数据结构与算法
（七）
图基础算法

杜育根

Ygdu@sei.ecnu.edu.cn

7. 图基础

- 这节课介绍了图结构的存储（包括邻接矩阵、邻接表、链式前向星）方法、图的各种常用概念、深度优先遍历、广度优先遍历以及树结构的存储和遍历。

7.1. 什么是图

在一个社交网络中，每个帐号和他们之间的关系构成了一张巨大的网络，就像右面这张图：

那么在电脑中，我们要用什么样的数据结构来保存这个网络呢？这个网络需要用一个之前课程里未提到过的数据结构，也就是接下来要讲解的图结构来保存。

到底什么是图？图是由一系列顶点和若干连结顶点集合内两个顶点的边组成的数据结构。数学意义上的图，指的是由一系列点与边构成的集合，这里我们只考虑有限集。通常我们用 G = (V, E) 表示一个图结构，其中 V表示点集，E表示边集。
在顶点集合所包含的若干个顶点之间，可能存在着某种两两关系——如果某两个点之间的确存在这样的关系的话，我们就在这两个点之间连边，这样就得到了边集的一个成员，也就是一条边。对应到社交网络中，顶点就是网络中的用户，边就是用户之间的好友关系。

有向边和无向边

- 如果用边来表示好友关系的话，对于微信这种双向关注的社交网络没有问题，但是对于微博这种单向关注的要如何表示呢?
- 于是引出了两个新的概念：有向边和无向边。
- 简而言之，一条有向边必然是从一个点指向另一个点，而相反方向的边在有向图中则不一定存在；而有的时候我们并不在意构成一条边的两个顶点具体谁先谁后，这样得到的一条边就是无向边。就像在微信中，A 是 B 的好友，那 B 也一定是 A 的好友；而在微博中，A 关注 B 并不意味着 B 也一定关注 A。
- 对于图而言，如果图中所有边都是无向边，则称为无向图，反之称为有向图。简而言之，无向图中的边是“好友”，而有向图中的边是“关注”。一般而言，我们在数据结构中所讨论的图都是有向图，因为有向图相比无向图更具有代表性。
- 实际上，无向图可以由有向图来表示。如果 AB 两个点之间存在无向边的话，那用有向图也可以表示为：AB 两点之间同时存在 A 到 B 与 B 到 A 两条有向边。
- 仍然以社交网络举例：虽然微博中并不存在明确定义的好友关系，但是一般情况下，如果你和另一个 ID 互相关注的话，那么我们也可以近似认为，你和 TA 是好友。

图的种类

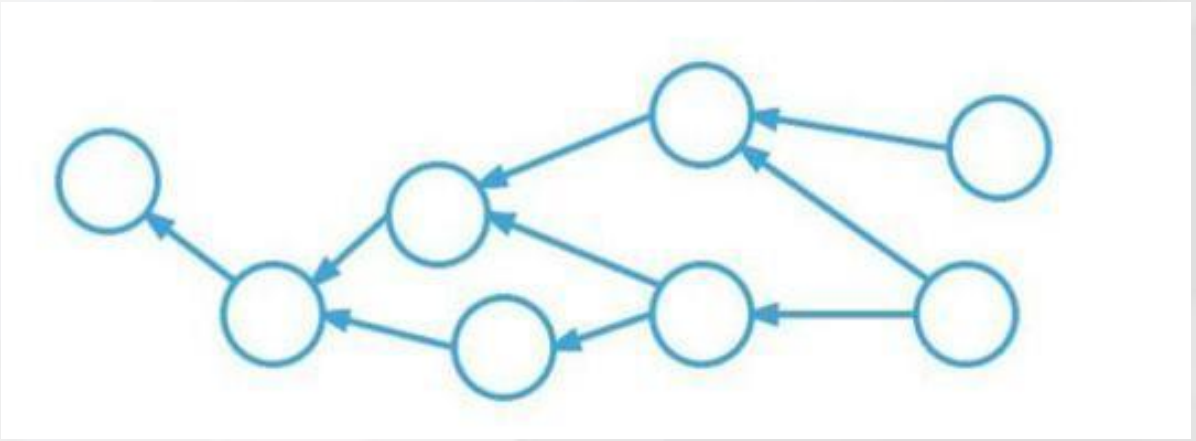
(1) 无向无权图，边没有权值、没有方向；

(2) 有向无权图，边有方向、无权值；

(3) 加权无向图，边有权值，但没有方向；

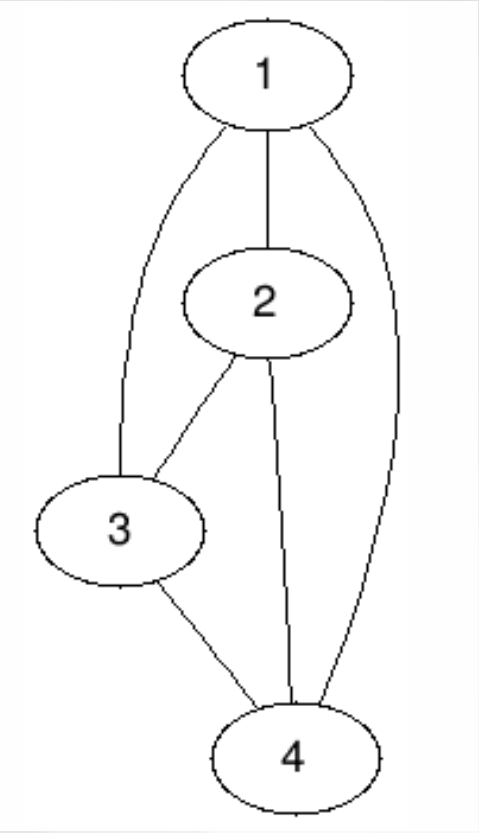
(4) 加权有向图；

(5) 有向无环图（Directed Acyclic Graph，DAG）。



图的常用概念--图的分类

- **稀疏图**: 有很少边或弧（如 e<nlogn，e 指边数，n 指顶点数）的图称为 稀疏图，反之称为 稠密图。如果图中边集为空，则称该图为 零图。
- 对应到微博里，如果在一个圈内，同学们都互相关注，则我们可以认为该关系图是一个稠密图，如果只有几个人关注了别人，则我们可以认为这是一个稀疏图。
- **完全图**:如果无向图中任何一对顶点之间都有一条边相连，也就是有 n×(n−1)/2 不重复的边，则这个无向图被称为 完全图。右图就是由 5个顶点组成的无向完全图。



- **有向完全图**：类似地，如果有向图中任何一对顶点 u,v之间都有两条有向边<u,v>,<v,u> 相连，则称这个有向图为 有向完全图。

图的常用概念--度

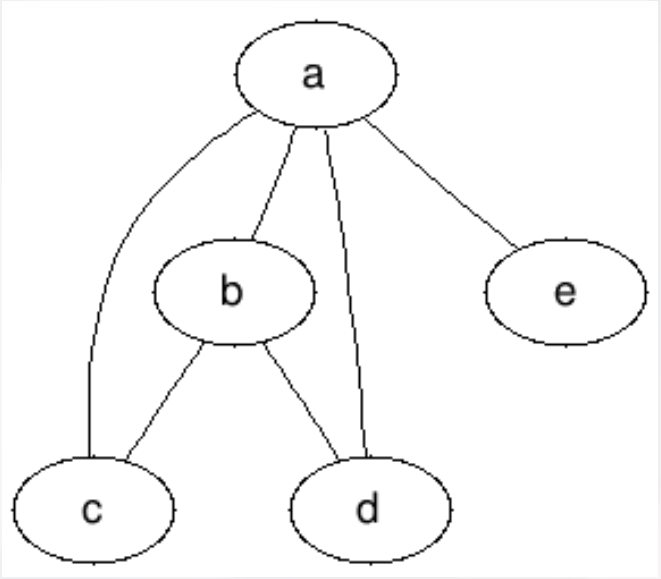
在无向图中，顶点的 **度** 是指某个顶点连出的边数。例如在左下图中，顶点 a 的度数为 4，顶点 b 的度数为 3。

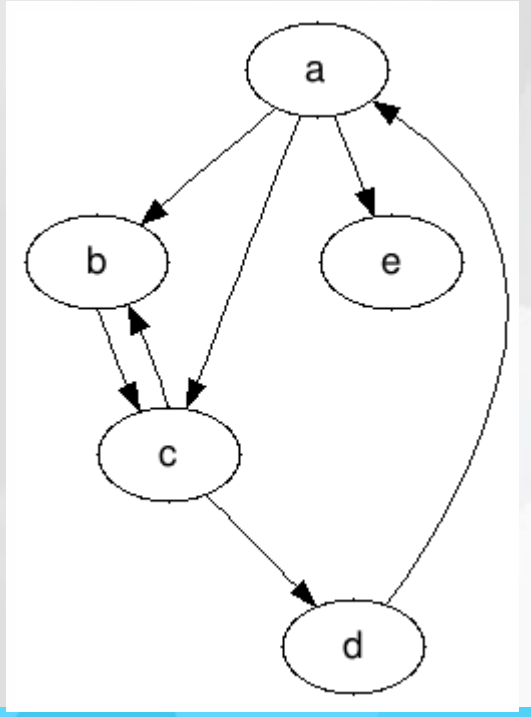
在有向图中，和度对应的是 **入度** 和 **出度** 这两个概念。

顶点的入度是指以该顶点为终点的有向边数量；

顶点的出度是指以顶点为起点的有向边数量。

需要注意的是，在有向图里，顶点的 度 为入度与出度之和。例如在右下图中，顶点 a 的入度为 1，出度为 3；顶点 c 的入度为 2，出度为 2。





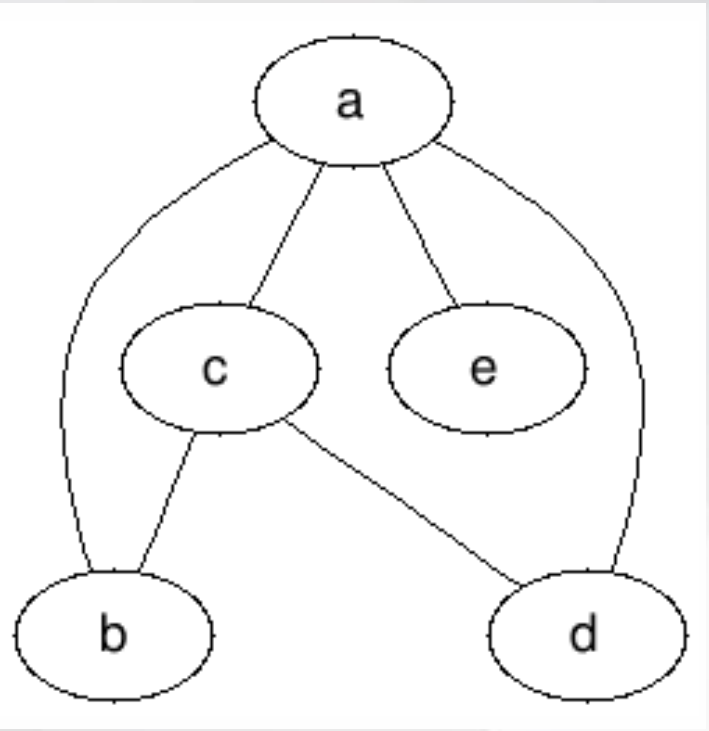
度的性质

在无向图或有向图中，顶点的度数总和为边数的两倍，即：

$$|E| = \frac{1}{2}\sum_{i=1}^{n} deg(u_i)$$

在无向图中，**度序列** 的定义为：将图 G 中所有顶点的度数排成一个序列 s，则称 s 为图 G 的**度序列**。

例如右面这张图中，其对应的一个度序列为 4,2,3,2,1，而 1,2,2,3,4也是其对应的度序列。换句话说，每个无向图对应的度序列不一定是唯一的。



可图的序列

- 一个序列是可图的，是指这个序列是某个无向图的度序列。判断一个序列是不是可图的，可以借助Havel-Hakimi 定理
- Havel-Hakimi 定理：由非负整数组成的不递增序列 s: $d_1, d_2, \ldots d_n (n \geq 2, d_1 \geq 1)$，是可图的，当且仅当序列

$$s_1 : d_2 - 1, d_3 - 1, \cdots, d_{d_1+1} - 1, d_{d_1+2}, \cdots, d_n$$

是可图的。序列 S_1中有 n−1 个非负整数， s序列中 d_1 后的前 d_1个度数（即 $d_2 \sim d_{d_1+1}$）减 1 后构成 S_1 中的前 d_1个数。

举例

例1：判断序列S:=6，5，4，3，3，3，2，0 是否可图。
证：a. 删除首元素6，将除去第一个元素后面的6个元素减一，得到：S1 = 4,3,2,2,2,1,0
b.删除首元素4，将除去第一个元素后面的4个元素减一，得到：S2 = 2,1,1,1,1,0
c,删除首元素2，将除去第一个元素后面的2个元素减一，得到：S3 = 0,0,1,1,0
d.重新排序：S4 = 1,1,0,0,0
e.删除首元素1，将除去第一个元素后面的1个元素减一，得到：S3 = 0,0,0,0

例2：判断序列S:=7，6，4，3，3，3，2，1 是否可图。
证：a. 删除首元素7，将除去第一个元素后面的7个元素减一，得到：S1 = 6,3,2,2,2,1,0
b.删除首元素6，将除去第一个元素后面的6个元素减一，得到：S2 = 2,1,1,1,0,-1
最后得到的是存在负数的序列，证明 序列式不可图的！

Havel-Hakimi 定理

```
bool Havel_Hakimi(int arr[])
{      for(int i=0; i<n-1; ++i)
      {     sort(arr+i,arr+n);// 从第i个元素开始非递增排序
            if(i+arr[i] >= n) return false;
                  //若第i个元素+arr[i]的值超过原数组长度，那么将溢出。
            for(int j=i+1; j<=i+arr[i] ; ++j)
            {
                  --arr[j];
                  if(arr[j] < 0) return false;
            }

      }
      if(arr[n-1]!=0) return false;
     return true;
 }
```

路径

- 在无向图G中，如果从顶点vi出发，沿着图中的边经过一些顶点vp1,vp2,...,vp到达顶点 vj，则称顶点序列（vi,vp1,...,vpm,vj）为从顶点 vi到顶点vj的一条 **路径**（Path），其中（vi,vp1),(vp1,vp2),...,(vpm,vj）均为G 中的边。如果 G是有向图，则 <vi,vp1>,<vp1,vp2>,...,<vpm,vj> 均为 G中的有向边。
- 路径中边的数量被称为 **路径长度**。如果路径中的顶点均不重复，则称这条路径为 **简单路径**。如果路径中的第一个顶点 vi 和最后一个顶点 vj是同一个顶点，则称这条路径为 **回路**。

子图

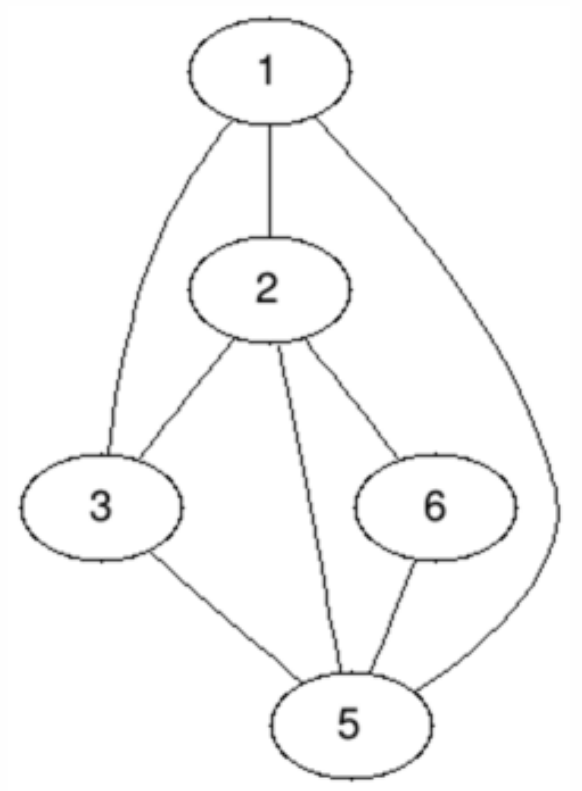
- 对于两个图 G1(V1,E1) 和 G2(V2,E2)，如果 V1⊆V2，且 E1⊆E2，则称图 G1 是图 G2 的子图。

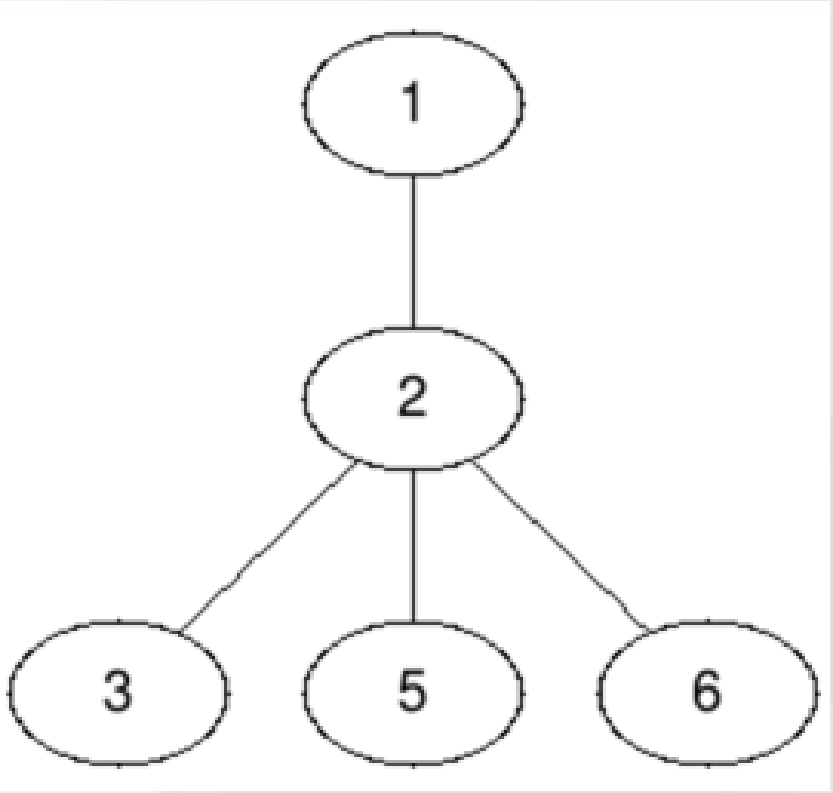
连通性

- 在无向图中，如果从顶点 u 到顶点 v 有路径，则称点 u 和点 v 是 **连通** 的。
- **连通图**: 如果无向图中任意一对顶点之间都是连通的，那么这个无向图就是**连通图**。如果一个无向图 G 的子图是连通图，则称该子图为图 G的 连通子图。
- **极大连通子图**: 如果一个连通子图加上任何一个不在子图中的顶点后都不能成为连通子图，则称其为 **极大连通子图**，又名 **连通分量**。
- 在有向图中，如果每对顶点之间都互相有路径可达，则称此图为 **强连通图**。如果一个有向图 G的子图是强连通图，则称该子图为图 G 的 **强连通子图**。如果一个强连通子图加上任何一个不在子图中的顶点后都不能成为强连通子图，则称其为 **极大强连通子图**，又名 **强连通分量**。

生成树

- 对一个无向图G(V,E) 来说，它的一个**生成树**指的是包含 |V| 个顶点、|V|–1 条边的连通子图。对于下面这个图 G 来说，右图就是图G 的一个生成树。





◆ 图的存储

- 能快速访问：图的存储，能让程序很快定位结点u和v的边(u, v) 。

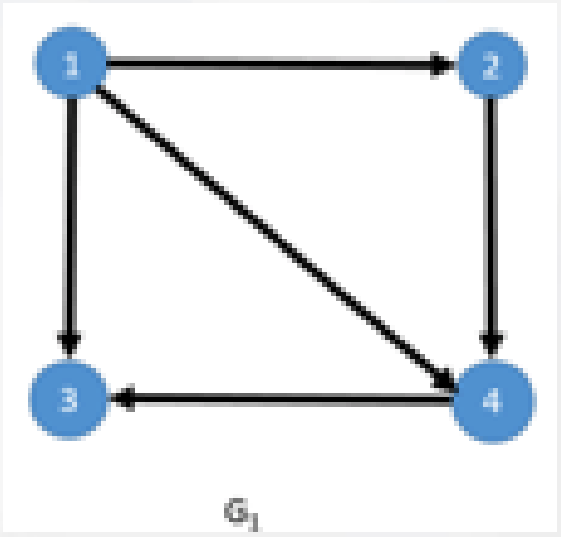
- 三种数据结构存图：
  邻接矩阵

  邻接表

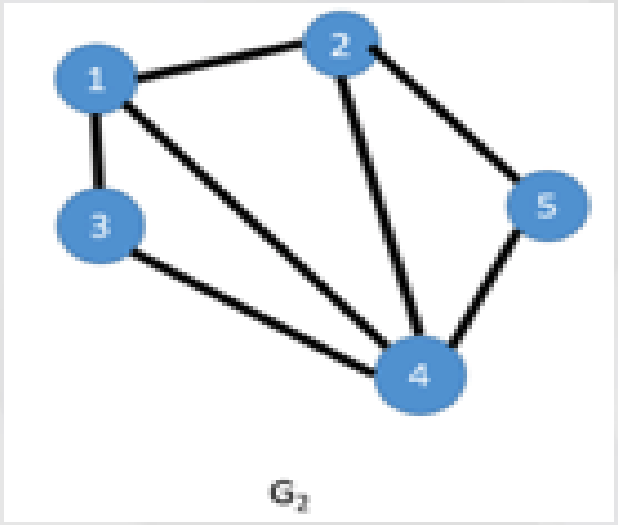
  链式前向星

邻接矩阵

- 邻接矩阵存储结构就是用矩阵表示图中各顶点之间的邻接关系。
- 对于有n个顶点的图 G=(V,E) 来说，我们可以用一个 n×n 的矩阵A来表示 G 中各顶点的相邻关系，如果 vi和vj 之间存在边（或弧），则 A[i][j]=1，否则 A[i][j]=0。无向图的邻接矩阵是一个对称矩阵。下图为有向图 G1 和无向图 G2 对应的邻接矩阵.也可以A[i][j]表示节点vi到vj边的权值， A[i][j]=INF表示vi和vj无边。
- 顶点的出度，即为邻接矩阵上点对应行上所有值的总和。



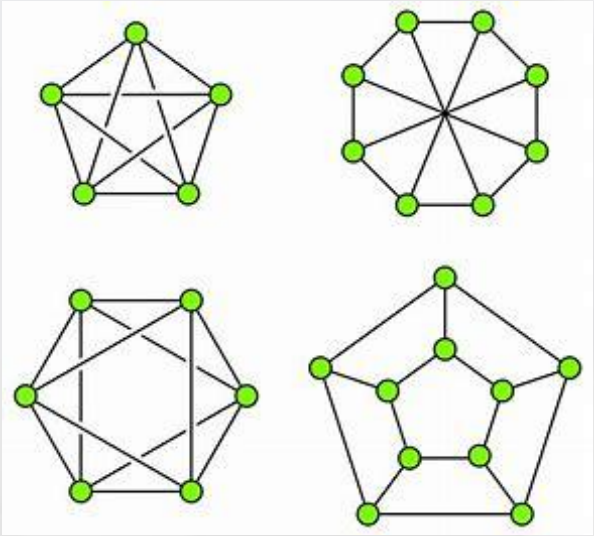
0	1	1	1
0	0	0	1
0	0	0	0
0	0	1	0



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

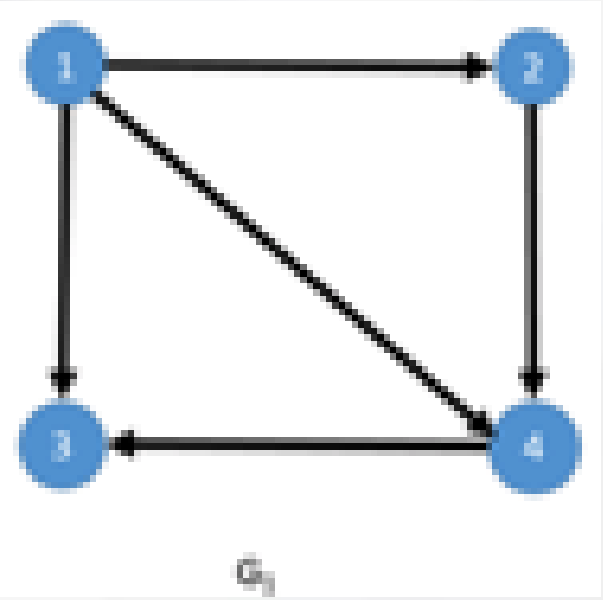
优点和缺点

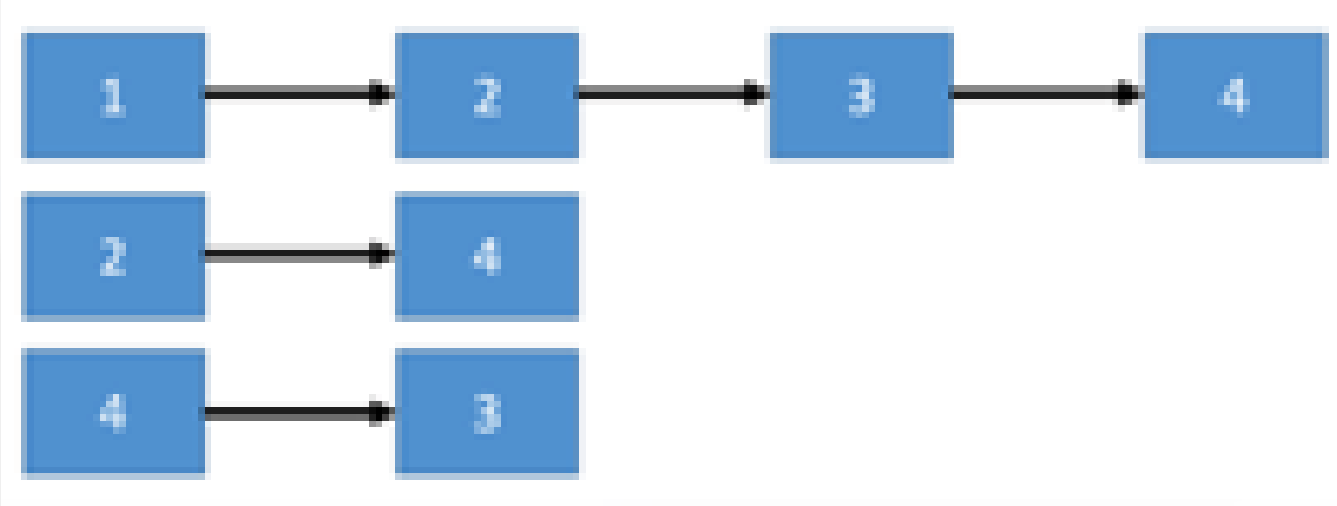
- 优点：
  适合稠密图；
  编码非常简短；
  对边的存储、查询、更新等操作又快又简单。
- 缺点：
  存储复杂度$O(|V|^2)$太高。$|V|=10000$时，空间100M。
  不能存储重边。



邻接表

- 邻接表是图的一种顺序存储与链式存储相结合的存储方法。我们给图G中的每个顶点建立一个单链表，第 i 个单链表中的结点表示依附于顶点vi的边（对于有向图是以vi为起点的弧）。所有单链表的表头结点都存储在一个一维数组中，以便于顶点的访问。下图为图G1对应的邻接表。





- 在无向图的邻接表中，顶点vi的度为第i个单链表中的结点数；而在有向图中，第i个单链表中的结点数表示的是顶点vi的出度，如果要求入度，则要遍历整个邻接表。另外，在邻接表中，我们很容易就能知道某一顶点和哪些顶点相连接。

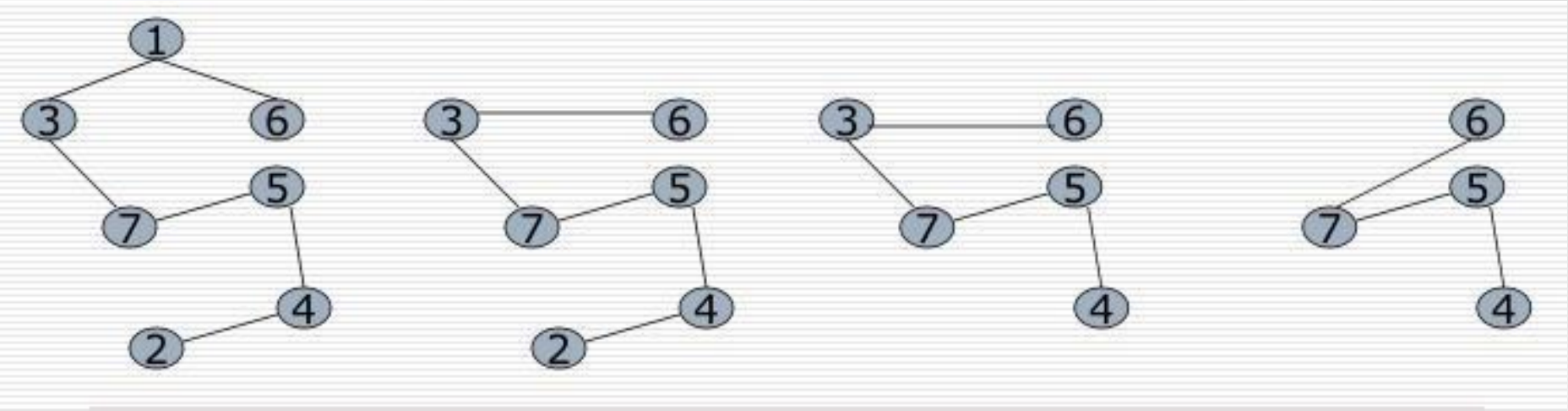
邻接表优缺点

- **应用场景：大稀疏图。**
- **优点：**

  存储效率非常高，存储复杂度O(|V|+|E|)；

  能存储重边。

- **缺点：编程比较麻烦。**



用vector实现图的邻接表

```
//定义边
struct edge{
    int from, to, w;  //边：起点from，终点to，权值w
    edge(int a, int b,int c){from=a; to=b; w=c;}  //对边赋值
};
vector<edge> e[NUM];  //e[i]：存第i个结点连接的所有的边
//初始化
for(int i=1; i<=n; i++)     e[i].clear();
//存边
e[a].push_back(edge(a,b,c)); // 把边(a,b) 存到结点a的邻接表中

//检索结点u的所有邻居
    for(int i=0; i < e[u].size(); i++){
            //结点u的邻居有e[u].size()个
        …
    }
```

邻接矩阵和邻接表使用场合

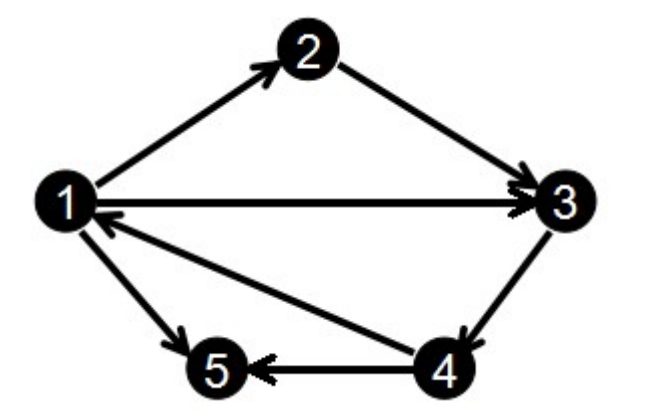
- 那我们什么时候用邻接矩阵，什么时候用邻接表呢?
- 我们可以看到，邻接矩阵存储结构最大的优点就是简单直观，易于理解和实现。其适用范围广泛，有向图、无向图、混合图、带权图等都可以直接用邻接矩阵表示。另外，对于很多操作，比如获取顶点度数，判断某两点之间是否有连边等，都可以在常数时间内完成。
- 然而，它的缺点也是显而易见的：从以上的例子我们可以看出，对于一个有n个顶点的图，邻接矩阵总是需要 $O(n^2)$ 的存储空间。当边数很少的时候，就会造成空间的浪费。
- 因此，具体使用哪一种存储方式，要根据图的特点来决定：如果是稀疏图（点多边少），我们一般用邻接表来存储，这样可以节省空间；如果是稠密图（点少边多），当需要频繁判断图中的两点之间是否存在边时往往用邻接矩阵来存储，其他时候用邻接表或邻接矩阵皆可。

链式前向星存储结构

- 链式前向星又称为邻接表的静态建表方式，其最开始确实是基于前向星，是以提高前向星的构造效率为目的设计的存储方式，最终演变成了一个变形的邻接表这一数据结构。
- 链式前向星采用数组模拟链表的方式实现邻接表的功能（数组模拟链表的主要方式就是记录下一个节点在数组的哪个位置。），并且使用很少的额外空间，可以说是目前建图和遍历效率最高的存储方式了.

前向星

- 前向星：其实就是一种边集数组，我们把边集数组中的每一条边按照起点从小到大排序,如果起点相同就按照终点从小到大排序,并记录下以某个点为起点的所有边在数组中的起始位置和存储长度,那么前向星就构造好了.其中len[i]表示以第i个点为起点的边的数目，head[i]存储第i个点的第一条边的位置。于是这个图就被存起来了。
- 举个例吧：



对于左图，假设输入顺序：

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

那么排完序后就得到:

编号:	1	2	3	4	5	6	7
起点u:	1	1	1	2	3	4	4
终点v:	2	3	5	3	4	1	5

前向星就是这样

head[1] = 1　len[1] = 3
head[2] = 4　len[2] = 1
head[3] = 5　len[3] = 1
head[4] = 6　len[4] = 2

利用前向星会有排序操作,如果用快排时间至少为O(nlog(n))

用链式前向星,就可以避免排序

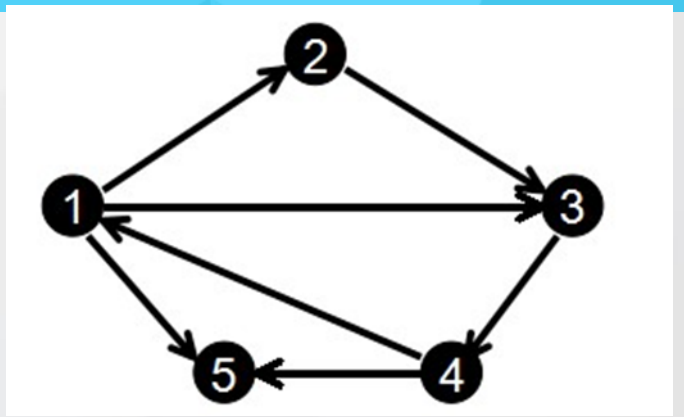
建立边结构体为:

```
struct Edge
{
    int to;//第i条边的终点
    int w;//第i条边的权值
    int next;//与第i条边同起点的下一条边的存储位置(边的读入顺序号)
} edge[MAXN];
int head[MAXN];//head[i]表示以i为起点的第一条边的存储位置,
                        //实际上最后存的是以i为起点的所有边的最后输入的那个编号.
```

```
5 7
1 2 1
2 3 2
3 4 3
1 3 4
4 1 5
1 5 6
4 5 7
```

添加边

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



tot为边数，初始化tot = 0,head[]=-1。这样,按照上面的图和输入来模拟如右所示:

```
5 7
1 2 1  edge[1].to=2;edge[1].next=-1;head[1]=1
2 3 2  edge[2].to=3;edge[2].next=-1;head[2]=2
3 4 3  edge[3].to=4;edge[3].next=-1;head[3]=3
1 3 4  edge[4].to=3;edge[4].next=1;head[1]=4
4 1 5  edge[5].to=1;edge[5].next=-1;head[4]=5
1 5 6  edge[6].to=5;edge[6].next=4;head[1]=6
4 5 7  edge[7].to=5;edge[7].next=5;head[4]=7
```

遍历点及边

```
for(int u=1;u<=n;u++)
        for (int i=head[u];i>0;i=edge[i].next){
          cout<< “edge No:” <<i<< “;” <<u<< “->” <<edge[i].to;
          cout<<";.next:"<<edge[i].next<<" 权值: "<<edge[i].w<<endl;
      }
```

先遍历编号head[1]=6 的边, 然后就是edge[6].next,也就是编号4的边,然后继续edge[4].next,也就是编号1的边,可以看出是逆序的

```
edge No:6;1->5;.next:4 权值: 6
edge No:4;1->3;.next:1 权值: 4
edge No:1;1->2;.next:-1 权值: 1
edge No:2;2->3;.next:-1 权值: 2
edge No:3;3->4;.next:-1 权值: 3
edge No:7;4->5;.next:5 权值: 7
edge No:5;4->1;.next:-1 权值: 5
```

习题：邻接矩阵的使用

- 给出一个包含有向图和无向图的混合图 G，图上有 n 个点和 m 条边，现在你需要使用邻接矩阵来存储该混合图 G 并按格式输出邻接矩阵。
- 输入格式：输入第一行为两个正整数 n 和 m（1≤n,m≤100），表示混合图上的 n 个点和 m 条边。接下来输入 m 行，每行输入三个整数 a，x，y（0≤x,y< n），表示点 x 和点 y 之间有一条边。如果 a=0，则表示该边为有向边，如果 a=1，则表示该边为无向边。
- 输出格式：输出一个 n×n 的邻接矩阵，矩阵中第 i 行第 j 列的值描述了点 i 到点 j 的连边情况。如果值为 0 表示点 i 到点 j 没有边相连，值为 1 表示有边相连。在每一行中，每两个整数之间用一个空格隔开，最后一个整数后面没有空格。
- 样例输入
- 4 4
- 0 0 1
- 1 0 2
- 0 3 1
- 1 2 3
- 样例输出
- 0 1 1 0
- 0 0 0 0
- 1 0 0 1
- 0 1 1 0

习题：邻接表的使用

- 给出一个包含有向图和无向图的混合图 G，图上有 n个点和 m条边，现在你需要使用邻接表来存储该混合图 G 并按格式输出邻接表。
- 输入格式：输入第一行为两个正整数 n和m(1≤n,m≤100），表示混合图上的n个点和m条边。接下来输入 m 行，每行输入三个整数 a，x，y（0≤a≤10，0≤x,y<n），表示点x和点y之间有一条边。a=0，则表示该边为有向边，a=1，则表示该边为无向边。
- 输出格式：输出邻接表，输出 n行，第i行表示第i个点连接边的情况，首先输出 i，接着输出:，然后输出所有点 i能到达的点的编号，边关系中后出现的点先输出。每个整数前有一个空格，具体格式见样例。
- 样例输入
- 4 4
- 0 0 1
- 1 0 2
- 0 3 1
- 1 2 3
- 样例输出
- 0: 2 1
- 1:
- 2: 3 0
- 3: 2 1

习题：画图游戏

- 小明想让你帮他画一个无向图，图中有 n 个顶点，小明告诉你这 n 个顶点的度数。为了简化问题，你只需要告诉小明图的邻接矩阵就可以了，图中不存在自环的情况，也就是说邻接矩阵的正对角线均为0。（矩阵可能不唯一，只要符合要求即可）
- 输入格式：第一行输入一个整数n，代表顶点的个数（1≤n≤15），第二行是n个整数，分别代表这n个顶点的度数。
- 输出格式：输出一个 n×n 的01矩阵，代表图的邻接矩阵。如果没有符合要求的图，则输出“None”。
- 样例输入1
- 4
- 2 3 2 1
- 样例输出1
- 0 1 1 0
- 1 0 1 1
- 1 1 0 0
- 0 1 0 0
- 样例输入2
- 4
- 3 2 1 1
- 样例输出2
- None

思路

- 应用Havel-Hakimi定理。利用结构体来记录每个节点的编号和度。将节点按度数排序，每个节点的度为a[i]。套用公式，对于度数最大的那个点，我们将其与度数次大的a[i]个点用边连起来（即公式中将次大的a[i]个点的度值减一），然后这个点就可以不管了（即公式中删除最大的点）。这样一直做，如果出现有边的度数为负，则这个序列不可图，输出None，若一直减少到所有点度数为0，则得到一个可行的图。
- 注意：

1、由于只有部分点的度减一，顺序可能会有所不同，每次都应该重新排序。

2、可能会出现度数超过节点数量的情况，也要考虑。

回顾下我们之前介绍的Havel-Hakimi()函数，稍微改下，就能完成我们的任务

结构体内嵌比较函数

```
struct Node
{
    int d, id;
    bool operator<(const Node &rhs) const
    {
        return d > rhs.d; }
};
Node arr[maxn];


sort(node,node+n)
```

```
struct Node
{
        int d, id;
 };
Node arr[maxn];

bool cmp(Node a,Node b){
    return a.d>b.d;
 }

sort(node,node+n,cmp)
```

Havel_Hakimi()稍微改了一下

```
int G[maxn][maxn];//邻接矩阵
bool Havel_Hakimi(Node  arr[]){
    for(int i=0; i<n-1; ++i){
        sort(arr+i,arr+n);// 从第i个元素开始非递增排序
            if(i+arr[i].d >= n) return false;
                //若第i个元素+arr[i].d的值超过原数组长度，那么将溢出。
            for(int j=i+1; j<=i+arr[i].d ; ++j){
                --arr[j].d;
                if(arr[j].d < 0) return false;//度数为负
                G[arr[i].id][arr[j].id]=G[arr[j].id][arr[i].id]=1;
            }
      }
    if(arr[n-1].d!=0) return false;
    return true;
 }
```

main()

```
int main(void) {
    scanf("%d", &n);
    for (int i = 0; i <n; ++i) {
        scanf("%d", &arr[i].d);
        arr[i].id = i;
    }

    if (!Havel_Hakimi(arr) ) { printf("None\n"); }
    else {
        for (int i = 0; i <n; ++i) {
            for (int j = 0; j < n-1; ++j) { printf("%d ", G[i][j]); }
            printf("%d\n", G[i][n-1]);
        }
    }
    return 0;
}
```

图的深度和广度优先搜索

- 在现实生活中，一系列个体与个体之间存在的两两关系，可以抽象成图这种数据结构来表示。现在，让我们来讨论一个有趣的例子：维基百科的“终极真理之路”。
- 维基百科很多人都用过，但是很少有人知道它有一个特别的“玩法”：任意点开一个词条，比如“WeChat（微信）”，然后每次只点击词条中的第一个链接，在经历了一定的跳转次数之后，你最后一定能抵达“Philosophy（哲学）”这一词条。这个有趣的现象就被称为“维基百科的终极真理”。
- 和人际交往之间的六度分割原理类似，维基百科遵循的是一种三度分割原理：两个词条之间的最短路径普遍为三个，比如从“我”到“奥巴马”：
- I → letter → Federal Bureau of Investigation → Barack Obama
- 再比如从“微软”到“理查德.F.赫克（2010年诺贝尔化学奖得主）”：
- Microsoft → Intel → Chemist → Richard F.Heck
- 那么，对于计算机来说，怎样才能知道从一个词条出发可以跳转到哪些词条呢？整个维基百科可以抽象成一个图结构——每一个词条都是一个顶点，而指向其他词条的链接就是图中的边。运用计算机求解这一问题，需要用到我们接下来要介绍的图的遍历算法。
- 什么是图的遍历呢？从图的某个顶点出发，沿图中的路径依次访问图中所有顶点，并且使得图中所有顶点都恰好被访问一次，这一过程即为图的遍历。需要注意的是，接下来讨论图的遍历时，都是特指在一个连通图上进行遍历。
- 图有两种最常见的遍历算法：深度优先搜索（DFS）和广度优先搜索（BFS），在这节课里，我们将首先介绍深度优先搜索。

图的深度优先搜索

- 对于算法的具体实现，因深度优先搜索的优先遍历深度更大的顶点，所以我们可以借助栈这一数据结构来实现：
  1. 将要访问的第一个顶点 v入栈，然后首先对其进行访问。
  2. 将顶点 v出栈，依次将与顶点 v相邻且未被访问的顶点 c压入栈中。
  3. 重复第一步操作，直至栈为空。
- 为了方便，我们通常以递归的形式实现深度优先搜索。

C++ 示例代码如下

```cpp
const int MAX_N = 100;
const int MAX_M = 10000;
struct edge {
    int v, next;
} e[MAX_M];
int p[MAX_N], eid;
void init() {
    memset(p, -1, sizeof(p));
    eid = 0;
}
void insert(int u, int v) {
    e[eid].v = v;
    e[eid].next = p[u];
    p[u] = eid++;
}
bool vst[MAX_N];

// 在每次 dfs 之前需要将 vst 数组中的元素全部初始化为 false
void dfs(int u) {
    cout << "visiting " << u << endl;  // 访问顶点 u
    vst[u] = true;
    for (int i = p[u]; i != -1; i = e[i].next) {
        if (!vst[e[i].v]) {
            dfs(e[i].v);
        }
    }
}
```

图的广度优先搜索

- 结合队列先进先出的特性，我们可以借助它来具体实现广度优先搜索：
- 1. 任意选择一个顶点 v作为起点，加入队列。
- 2. 访问队首元素 v并标记，将其从队列中删除。
- 3. 遍历与顶点 v相邻且未被访问的所有顶点 c1，c2，…，ck，并依次加入到队列中。
- 4. 重复第二步和第三步操作，直到队列为空。

BFS的C++ 示例代码如下：

```cpp
const int MAX_N = 100;
const int MAX_M = 10000;
struct edge {
    int v, next;
} e[MAX_M];
int p[MAX_N], eid;
void init() {
    memset(p, -1, sizeof(p));
    eid = 0;
}
void insert(int u, int v) {
    e[eid].v = v;
    e[eid].next = p[u];
    p[u] = eid++;
}
```

```cpp
bool vst[MAX_N];
int q[MAX_N], l, r;  // 用数组实现队列

void bfs(int v) {
    memset(vst, 0, sizeof(vst));
    l = 0, r = -1;  // 初始化队列
    q[++r] = v;
    vst[v] = 1;
    while (l <= r) {
        int u = q[l++];
        cout << "visiting " << u << endl;  // 访问顶点 u
        for (int i = p[u]; i != -1; i = e[i].next) {
            if (!vst[e[i].v]) {
                vst[e[i].v] = 1;
                q[++r] = e[i].v;
            }
        }
    }
}
```

习题：修建大桥

- 小明来到一个由 n 个小岛组成的世界，岛与岛之间通过修建桥，来让岛上的居民可以去其他的小岛。已知已经修建了 m 座桥，居民们想让小明帮忙计算，最少还要在修建几座桥，居民们才能去所有的岛。
- 输入格式: 第一行输入俩个数字 n，m（$1 \le n \le 1000$，$0 \le m \le n \times (n-1)/2$），分别代表岛的个数，和已经修建的桥的个数，岛的编号分别是 1...n。接下来的 m 行，每行俩个数字，代表这俩个编号的岛之间已经有一座桥了。
- 输出格式: 输出最少还需要修建多少座桥，居民才能去所有的岛。
- 样例输入
- 5 4
- 1 2
- 2 3
- 4 5
- 1 3
- 样例输出
- 1

习题：农场看守

- 小明最近做了农场看守，他每天晚上的工作就是巡视农场并且保证没有人破坏农场。从谷仓出发去巡视，并且最终回到谷仓。
- 小明视力不太好，其他农场守卫只需要对农场的每一条连接不同场地的路走一遍就可以发现是不是有异常情况了。但是他很仔细和耐心，对农场的每一条连接不同场地的路需要走两遍，并且这两遍必须是不同的方向，因为他觉得应该不会两次都忽略农场中的异常情况。
- 每两块地之间一定至少有一条路连接。现在的任务就是帮他制定巡视路径。
- 输入格式: 第一行输入两个整数 N(2≤N≤10000) 和 M(1≤M≤50000)，表示农场一共有N 块地 M 条边。第二到 M+1 行输入两个整数，表示对应的两块地之间有一条边。
- 输出格式: 输出 2M+1个数，一个数占一行，表示小明巡查路径上地的标号，1 号为谷仓，从 1 开始，以 1 结束。如果有多种答案，输出任意一种。
- 本题答案不唯一，符合要求的答案均正确
- 样例输入
  - 4 5
  - 1 2
  - 1 4
  - 2 3
  - 2 4
  - 3 4
- 样例输出
  - 1
  - 2
  - 3
  - 4
  - 2
- 1
  - 4
  - 3
  - 2
  - 4
  - 1

习题：互粉攻略

- 小明和他的同事们最近在玩一个好玩的游戏：互粉攻略。一共有 N 个人参加游戏，编号从 0 到 N−1，游戏前每个人都会展示自己最靓丽的一面。当游戏开始时，每个人可以选择去关注别人。当 A 关注了 B，则 A 就成了 B 的粉丝，但是并不意味着 B 同时关注了 A。当所有人都选好后，游戏结束，人气指数最高的人成为冠军。小明制定了奇怪的规定：一个人的人气指数等于他的粉丝数减去关注数，因为小明觉得人气高的人，往往有很多粉丝，并且一般都非常高冷，很少去关注别人。
- 小明发现一共有 M 条关注，粗心的他在统计时出了点小问题，所以可能会出现重复的关注。现在小明想知道每个人的人气指数，聪明的你能帮帮他么?
- 输入格式: 第一行输入两个数 n 和 m，$1 \le n \le 1000$，$1 \le m \le 100000$。接下来输入 m 行，每行输入两个数 a 和 b，表示编号 a 的人关注了编号 b 的人，$0 \le a,b \le n-1$，$a \ne b$。
- 输出格式: 输出 N 行，每行输出每个人的人气指数，按编号依次输出即可。
- 样例输入
- 4 3
- 0 2
- 2 3
- 0 1
- 样例输出
- -2
- 1
- 0
- 1

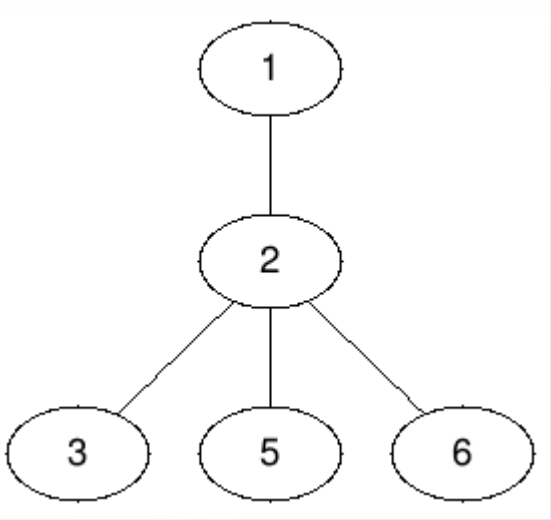
习题：最短路简化版

- 经历一周忙碌的工作后，小明想趁着周末好好游玩一番。小明想去好多好多地方，他想去南锣鼓巷吃各种好吃的，想去颐和园滑冰，还想去怀柔滑雪场滑雪……可是时间有限，小明并不能玩遍所有的地方，最后他决定去几个离他较近的。
- 我们知道小明一共想去 N个地方玩耍，编号从 1到 N，并且知道了小明所在地方的编号 C，以及M条路径。现在小明想让你帮他算一算，他到每个地方分别需要经过多少个地方?
- 输入格式: 第一行输入三个正整数 N,M,C。代表小明想去 N个地方，有 M条路径，小明在编号为C的地方。1≤N,C≤1000, 1≤C≤N, 1≤M≤10000。保证没有重复边，且图中所有点互相连通。
- 输出格式: 输出 N行，按编号从小到大，输出结果。第 i行表示小明到编号为 i的地方，需要经过多少个地方。

样例输入	样例输出
5 5 2	1
1 2	0
2 3	1
2 4	1
3 4	2
3 5	

7.2. 树的存储和遍历

- 用图来定义树：如果一个无向连通图中不存在回路，则称这个图为 树。例如下图就是一棵树：

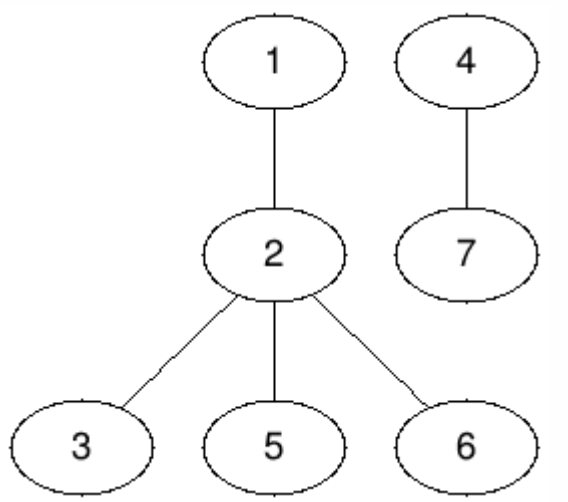


- 在这棵树上，1是树中的 根结点，3是 2 的 子结点，1是 2 的 父结点。除根结点外，每个结点有且只有一个父结点。如果一个结点没有子结点，则该结点被称为 叶结点。树中结点的子结点个数被称为结点的 度（注意，这个度和图中的度不一样），最大度为 2的树被称为 二叉树。

树有一些常用的性质

1. 若树上的结点数为 n，则边数一定为 n−1。
2. 树上的任意一对结点之间有且仅有一条路径。

- 对于树上的结点u，其到根结点的路径上的所有结点都是结点u的 祖先。对于树上的两个结点 u,v，如果一个结点同时为u和v的祖先，则称之为 u,v的公共祖先。所有公共祖先中距离根结点最远的那个被称为 最近公共祖先（Least Common Ancestors, LCA）。
- 如果一个无向图中包含了几棵不互相连通的树，则称该无向图为 森林。很显然，森林是一个非连通图，例如下面这张图，就表示了一个森林：



- 树的存储和图没有区别，由于树是稀疏图，所以通常使用邻接表来存储。

树的遍历

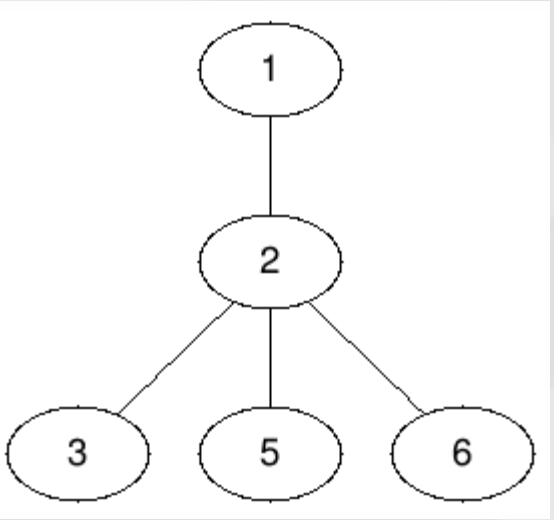
因为可以把树当成一个稀疏无向图来处理，所以在树上进行深度优先搜索和广度优先搜索与在无向图中没有任何区别。特别地，在对树进行深度优先搜索（dfs）时，有一个常用的概念——dfs 序，它的生成过程举例如下：

对于如右这棵树，从顶点1开始进行深度优先搜索，

如果当每次开始访问某个顶点、和结束访问某个顶

点时均将这个顶点插入序列中，则生成的序列如下：

1,2,3,3,5,5,6,6,2,1



这个序列就是 dfs 序。dfs 序有一个非常重要的性质：

子树内所有结点在 dfs 序中是连续的一段。例如2所在子树内的所有顶点对应着 dfs 序中 2,3,3,5,5,6,6,2这段区间，且左右两端一定是该子树的根结点。

有很多树上的问题都可以借助 dfs 序转化为数组中的问题，而和 dfs 序结合最多的就是各种高级数据结构，如线段树等。

例题

- 给定一棵树，算出每个结点所在子树中包括多少个叶结点。
- 解析：我们可以用cnt[u]来记录结点u所在子树的叶结点个数，有如下的计算公式：

• 若 u 非叶结点：$cnt_u = \sum_{v \in childs_u} cnt_v$；

•若u 为叶结点：$cnt_u = 1$。

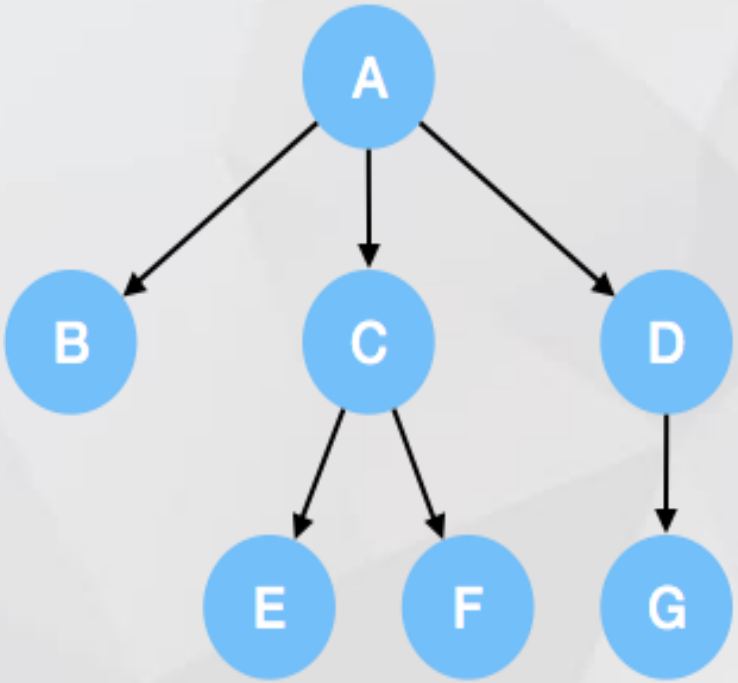
于是，我们可以基于深度优先搜索，用如下的代码来递归地算出答案：

```
// 邻接表的基础代码省略，可以参考前面课节中的内容
int cnt[MAX_N]; // 统计结果
void dfs(int u) {
    vst[u] = true;
    bool is_leaf = true;
    for (int i = p[u]; i != -1; i = e[i].next) {
        int v = e[i].v;
        if (!vst[v]) {
            dfs(v);
            is_leaf = false;
            cnt[u] += cnt[v];
        }
    }
    if (is_leaf) {
        cnt[u] = 1;
    }
}
```

N叉树遍历

一棵二叉树可以按照前序、中序、后序或者层序来进行遍历。在这些遍历方法中，前序遍历、后序遍历和层序遍历同样可以运用到N叉树中。N叉树的中序遍历没有标准定义，中序遍历只有在二叉树中有明确的定义。

如图所示的三叉树来举例说明:



1.前序遍历

在N叉树中，前序遍历指先访问根节点，然后逐个遍历以其子节点为根的子树。

例如，上述三叉树的前序遍历是: A->B->C->E->F->D->G.

2.后序遍历

在N叉树中，后序遍历指前先逐个遍历以根节点的子节点为根的子树，最后访问根节点。

例如，上述三叉树的后序遍历是: B->E->F->C->G->D->A.

3.层序遍历

N叉树的层序遍历与二叉树的一致。通常，当我们在树中进行广度优先搜索时，我们将按层序的顺序进行遍历。

例如，上述三叉树的层序遍历是: A->B->C->D->E->F->G.

N叉树的存储

用指针实现。

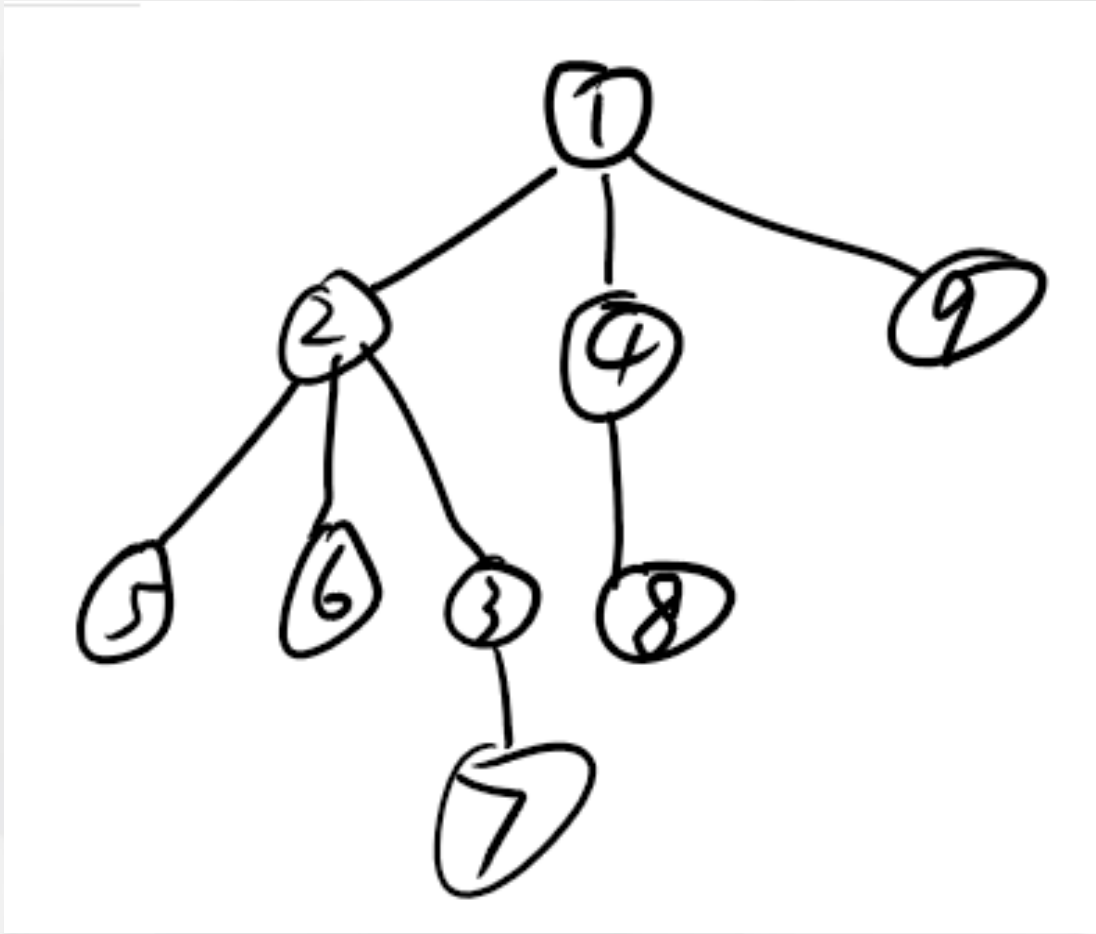
```
struct Node{
    int val;     //结点的值
    vector<Node*> children;//指向子结点
};
```

例题：建立多叉树

输入建立多叉树，输入的格式是每个节点的具体数据t和拥有的孩子数目n，输入顺序按节点的先序顺序。分别输出节点的先序、后序、层序的顺序。

输入样例（建立如图所示的多叉树）：

1 3
2 3
5 0
6 0
3 1
7 0
4 1
8 0
9 0



输出样例

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

建树的代码

```
struct Node
{
    int val;
    vector<Node*> children;
};

Node* build()//递归建立多叉树
{
    int t, n;
    cin >> t >> n;
    Node* p = 0;
    p = new Node;
    p->val = t;
    for (int i = 0; i != n; ++i)
    {
        p->children.push_back(build());
    }
    return p;
}
```

```
int    main()
{
    vector<int> pre,in,post;
    Node* root = 0;
    root = build();
    PreOrder(root,pre);
    for(vector<int>::iterator iter = pre.begin();
          iter != pre.end(); iter++)
       cout << *iter<< ' ';
    cout<<endl;

    PostOrder(root,in);
          …

    LevelOrder(root,post);
          …
      return 0;
}
```

先序遍历的算法：

```
void PreOrder(Node* T,vector<int> &res){
    //这里的&必须要加上，不然会出现这样的结果
    if(T==NULL) return;
    res.push_back(T->val);
    for(int i=0;i!=(T->children).size();i++){
        PreOrder((T->children)[i],res);
    }
    /*上面for循环也可以简化下：
        for(Node* next: T->children {
            PreOrder(next,res);}
    */


}
```

后序遍历

提问：后序遍历下面的代码有没有问题?

```
void PostOrder(Node* root,vector<int> &vec){
    //这里的&必须要加上，参数引用
      if(root==NULL){
         return;
      }
      for(int i=0;i!=root->children.size();i++){
         PostOrder((root->children)[i],vec);
      }
         /*上面for循环也可以简化下:
         for(Node* next: root->children {
                  PostOrder(next,vec);}
         */
      vec.push_back(root->val);
   }
```

层序遍历

```
void LevelOrder(Node* root,vector<int> &vec){//层序遍历
    if (root != 0){
        Node* t;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            t = q.front();
            vec.push_back(t->val);
            q.pop();
            for (vector<Node*>::size_type i = 0; i != t->children.size(); ++i){
                q.push(t->children[i]);
            }
        }
    }
}
```

树的直径

- 给定一棵树，树中每条边都有一个权值，树中两点之间的距离定义为连接两点的路径边权之和。树中最远的两个节点之间的距离被称为树的直径，连接这两点的路径被称为树的最长链。后者通常也可称为直径，即直径是一个数值概念，也可代指一条路径。
- 树的直径通常有两种求法：1.树形DP求树的直径；2.两次BFS（DFS）求树的直径
- 时间复杂度均为O(n)。我们假设树以N个点N-1条边的无向图形式给出，并存储在邻接表中。

树形DP求树的直径

设1号节点为根,"N个点N-1条边的无向图"就可以看做“有根树”
设d[x]表示从节点x出发走向以x为根的子树，能够到达的最远节点的距离。设x的子节点为$y1, y2, y3, \ldots, yt$，$edge(x, y)$表示边权，显然有：

$$d[x] = max\{d[yi] + edge(x, yi)\}(1 <= i <= t)$$

接下来，我们可以考虑对每个节点x求出"经过节点x的最长链的长度"f[x]，整棵树的直径就是max{f[x]}(1 <= x <= n)
对于x的任意两个节点yi和yj，"经过节点x的最长链长度"可以通过四个部分构成：从yi到yi子树中的最远距离，边(x, yi)，边(x, yj)，从yj到yj子树中的最远距离。设j < i，因此：

$$f[x] = max\{d[yi] + d[yj] + edge(x, yi) + edge(x, yj)\}(1 <= j < i <= t)$$

但是我们没有必要使用两层循环来枚举i，j。在计算d[x]的过程中，子节点的循环将要枚举到i时d[x]恰好就保存了从节点x出发走向“以yj(j < i)为根的子树”，能够到达的最远节点的距离，这个距离就是max{d[yi] +edge(x, yi)}(1<=j< i)。所以我们先用d[x] + d[yi] + edge(x, yi)更新f[x]，再用d[yi] + edge(x, yi)更新d[x]即可

两次BFS（DFS）求树的直径

- 通过两次BFS或者两次DFS也可以求树的直径，并且更容易计算出直径上的具体节点
- 详细地说，这个做法包含两步：
- 1.从任意节点出发，通过BFS和DFS对树进行一次遍历，求出与出发点距离最远的节点记为p
- 2.从节点p出发，通过BFS或DFS再进行一次遍历，求出与p距离最远的节点，记为q。
- 从p到q的路径就是树的一条直径。因为p一定是直径的一端，否则总能找到一条更长的链，与直径的定义矛盾。显然地脑洞一下即可。p为直径的一端，那么自然的，与p最远的q就是直径的另一端。
- 在第2步的遍历中，可以记录下来每个点第一次被访问的前驱节点。最后从q递归到p，即可得到直径的具体方案

习题：子树的结点个数

- 有一个棵树，树上有 n 个结点。结点的编号分别为 1...n，其中 1是树的根结点。现在希望你帮忙计算每个结点作为根结点的子树分别有多少结点。
- 输入格式：第一行输入一个数字 n（2≤n≤1000），代表树上结点的个数。接下来的 n−1 行，每行俩个数字 a，b，代表结点 a到结点 b 有一条边。
- 输出格式：按编号顺序输出每个结点作为根结点的子树，分别有多少结点，中间用空格分开。
- 样例输入
- 5
- 1 4
- 1 3
- 3 2
- 3 5
- 样例输出
- 5 1 3 1 1

习题：网络延时

- 某计算机网络中存在 n个路由，每个路由代表一个子网。路由之间有 n−1 条互通关系，使得这 n 个网络之间任意两个网络都可以直接联通，或者通过其他网络间接连通。
- 为了测试组建的网路的性能，假设相邻的路由之间的数据传输需要一单位时间，现在需要知道任意两个路由之间传输数据最多需要多长时间。
- 输入格式: 第一行一个整数 n(2≤n≤50000) 表示网络中路由个数。接下来 n−1 行，每行输入 u,v(1≤u,v≤n) ，表示路由 u,v 相连。
- 输出格式: 输出一行表示答案。
- 样例输入
- 8
- 6 3
- 3 7
- 3 4
- 7 5
- 5 1
- 6 8
- 5 2
- 样例输出
- 5