component[i]为0的i点出发开始dfs
    for (int i=1;i<=n;i++)
        if (component[i]==0) {
            componentnumber++;
            dfsfill(i);
        }

    cout<<componentnumber-1<<endl;
    return 0;
}

```

```
void dfsfill(int sp)
```

/*sp为当前访问的节点编号

DFS函数从这个sp节点出发, 通过grid[][]寻找相邻节点的方法, 找到sp出发可以到达的所有节点, 他们的component[]值都为componentnumber。

```

*/
{
    for (int i=1;i<=n;i++) //寻找sp的相邻节点i
        if (grid[sp][i]&& component[i]==0)
        {
            component[i]=componentnumber;
            dfsfill(i);
        }

    return;
}

```

```
int n,m;
```

```
bool grid[1001][1001]; //保存连接信息
```

```
int component[1001]={0}; //记录输入哪个连通子图
```

```
int componentnumber; //当前处理第几个连通子图
```

BFS

思路和DFS基本类似

不过是找一个节点所有相邻节点的工作用queue循环处理

我们可以一起修改一下看看

并查集

并查集顾名思义就是有“合并集合”和“查找集合中的元素”两种操作的关于数据结构的一种算法。

图中连通的节点可以认为在1个集合，他们最后指向的根是一个

无向图的连通分量个数，初始时，将无向图中各个顶点视为不连通的子集合，对图中每一条边，相当于union这条边对应的两个顶点分别所在的集合，直至所有的边都处理完后，还剩下的集合的个数即为连通分量的个数。

给定若干个城市，编号1到n，（ $n \leq 10^3$ ），他们之间m条双向道路($m \leq 10^5$)。

输入：第1行2个整数n, m空格隔开，后面第2行倒m+1行为m条边，每行两个整数a b表示节点a和节点b有双向道路。

输出：一个整数，最少还需要修建多少条道路才能保证任意两个城市之间都可以实现交通（不一定有直接的道路相连，只要互相间接通过道路可达即可）不需要输出0.

示例输入1:

9 9

1 2

3 1

2 4

5 2

3 6

4 7

8 4

4 9

5 4

示例输出1:

0

示例输入2:

9 8

1 2

3 1

2 4

5 2

3 6

4 7

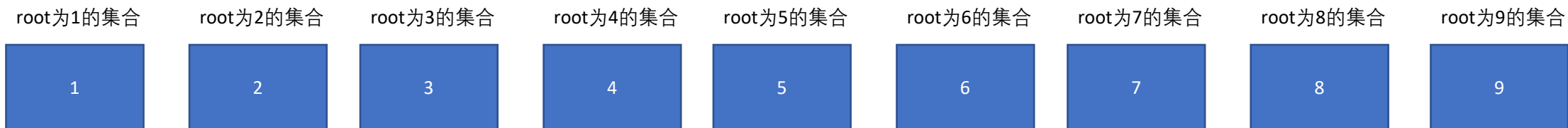
8 9

5 4

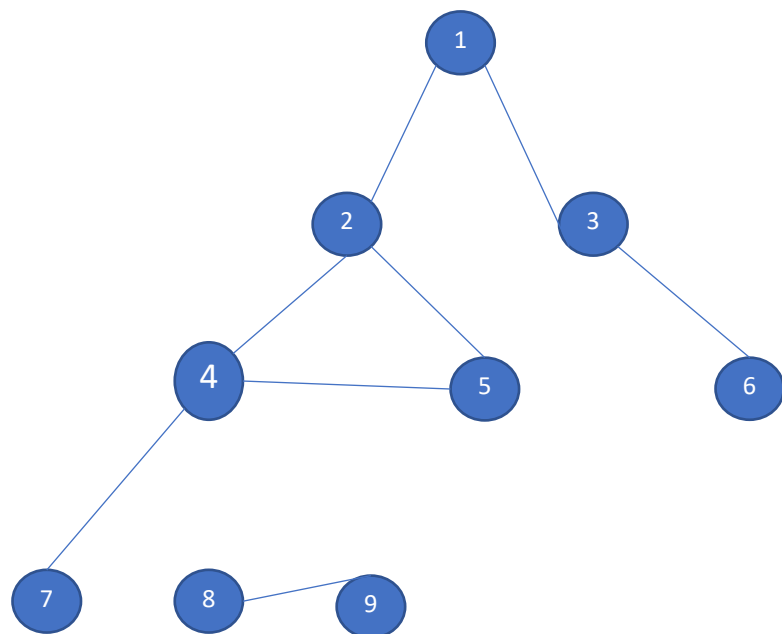
示例输出2:

1

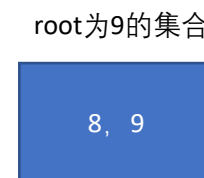
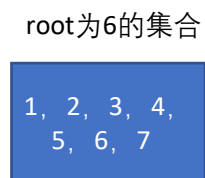
例题



9 8
1 2
3 1
2 4
5 2
3 6
4 7
8 9
5 4



从上面9个集合合并成下面的2个集合。
合并的原则是：两个点相邻（一个边的两个点）就把2个点所在的集合合并



如果合并成了2个集合，那么连通子图数目就是集合数目

基本做法

- 首先初始化parent[]数组让每一个节点都有一个parent，实际上就是每一个节点单独一个集合。然后我们有两个操作：
 - findroot(i) 就是找到节点i所在集合的根（parent等于自己的叫做根）；
 - union (i,j) 就是把i和j所在的集合合并（让i所在集合的根的parent等于j所在集合的根）。
- 判断i和j是否在一个集合就是看findroot(i)是不是等于findroot(j);
- 一起看刚才的示例数据的处理：



```
int n,m;  
int parent[1001];  
int findroot(int x)  
{  
    if (parent[x]==x) return x;  
    return (findroot(parent[x]));  
    //递归查找集合的代表元素也就是根元素  
}
```



依次读入每一条边并且处理，每一个集合内的元素递归找parent都能找到同一个根root

- 1 2 findroot(1)=1 findroot(2)=2 parent[1] =2 根2的集合{1, 2}
- 3 1 findroot(3)=3 findroot(1)=2 parent[3] =2 根2的集合{1, 2, 3}
- 2 4 findroot(2)=2 findroot(4)=4 parent[2] =4 根4的集合{1, 2, 3, 4}
- 5 2 findroot(5)=5 findroot(2)=4 parent[5] =4 根4的集合{1, 2, 3, 4, 5}
- 3 6 findroot(3)=4 findroot(6)=6 parent[4] =6 根6的集合{1, 2, 3, 4, 5, 6}
- 7 4 findroot(7)=7 findroot(4)=6 parent[7] =6 根6的集合{1, 2, 3, 4, 5, 6, 7}
- 8 9 findroot(8)=8 findroot(9)=9 parent[8] =9 根6的集合{1, 2, 3, 4, 5, 6, 7}, 根9的集合{8, 9}
- 5 4 findroot(5)=6 findroot(4)=6 parent[6] =6 根6的集合{1, 2, 3, 4, 5, 6, 7}, 根9的集合{8, 9}

上面每一个边处理过后，就修改了所有节点的parent



`parent[1] = 2; parent[2] = 4; parent[3] = 2; parent[4] = 6; parent[5] = 4; parent[6] = 6; parent[7] = 6;`
{1, 2, 3, 4, 5, 6, 7}里面每一个节点都可以递归访问parent也就是findroot()函数返回6;



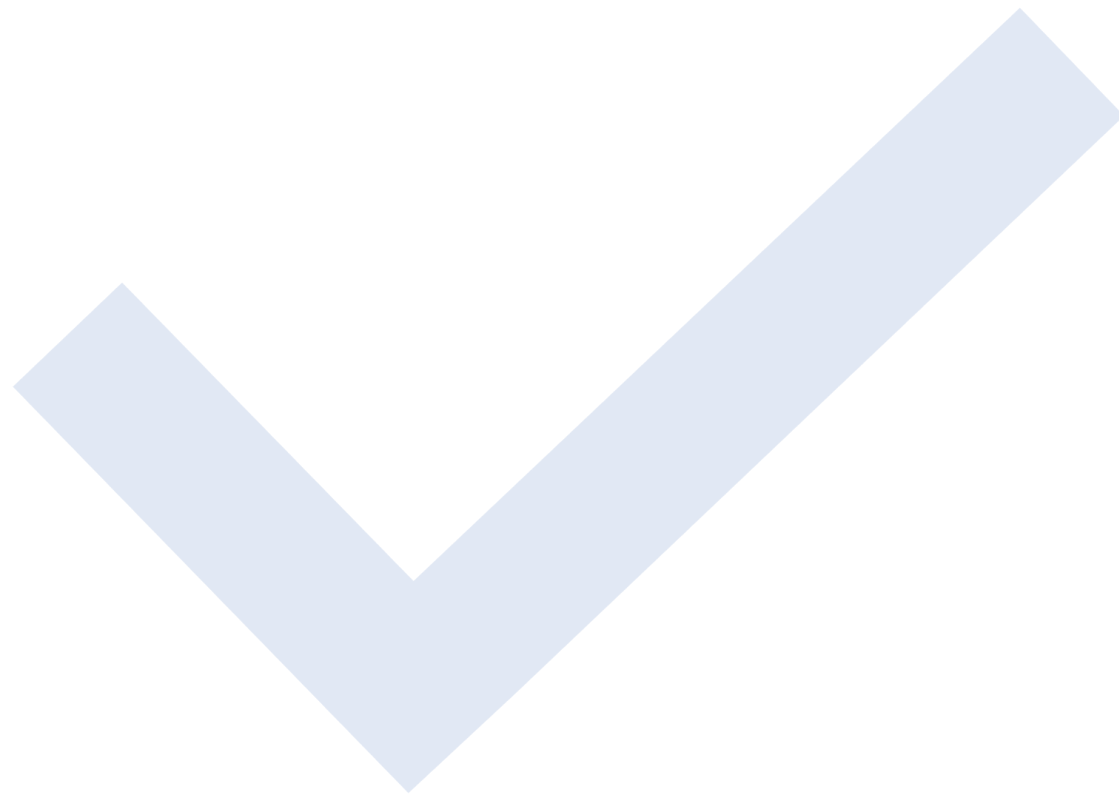
`parent[8] = 9; parent[9] = 9;` 根6的集合，根9的集合{8, 9}里面每一个节点都可以递归访问parent也就是findroot()函数返回9;



这样最后我们有几个连通就是有几个集合（或者说有几个`parent[i] == i`），而且访问每一个节点都能找到他的根，还可以通过比较根确定两个节点是否在一个连通（集合）里面。



一起写？



```
for(int i=1;i<=n;i++) parent[i]=i; //初始化n个集合
for (int i=1;i<=m;i++)      { //union的过程
    int a,b;  cin>>a>>b; //读入两个节点
    int x=findroot(a),y=findroot(b); //找到ab所在集合的根元素
    if(x!=y) parent[x]=y; //合并两个根所在集合
}
int cnt=0;

for(int i=1;i<=n;++i) findroot(i); //压缩

for(int i=1;i<=n;++i){ //统计集合个数，即为连通分量个数
    if(parent[i]==i) ++cnt;
}
cout<<cnt-1<<endl;
```

并查集：

读一条边就处理，边读便处理；

不需要存储整个图

压缩:

```
int n,m;
```

```
int parent[1001];
```

```
int findroot(int x)
```

```
{
```

```
    if (parent[x]==x) return x;
```

```
    //return (findroot(parent[x]));
```

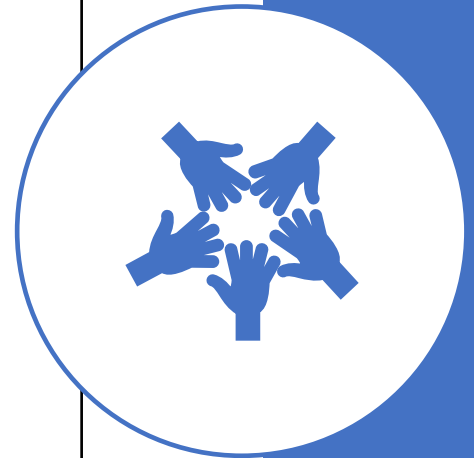
```
    return (parent[x]=findroot(parent[x]));
```

```
    //递归查找集合的代表元素也就是根元素
```

```
    //上面两种方式最后输出的可以比较一下，第二个层次更少
```

```
    //第二个和第一个的不同在于除了递归返回findroot还会赋值给当前的parent[x]
```

```
}
```



压缩，为什么要压缩？

i 1parent[i]2
i 2parent[i]4
i 3parent[i]4
i 4parent[i]6
i 5parent[i]6
i 6parent[i]6
i 7parent[i]6
i 8parent[i]9
i 9parent[i]9

for(int i=1;i<=n;++i) findroot(i);



i 1parent[i]6
i 2parent[i]6
i 3parent[i]6
i 4parent[i]6
i 5parent[i]6
i 6parent[i]6
i 7parent[i]6
i 8parent[i]9
i 9parent[i]9

迷宫（棋盘）问题，类似图，找联通是一样的都是找相邻。

不同的地方：

1.找相邻节点的方法不同。不是通过前面讲的三种边的表示方法，而是通过类似迷宫问题中的障碍矩阵来判断上下左右四个方向是否是合法相邻节点。

2.现在的每一个节点是矩阵的一个位置，有行和列，或者说定位一个节点不是通过一个整数节点编号，而是通过行和列两个整数，所以用来标记单元号的component数组就是二维的了。

3.用并查集就不是很好做。需要把二维的位置转换为一维的整数节点编号。

作业2

找房间

- 给出一个大楼的地图，计算房间的数目。地图是 $n*m$ 个方块组成的方阵结构，每一个方块要么是强要么是地板。你可以通过有地板的方块往左，右，上，下四个方向走。
- 输入：
- 第一行2个整数 n 和 m 。
 $1 \leq n, m \leq 1000$
- 接下来 n 行，每一行 m 个字符描述地图结构。每一个字符不是.就是#，分别表示地板和墙。
- 输出：
- 一个整数，房间的数目。

示例输入
5 8

#..#...#
####.#.#
#..#...#

示例输出
3

下节课例题预习

- 不需要做，最好看看题目，思考一下。
- USACO 2019 US Open Contest, Bronze
Milk Factory
- USACO 2018 February Contest, Bronze Hoofball



Tips 有错么？

本地编译通过，考试提交后0分。

缺少头文件：

```
#include <stdio.h>
```

或者#include <cstdio>

或者干脆#include <bits/stdc++.h>

最好使用fin fout

作业1

图搜索

findroad.cpp

给定 n 个节点 ($n \leq 10^3$)，编号1到 n ，他们之间 m 条双向路径($m \leq 10^5$)。

输入：第1行4个整数 n, m, p, q 空格隔开 ($p \neq q$)，后面第2行到 $m+1$ 行为 m 条边，每行两个整数 a, b 表示节点 a 和节点 b 有双向连接。

输出：请输出节点 p 和 q 之间有多少条不同不重复的路线（一个点只能经过一次），如果没有输出0。

示例输入：

9 9 7 6

1 2

3 1

2 4

5 2

3 6

4 7

8 4

4 9

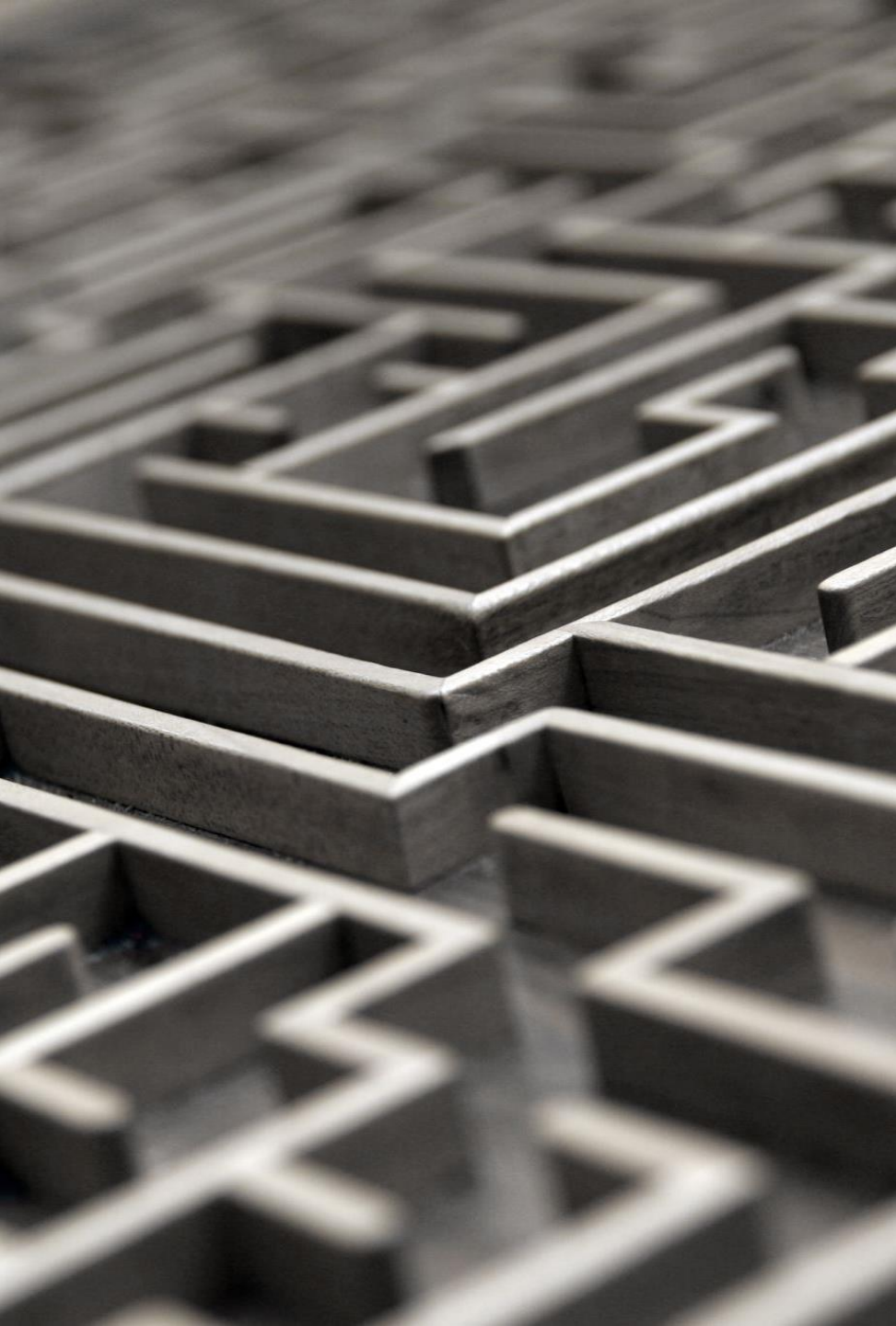
5 4

示例输出：

2

示例说明：7号点到6号点的两条线路分别为：7-4-5-2-1-3-6 7-4-2-1-3-6

思考：如果两个点之间出现两条以上的路线，说明有环出现。对么？



作业1提示

- 前面二阶第八课讲过一个类似的例题，当时实现的是判断两个点是否连接。这里做了修改，需要统计所有路线，所以前面的做法是搜索到了目标节点就退出输出可以找到，现在的做法就是找到了目标节点，说明找到了一条路，计数器++。非常类似。
- 注意和迷宫问题一样，如果要寻找所有路线，那么一个点被一条路线访问后其他路线还有可能访问，所以需要回溯visited节点访问状态。

作业2

找房间

findroom.cpp

- 给出一个大楼的地图，计算房间的数目。地图是 $n*m$ 个方块组成的方阵结构，每一个方块要么是强要么是地板。你可以通过有地板的方块往左，右，上，下四个方向走。
- 输入：
- 第一行2个整数 n 和 m 。
 $1 \leq n, m \leq 1000$
- 接下来 n 行，每一行 m 个字符描述地图结构。每一个字符不是.就是#，分别表示地板和墙。
- 输出：
- 一个整数，房间的数目。

示例输入
5 8

#..#...#
####.#.#
#..#...#

示例输出
3

作业2提示

- 这就是一个迷宫（棋盘）问题了。
- 什么叫做房间？是不是不被墙隔离而能够连接在一起的地板（矩阵中的格子）。如果把有墙的地方当作障碍，我们要找的就是有多少个联通子图（如果两个位置联通就是说他们可以通过找相邻节点（不是墙的）的方法走到对方）

由易到难，思维体系训练
实战结合，创新协作培养
兴趣导向，未来职业引领

<https://www.35tang.com>



扫码关注公众号

<https://www.三五堂.com>



添加辅导老师