# 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、三种最小生成树算法、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。

## 图的概念

### 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。

### 图的基本术语

一般情况下，我们将顶点集合为，边集合为的图记为，其顶点数为，边数为。连接顶点的边记为。在有向图中和是不同的边，边的权值记为。

#### 度数

与一个顶点相连的边的条数称为度，记做。特别的，对于边(v, v)，则每条这样的边要对产生22的贡献。

在有向图中，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。显然。

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。

#### 简单图

自环：对E中的边，若，则e被称作一个自环。

重边：若中存在两个完全相同的元素（边），则它们被称作（一组）重边。值得注意的是，对有向图来说，边与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（边的条数远小于）的图称为稀疏图。

稠密图：边的条数接近，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。

#### 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，一条有限途径是一个边的序列，使得存在一个顶点序列v_0, v_1, \ldots, v_k满足e_i = (v_{i-1}, v_i)，其中。这样的途径可以简写为v_0 \to v_1 \to v_2 \to \cdots \to v_k，有时我们也叫做到的一条通路。通常来说，边的数量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，则称是一条回路。

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。

#### 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：

1. 欧拉图均为连通图；
2. 无向连通图 是欧拉图，则其不含奇数度结点(所有结点度数均为偶数)；
3. 无向连通图 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；
4. 有向连通图 是欧拉图，则其每个结点的入度=出度；
5. 有向连通图 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。

#### 连通

对于一张无向图，对于，若存在一条途径使得，则称u和是连通的。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，满足其中任意两个顶点均连通，则称是连通图，G的这一性质称作连通性。

[无向图](https://baike.baidu.com/item/%E6%97%A0%E5%90%91%E5%9B%BE?fromModule=lemma_inlink)的极大连通子图称为的连通分量。任何[连通图](https://baike.baidu.com/item/%E8%BF%9E%E9%80%9A%E5%9B%BE?fromModule=lemma_inlink)的连通分量只有一个，即是其自身，非连通的[无向图](https://baike.baidu.com/item/%E6%97%A0%E5%90%91%E5%9B%BE?fromModule=lemma_inlink)有多个连通分量。

对于一张有向图G = (V, E)，对于，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）。

#### 二分图

设是一个无向图，如果顶点可分割为两个互不相交的子集，并且图中的每条边所关联的两个顶点和分别属于这两个不同的顶点集，则称图为一个二分图。

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。

## 图的存储与遍历

### 图的存储

在本文中，将用表示图的点数，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。

#### 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

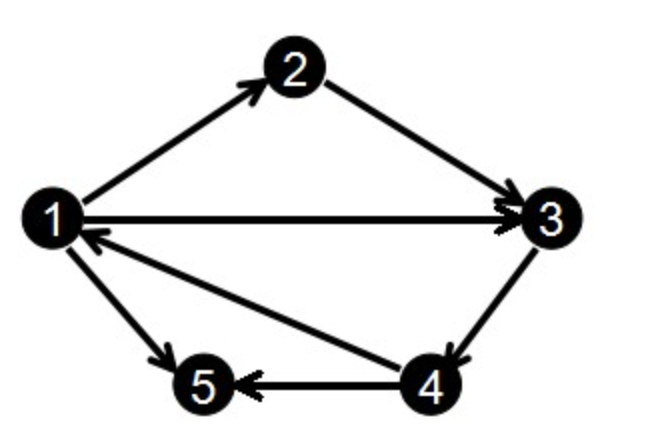
3 4

1 3

4 1

1 5

4 5



图表 3.2‑1

其中和各表示点的数量和边的数量，结构体存储边的起点和终点，作为容器存储边，是图的遍历中使用的标记数组，使用函数判断是否有以为起点，为终点的边存在，存在返回，否则返回，使用函数实现图的遍历，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函数中体现。

1. struct Edge {

2. int u, v;

3. };

4.

5. int n, m;

6. vector<Edge> e;

7. vector<bool> vis;

8.

9. bool find\_edge(int u, int v) {

10. for (int i = 1; i <= m; ++i) {

11. if (e[i].u == u && e[i].v == v) {

12. return true;

13. }

14. }

15. return false;

16. }

17.

18. void dfs(int u) {

19. if (vis[u]) return;

20. vis[u] = true;

21. for (int i = 1; i <= m; ++i) {

22. if (e[i].u == u) {

23. dfs(e[i].v);

24. }

25. }

26. }

时间复杂度：

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，时间复杂度为O(m)。

遍历一个点的所有出边：需要将所有边都查找一遍，时间复杂度为。

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，对结点为的图，需要循环次所有的边，其时间复杂度为。

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](https://oi-wiki.org/graph/mst/#kruskal)（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。

#### 邻接矩阵

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，表示是否有从结点到的边存在（无向图中表示结点和双向可达），存储的数字可以表示从到的边的数量，在没有重边情况下可以存储从到的边的权值，或者在重边的情况下存储从到的边的最大（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 |

图表 3.2‑2

代码：

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

1. int n, m;

2. vector<bool> vis;

3. vector<vector<bool> > adj;

4.

5. bool find\_edge(int u, int v) {

6. return adj[u][v];

7. }

8.

9. void dfs(int u) {

10. if (vis[u]) return;

11. vis[u] = true;

12. for (int v = 1; v <= n; ++v) {

13. if (adj[u][v]) {

14. dfs(v);

15. }

16. }

17. }

18.

19. int main() {

20. cin >> n >> m;

22. vis.resize(n + 1, false);

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

25. for (int i = 1; i <= m; ++i) {

26. int u, v;

27. cin >> u >> v;

28. adj[u][v] = true;

29. }

31. return 0;

32. }

时间复杂度：

查询是否存在某条边：O(1)只需要查看一下的值就可以得知节点到的边的情况。时间复杂度为。

遍历一个点的所有出边：需要遍历查找所有点，查看点是否能够到达点，时间复杂度为。

遍历整张图：需要遍历所有点，对点，查找所有点，查看点是否能够到达点，时间复杂度为 O(n^2)。

空间复杂度：

使用的二维数组表示图，空间复杂度为O(n^2)。

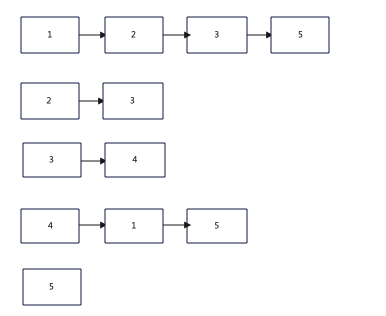
应用：

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 O(1)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。

#### 邻接表

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，如来存边，其中存储的是点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：



图表 3.2‑3

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。

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库中的容器套的形式代替数组+链表。

1. int n, m;

2. vector<bool> vis;

3. vector<vector<int> > adj;

4.

5. bool find\_edge(int u, int v) {

6. for (int i = 0; i < adj[u].size(); ++i) {

7. if (adj[u][i] == v) {

8. return true;

9. }

10. }

11. return false;

12. }

13.

14. void dfs(int u) {

15. if (vis[u]) return;

16. vis[u] = true;

17. for (int i = 0; i < adj[u].size(); ++i) dfs(adj[u][i]);

18. }

19.

20. int main() {

21. cin >> n >> m;

23. vis.resize(n + 1, false);

24. adj.resize(n + 1);

26. for (int i = 1; i <= m; ++i) {

27. int u, v;

28. cin >> u >> v;

29. adj[u].push\_back(v);

30. }

32. return 0;

33. }

时间复杂度：

查询是否存在结点到的边：O(d^+(u))（如果事先进行了排序就可以使用二分查找做到）。

遍历点的所有出边：对于点，遍历一遍结点所连接的长度为的链表，时间复杂度为O(d^+(u))。

遍历整张图：遍历点的所有出边的复杂度为，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，故遍历整张图的时间复杂度为。

空间复杂度：。

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。

#### 链式前向星

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和数组的含义，

我们只要知道，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：

对于这条边：

对于这条边：

对于这条边：

对于这条边：

对于这条边：

对于这条边：

对于这条边：

遍历以为顶点的所有边时，存储以为顶点的最后一条边的编号，通过找到下一条边的编号，初始化为，所以找到最后一条边（以为顶点的第一条边）时，作为终止条件。

【代码】

表示第条边的终点，表示与第条边同一个起点的下一条边，表示第条边的边权，数组来储存第一个以为起点的边的位置。函数表示添加一条结点到边权为的边。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， 数组需要初始化为-1。

1. int n, m, cnt;

2. vector<bool> vis;

3. struct E{

4. int to, next;

5. }edge[N];

6. int head[N];

7.

8. void add\_edge(int u, int v, int w)

9. {

10. edge[cnt].to = v;

11. edge[cnt].w = w; edge[cnt].next = head[u];

12. head[u] = cnt++;

13. }

14.

15. bool find\_edge(int u, int v) {

16. for (int i = head[u]; ~i; i = edge[i].next) { // ~i 表示 i != -1

17. if (edge[i].to == v) {

18. return true;

19. }

20. }

21. return false;

22. }

23.

24. void dfs(int u) {

25. if (vis[u]) return;

26. vis[u] = true;

27. for (int i = head[u]; ~i; i = edge[i].next) dfs(edge[i].to);

28. }

时间复杂度：

查询是否存在结点到的边：。

遍历点的所有出边：O(d^+(u)) 。

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），时间复杂度为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。

### 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](https://en.wikipedia.org/wiki/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，从图中某顶点出发：

1. 访问顶点；
2. 依次从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；空间复杂度为；

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

3. for u in v 的相邻节点

4. if u 没有打过访问标记 then

5. DFS(u)

6. end

7. end

8. 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。

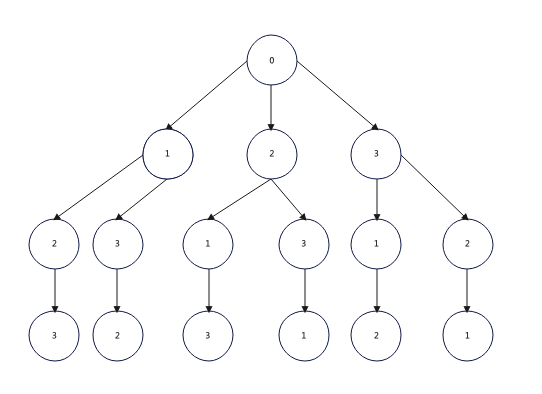
【输入格式】

一个整数。

【输出格式】

由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】



图表 3.2‑4

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。

用数组保存排列，当排列的长度为 时，是一种方案，进行输出；

用数组表示数字是否使用过。当为1时，已经被使用过，为0时，没有被使用过；

表示的含义是：在处填写数字，然后递归到下一个位置填写数字。

回溯：第个位置填写某个数字的所有情况都遍历后， 第个位置填写下一个数字。

关键代码如下：

1. const int N = 20;

2. int path[N], m;

3. bool b[N];

4.

5. void dfs(int x){

6. if (x == m){ //打印方案

7. for (int i = 0; i < m; i++){

8. cout << path[i] << " ";

9. }

10. cout << endl;

11. return;

12. }

13. for (int i = 1; i <= m; i++){

14. if (b[i] != true){

15. path[x] = i;

16. b[i] = true;

17. dfs(x + 1);

18. b[i] = false;

19. }

20. }

21. return;

22. }

23.

24. int main(){

25. cin >> m;

26. dfs(0);

27. return 0;

28. }

例题 油田（**Oil Deposits, UVa 572**）

输入一个行列的字符矩阵，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行都有两个正整数和（），表示地域的行数和列数。如果，则表示输入结束；否则此后有行，每行都有个字符。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】

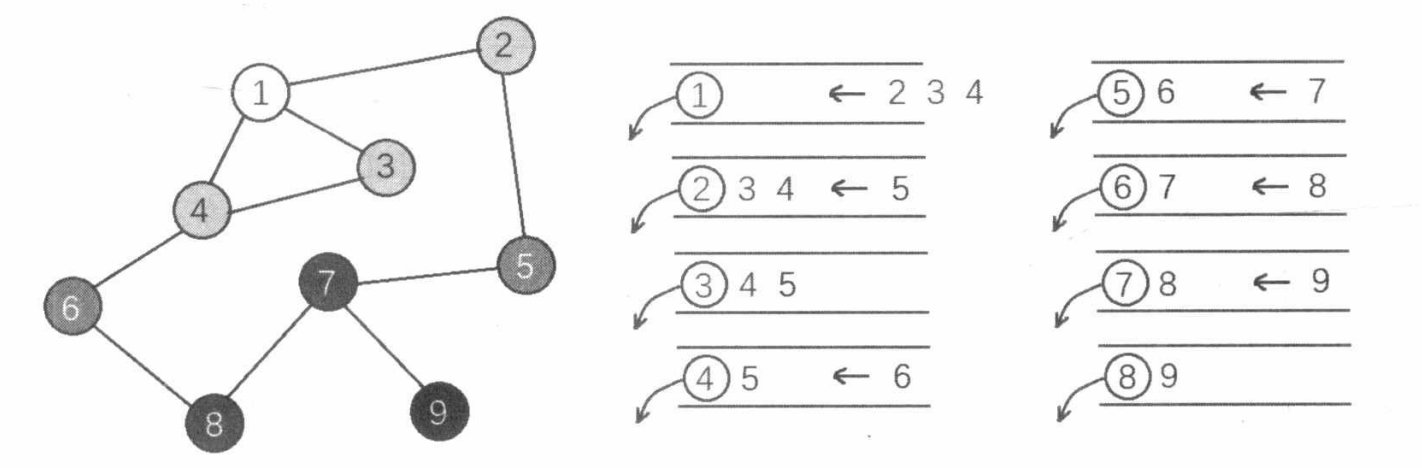
虽然图有DFS和BFS遍历两种方法。由于DFS更容易编写，一般用DFS找连通块：从每个格子出发，递归遍历它周围的格子。每次访问一个格子时就给它写上一个“连通分量编号”（即下面代码中的数组），这样就可以在访问之前检查它是否已经有了编号，从而避免同一个格子访问多次。本节提供的代码用一个二重循环来找到当前格子的相邻8个格子，也可以用常量数组或者写8条DFS调用。

### 广度优先搜索

已知图和一个起始顶点,宽度优先搜索以一种系统的方式探寻的边从 而“发现" 所能到达的所有顶点，并计算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且包括所有可达顶点的宽度优先树。对从可达的任意顶点 ,宽度优先树中从到的路径对应于图中从到的最短路径，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，算法首先搜索和距离为的所有顶点，然后再去搜索和 距离为的其它顶点。

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，对于面对的多条分支,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。



图表 3.2‑5

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. memset(d, 0, sizeof(d));

3. queue<int> q;

4. q.push(1);

5. d[1] = 1;

6. while (q.size() > 0) {

7. int x = q.front();

8. q.pop();

9. for (int i = head[x]; i; i = next[i]) {

10. int y = ver[i];

11. if (d[y]) continue;

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

13. q. push(y);

14. }

15. }

16. }

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数组。对于一棵树来讲，就是点在树中的深度。对于一张图来讲，被称为点的层次(从起点走到点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.在访问完所有的第层节点后，才会开始访问第层节点。

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层，则另一部分节点属于第层，并且所有第层节点排在第层节点之前。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。

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 。每组数据第一行为两个整数和（）。以 下行每行有个字符，即迷宫。其中表示墙和障碍物，表示空地，是 Joe 的初始位置（也是一个空地），是着火格。每组数据的迷宫中恰好有一个格子是。

【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 。

### 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:

**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。

**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。

**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-一个区间，此时可行性剪枝也被称为“上下界剪枝”。

**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。

**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 (也就是枚举答案)。当然，应该是所有木棍长度总和的约数，并且原始木棒的根数就等于。

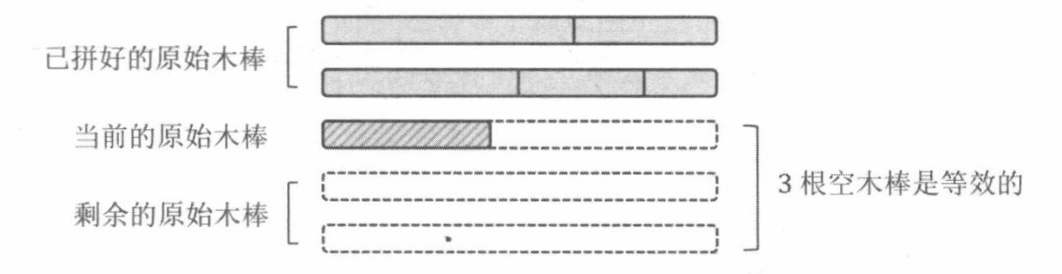
对于枚举的每个,我们可以依次搜索每根原始木棒由哪些木棍拼成。具体地讲，搜索所面对的状态包括:已经拼好的原始木棒根数,正在拼的原始木棒的当前长度，每个木棍的使用情况。在每个状态下，我们从尚未使用的木棍中选择一个, 尝试拼到当前的原始木棒里，然后递归到新的状态。递归边界就是成功拼好根原始木棒，或者因无法继续拼接而宣告失败。

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:

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。

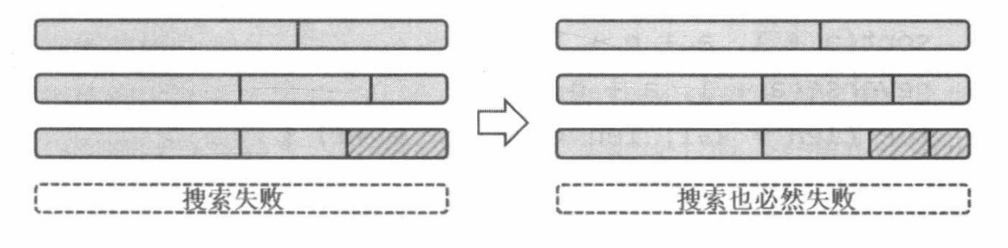
排除等效冗余

1. 可以限制先后加入一根原始木棒的木棍长度是递减的。这是因为先拼上一根长度为的木棍，再拼上一根长为的木棍，与先拼上再拼上显然是等效的，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。



图表 3.2‑6

1. 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。



图表 3.2‑7

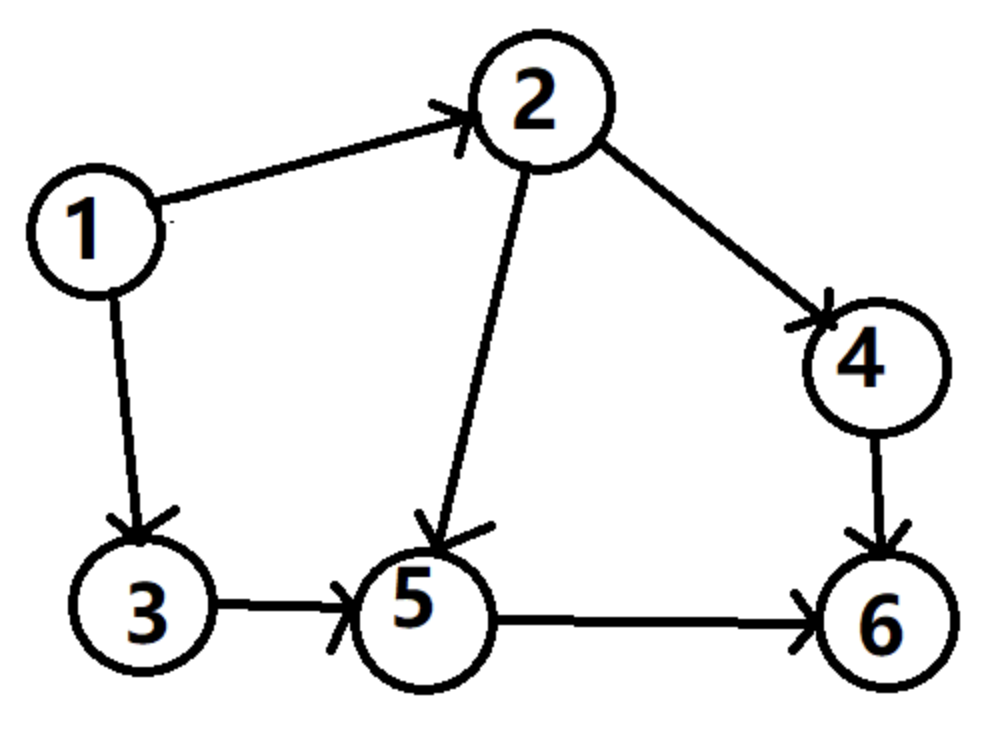
上述(1)至(4)分别利用“同一根木棒上木棍顺序的等效性”“ 等长木棍的等性”“空木棒的等效性”和“贪心”，剪掉了搜索树上诸多分支，使得搜索效率大大提升。

### 拓扑排序

对一个有向无环图 (Directed Acyclic Graph 简称 DAG)进行拓扑排序，是将中所有顶点排成一个线性序列，使得图中任意一对顶点和，若边，则 在线性序列中出现在之前。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。

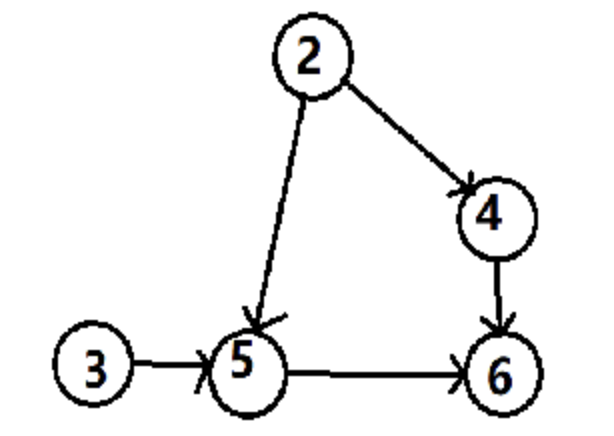
拓扑排序的流程如下：

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；



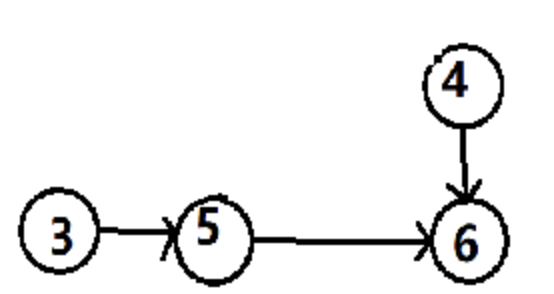
图表 3.2‑8

以图3.2-7为例，第一步输出“1”。



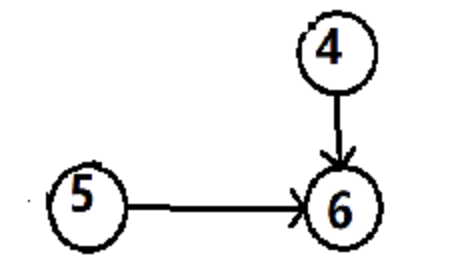
图表 3.2‑9

第二步输出“2”。



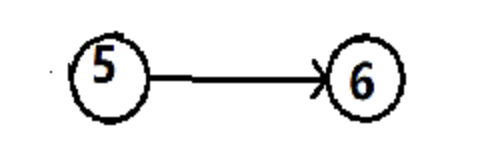
图表 3.2‑10

第三步输出“3”。



图表 3.2‑11

第四步输出“4”。



图表 3.2‑12

第五步输出“5”。



图表 3.2‑13

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，一个有序的数列表示。在这道题中，我们将给你一系列形如的关系，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，表示需要排序的元素数量，。第到个元素将用大写的表示。表示将给出的形如的关系的数量。

接下来有行，每行有个字符，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】

若根据前个关系即可确定这个元素的顺序（如），输出

若根据前个关系即发现存在矛盾（如），输出

若根据这个关系无法确定这个元素的顺序，输出

（提示：确定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（入度）为的点加入队列，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，或当取同一个队首时有不止一个的入度变成了零，则条件不足。

### 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：

1. const int N = 1e5 + 10, M = 2 \* N;

2. int h[N], e[M], ne[M], idx;

3. int color[N];

4. int n, m;

5.

6. void add(int a, int b){

7. e[idx] = b;

8. ne[idx] = h[a];

9. h[a] = idx ++;

10. }

11.

12. bool dfs(int u, int c){

13. color[u] = c;//记录颜色

14.

15. for(int i = h[u]; i != -1; i = ne[i]){

16. int j = e[i];

17. if(!color[j]) {//如果没染过颜色

18. //dfs深搜 染色 1 或者 2

19. //如果不可以将j成功染色

20. if(!dfs(j, 3 - c))

21. return false;

22. }

23. else if(color[j] == c)

24. return false; //如果染过颜色且和c相同

25. }

26.

27. return true;

28. }

29.

30. int main(){

31. cin >> n >> m;

32. memset(h, -1, sizeof h);

33.

34. while (m -- ){

35. int a, b;

36. cin >> a >> b;

37. add(a, b);

38. add(b, a);

39. }

40.

41. //遍历判断所有点，染色

42. bool flag = true;

43. for(int i = 1; i <= n; i ++) {

44. if(!color[i]) {//如果这个点未染色

45. if(!dfs(i, 1)) { //dfs过程中要是返回false,认为有矛盾发生，不是二分图

46. flag = false;

47. break;

48. }

49. //不发生矛盾则继续搜

50. }

51. }

52.

53. if(flag) puts("Yes");

54. else puts("No");

55.

56. return 0;

57. }

## 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, 是点集，是边集，, ，节点以之间的连续整数编号，描述一条从节点出发，到达节点，长度为的有向边。设号点为起点，求长度为的数组, 其中表示从起点到节点的最短路径的长度。

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。

为图上点的数目，为图上边的数目；为最短路的源点；为节点到节点的实际最短路长度；为节点到节点的估计最短路长度。任何时候都有 。特别地,当最短路算法终止时，应有。为这一条边的边权。

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。

### 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，表示只允许经过结点到（也就是说，在子图中的路径，注意，与不一定在这个子图中），结点到结点的最短路长度。很显然，就是结点到结点的最短路长度（因为即为本身，其表示的最短路径就是所求路径）。

接下来考虑如何求出数组的值。

：与的边权，或者，或者（当与间有直接相连的边的时候，为它们的边权；当的时候为零，因为到本身的距离为零；当与没有直接相连的边的时候，为）。

（其中为不经过点的最短路径，而为经过了点的最短路）。

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

代码：

1. for (k = 1; k <= n; k++) {

2. for (x = 1; x <= n; x++) {

3. for (y = 1; y <= n; y++) {

4. f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y]);

5. }

6. }

7. }

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，从而将算法的空间复杂度降低至。

代码如下：

for (k = 1; k <= n; k++) {

2. for (x = 1; x <= n; x++) {

3. for (y = 1; y <= n; y++) {

4. f[x][y] = min(f[x][y], f[x][k] + f[k][y]);

5. }

6. }

7. }

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 ，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 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： 和 。

第行: 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,输了或间接输掉的牛数为，如果要确认一头牛的排名，很显然当且仅当满足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。

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。只需要结点与其他个结点的关系都确定，就可以确定他的排名。

关键代码：

1. int main(){

2. scanf("%d%d", &n, &m);

3. for (int i = 1; i <= m; i++) {

4. scanf("%d%d", &a, &b);

5. f[a][b] = 1;

6. }

7.

8. for (int k = 1; k <= n; k++)

9. for (int i = 1; i <= n; i++)

10. for (int j = 1; j <= n; j++)

11. f[i][j] = f[i][j] | f[i][k] & f[k][j];

12.

13. for (int i = 1; i <= n; i++){

14. int gg = 1;

15. for (int j = 1; j <= n; j++)

16. if (i == j)

17. continue;

18. else

19. gg = gg & (f[i][j] | f[j][i]);

20. ans += gg;

21. }

22. printf("%d\n", ans);

23. return 0;

24. }

### 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,有成立，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, 若,则用更新。
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。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，的距离才可能变小。

核心代码：

1. int spfa()

2. {// bool st[N]： 存第 i 个点是不是在队列中，防止存重复的点

3. memset(dist,0x3f,sizeof(dist));

4. dist[1] = 0;

5.

6. queue<int> q; //存储所有待更新的点

7. q.push(1); // 1号点入队

8. st[1] = true;

9. while(q.size()) // 队列不空

10. {

11. int t = q.front(); //取队头

12. q.pop();

13. st[t] = false; // 代表这个点已经不在队列了，因为存在边权为负数，某个点可能会被更新多次，所以可以多次入队和出队。

14.

15. for(int i = h[t]; i!=-1; i=ne[i]) // 更新 t 的所有临边结点的最短路

16. {

17. int j = e[i];

18. if(dist[j] > dist[t]+w[i])

19. {

20. dist[j] = dist[t] + w[i];

21. if(!st[j]) //如果 j 不在队列，让 j 入队

22. {

23. q.push(j);

24. st[j] = true; // 标记 j 在队中

25. }

26. }

27. }

28. }

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

30. return dist[n];

31. }

32.

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，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，当经过了至少条边时，说明点可以抵达一个负环。

### Dijkstra算法

Dijkstra算法的流程如下:

1. 初始化， 其余节点的 值为正无穷大。
2. 找出一个未被标记的、最小的节点，然后标记节点。
3. 扫描节点的所有出边, 若,则使用更新。
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步中选出的节点必然满足: 已经是起点到的最短路径。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,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、优先队列等方式对数组进行维护，用的时间获取最小值并从堆中删除，用的时间执行一条边的扩展和更新，最终可在的时间内实现Dijkstra算法。

这里优先队列优化的关键代码：

1. struct edge {

2. int v, w;

3. };

4.

5. struct node {

6. int dis, u;

7. bool operator>(const node& a) const {

8. return dis > a.dis;

9. }

10. };

11.

12. vector<edge> e[maxn];

13. int dis[maxn], vis[maxn];

14. priority\_queue<node, vector<node>, greater<node> > q;

15.

16. void dijkstra(int n, int s) {

17. memset(dis, 63, sizeof(dis));

18. dis[s] = 0;

19. q.push({0, s});

20. while (!q.empty()) {

21. int u = q.top().u;

22. q.pop();

23. if (vis[u]) continue;

24. vis[u] = 1;

25. for (auto ed : e[u]) {

26. int v = ed.v, w = ed.w;

27. if (dis[v] > dis[u] + w) {

28. dis[v] = dis[u] + w;

29. q.push({dis[v], v});

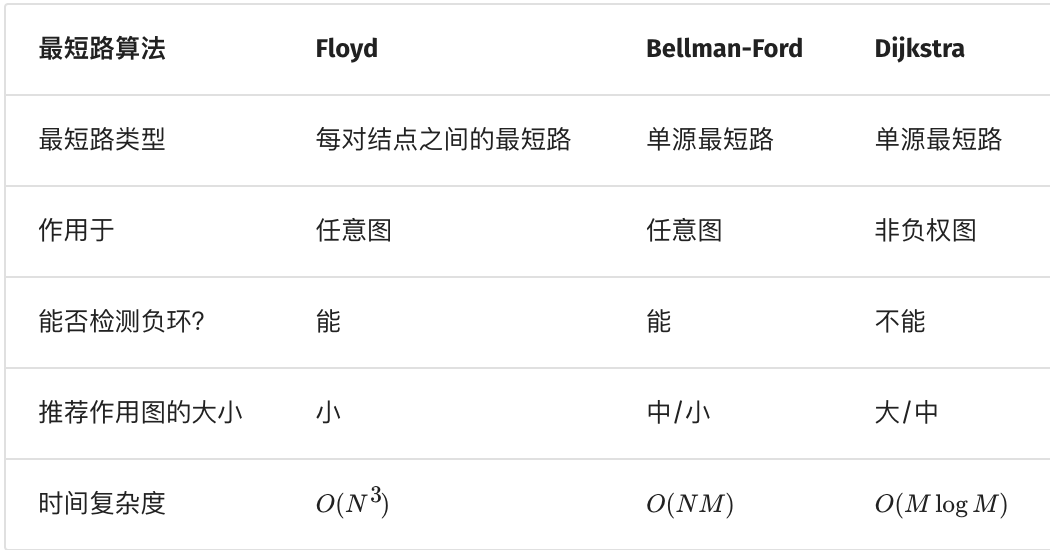
30. }

31. }

32. }

33. }

不同算法之间的比较：



图表 3.3‑1

## 最小生成树

### 树和生成树

定义无向连通图的最小生成树（Minimum Spanning Tree，MST）为边权和最小的生成树。注意：只有连通图才有生成树，而对于非连通图，只存在生成森林。

定理：

任意一棵最小生成树一定包含无向图中权值最小的边。

证明:

反证法：假设无向图 存在一棵最小生成树不包含权值最小的边。设是无向图中权值最小的边。把添加到树中，会和树上从到的路径一起构成一个环，并且环上其他边的权值都比大。因此用e代替环上的其他任意一条边，会形成一棵权值和更小的生成树，与假设矛盾。故假设不成立，原命题成立。

证毕。

推论：给定一张无向图，, 。从中选出条边构成的一个生成森林。若再从剩余的条边中选条添加到生成森林中，使其成为的生成树，并且选出的边的权值之和最小，则该生成树一定包含这条边中连接生成森林的两个不连通节点的权值最小的边。

对于求解最小生成树问题，以下介绍三个主要的算法。

### Kruskal算法

Kruskal算法就是基于上述推论的。Kruskal算法总是维护无向图的最小生成森林。最初，可认为生成森林由零条边构成，每个节点各自构成一棵仅包含一个点的树。

在学习Kruskal算法之前，请读者确保已经理解并查集的原理，并能够熟练应用并查集，并查集是理解整个Kruskal算法的核心和关键。

在任意时刻，Kruskal 算法从剩余的边中选出一条权值最小的，并且这条边的两个端点属于生成森林中两棵不同的树(不连通)，把该边加入生成森林。图中节点的连通情况可以用并查集维护。

详细来说，Kruskal 算法的流程如下:

1. 建立并查集，每个点各自构成一个集合。
2. 把所有边按照权值从小到大排序，依次扫描每条边。
3. 若属于同一集合(连通)，则忽略这条边，继续扫描下一条。
4. 若属于不同集合，合并所在的集合，并把累加到答案中。
5. 所有边扫描完成后，第4步中处理过的边就构成最小生成树。

时间复杂度为。

关键代码：

1. int n, m;

2. struct node{

3. int u;

4. int v;

5. int w;

6. }e[maxn];

7.

8. int fa[maxn], cnt, sum, num;

9.

10. void add(int x, int y, int w)

11. {

12. e[++ cnt].u = x;

13. e[cnt].v = y;

14. e[cnt].w = w;

15. }

16.

17. bool cmp(node x, node y)

18. {

19. return x.w < y.w;

20. }

21.

22. int find(int x)

23. {

24. return fa[x] == x ? fa[x] : fa[x] = find(fa[x]);//路径压缩

25. }

26.

27. /\*

28. int find(int x)

29. {

30. if(fa[x] == x) return x;

31. else

32. {

33. fa[x] = find(fa[x]);//路径压缩

34. return fa[x];

35. }

36. }

37. \*/

38.

39. void kruskal()

40. {

41. for(int i = 1; i <= cnt; i ++)

42. {

43. int x = find(e[i].u);

44. int y = find(e[i].v);

45. if(x == y) continue;

46. fa[x] = y;

47. sum += e[i].w;

48. if(++ num == n - 1) break;//如果构成了一颗树

49. }

50. }

51.

52. int main()

53. {

54. cin>>n>>m;

55. for(int i = 1; i <= n; i ++) fa[i] = i;

56. while(m--)

57. {

58. int x, y, w;

59. cin>>x>>y>>w;

60. add(x, y, w);

61. }

62. std:: sort(e + 1, e + 1 + cnt, cmp);

63. kruskal();

64. printf("%d",sum);

65. return 0;

66. }

例题 局域网（洛谷P2820）

某个局域网内有台计算机，由于搭建局域网时工作人员的疏忽，现在局域网内的连接形成了回路，我们知道如果局域网形成回路那么数据将不停的在回路内传输，造成网络卡的现象。因为连接计算机的网线本身不同，所以有一些连线不是很畅通，我们用表示之间连接的畅通程度，值越小表示之间连接越通畅，为示之间无网线连接。

现在需要解决回路问题，我们将除去一些连线，使得网络中没有回路，不改变原图节点的连通性，并且被除去网线的最大，请求出这个最大值。

【输入格式】

第一行两个正整数。

接下来的行每行三个正整数表示两台计算机之间有网线联通，通畅程度为。

【输出格式】

一个正整数，的最大值。

【分析】

题目要求使删去的边最大，等价于使剩下的边长度最小。那么就可以使用最小生成树求解此题，只需要在输入时算出边长总和，再减去最小生成树的长度即可。

很显然是一道典型的最小生成树问题。一般来说，大部分生成树问题都采用Kruskal算法解决。

### Prim算法

Prim算法同样基于上述推论，但思路略有改变。Prim算法总是维护最小生成树的一部分。最初，Prim算法仅确定1号节点属于最小生成树。

在任意时刻，设已经确定属于最小生成树的节点集合为T，剩余节点集合为S。Prim算法找到， 即两个端点分别属于集合,的权值最小的边，然后把点从集合中删除，加入到集合，并把累加到答案中。

具体来说，可以维护数组;若,则表示节点与集合中的节点之间权值最小的边的权值。若,则就等于被加入时选出的最小边的权值。

可以类比Dijkstra 算法，用一个数组标记节点是否属于。每次从未标记的节点中选出值最小的，把它标记(新加入)，同时扫描所有出边，更新另一个端点的值。最后最小生成树的权值总和就是。

Prim算法的时间复杂度为， 可以用二叉堆优化到。 但用二叉堆优化不如直接使用Kruskal 算法更加方便。因此，Prim 主要用于稠密图，尤其是完全图的最小生成树的求解。

关键代码：

1. struct edge{

2. int v,w,next;

3. }e[maxm<<1];

4. int head[maxn],dis[maxn],cnt,n,m,tot,now=1,ans;

5. bool vis[maxn];

6.

7. void add(int u,int v,int w){

8. e[++cnt].v=v;

9. e[cnt].w=w;

10. e[cnt].next=head[u];

11. head[u]=cnt;

12. }

13.

14. void init(){

15. cin>>n>>m;

16. for(re int i=1,u,v,w;i<=m;++i)

17. {

18. cin>>u>>v>>w;

19. add(u,v,w),add(v,u,w);

20. }

21. }

22.

23. int prim()

24. {

25. for(int i=2;i<=n;++i){

26. dis[i]=inf;

27. }

28. for(int i=head[1];i;i=e[i].next){

29. dis[e[i].v]=min(dis[e[i].v],e[i].w);

30. }

31. while(++tot<n){

32. int minn=inf;

33. vis[now]=1;

34. for(int i=1;i<=n;++i)

35. {

36. if(!vis[i]&&minn>dis[i])

37. {

38. minn=dis[i];

39. now=i;

40. }

41. }

42. ans+=minn;

43.

44. for(int i=head[now];i;i=e[i].next){

45. re int v=e[i].v;

46. if(dis[v]>e[i].w&&!vis[v]){

47. dis[v]=e[i].w;

48. }

49. }

50. }

51. return ans;

52. }

### Boruvka算法

Boruvka 算法的思想是前两种算法的结合。它可以用于求解边权互不相同的无向图的最小生成森林。(无向连通图就是最小生成树。)

为了描述该算法，我们需要引入一些定义:

定义为我们当前找到的最小生成森林的边。在算法执行过程中，我们逐步向加边，定义连通块表示一个点集，且这个点集中的任意两个点，在中的边构成的子图上是连通的(互相可达)。

定义一个连通块的最小边为它连向其它连通块的边中权值最小的那一条。

初始时，,每个点各自是一个连通块:

1. 计算每个点分别属于哪个连通块。将每个连通块都设为“没有最小边”。
2. 遍历每条边，如果和不在同一个连通块，就用这条边的边权分别更新和所在连通块的最小边。
3. 如果所有连通块都没有最小边，退出程序，此时的就是原图最小生成森林的边集。否则,将每个有最小边的连通块的最小边加入,返回第一步。

当原图连通时，每次迭代连通块数量至少减半，算法只会迭代不超过次，而原图不连通时相当于多个子问题，因此算法复杂度时的。

例题 【模板】最小生成树（洛谷P3366）

给出一个无向图，求出最小生成树，如果该图不连通，则输出 orz。

【输入格式】

第一行包含两个整数,，表示该图共有个结点和条无向边。

接下来行，每行包含三个整数​,​,​，表示有一条长度为的无向边连接了结点​,​​。

【输出格式】

如果该图连通，则输出一个整数表示最小生成树的各边的长度之和。如果该图不连通则输出orz。

1. const int MaxN = 5000 + 5, MaxM = 200000 + 5;

2. int N, M;

3. int U[MaxM], V[MaxM], W[MaxM];

4. bool used[MaxM];

5. int par[MaxN], Best[MaxN];

6.

7. void init() {

8. scanf("%d %d", &N, &M);

9. for (int i = 1; i <= M; ++i)

10. scanf("%d %d %d", &U[i], &V[i], &W[i]);

11. }

12.

13. void init\_dsu() {

14. for (int i = 1; i <= N; ++i)

15. par[i] = i;

16. }

17.

18. int get\_par(int x) {

19. if (x == par[x]) return x;

20. else return par[x] = get\_par(par[x]);

21. }

22.

23. bool Better(int x, int y) {

24. if (y == 0) return true;

25. if (W[x] != W[y]) return W[x] < W[y];

26. return x < y;

27. }

28.

29. void Boruvka() {

30. init\_dsu();

31.

32. int merged = 0, sum = 0;

33.

34. bool update = true;

35. while (update) {

36. update = false;

37. memset(Best, 0, sizeof Best);

38.

39. for (int i = 1; i <= M; ++i) {

40. if (used[i] == true) continue;

41. int p = get\_par(U[i]), q = get\_par(V[i]);

42. if (p == q) continue;

43.

44. if (Better(i, Best[p]) == true) Best[p] = i;

45. if (Better(i, Best[q]) == true) Best[q] = i;

46. }

47.

48. for (int i = 1; i <= N; ++i)

49. if (Best[i] != 0 && used[Best[i]] == false) {

50. update = true;

51. merged++; sum += W[Best[i]];

52. used[Best[i]] = true;

53. par[get\_par(U[Best[i]])] = get\_par(V[Best[i]]);

54. }

55. }

56.

57. if (merged == N - 1) printf("%d\n", sum);

58. else puts("orz");

59. }

60.

61. int main() {

62. init();

63. Boruvka();

64. return 0;

65. }