图论基础

一、图的概念

1.1 图的定义

图是由一个顶点集 V 和一个弧集 R 构成的数据结构。

ADT Graph {

数据对象 V：V 是具有相同特性的数据元素的集合，称为顶点集。

数据关系 R：

R＝{VR}

VR＝{<v,w>| v,w∈V 且 P(v,w)，<v,w>表示从 v 到 w 的弧，

谓词 P(v,w)定义了弧<v,w>的意义或信息 }

1.2 图的重要术语

1.2.1 无向图

无向图:在一个图中，如果任意两个顶点构成的偶对（v,w）∈E 是无序的，即顶点之间的连线是没有方向的，则称该图为无向图。

1.2.2 有向图

有向图：在一个图中，如果任意两个顶点构成的偶对（v,w）∈E 是有序的，即顶点之间的连线是有方向的，则称该图为有向图。

1.2.3 无向完全图

无向完全图：在一个无向图中，如果任意两顶点都有一条直接边相连接，则称该图为无向完全图。在一个含有 n 个顶点的无向完全图中，有 n(n-1)/2 条边。

1.2.4 有向完全图

有向完全图：在一个有向图中，如果任意两顶点之间都有方向互为相反的两条弧相连接，则称该图为有向完全图。在一个含有 n 个顶点的有向完全图中，有 n(n-1)条边。

1.2.5 稠密图、稀疏图

稠密图、稀疏图：若一个图接近完全图，称为稠密图；称边数很少（e<nlogn）的图为稀疏图。

1.2.6 顶点的度、入度、出度

顶点的度（degree）是指依附于某顶点 v 的边数，通常记为 TD (v)。

在有向图中，要区别顶点的入度与出度的概念。顶点 v 的入度是指以顶点为终点的弧的数目，记为 ID (v)；顶点 v 出度是指以顶点 v 为始点的弧的数目，记为 OD (v)。

TD (v)=ID (v)＋OD (v)。

可以证明，对于具有 n 个顶点、e 条边的图，顶点 v_i 的度 TD (v_i)与顶点的个数以及边的数目满足关系：

$$e=\left(\sum_{i=1}^{n} TD(v_i)\right)/2$$

因为无论有向图或者无向图，度都计算了两次

1.2.7 边的权、网图

边的权、网图：与边有关的数据信息称为权（weight）。在实际应用中，权值可以有某种含义。边上带权的图称为网图或网络（network）。如果边是有方向的带权图，则就是一个有向网图。

1.2.8 路径、路径长度

路径、路径长度：顶点 v_p 到顶点 v_q 之间的路径（path）是指顶点序列 $v_p,v_{i1},v_{i2},\ldots,v_{im},v_q$。其中，($v_p,v_{i1}$)，($v_{i1},v_{i2}$)，…,($v_{im},v_q$)分别为图中的边。

路径上边的数目称为路径长度。

1.2.9 简单路径、简单回路

简单路径、简单回路：序列中顶点不重复出现的路径称为简单路径。除第一个顶点与最后一个顶点之外，其他顶点不重复出现的回路称为简单回路，或者简单环。

1.2.10 子图

子图：对于图 G=（V，E），G'=（V'，E'），若存在 V'是 V 的子集 ，E'是 E 的子集，则称图 G'是 G 的一个子图。

1.2.11 连通图、连通分量

连通图、连通分量：在无向图中，如果从一个顶点 v_i 到另一个顶点 $v_j(i\neq j)$ 有路径，则称顶点 v_i 和 v_i 是连通的。如果图中任意两顶点都是连通的，则称该图是连通图。无向图的极大连通子图称为连通分量。

1.2.12 强连通图、强连通分量

强连通图、强连通分量：对于有向图来说，若图中任意一对顶点 v_i 和 $v_j(i\neq j)$均有从一个顶点 v_i 到另一个顶点 v_j 有路径，也有从 v_j 到 v_i 的路径，则称该有向图是强连通图。有向图的极大强连通子图称为强连通分量

1.2.13 生成树

生成树：所谓连通图 G 的生成树，是 G 的包含其全部 n 个顶点的一个极小连通子图，它必定包含且仅包含 G 的 n-1 条边。在生成树中添加任意一条属于原图中的边必定会产生回路，因为新添加的边使其所依附的两个顶点之间有了第二条路径。若生成树中减少任意一条边，则必然成为非连通的。

1.2.14 生成森林

生成森林：在非连通图中，由每个连通分量都可得到一个极小连通子图，即一棵生成树。这些连通分量的生成树就组成了一个非连通图的生成森林。

二、图的存储

2.1 邻接矩阵

所谓邻接矩阵存储结构，就是用一维数组存储图中顶点的信息，用矩阵表示图中各顶点之间的邻接关系。

假设图 G=（V，E）有 n 个确定的顶点，即 $V=\{v_0,v_1,\ldots,v_{n-1}\}$,则表示 G 中各顶点相邻关系为一个 n×n 的矩阵，矩阵的元素为：

$$A[i][j]=\begin{cases}1 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 E(G)中的边}\\ 0 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 E(G)中的边}\end{cases}$$

若 G 是网图，则邻接矩阵可定义为：

$$A[i][j]=\begin{cases}w_{ij} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 E(G)中的边}\\ 0\text{ 或}\infty & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 E(G)中的边}\end{cases}$$

其中，w_{ij} 表示边(v_i,v_j)或$<v_i,v_j>$上的权值；∞表示一个计算机允许的、大于所有边上权值的数。

从图的邻接矩阵存储方法容易看出，这种表示具有以下特点：

（1）无向图的邻接矩阵一定是一个对称矩阵。因此，在具体存放邻接矩阵时只需存放上（或下）三角矩阵的元素即可。

（2）对于无向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的度 $TD(v_i)$。

（3）对于有向图，邻接矩阵的第 i 行（或第 i 列）非零元素（或非∞元素）的个数正好是第 i 个顶点的出度 $OD(v_i)$（或入度 $ID(v_i)$）。

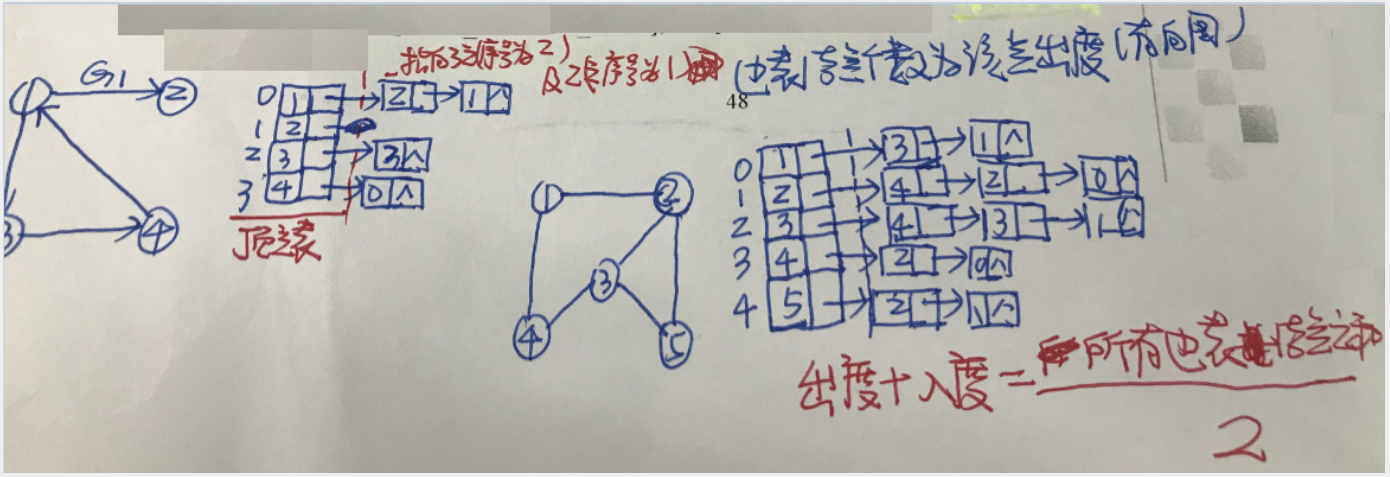
（4）用邻接矩阵方法存储图，很容易确定图中任意两个顶点之间是否有边相连；但是，要确定图中有多少条边，则必须按行、按列对每个元素进行检测，所花费的时间代价很大，这是用邻接矩阵存储图的局限性。

2.2 邻接表

邻接表(Adjacency List)是图的一种顺序存储与链式存储结合的存储方法。邻接表表示法类似于树的孩子链表表示法。就是对于图 G 中的每个顶点 v_i，将所有邻接于 v_i 的顶点 v_i 链成一个单链表，这个单链表就称为顶点 v_i 的邻接表，再将所有点的邻接表表头放到数组中，就构成了图的邻接表。在邻接表表示中有两种结点结构，如图所示。

邻接矩阵表示的结点结构

一种是顶点表的结点结构，它由顶点域（data）和指向第一条邻接边的指针域（firstarc）构成，另一种是边表（即邻接表）结点，它由邻接点域(adjvex)和指向下一条邻接边的指针域(nextarc)构成。对于网图的边表需再增设一个存储边上信息（如权值等）的域（info）。



从图的邻接表存储方法容易看出，这种表示具有以下特点：

（1）若无向图中有 n 个顶点、e 条边，则它的邻接表需 n 个头结点和 2e 个表结点。显然，在边稀疏(e<<n(n-1)/2)的情况下，用邻接表表示图比邻接矩阵节省存储空间，当和边相关的信息较多时更是如此。

（2）在无向图的邻接表中，顶点 v_i 的度恰为第 i 个链表中的结点数.

（3）而在有向图中，第 i 个链表中的结点个数只是顶点 v_i 的出度，为求入度，必须遍历整个邻接表。在所有链表中其邻接点域的值为 i 的结点的个数是顶点 v_i 的入度。

有时，为了便于确定顶点的入度或以顶点 v_i 为头的弧，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个链接以 v_i 为头的弧的链表。

在建立邻接表或逆邻接表时，若输入的顶点信息即为顶点的编号，则建立邻接表的复杂度为 O（n+e），否则，需要通过查找才能得到顶点在图中位置，则时间复杂度为 O（n·e）。

（4）在邻接表上容易找到任一顶点的第一个邻接点和下一个邻接点，但要判定任意两个顶点（v_i 和 v_j）之间是否有边或弧相连，则需搜索第 i 个或第 j 个链表，因此，不及邻接矩阵方便。

三、图的遍历

3.1 图的深度优先(DFS)遍历

深度优先搜索（Depth_Fisrst Search）遍历类似于树的先根遍历，是树的先根遍历的推广。

假设初始状态是图中所有顶点未曾被访问，则深度优先搜索可从图中某个顶点发 v 出发，访问此顶点，然后依次从 v 的未被访问的邻接点出发深度优先遍历图，直至图中所有和 v 有路径相通的顶点都被访问到；若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。

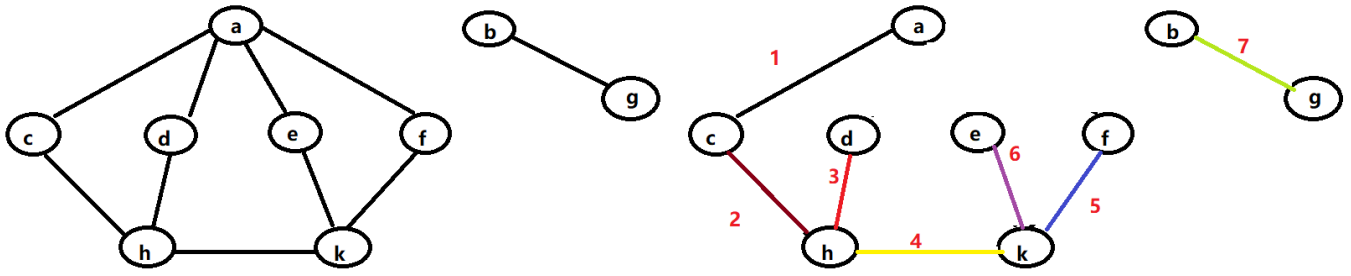
第一个邻节点找到，访问，再找到第一个邻节点的邻节点，访问，如果第一个领节点的第一个领节点被访问过了，就找第一个邻节点的第二个领节点，以此类推。这里的领节点就相当于二叉树的左子树，前序遍历不遍历完所有左子树不会去遍历右子树

显然，这是一个递归的过程。为了在遍历过程中便于区分顶点是否已被访问，需附设访问标志数组 visited[0:n-1]，，其初值为 FALSE ，一旦某个顶点被访问，则其相应的分量置为 TRUE。

从图的某一点 v 出发，递归地进行深度优先遍历的过程算法如下。

```
void DFSTraverse (Graph G) {          //深度优先遍历图 G
    for (v=0; v<G.vexnum; ++v)
        visited[v] = FALSE;           //访问标志数组初始化
    for (v=0; v<G.vexnum; ++v)
        if (!visited[v]) DFS(G,v);    //对尚未访问的顶点调用 DFS
}
void DFS(Graph G,int v ) {            //从第 v 个顶点出发递归地深度优先遍历图 G
    visited[v]=TRUE;Visit(v);         //访问第 v 个顶点
    for (w=FisrtAdjVex(G,v);w>=0; w=NextAdjVex(G,v,w))
        if (!visited[w]) DFS(G,w);    //对 v 的尚未访问的邻接顶点 w 递归调用 DFS
}
```

也就是一直找第一个邻节点的第一个邻节点的第一个邻节点



3.2 图的深度优先遍历的java代码实现

3.2.1 图的建立

```java
public class Graph {
    private int vertexSize;//顶点数量
    private int [] vertexs;//顶点数组
    private int[][]  matrix; //邻接矩阵
    private static final int MAX_WEIGHT = 1000;//用来表示没有连接
    private boolean [] isVisited; //访问的标志
    public Graph(int vertextSize){
        this.vertexSize = vertextSize;//邻接矩阵初始化
        matrix = new int[vertextSize][vertextSize];
        vertexs = new int[vertextSize];//顶点数组初始化
        for(int i = 0;i<vertextSize;i++){
            vertexs[i] = i;
        }                    //是否被访问的数组初始化
        isVisited = new boolean[vertextSize];
    }
```

3.2.2 找到第一个邻节点的代码

```java
    /*
     * *
     * 获取某个顶点的第一个邻接点
     */
    public int getFirstNeighbor(int index){
        for(int j = 0;j<vertexSize;j++){
            //因为是顺序遍历的，所以找到的“合法”的节点必定是第一个节点
            if(matrix[index][j]>0&&matrix[index][j]<MAX_WEIGHT){
                return j;
            }
        }
        return -1;
    }
```

3.2.3 找到下一个邻节点的代码

```java
    /*
     * *
     * 根据前一个邻接点的下标来取得下一个邻接点
     * @param v1表示要找的顶点
     * @param v2 表示该顶点相对于哪个邻接点去获取下一个邻接点
     */
    public int getNextNeighbor(int v,int index){
        for(int j = index+1;j<vertexSize;j++){
            if(matrix[v][j]>0&&matrix[v][j]<MAX_WEIGHT){
                return j;
            }
        }  同样的代码而已，只不过从该标号往下查找下一个邻节点
        return -1;
    }
```

3.2.4 单个点的深度优先代码实现

```
75      /*
76       * *
77       * 图的深度优先遍历算法
78       */
79      private void depthFirstSearch(int i){
80          isVisited[i] = true;
81          int w = getFirstNeighbor(i);//
82          while(w!=-1){
83              if(!isVisited[w]){
84                  //需要遍历该顶点
85                  System.out.println("访问到了："+w+"顶点");
86                  depthFirstSearch(w);
87              }
88              w = getNextNeighbor(i, w);//第一个相对于w的邻接点
89          }
90      }
```

3.2.5 整个图的深度优先代码实现

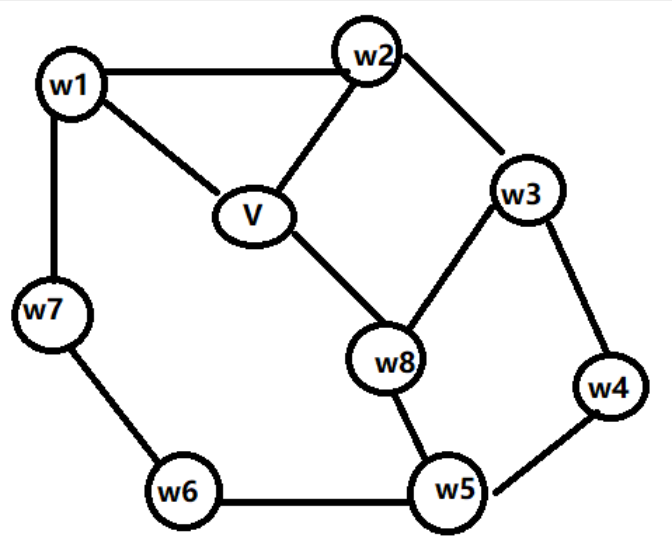
```
94       * 对外公开的深度优先遍历
95       */
96
97      public void depthFirstSearch(){
98          isVisited = new boolean[vertexSize];
99          for(int i = 0;i<vertexSize;i++){
100             if(!isVisited[i]){
101                 System.out.println("访问到了："+i+"顶点");
102                 depthFirstSearch(i);
103             }
104         }
105         isVisited = new boolean[vertexSize];
106     }
```

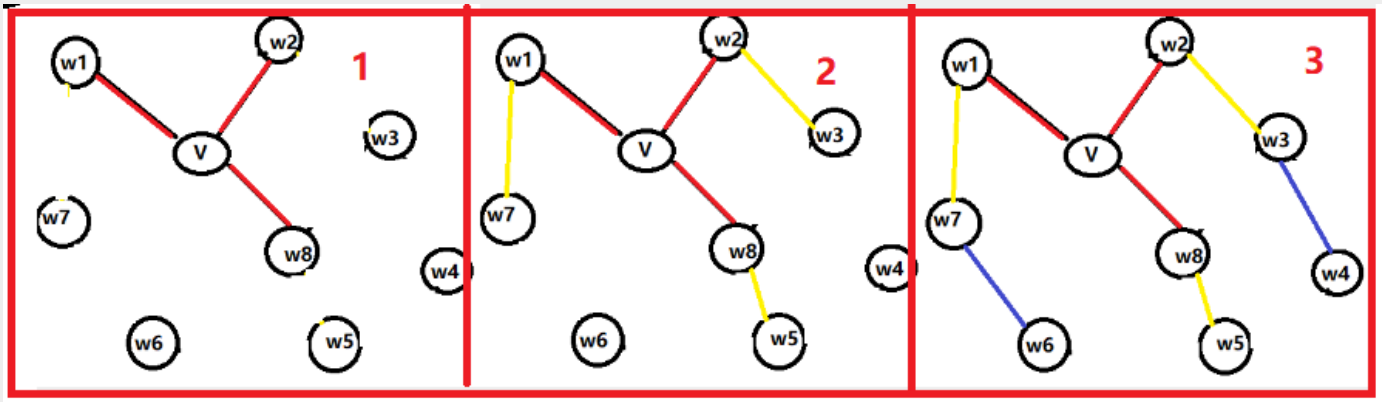
3.3 图的广度优先(BFS)遍历

广度优先搜索（Breadth_First Search） 遍历类似于树的按层次遍历的过程。

假设从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各个未曾访问过和邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。换句话说，广度优先搜索遍历图的过程中以 v 为起始点，由近至远，依次访问和 v 有路径相通且路径长度为 1,2,…的顶点。

广度优先搜索和深度优先搜索类似，在遍历的过程中也需要一个访问标志数组。并且，为了顺次访问路径长度为 2、3、…的顶点，需附设队列以存储已被访问的路径长度为 1、2、…的顶点。





3.3.1 单个点的广度优先遍历代码

3.3.2 整张图的广度优先遍历代码