第 3 章 图论基础

图论是算法竞赛中的一个重要内容，其特点在于概念与基础算法较多，更加考验读者的建模能力，要求可以从问题中抽象出图论模型并运用相关算法解决。而对于初学图论的同学们来说，熟悉基本算法，熟悉常见模型是快速入门的关键。

本章将介绍图与树的一些基本概念、图的四种存储方法与两种遍历方法以及最短路径算法、二分图判定等等一系列常见的基础算法。帮助读者由浅入深，系统了解与学习图论的内容。

3.1 图的概念

3.1.1 图的分类

图作为一种特殊的数据结构，表现的是若干对象以及这些对象间关系的一个集合，通过图，可以直观的显示出各个对象的关系。比如用图表现一个班级里各个同学的关系，则每个同学就是一个对象，所有的同学以及他们之间所有的关系的集合就是图。

图中的对象称为“结点”或“顶点”，一般用圆点来表示，顶点间的关系称为“边”，用连线或箭头来表示。

图大概可以分为四种：

- 无向图：边没有方向，以连线表示。例如：A 和 B 是朋友，则 B 和 A 也是朋友。调转初始与末尾结点后，关系不变。
- 有向图：边有方向，以箭头表示。 例如：要先学习 A 知识后才能学 B 知识。调转初始与末尾结点后，关系不同。
- 加权无向图：边无方向，有权值。 例如：A 与 B 直线距离 50 公里，B 与 A 直线距离也是 50 公里。
- 加权有向图：边有方向，有权值。例如：从 A 到 B 只能坐车，需要花费 50 分钟。从 B 到 A 只能步行，需要花费 100 分钟。

3.1.2 图的基本术语

一般情况下，我们将顶点集合为V，边集合为E的图记为$G = (V, E)$，其顶点数为 $|V|$，边数为 $|E|$。连接顶点a、b 的边记为$e = (a, b)$。在有向图中(a, b) 和 (b, a) 是不同的边，边的权值记为 $w(a, b)$。

3.1.2.1 度数

与一个顶点v相连的边的条数称为度，记做$d(v)$。特别的，对于边(v,v)，则每条这样的边要对 $d(v)$产生2 的贡献。

在有向图$G=(V,E)$中，以一个顶点v为起点的边的条数称为该顶点的出度，记作$d^+(v)$。以一个顶点v为终点的边的条数称为该节点的入度，记作$d^-(v)$。显然$d^+(v)+d^-(v)=d(v)$。

通过对图的遍历，我们可以轻松的得到图中每个顶点的度数，在很多图论算法中都运用了度的概念。希望读者在学习图的遍历时，可以尝试记录图的各个定点的度数。

3.1.2.2 简单图

自环：对E中的边$e=(u,v)$，若$u=v$，则e被称作一个自环。

重边：若E中存在两个完全相同的元素（边）e_1,e_2，则它们被称作（一组）重边。值得注意的是，对有向图E来说，边$e_1=(u,v)$与$e_2=(v,u)$虽然具有两个相同的端点，但两条边方向不同，不能认定为重边。

简单图：若一个图中没有自环和重边，它被称为简单图。具有至少两个顶点的简单无向图中一定存在度相同的结点。

多重图：如果一张图中有自环或重边，则称它为多重图。

稀疏图：有很少条边或弧（边的条数$|E|$远小于$|V|^2$）的图称为稀疏图。

稠密图：边的条数$|E|$接近$|V|^2$，称为稠密图。稀疏图与稠密图的判断没有绝对的标准，只是说对结点数目相同的图，边的数量越多图越稠密。

3.1.2.3 路径

途径：途径是连接一连串顶点的边的一个序列，可以为有限或无限长度。形式化地说，一条有限途径w是一个边的序列$e_1,e_2,\ldots,e_k$，使得存在一个顶点序列$v_0,v_1,v_2,\ldots,v_k$满足$e_i=(v_{i-1},v_i)$，其中$i\in[1,k]$。这样的途径可以简写为$v_0\to v_1\to v_2\to\ldots\to v_k$，有时我们也叫做$v_0$到$v_k$的一条通路。通常来说，边的数量$k$被称作这条途径的长度（如果边是带权的，长度通常指途径上的边权之和，题目中也可能另有定义）。

迹：对于一条途径w，若$e_1,e_2,\ldots,e_k$两两互不相同，则称w是一条迹。

路径（又称简单路径)：对于一条迹w，若其连接的点的序列中点两两不同，则称w是一条路径。

回路：对于一条迹w，若$v_0=v_k$，则称w 是一条回路。

环（又称简单回路/简单环）：对于一条回路w，若$v_0 = v_k$ 是点序列中唯一重复出现的点对，则称w是一个环。

在此处值得注意的是，在具体题目背景中的“路径”，“回路”等类似词汇，在没有“简单路径”、“简单回路”这种特殊说明时，也需要分析一下是题目背景中的含义还是本章节中的基础定义。这对题意的解读，模型的建立都至关重要。

3.1.2.4 欧拉图

欧拉通路：通过图中所有边恰好一次且行遍所有顶点的通路称为欧拉通路（又称欧拉路径）。

欧拉回路：通过图中所有边恰好一次且行遍所有顶点的回路称为欧拉回路。

欧拉图：具有欧拉回路的图。

欧拉回路、欧拉通路和欧拉图有一些常见的性质如下：

- ✓ 欧拉图均为连通图；
- ✓ 无向连通图 G 是欧拉图，则其不含奇数度结点(所有结点度数均为偶数)；
- ✓ 无向连通图 G 是欧拉通路，则其没有或有两个奇数度的结点，这两个节点为欧拉通路的起点与终点；
- ✓ 有向连通图 D 是欧拉图，则其每个结点的入度=出度；
- ✓ 有向连通图 D 是欧拉通路，则其除起点与终点外，其余每个结点的入度=出度。

3.1.2.5 连通

对于一张无向图$G = (V, E)$，对于$u, v \in V$，若存在一条途径使得$v_0 = u, v_k = v$，则称u和v是连通的。由定义，任意一个顶点和自身连通，任意一条边的两个端点连通。

若无向图$G = (V, E)$，满足其中任意两个顶点均连通，则称G是连通图，G的这一性质称作连通性。

无向图G的极大连通子图称为G的连通分量。任何连通图的连通分量只有一个，即是其自身，非连通的无向图有多个连通分量。

对于一张有向图$G = (V, E)$，对于$u, v \in V$，若存在一条途径使得$v_0 = u, v_k = v$，则称u可达v。由定义，任意一个顶点可达自身，任意一条边的起点可达终点。（无向图中的连通也可以视作双向可达。）

若一张有向图G的节点两两互相可达，则称这张图是强连通的。

若一张有向图G的边替换为无向边后可以得到一张连通图，则称原来这张有向图是弱连通的。

与连通分量类似，有向图同样也有弱连通分量（极大弱连通子图）和强连通分量（极大强连通子图）。

3.1.2.6 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。

二分图拥有十分多的性质与应用，比如求解最大匹配数、最小覆盖数、最大独立集等等，具体一些的将在图论提高章节中展开介绍。在本章中，需要掌握的是通过染色法判断图是否是二分图。

图论涉及到的概念还有很多，例如割点、桥、补图、反图等等，在本章节中只选取了部分出现频率较高的进行了介绍，在之后的学习过程中，会针对具体算法对一些遗漏的概念进行补充介绍。

3.2 图的存储与遍历

3.2.1 图的存储

在本文中，将用n表示图的点数，m表示图的边数。

本章节介绍图的四种存储方式，提供具体边的查找函数（find_edge）、图的遍历函数（DFS）以及图的存储部分的核心代码，对四种方式的应用也做了具体分析，读者在解题时可择优选取合适的存储方式。其中图的遍历部分代码希望读者可以在学习完搜索后尝试实现运行。

3.2.1.1 直接存边

思路：

使用一个结构题数组或者多个数组，直接将边的信息（例如起点、终点和权值）存储在数组中。对于图 3.1 的不带权有向图，输入的边为：

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

4 1

1 5

4 5

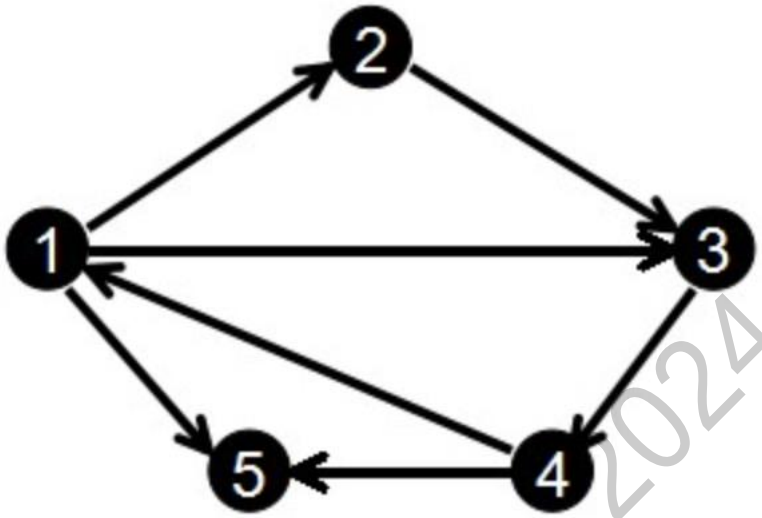


图 3.1

代码（此处只展示部分核心代码）：

其中n和m各表示点的数量和边的数量，$Edge$结构体存储边的起点和终点，e作为容器存储边，vis是图的遍历中使用的标记数组，使用$find_edge()$函数判断是否有以u为起点，v为终点的边存在，存在返回$true$，否则返回$false$，使用$DFS()$函数实现图的遍历，请读者在学习完搜索后尝试实现运行此代码，图的存储部分的代码在$main()$函数中体现。

```
struct Edge{
    int u, v;
};

int n, m;
vector<Edge> e;
vector<bool> vis;

bool find_edge(int u, int v){
    for (int i = 1; i <= m; ++i){
        if (e[i].u == u && e[i].v == v){
            return true;
        }
    }
    return false;
}
```

```
17.
18. void dfs(int u){
19.     if (vis[u])
20.         return;
21.     vis[u] = true;
22.     for (int i = 1; i <= m; ++i){
23.         if (e[i].u == u){
24.             dfs(e[i].v);
25.         }
26.     }
27. }
28.
29. int main(){
30.     cin >> n >> m;
31.
32.     vis.resize(n + 1, false);
33.     e.resize(m + 1);
34.
35.     for (int i = 1; i <= m; ++i)
36.         cin >> e[i].u >> e[i].v;
37.
38.     return 0;
39. }
```

复杂度：

- 查询是否存在某条边：最坏情况下需要将所有边都查找一遍是否有满足条件的边，故复杂度为$O(m)$。
- 遍历一个点的所有出边：需要将所有边都查找一遍，复杂度为$O(m)$。
- 遍历整张图：对于每个点，需要查找所有的出边，遍历一次所有的边，对结点为n的图，需要循环n次所有的边，其复杂度为$O(nm)$。
- 空间复杂度：边存储在结构体数组中，没有其他多余的空间开销，空间复杂度为$O(m)$。

应用：

由于直接存边的遍历效率低下，一般不用于遍历图。

在 Kruskal 算法（解决最小生成树问题，参见 3.4.3）中，由于需要将边按边权排序，需要直接存边。

在有的题目中，需要多次建图（如建一遍原图，建一遍反图），此时既可以使用多个其它数据结构来同时存储多张图，也可以将边直接存下来，需要重新建图时利用直接存下的边来建图。

3.2.1.2 邻接矩阵

思路：

顾名思义，邻接矩阵将图中的边汇集到一个矩阵中去，使用一个二维数组$adj[i][j]$，表示是否有从结点i到j的边存在（无向图中表示结点i和j双向可达），存储的数字可以表示从i到j的边的数量，在没有重边情况下可以存储从i到j的边的权值，或者在重边的情况下存储从i到j的边的最大（小）权值。具体情况需要读者依据题目要求和解题思路而定。对于图 2-1 的不带权有向图，其对应的邻接矩阵为：

0	1	1	0	1
0	0	1	0	0
0	0	0	1	0
1	0	0	0	1
0	0	0	0	0

代码：

n、m与各个函数的含义与 3.2.1.1 相同，在此便不再赘述，不同的是采用了$vector$套$vector$来存储邻接矩阵，当然使用二维数组的效果是相同的。

```
int n, m;
vector<bool> vis;
vector<vector<bool>> adj;

bool find_edge(int u, int v) {
    return adj[u][v];
}

void dfs(int u){
    if (vis[u])
        return;
    vis[u] = true;
    for (int v = 1; v <= n; ++v){
        if (adj[u][v]) {
```

```
            dfs(v);
        }
    }
}

int main()
{
    cin >> n >> m;

    vis.resize(n + 1, false);
    adj.resize(n + 1, vector<bool>(n + 1, false));

    for (int i = 1; i <= m; ++i){
        int u, v;
        cin >> u >> v;
        adj[u][v] = true;
    }

    return 0;
}

```

复杂度：

- 查询是否存在某条边：只需要查看一下$d[i][j]$的值就可以得知节点i到j的边的情况。复杂度为$O(1)$。
- 遍历一个点的所有出边：需要遍历查找所有点j，查看点i是否能够到达点j，复杂度为$O(n)$。
- 遍历整张图：需要遍历所有点，对点i，查找所有点j，查看点i是否能够到达点j，复杂度为 $O(n^2)$。
- 空间复杂度：使用$n*n$的二维数组表示图，空间复杂度为$O(n^2)$。

应用：

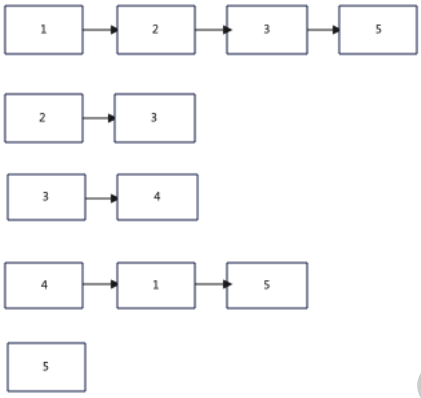
邻接矩阵只适用于没有重边（或重边可以忽略）的情况，其最显著的优点是可以　查询一条边是否存在。

由于邻接矩阵在稀疏图上效率很低（尤其是在点数较多的图上，空间无法承受），所以一般只会在稠密图上使用邻接矩阵。

3.2.1.3 邻接表

思路：

邻接表是图的一种链式存储方法，其数据结构包括两部分：节点和邻接点，使用数组（顶点集）+ 链表（边集）的形式。具体实现中，可以使用一个支持动态增加元素的数据结构构成的数组，如$vector < int > d[n + 1]$来存边，其中$d[u]$存储的是点u的所有出边的相关信息（终点、边权等）。对于图 3.1 的不带权有向图，对应的邻接表为：



节点 1 的邻接点（只看出边，即出弧）是节点 2、3、5，其邻接点的存储下标为 1、2、5，按照头插法（逆序）将其放入节点 1 后面的单链表中；

节点 2 的邻接点是节点 3，将其放入节点 2 后面的单链表中；

节点 3 的邻接点是节点 4，将其放入节点 3 后面的单链表中；

节点 4 的邻接点是节点 1、5，将其放入节点 4 后面的单链表中；

节点 5 没有邻接点，其后面的单链表为空。

注意： 对有向图中节点的邻接点，只看该节点的出边（出弧）。而对于无向图来说，一条边会被重复存储两次。

代码：

各变量及函数含义同 3.2.1.2，邻接表为了方便实现，使用 STL 库中的容器$vector$套$vector$的形式代替数组+链表。

```
int n, m;
vector<bool> vis;
vector<vector<int>> adj;

bool find_edge(int u, int v){
    for (int i = 0; i < adj[u].size(); ++i){
        if (adj[u][i] == v){
            return true;
        }
    }
    return false;
}
```

```
13.
14. void dfs(int u){
15.     if (vis[u])
16.         return;
17.     vis[u] = true;
18.     for (int i = 0; i < adj[u].size(); ++i)
19.         dfs(adj[u][i]);
20. }
21.
22. int main(){
23.     cin >> n >> m;
24.
25.     vis.resize(n + 1, false);
26.     adj.resize(n + 1);
27.
28.     for (int i = 1; i <= m; ++i)
29.     {
30.         int u, v;
31.         cin >> u >> v;
32.         adj[u].push_back(v);
33.     }
34.
35.     return 0;
36. }
```

复杂度：

- 查询是否存在结点i到j的边：$O(d^+(i))$（如果事先进行了排序就可以使用二分查找$O(log(d^+(i)))$做到）。
- 遍历点i的所有出边：对于点i，遍历一遍结点i所连接的长度为$d^+(i)$的链表，复杂度为$O(d^+(i))$。
- 遍历整张图：遍历点i的所有出边的复杂度为$O(d^+(i))$，有向图的所有点的出度之和为边的数目，无向图的所有点的度数之和为边的数目的两倍，故遍历整张图的复杂度为$O(n+m)$。
- 空间复杂度：$O(m)$。

应用

对于邻接矩阵来说，如果图的边比较少，会造成一定程度的空间浪费，而邻接表弥补了这个缺点，存各种图都很适合，除非有特殊需求（如需要快速查询一条边是否存在，且点数较少，可以使用邻接矩阵）。尤其适用于需要对一个点的所有出边进行排序的场合。

相对而言，邻接矩阵适合稠密图，邻接表适合稀疏图，读者需要在做题中根据结点数目和边的数目做出合适的选择。

3.2.1.4 链式前向星

思路：

链式前向星采用了一种静态链表存储方式，将边集数组和邻接表相结合，可以快速访问一个节点的所有邻接点。对于图 3.1 的不带权有向图，我们只需要知道结构体数组$edge$和$head$数组的含义，

我们只要知道$next$，$head$数组表示的含义，根据上面的数据就可以写出下面的过程，对图 3.1 的不带权有向图，模拟添加边的过程：

对于1 2这条边：$edge[0].to = 2; edge[0].next = -1; head[1] = 0;$

对于1 3这条边：$edge[3].to = 3; edge[3].next = 0; \quad head[1] = 3;$

对于1 5这条边：$edge[5].to = 5; edge[5].next = 3; \quad head[1] = 5;$

对于2 3这条边：$edge[1].to = 3; edge[1].next = -1; head[2] = 1;$

对于3 4这条边：$edge[2].to = 4; edge[2],next = -1; head[3] = 2;$

对于4 1这条边：$edge[4].to = 1; edge[4].next = -1; head[4] = 4;$

对于4 5这条边：$edge[6].to = 5; edge[6].next = 4; \quad head[4] = 6;$

遍历以u为顶点的所有边时，$head[u]$存储以u为顶点的最后一条边的编号，通过$edge[i].next$找到下一条边的编号，初始化$head[i]$为-1，所以找到最后一条边（以u为顶点的第一条边）时，$edge[i].next = 1$作为终止条件。

代码：

$edge[i].to$表示第i条边的终点，$edge[i].next$ 表示与第i条边同一个起点的下一条边，$edge[i].w$表示第i条边的边权，$head$数组来储存第一个以i为起点的边的位置。$add_edge()$函数表示添加一条结点u到v边权为w的边。对于无向图，每输入一条边，都需要添加两条边，互为反向边。

值得注意的是， $head$数组需要初始化为-1。

```
struct E{
    int to, w, next;
} edge[10010];

int n, m, cnt;
bool vis[10010];
```

```cpp
int head[10010];

void add_edge(int u, int v, int w){
    edge[cnt].to = v;
    edge[cnt].w = w;
    edge[cnt].next = head[u];
    head[u] = cnt++;
}

bool find_edge(int u, int v){
    for (int i = head[u]; ~i; i = edge[i].next){ // ~i 表示 i != -1
        if (edge[i].to == v){
            return true;
        }
    }
    return false;
}

void dfs(int u){
    if (vis[u])
        return;
    vis[u] = true;
    for (int i = head[u]; ~i; i = edge[i].next)
        dfs(edge[i].to);
}

int main(){
    cin >> n >> m;

    memset(vis, 0, sizeof(vis));
    memset(head, -1, sizeof(head));

    for (int i = 1; i <= m; ++i){
        int u, v, w;
        cin >> u >> v >> w;
        add_edge(u, v, w);
    }

    return 0;
}
```


复杂度：

- 查询是否存在结点i到j的边：$O(d^+(i))$。
- 遍历点i的所有出边： $O(d^+(i))$。

- 遍历整张图：访问每个点和边恰好 1 次（如果是无向图，正反向各访问 1 次），复杂度为$O(n+m)$。
- 空间复杂度： $O(m)$。

应用：

存各种图都很适合,是最常用的存储方式，但不能快速查询一条边是否存在，也不能方便地对一个点的出边进行排序。优点是边是带编号的，有时会非常有用。

3.2.2 深度优先搜索

在学习本节之前，请确保自己对递归思想有了一定的认识，并懂得利用栈、队列等数据结构，这些内容将会贯穿于搜索算法学习的始末。

DFS 全称是 Depth First Search，中文名是深度优先搜索，顾名思义，就是按照深度优先的顺序对“问题状态空间”进行搜索的算法，一般用于遍历或搜索树或图。所谓深度优先，就是说每次都尝试向更深的节点走。如果把一个问题的求解看作对问题状态空间的遍历与映射。我们可以进一步把“问题空间”类比为一张图，其中的状态类比为节点，状态之间的联系与可达性就用图中的边来表示，那么使用深度优先搜索算法求解问题，就相当于在一张图上进行深度优先遍历。

深度优先遍历图的方法是，从图中某顶点v出发：

（1）访问顶点v；

（2）依次从v的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和v有路径相通的顶点都被访问；

（3）若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。当然，当人们刚刚掌握深度优先搜索的时候常常用它来走迷宫。

递归的主要思想在于不断调用本身的函数，层层深入，直到遇到递归终止条件后层层回溯，其思想与 DFS 基本吻合，从而针对深度优先搜索的特点，既容易理解又容易实现的便是调用递归实现；回溯是在计算机底层执行的（系统有一个隐藏的栈帮我们做回溯），无法看到，也不需要读者操作。因此，理解并完成递归是 DFS 的一个难点;

一般来说，DFS 的时间复杂度为 $O(n*n!)$；空间复杂度为 $O(n)$；

DFS 模版如下：

```
1. DFS(v) // v 可以是图中的一个顶点，也可以是抽象的概念，如一种动态规划的 dp 状态等。
2.   在 v 上打访问标记
```

```
  for u in v 的相邻节点
    if u 没有打过访问标记 then
      DFS(u)
    end
  end
end
```

在图的存储中，对 4 种存储方式，都提供了对应图的搜索的代码，读者可以尝试实现该过程。DFS 应用不止体现在图论中，事实上，许多问题可以用搜索来解决小范围数据的情况，在很多情况下，DFS 被称为暴力算法，被用来与代码进行对拍。

例题　　全排列问题（洛谷 P1706）

按照字典序输出自然数 1 到 n 所有不重复的排列，即 n 的全排列，要求所产生的任一数字序列中不允许出现重复的数字。

【输入格式】

一个整数n。

【输出格式】

由$1 \sim n$组成的所有不重复的数字序列，每行一个序列。每个数字保留 5 个场宽。

【分析】

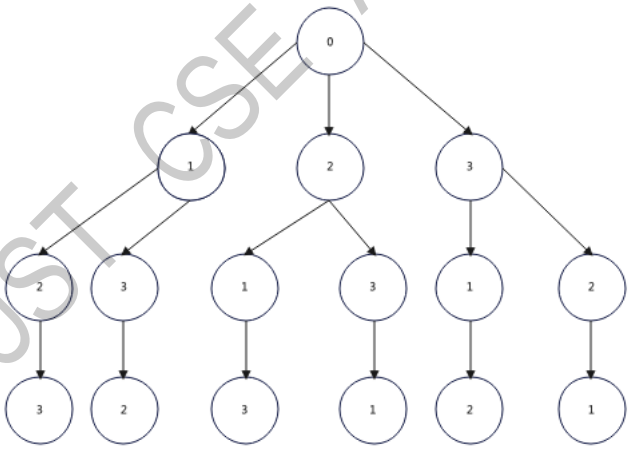


图 2-2

将数字的排列问题抽象成一棵树的遍历问题，如图 2-2，将 3 的排列问题转化成了树的搜索问题。

用$path$数组保存排列，当排列的长度为 n 时，是一种方案，进行输出；

用$bool$数组b表示数字i是否使用过。当$b[i]$为 1 时，i已经被使用过，$b[i]$ 为 0 时，i 没有被使用过；

$dfs(i)$ 表示的含义是：在 $path[i]$ 处填写数字，然后递归到下一个位置填写数字。

回溯：第 i 个位置填写某个数字的所有情况都遍历后， 第 i 个位置填写下一个数字。

关键代码如下：

```
const int N = 20;
int path[N], m;
bool b[N];

void dfs(int x){
    if (x == m){ //打印方案
        for (int i = 0; i < m; i++){
            cout << path[i] << " ";
        }
        cout << endl;
        return;
    }
    for (int i = 1; i <= m; i++){
        if (b[i] != true){
            path[x] = i;
            b[i] = true;
            dfs(x + 1);
            b[i] = false;
        }
    }
    return;
}

int main(){
    cin >> m;
    dfs(0);
    return 0;
}
```

例题　油田（Oil Deposits, UVa 572）

输入一个m行n列的字符矩阵，统计字符“@”组成多少个八连块。如果两个字符“@”所在的格子相邻（横、竖或者对角线方向），就说它们属于同一个八连块。

【输入格式】

一个或多个长方形地域。每个地域的第 1 行都有两个正整数m 和n（$1 \le m,n \le 100$），表示地域的行数和列数。如果$m = 0$，则表示输入结束；否则此后有m 行，每行都有n 个字符。每个字符都对应一个正方形区域，字符“*”表示没有油，字符“@”表示有油。

【输出格式】

对于每个长方形地域，都单行输出油藏的个数。

【分析】

虽然图有 DFS 和 BFS 遍历两种方法。由于 DFS 更容易编写，一般用 DFS 找连通块：从每个“@”格子出发，递归遍历它周围的“@”格子。每次访问一个格子时就给它写上一个“连通分量编号”（即下面代码中的idx数组），这样就可以在访问之前检查它是否已经有了编号，从而避免同一个格子访问多次。本节提供的代码用一个二重循环来找到当前格子的相邻 8 个格子，也可以用常量数组或者写 8 条 DFS 调用。

关键代码如下：

```
const int maxn = 100 + 5;
char pic[maxn][maxn];
int m, n, idx[maxn][maxn];

void dfs(int r, int c, int id) {
  if(r < 0 || r >= m || c < 0 || c >= n)
     return; //"出界"的格子
  if(idx[r][c] > 0 || pic[r][c] != '@')
     return; //不是"@"或者已经访问过的格子
  idx[r][c] = id; //连通分量编号
  for(int dr = -1; dr <= 1; dr++)
     for(int dc = -1; dc <= 1; dc++)
        f(dr != 0 || dc != 0)
           dfs(r+dr, c+dc, id);
}

int main() {
  while(scanf("%d%d", &m, &n) == 2 && m && n) {
     for(int i = 0; i < m; i++)
       scanf("%s", pic[i]);
     memset(idx, 0, sizeof(idx));
     int cnt = 0;
     for(int i = 0; i < m; i++)
        for(int j = 0; j < n; j++)
           if(idx[i][j] == 0 && pic[i][j] == '@')
             dfs(i, j, ++cnt);
     printf("%d\n", cnt);
  }
return 0;
}
```

3.2.3 广度优先搜索

已知图$G = (V, E)$和一个起始顶点S，宽度优先搜索以一种系统的方式探寻G的边从 而“发现”S所能到达的所有顶点，并计算S到所有这些顶点的最短矩离(最少边数)。该算法同时能生成一棵根为S且包括所有可达顶点的宽度优先树。对从S可达的任意顶点 V，宽度优先树中从S到V的路径对应于图G中从S到V的最短路径，即包含最少边数的路径。该算法对有向图和无向图同样适用。

之所以称为宽度优先算法是因为算法自始至终一直通过已找到和未找到的顶点之间的边界向外扩展。就是说，算法首先搜索和S距离为K的所有顶点，然后再去搜索和S 距离为$K+1$的其它顶点。

树与图的广度优先遍历需要使用一个队列来实现，起初，队列中仅包含一个起点（比如节点 1）。在广度优先遍历的过程中，不断从队头取出一个节点x，对于x面对的多条分支，把沿着每条分支到达的下一个节点(如果尚未访问过)插入队尾。重复执行上述过程直到队列为空。如图 3.2。

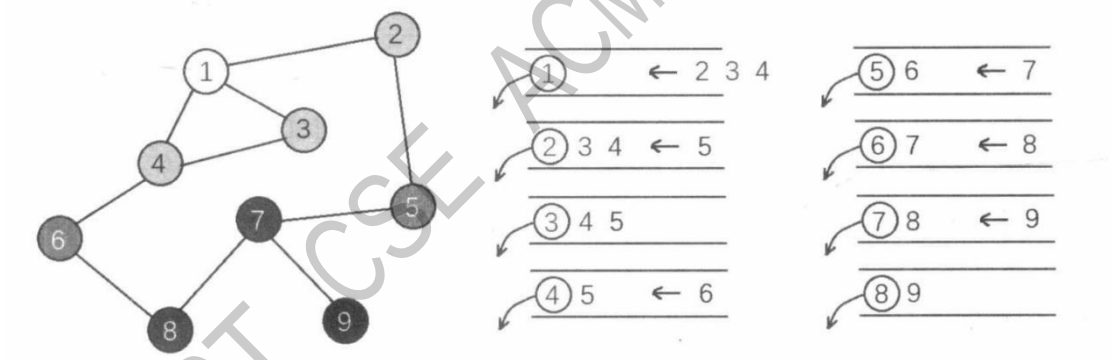


图 3.2

可以采用下面的代码对一张图进行广度优先遍历：

```
1. void bfs(){
2.
3.     memset(d, 0, sizeof(d));
4.
5.     queue<int> q;
6.
7.     q.push(1);
8.     d[1] = 1;
9.
10.    while (q.size() > 0){
11.
12.        int x = q.front();
```

```
        q.pop();

        for (int i = head[x]; i; i = edge[i].next){
            int y = edge[i].to;

            if (d[y])
                continue;

            d[y] = d[x] + 1;

            q.push(y);
        }
    }
}
```

在上面的代码中，我们在广度优先遍历的过程中顺便求出了一个d数组。对于一棵树来讲，$d[x]$就是点x在树中的深度。对于一张图来讲，$d[x]$ 被称为点x的层次(从起点1走到点x需要经过的最少的点数)。从代码和示意图中我们可以发现，广度优先遍历是一种按照层次顺序进行访问的方法，它具有如下两个重要性质：

1. 在访问完所有的第i层节点后，才会开始访问第$i+1$层节点。

2. 任意时刻，队列中至多有两个层次的节点。若其中一部分节点属于第i层，则另一部分节点属于第$i+1$层，并且所有第i层节点排在第$i+1$层节点之前。也就是说，广度优先遍历队列中的元素关于层次满足“两段性”和“单调性”。

这两条性质是所有广度优先思想的基础，请读者务必牢记，与深度优先遍历一样，上文的代码的时间复杂度也为$O(n+m)$。

例题　大火蔓延的迷宫（Fire!, UVa 11624）

你的任务是帮助 Joe 走出一个大火蔓延的迷宫。Joe 每分钟可以走到上下左右 4 个方向的相邻格之一，而所有着火的格子都会往四周蔓延（即如果某个空格与着火格有公共边，则下一分钟这个空格将着火）。迷宫中有一些障碍格，Joe 和火都无法进入。当 Joe 走到一个迷宫的边界格子时，我们认为他已经出了迷宫。

【输入格式】

输入第一行为数据组数 T。每组数据第一行为两个整数 R 和 C（$1 \le R, C \le 1\,000$）。以 下 R 行每行有 C 个字符，即迷宫。其中“#”表示墙和障碍物，“.”表示空地，“J”是 Joe 的初始位置（也是一个空地），“F”是着火格。每组数据的迷宫中恰好有一个格子是“J”。

【输出格式】

对于每组数据，输出走出迷宫的最短时间（单位：分钟）。如果无法走出迷宫，则输出IMPOSSIBLE。

【分析】

如果没有火，那么本题是一个标准的迷宫问题，可以用 BFS 解决。加上了火，难度增加了多少呢？其实没多少。注意，火是不会自动熄灭的，因此只要某个格子在某时刻起火了，以后将一直如此。所以只需要预处理每个格子起火的时间，在 BFS 扩展结点的时候加一个判断，当到达新结点时该格子没着火才真的把这个新结点加到队列中。 最后需要考虑一下如何求出每个格子起火的时间，其实这也是一个最短路问题，只不过起点不是一个，而是多个（所有的初始着火点）。这只需要在初始化队列时把所有着火点都放进去即可。

两个步骤的时间复杂度均为 $O(RC)$。

核心代码：

其中，dx和dy数组代表每个点向上下左右 4 个方向扩展时，横纵坐标的变化值。可以很好的简化代码。

```
typedef long long ll;
typedef pair<int, int> P;

const int N = 1000 + 10;
const int INF = 1000000000;
char g[N][N];
int dist[N][N][2];
int dx[] = {-1, 1, 0, 0};
int dy[] = {0, 0, -1, 1};
int r, c;
int sx, sy;
int ans;
queue<P> q;

bool inside(int x, int y) {
return x >= 0 && x < r && y >= 0 && y < c;
}

void bfs(int id)  {// id=1 时候处理每个格子首次着火的时间，id=0 处理最短到达该方格的时间
    while (!q.empty()) {
        P u = q.front();
        q.pop();
        for (int i = 0; i < 4; i++){
            int nx = u.first + dx[i], ny = u.second + dy[i];
            if (inside(nx, ny) && g[nx][ny] != '#' && dist[nx][ny][id] < 0) {
```

```
                dist[nx][ny][id] = dist[u.first][u.second][id] + 1;
                q.push(P(nx, ny));
            }
        }
    }
}

void check(int x, int y){
    if (dist[x][y][0] < 0)
        return;
    if (dist[x][y][1] < 0 || dist[x][y][0] < dist[x][y][1])
        ans = min(ans, dist[x][y][0] + 1); // 如果这个格子没有着火，或晚于到达前着火，那么更新答案
}

int main(){
    int T;
    scanf("%d", &T);
    while (T--){
        scanf("%d%d", &r, &c);
        memset(dist, -1, sizeof(dist));
        for (int i = 0; i < r; i++){
            scanf("%s", g[i]);
            for (int j = 0; j < c; j++)
                if (g[i][j] == 'J')
                    sx = i, sy = j;
                else if (g[i][j] == 'F') {
                    dist[i][j][1] = 0;
                    q.push(P(i, j));
                }
        }

        bfs(1);
        q.push(P(sx, sy));
        dist[sx][sy][0] = 0;
        bfs(0);
        ans = INF;
        for (int i = 0; i < r; i++) {
            check(i, 0);
            check(i, c - 1);
        }
        for (int i = 0; i < c; i++) {
            check(0, i);
            check(r - 1, i);
```

```
        }
        if (ans == INF)
            puts("IMPOSSIBLE");
        else
            printf("%d\n", ans);
    }
    return 0;
}
```

3.2.4 搜索的优化

通过上面两节的内容，读者可能已经发现，一般搜索解法的基本框架并不难，但如何减小搜索树的规模并快速遍历搜索树却是一门高深的学问。在本节中，我们就重点研究搜索的优化。

剪枝，就是减小搜索树规模、尽早排除搜索树中不必要的分支的一种手段。形象地看，就好像剪掉了搜索树的枝条，故被称为“剪枝”。在深度优先搜索中，有以下几类常见的剪枝方法：

1. 优化搜索顺序

在一些搜索问题中，搜索树的各个层次、各个分支之间的顺序不是固定的。不同的搜索顺序会产生不同的搜索树形态，其规模大小也相差甚远。

2. 排除等效冗余

在搜索过程中，如果我们能够判定从搜索树的当前节点上沿着某几条不同分支到达的子树是等效的，那么只需要对其中的一条分支执行搜索。另外，初学者一定要避免重叠、混淆“层次”与“分支”，避免遍历若干棵覆盖同一状态空间的等效搜索树。

3. 可行性剪枝

在搜索过程中，及时对当前状态进行检查，如果发现分支已经无法到达递归边界，就执行回溯。这就好比我们在道路上行走时，远远看到前方是一个死胡同，就应该立即折返绕路，而不是走到路的尽头再返回。某些题目条件的范围限制是一一个区间，此时可行性剪枝也被称为“上下界剪枝”。

4. 最优性剪枝

在最优化问题的搜索过程中，如果当前花费的代价已经超过了当前搜到的最优解，那么无论采取多么优秀的策略到达递归边界，都不可能更新答案。此时可以停止对当前分支的搜索，执行回溯。

5. 记忆化

可以记录每个状态的搜索结果，在重复遍历一个状态时直接检索并返回。这就好比我们对图进行深度优先遍历时，标记一一个节点是否已经被访问过。

除此之外，还有很多特殊的搜索，例如折半搜索，A*搜索等等，感兴趣的读者可以自行学习。

例题　Sticks（POJ1011）

乔治拿来一组等长的木棒，将它们随机地砍断,得到若干根小木棍，并使每一节木棍的长度都不超过 50 个长度单位。然后他又想把这些木棍拼接起来，恢复到裁剪前的状态，但他忘记了初始时有多少木棒以及木棒的初始长度。请你设计一个程序，帮助乔治计算木棒的可能最小长度。每一节木棍的长度都用大于零的整数表示。

【输入格式】

输入包含多组数据，每组数据包括两行。第一行是一个不超过 64 的整数，表示砍断之后共有多少节木棍。第二行是截断以后，所得到的各节木棍的长度。在最后一组数据之后，是一个 0。

【输出格式】

对于每组数据，分别输出原始木棒的可能最小长度。

【分析】

我们可以从小到大枚举原始木棒的长度len（也就是枚举答案)。当然，len 应该是所有木棍长度总和sum的约数，并且原始木棒的根数cnt 就等于sum/len。

对于枚举的每个len,我们可以依次搜索每根原始木棒由哪些木棍拼成。具体地讲，搜索所面对的状态包括:已经拼好的原始木棒根数,正在拼的原始木棒的当前长度,每个木棍的使用情况。在每个状态下，我们从尚未使用的木棍中选择一个，尝试拼到当前的原始木棒里，然后递归到新的状态。递归边界就是成功拼好cnt 根原始木棒，或者因无法继续拼接而宣告失败。

但这个朴素的搜索算法的效率比较低，我们来依次考虑几类剪枝:

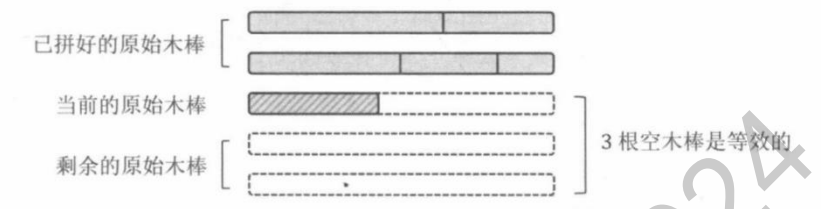
1.优化搜索顺序：把木棍长度从大到小排序，优先尝试较长的木棍。

2.排除等效冗余

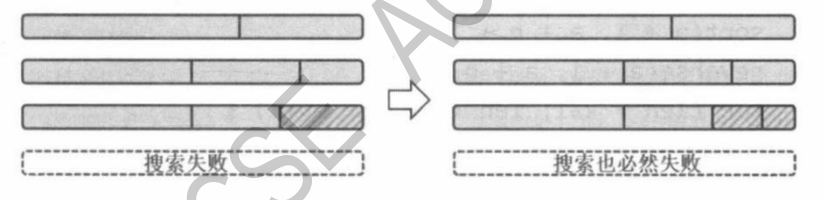
(1)可以限制先后加入一根原始木棒的木棍长度是递减$(\geq)$的。这是因为先拼上一根长度为x的木棍，再拼上一根长为y的木棍$(x < y)$，与先拼上y再拼上x显然是等效的，只需要搜索其中一种。

(2)对于当前原始木棒，记录最近一次尝试拼接的木棍长度。如果分支搜索失败回溯，不再尝试向该木棒中拼接其他相同长度的木棍(必定也会失败)。

(3)如果在当前原始木棒中“尝试拼入的第一根木棍”的递归分支就返回失败，那么直接判定当前分支失败，立即回溯。这是因为在拼入这根木棍前，面对的原始木棒都是“空”的(还没有进行拼接)，这些木棒是等效的。木棍拼在当前的木棒中失败，拼在其他木棒中一样会失败。如下图所示。



(4)如果在当前原始木棒中拼入一根木棍后，木棒恰好被拼接完整，并且“接下来拼接剩余原始木棒”的递归分支返回失败，那么直接判定当前分支失败，立即回溯。该剪枝可以用贪心来解释，“再用 1 根木棍恰好拼完当前原始木棒”必然比“再用若干根木棍拼完当前原始木棒”更好。如下图所示。



上述(1)至(4)分别利用“同一根木棒上木棍顺序的等效性”“ 等长木棍的等性”“空木棒的等效性”和“贪心”，剪掉了搜索树上诸多分支，使得搜索效率大大提升。

核心代码：

```
bool cmp(int x, int y){
    return x > y;
}

void dfs(int num1, int len, int pos){
    if (flag){
        return;
    }
    if (num1 == num){
        flag = 1;
        return;
    }
    for (int i = pos; i < n; i++){
        if (vis[i])
```

```
            continue;
        if (sum / num == a[i] + len){
            vis[i] = 1;
            // num++运算后 num 实际取值为 num+1，但显示的结果仍 num，
            // 下次运算时用 num+1 的取值运算，但显示 num+1 的值；
            // num=num+1 的值显示和实际的都为 num+1.
            dfs(num1 + 1, 0, 0);
            vis[i] = 0;
            return;
        }
        else if (sum / num > a[i] + len){
            vis[i] = 1;
            dfs(num1, len + a[i], i + 1);
            vis[i] = 0;
            if (len == 0)
                return;
            while (a[i] == a[i + 1])
                i++;
        }
    }
}

int main(){
    while (scanf("%d", &n) && n){
        sum = 0, flag = 0;
        memset(a, 0, sizeof(a));
        memset(vis, 0, sizeof(vis));
        for (int i = 0; i < n; i++){
            cin >> a[i];
            sum += a[i];
        }

        sort(a, a + n, cmp);
        for (int i = a[0]; i <= sum; i++){
            if (sum % i)
                continue;
            num = sum / i;
            dfs(0, 0, 0);
            if (flag){
                cout << i << endl;
                break;
            }
        }
    }
```

```
    return 0;
}
```

3.2.5 拓扑排序

对一个有向无环图（Directed Acyclic Graph 简称 DAG）G 进行拓扑排序，是将 G 中所有顶点排成一个线性序列，使得图中任意一对顶点 u 和 v，若边 $< u,v > \in E(G)$，则 u 在线性序列中出现在 v 之前。通常，这样的线性序列称为满足拓扑次序（Topological Order）的序列，简称拓扑序列。简单的说，由某个集合上的一个偏序得到该集合上的一个全序，这个操作称之为拓扑排序。

拓扑排序的流程如下：

1. 找一个入度为零（不需其他关卡通关就能解锁的）的端点，如果有多个，则从编号小的开始找；
2. 将该端点的编号输出；
3. 将该端点删除，同时将所有由该点出发的有向边删除；
4. 循环进行 2 和 3，直到图中的图中所有点的入度都为零；
5. 拓扑排序结束；

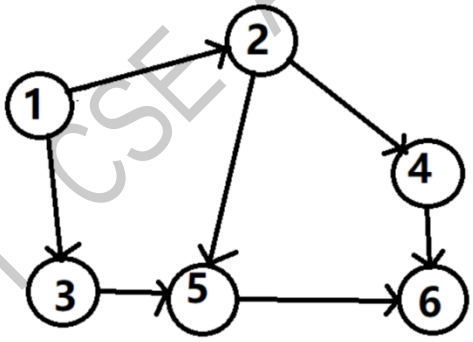
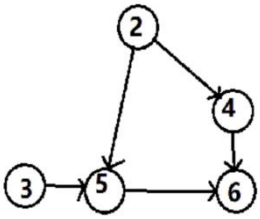
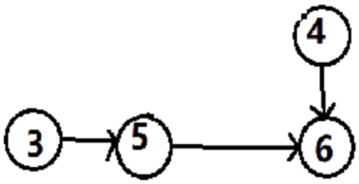


图 3.2

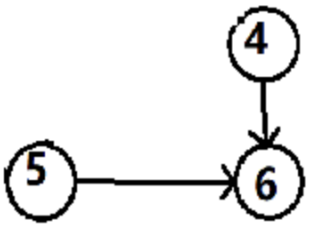
以图 3.2 为例，第一步输出“1”。



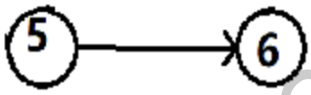
第二步输出“2”。



第三步输出“3”。



第四步输出“4”。



第五步输出“5”。

第六步输出“6”。排序结束。最终的拓扑序列为 1 2 3 4 5 6。

例题　排序（luoguP1346）

一个不同的值的升序排序数列指的是一个从左到右元素依次增大的序列，例如，一个有序的数列A,B,C,D表示$A<B,B<C,C<D$。在这道题中，我们将给你一系列形如$A<B$的关系，并要求你判断是否能够根据这些关系确定这个数列的顺序。

【输入格式】

第一行有两个正整数n,m，n表示需要排序的元素数量，$2\le n\le 26$。第 1 到n个元素将用大写的$A,B,C,D\ldots$表示。m表示将给出的形如$A<B$的关系的数量。

接下来有m行，每行有 3 个字符，分别为一个大写字母，一个 $<$ 符号，一个大写字母，表示两个元素之间的关系。

【输出格式】

若根据前x个关系即可确定这n个元素的顺序$yyy..y$（如ABC），输出

$$Sorted\ sequence\ determined\ after\ xxx\ relations:\ yyy\ldots y.$$

若根据前x个关系即发现存在矛盾（如$A<B,B<C,C<A$），输出

Inconsistency found after x relations.

若根据这 m 个关系无法确定这 n 个元素的顺序，输出

Sorted sequence cannot be determined.

（提示：确定 n 个元素的顺序后即可结束程序，可以不用考虑确定顺序之后出现矛盾的情况）。

【分析】

我们将每个字母转化为一个点，只需要考虑对整个图进行拓扑排序的情况。此外需要判断矛盾、得不到解的情况。

如果这道题目改为给出 X 条限制，判断这 X 条限制"是否矛盾"、"得到结果"与"条件不足"。bfs 版的拓扑排序是：首先将$du[i]$（入度）为0的点加入队列，然后每次取队列首，将所有队列首连向的点的入度减一，分别判断每个连向的点，若入度为 0，则再加入队列，直到队列为空为止。由于符合条件的点就入队，所以最终队列里的点的顺序就是拓扑排序中的队列所存的结果。

接下来考虑如何判断是否矛盾。当没有入度为零的点的时候，显然条件是有矛盾的。 同时，当出现矛盾时，有些点的入度永远不会减到零，这就意味着这些点永远入不了队。因此，当没有点入度为零，或当最终入队的点的个数小于n的时候，条件有矛盾。

最后考虑怎样判断条件不足。如果最开始入度为零的点有不止一个，那么条件一定不足，因为这两个点的顺序无法确定。 同时，如果每次取队首时，有不止一个点入队了，那么条件一定不足，因为这两个点都只有队首连向的边。所以这两个点无法确定顺序。

另外就是，如果同时满足条件矛盾与条件不足，这时应该判定为条件矛盾。因此，当有不止一个点入度为零，或当取同一个队首u时有不止一个v的入度变成了零，则条件不足。

关键代码：

```
const int maxn = 30;

int n, m;
vector<int> e[maxn];
int degree[maxn];
int a[maxn];
stack<int> s;
bool vis[maxn];
int mrk;

bool topo(int r){
```

```
    int sz = 0;
    bool finished = true;
int t[maxn];

    for (int i = 0; i < n; i++){
        t[i] = degree[i];
        if (!degree[i]){
            s.push(i);
vis[i] = true;
        }
}

    while (!s.empty()){
        if (s.size() > 1)
            finished = false;
        int k = s.top();
        a[sz++] = k;
        s.pop();

        for (int i = 0; i < e[k].size(); i++)
            t[e[k][i]]--;
        for (int i = 0; i < n; i++)
            if (!t[i] && !vis[i])
                s.push(i), vis[i] = true;
}

    if (sz < n)
        return false;
    if (finished && !mrk)
        mrk = r;
    return true;
}

int main(){
cin >> n >> m;

    for (int i = 1; i <= m; i++){
        char c[3];
        scanf("%s", c);
        int x = c[0] - 'A', y = c[2] - 'A';
        e[x].push_back(y);
        degree[y]++;

        if (!topo(i)){
```

```
            cout << "Inconsistency found after " << i << " relations.";
            return 0;
        }
        memset(vis, false, sizeof(vis));
}

    if (mrk){
        cout << "Sorted sequence determined after " << mrk << " relations: ";
        for (int i = 0; i < n; i++)
            cout << char(a[i] + 'A');
        cout << ".";
    }
    else
        cout << "Sorted sequence cannot be determined.";
    return 0;
}
```

3.2.6 二分图判定

在 3.1.2.6 中介绍了二分图，下面将介绍二分图的判定方法。

二分图判定是指给定一个具有 n 个顶点的图，要给图上每个顶点染色，并且要使相邻的顶点颜色不同。问是否能用最多 2 种颜色进行染色？保证图中没有重边和自环。

把相邻顶点染成不同颜色的问题叫做图的着色问题。对图进行染色所需要的最小颜色数称为最小着色数。最小着色数是 2 的图称作二分图，这也是二分图的另一种定义。

故只需要判断图能否只用两种颜色染色即可判断二分图。

关键代码：

```
const int N = 1e5 + 10, M = 2 * N;
int head[N], e[M], Next[M], idx;
int color[N];
int n, m;

void add(int a, int b){
    e[idx] = b;
    Next[idx] = head[a];
    head[a] = idx++;
}

bool dfs(int u, int c){
    color[u] = c; // 记录颜色

    for (int i = head[u]; i != -1; i = Next[i]){
        int j = e[i];
```

```
        if (!color[j]){ // 如果没染过颜色
            // dfs 深搜 染色 1 或者 2
            // 如果不可以将 j 成功染色
            if (!dfs(j, 3 - c))
                return false;
        }
        else if (color[j] == c)
            return false; // 如果染过颜色且和 c 相同
    }

    return true;
}

int main(){
    cin >> n >> m;
    memset(head, -1, sizeof(head));

    while (m--){
        int a, b;
        cin >> a >> b;
        add(a, b);
        add(b, a);
    }

    // 遍历判断所有点，染色
    bool flag = true;
    for (int i = 1; i <= n; i++){
        if (!color[i]){ // 如果这个点未染色
            if (!dfs(i, 1)){ // dfs 过程中要是返回 false——有矛盾发生，不是二分图
                flag = false;
                break;
            }
            // 不发生矛盾则继续搜
        }
    }

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

    return 0;
}
```

3.3 最短路径

最短路问题是一类经典问题，包括单源最短路和多源最短路。

单源最短路径问题(Single Source Shortest Path, SSSP 问题)是说，给定一张有向图$G=(V,E)$，V是点集，E是边集，$|V|=n$，$|E|=m$，节点以$[1,n]$之间的连续整数编号，(x,y,z)描述一条从节点x出发，到达节点y，长度为z的有向边。设1号点为起点，求长度为n的数组$dist$，其中$dist[i]$表示从起点1到节点i的最短路径的长度。

而多源最短路问题则是以多个节点为起点，求其余各点分别到这些点的最短路径长度。

为了方便叙述，这里先给出下文将会用到的一些记号的含义。

- n为图上点的数目，m为图上边的数目；
- s为最短路的源点；
- $D(u)$为节点s到节点u的实际最短路长度；
- $dis(u)$为节点s到节点u的估计最短路长度。任何时候都有$dis(u)\geq D(u)$。特别地，当最短路算法终止时，应有$dis(u)=D(u)$。
- $w(u,v)$为(u,v)这一条边的边权。

本节将介绍一个多源最短路算法 Floyd 算法和两个个单源最短路算法 Bellman-ford 算法和 Dijkstra 算法。

3.3.1 Floyd 算法

是用来求任意两个结点之间的最短路的。其复杂度比较高，但是常数小，容易实现。适用于任何图，不管有向无向，边权正负，但是最短路必须存在，无法处理图有负环的情况。

定义一个数组$f[k][x][y]$，表示只允许经过结点1到k（也就是说，在子图$V'=1,2,\dots k$中的路径，注意，x与y不一定在这个子图中），结点x到结点y的最短路长度。很显然，$f[n][x][y]$就是结点x到结点y的最短路长度（因为$V'=1,2,\dots n$即为V本身，其表示的最短路径就是所求路径）。

接下来考虑如何求出 f 数组的值。

$f[0][x][y]$：x与y的边权，或者0，或者$+\infty$（当x与y间有直接相连的边的时候，$f[0][x][y]$为它们的边权；当$x=y$的时候$f[0][x][y]$为零，因为到本身的距离为零；当x与y没有直接相连的边的时候，$f[0][x][y]$为$+\infty$）。

$f[k][x][y]=min(f[k-1][x][y],f[k-1][x][k]+f[k-1][k][y])$（其中$f[k-1][x][y]$为不经过$k$点的最短路径，而$f[k-1][x][k]+f[k-1][k][y]$为经过了$k$点的最短路）。

显然对于上面的算法，其时间复杂度为$O(n^3)$ ，我们需要依次增加问题规模（k从1到n），判断任意两点在当前问题规模下的最短路。

代码：

```
for (k = 1; k <= n; k++){
    for (x = 1; x <= n; x++){
        for (y = 1; y <= n; y++){
            f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y]);
        }
    }
}
```

可以发现第一维对算法的结果不产生影响，所以可以将数组的第一维省略，于是可以直接改成$f[x][y] = min(f[x][y], f[x][k] + f[k][y])$，从而将算法的空间复杂度降低至$O(n^2)$。代码如下：

```
for (k = 1; k <= n; k++){
    for (x = 1; x <= n; x++) {
        for (y = 1; y <= n; y++){
            f[x][y] = min(f[x][y], f[x][k] + f[k][y]);
        }
    }
}
```

虽然 Floyd 算法的时间和空间性能不够优秀，但作为一个多源最短路径算法，可扩展的应用和优化也很多。最常见的是使用 Floyd 计算传递闭包。

例题　Cow Contest S（luoguP2419）

在赛场上，奶牛们按$1..N$依次编号。每头奶牛的编程能力不尽相同，并且没有哪两头奶牛的水平不相上下，也就是说，奶牛们的编程能力有明确的排名。 整个比赛被分成了若干轮，每一轮是两头指定编号的奶牛的对决。如果编号为 A 的奶牛的编程能力强于编号为 B 的奶牛$(1 <= A <= N; 1 <= B <= N; A != B)$ ，那么她们的对决中，编号为 A 的奶牛总是能胜出。 FJ 想知道奶牛们编程能力的具体排名，于是他找来了奶牛们所有 $M(1 <= M <= 4{,}500)$轮比赛的结果，希望你能根据这些信息，推断出尽可能多的奶牛的编程能力排名。比赛结果保证不会自相矛盾。

【输入格式】

第 1 行：2 个用空格隔开的整数：N 和 M。

第2..$M+1$行：每行为 2 个用空格隔开的整数 A、B，描述了参加某一轮比赛的奶牛的编号以及结果（编号为 A，即为每行的第一个数的奶牛为胜者）。

【输出格式】

第 1 行：输出 1 个整数，表示排名可以确定的奶牛的数目。

【分析】

首先考虑如何才能确定一头牛的排名，假设赢了或间接赢了这个牛的牛数为sum_{win}，输了或间接输掉的牛数为sum_{lose}，如果要确认一头牛的排名，很显然当且仅当满足$sum_{win}+sum_{lose}=n-1$的时候才能确定一头牛的排名。

本题有多种解法，可以使用搜索，具体思路为建立两张图，第一个图我们用一个结点指向另一个结点表示一头牛输给了别人，另外一个图恰好相反。对每头牛遍历两张图累计遍历到的节点是否满足$sum_{win}+sum_{lose}=n-1$。

拓扑排序也是本题的解法之一。感兴趣的读者可自行探索本题的各种解法。

当然本题作为一个传递闭包问题可以使用 Floyd 算法解决。只需要结点与其他$n-1$个结点的关系都确定，就可以确定他的排名。

关键代码：

```
int main(){
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++) {
        scanf("%d%d", &a, &b);
        f[a][b] = 1;
}

    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                f[i][j] = f[i][j] | f[i][k] & f[k][j];

    for (int i = 1; i <= n; i++){
        int gg = 1;
        for (int j = 1; j <= n; j++)
            if (i == j)
                continue;
            else
                gg = gg & (f[i][j] | f[j][i]);
        ans += gg;
    }
    printf("%d\n", ans);
```

```
    return 0;
}
```

3.3.2 Bellman-ford 算法与 SPFA 算法

给定一张有向图，若对于图中的某一条边(x,y,z),有 $dist[y] \leq dist[x]+z$成立，则称该边满足三角形不等式。若所有边都满足三角形不等式，则$dist$ 数组就是所求最短路。

我们先介绍基于迭代思想的 Bellman-Ford 算法。它的流程如下:

1. 扫描所有边(x,y,z)，若$dist[y] > dist[x] + z$,则用$dist[x]+z$ 更新$dist[y]$。

2. 重复上述步骤，直到没有更新操作发生。

Bellman-Ford 算法的时间复杂度为$O(nm)$。其优点是可以求出有负权的图的最短路，并可以对最短路不存在的情况进行判断。

相比较于 Bellman-Ford 算法，我们使用更多的是基于 Bellman-Ford 算法优化的 SPFA 算法（Shortest Path Faster Algorithm）。在 Bellman-Ford 算法中，如果某个点未被更新过，我们还是会用这个点去更新其他点，其实该操作是不必要的，只需要拿更新过后的点去更新其他的点，因为只有用被更新过的点更新其他结点x，x的距离才可能变小。

核心代码：

其中 bool 类型的st数组表示第 i 个点是否在队列中，防止存储重复的点，队列q存储所有待更新的点。

```
int spfa(){
    memset(dist, 0x3f, sizeof(dist));
    dist[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;
    while (q.size()){
        int t = q.front();
        q.pop();
        st[t] = false;

        for (int i = head[t]; i != -1; i = Next[i]){
            int j = e[i];
            if (dist[j] > dist[t] + w[i]){
                dist[j] = dist[t] + w[i];
                if (!st[j]) {
                    q.push(j);
                    st[j] = true;
```

```
            }
        }
    }
}

    if (dist[n] == 0x3f3f3f3f)
        return -1; // 不存在最短路
    return dist[n];
}
```

虽然在大多数情况下 SPFA 的运行速度很快，但其最坏情况下的时间复杂度为$O(nm)$，将其卡到这个复杂度也是不难的，所以要谨慎使用（在没有负权边时最好使用 Dijkstra 算法，在有负权边且题目中的图没有特殊性质时，若 SPFA 是标算的一部分，题目不应当给出 Bellman-Ford 算法无法通过的数据范围）。当然 SPFA 也可以用来判断 s 点是否能抵达一个负环，只需记录最短路经过了多少条边，当经过了至少n条边时，说明s点可以抵达一个负环。

3.3.3 Dijkstra 算法

Dijkstra 算法的流程如下：

1. 初始化$dist[1]=0$， 其余节点的$dist$ 值为正无穷大。

2. 找出一个未被标记的、$dist$最小的节点x，然后标记节点x。

3. 扫描节点x的所有出边(x,y,z)， 若$dist[y] > dist[x]+z$, 则使用$dist[x]+z$更新$dist[y]$。

4. 重复上述 2～3 两个步骤，直到所有节点都被标记。

Dijkstra 算法基于贪心思想，它只适用于所有边的长度都是非负数的图。当边长z都是非负数时，全局最小值不可能再被其他节点更新，故在第 1 步中选出的节点x必然满足：$dist[x]$ 已经是起点到x的最短路径。我们不断选择全局最小值进行标记和扩展，最终可得到起点 1 到每个节点的最短路径的长度。

上面算法流程的时间复杂度为$0(n^2)$, 主要瓶颈在于第 1 步的寻找全局最小值的过程。可以用二叉堆、优先队列等方式对$dist$ 数组进行维护，用$0(logn)$的时间获取最小值并从堆中删除，用$0(logn)$ 的时间执行一条边的扩展和更新，最终可在$0(m\,logn)$的时间内实现 Dijkstra 算法。

这里优先队列优化的关键代码：

```cpp
struct edge{
    int v, w;
};

struct node{
    int dis, u;

    bool operator>(const node &a) const{
        return dis > a.dis;
    }
};

vector<edge> e[maxn];
int dis[maxn], vis[maxn];
priority_queue<node, vector<node>, greater<node>> q;

void dijkstra(int n, int s){
    memset(dis, 63, sizeof(dis));
    dis[s] = 0;
    q.push({0, s});
    while (!q.empty()){
        int u = q.top().u;
        q.pop();
        if (vis[u])
            continue;
        vis[u] = 1;
        for (auto ed : e[u]){
            int v = ed.v, w = ed.w;
            if (dis[v] > dis[u] + w)
            {
                dis[v] = dis[u] + w;
                q.push({dis[v], v});
            }
        }
    }
}
```


不同算法之间的比较：

最短路算法	Floyd	Bellman-Ford	Dijkstra
最短路类型	每对结点之间的最短路	单源最短路	单源最短路
作用于	任意图	任意图	非负权图
能否检测负环?	能	能	不能
推荐作用图的大小	小	中/小	大/中
时间复杂度	$O(N^3)$	$O(NM)$	$O(M \log M)$