数 据 结 构

第八章 图 [2]　图的遍历和应用

一生二
二生三
三生万物

摘自《道德经》老子 著

任课老师：郭艳

数据结构课程组

计算机学院

中国地质大学（武汉）2020年秋

一　　　　　数据结构
数据元素的表示、存储
数据元素关系的表示、存储
数据的逻辑操作、实现

二　　　　　数据元素的表示、数据元素关系的表示、数据的逻辑操作
线性表　一
树　　　二
图　　　三

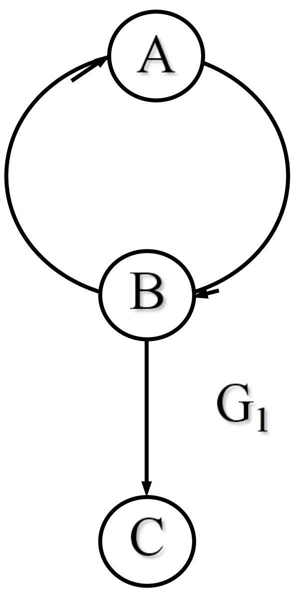
二　　　　　三
数据元素的存储、数据元素关系的存储、数据的操作实现
顺序表　　　有序
链表　　　　单链表　　　双向链表　　　循环单链表　　　循环双向链表
散列表　　　闭散列表　　分离链散列表

二　　　　　三
数据的操作
遍历　　　　构造　　　　输出　　　　查找　　　　找最小（大）　　　　排序

三　　　　　万物
　　　　　　各种数据处理、应用

上堂课要点回顾

- 图的基本概念与抽象数据类型定义
  - 连通图、连通分量、生成树
- 图的设计与实现



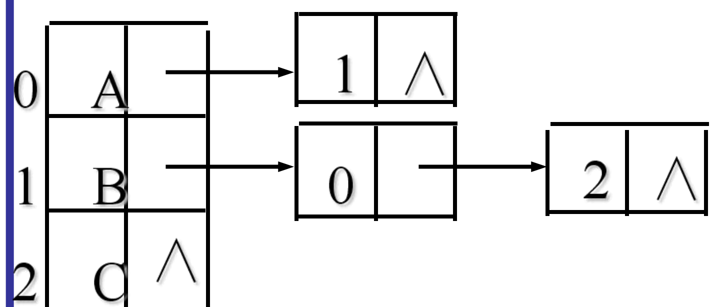
G_1

- 邻接矩阵

$$V_1 = [A, B, C]$$

$$A_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

- 邻接表



第十七次课

阅读：

般人昆，第364-365，366-368，373-375页

习题：

作业17

8.3 图的遍历

- 什么叫图的遍历?

已知图G(V,E)，从图中的任一顶点出发，按一定规则顺着某些边去访问图中其余顶点，使每一个顶点被访问一次且仅被访问一次。

遍历的方法:

深度优先搜索

前序

后序

广度优先搜索

8.3 图的遍历（续）

- 图的遍历从一个顶点v出发，试探性地访问其余顶点，必须考虑到下列情况：
  - 有可能会陷入死循环，如存在回路的图
  - 从一顶点出发，可能不能到达所有其它的顶点（只能到达v所在连通分量的所有顶点），例如非连通图

- 解决办法
  - 为每个顶点设置一个访问标志位（visit bit）。算法开始时，所有顶点的访问标志位置零；在遍历的过程中，当某个顶点被访问时，其标志位就被标记为已访问。
  - 检查图的所有顶点是否被访问过，如果未被访问，则从该未被访问的顶点出发开始继续遍历。

8.3 图的遍历（续）

- 图的生成树
  - 定义：G的所有顶点加上遍历过程中经过的边所构成的子图称作图G的生成树G’
  - 特征
    - 生成树G’是G的极小连通子图
    - 生成树G’含有图G中全部n个顶点，但只有n-1条边
    - 生成树G’没有回路

8.3 图的遍历（续）

- 图的生成森林
  - 若一个图G是非连通图或非强连通图，通过遍历可以得到图G的生成森林G’。
  - 特征
    - 若G有 n 个顶点，m 个连通分量或强连通分量，则可以遍历得到m棵生成树，合起来为生成森林G’，森林G’中包含n-m条树边。

- **深度优先搜索 (depth-first search, DFS)步骤**

① 首先访问出发顶点v，再访问一个未被访问过的v的邻居w_1;

② 再从w_1出发，递归地按照深度优先的方式遍历；设访问顶点序列w_1, w_2, w_3,……,

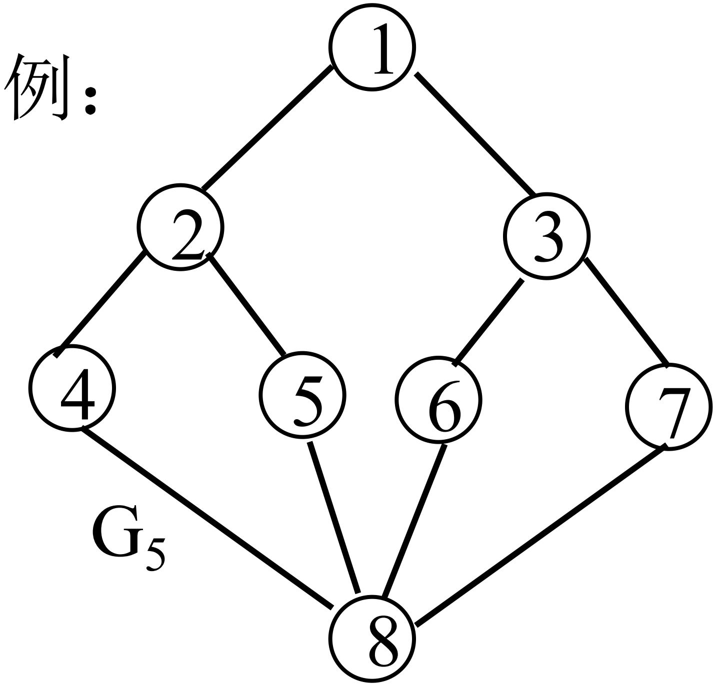
③ 当遇到一个所有邻居均被访问过的顶点w_t时则沿刚才访问的次序，反向回到已访问顶点序列中最后一个尚有邻居未被访问过的顶点w_s;

④ 再从w_s出发，递归地按照深度优先方式遍历；

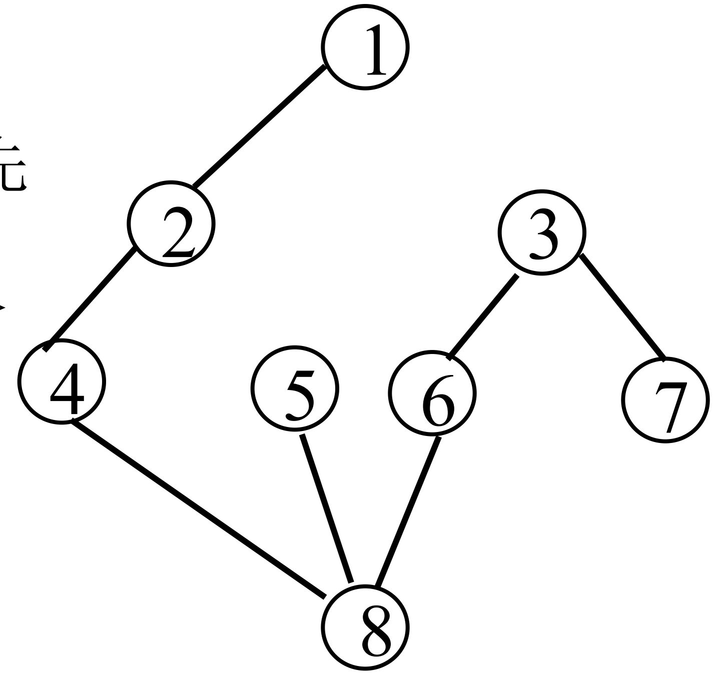
⑤ 当所有被访问过的顶点都没有未被访问的邻居时，出发顶点v所在连通分量的遍历结束。

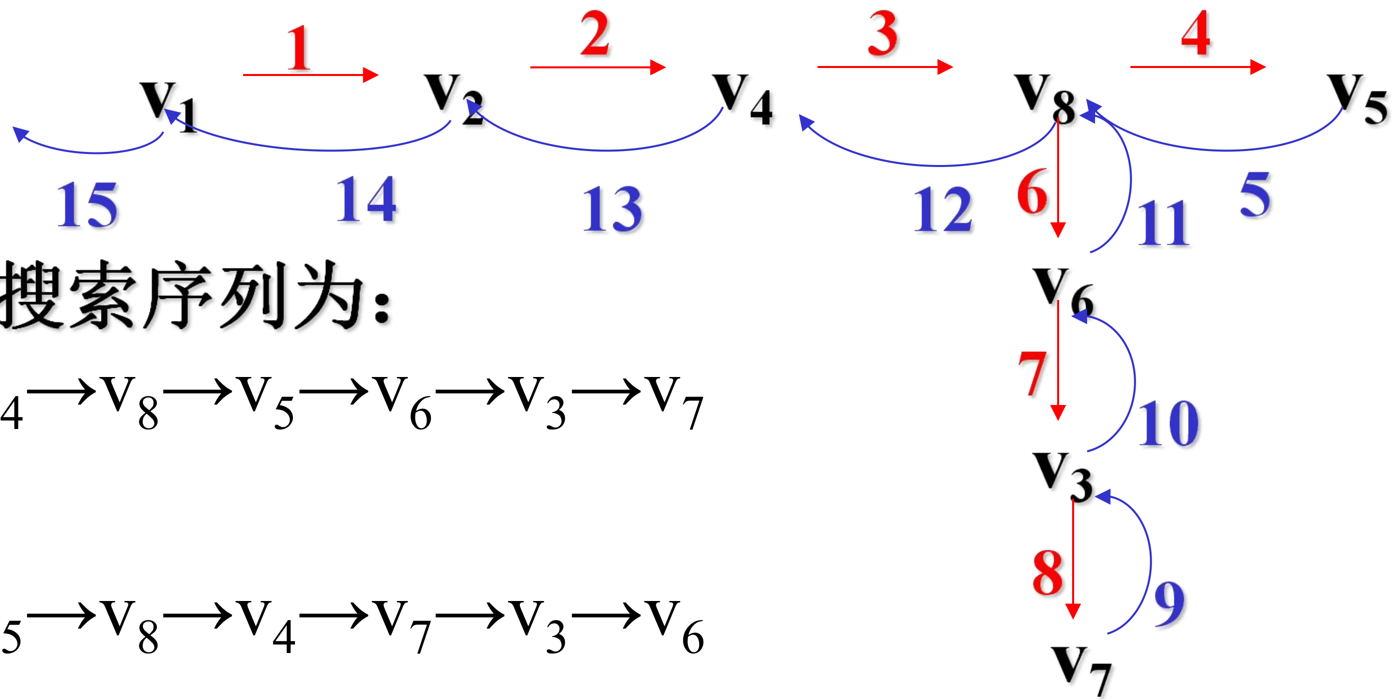
- **深度优先搜索树（depth-first search tree）**

例：



求G_5的深度优先生成树





深度优先搜索序列为：

$v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_8 \rightarrow v_5 \rightarrow v_6 \rightarrow v_3 \rightarrow v_7$

或

$v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_8 \rightarrow v_4 \rightarrow v_7 \rightarrow v_3 \rightarrow v_6$

图的深度优先搜索算法

```
template<class T, class E>
void DFS (Graph<T, E>& G, const T& v) {
//从顶点v出发对图G进行深度优先遍历的主过程
    int i, loc, n = G.NumberOfVertices();    //顶点个数
    bool *visited = new bool[n];             //创建辅助数组
    for (i = 0; i < n; i++) visited [i] = false; //辅助数组visited初始化
    loc = G.getVertexPos(v);
    for (i=loc; i<=(i+n-1)%n; i=(i+1)%n) //增加处理非连通图
    {    if (!visited [i])        //增加处理非连通图
             DFS (G, i, visited); //从第loc顶点开始深度优先搜索
    }
    delete [] visited;                       //释放visited
};//思考1：DFS (G, i, visited)的调用次数是（）？
```

图的深度优先搜索(前序)递归算法

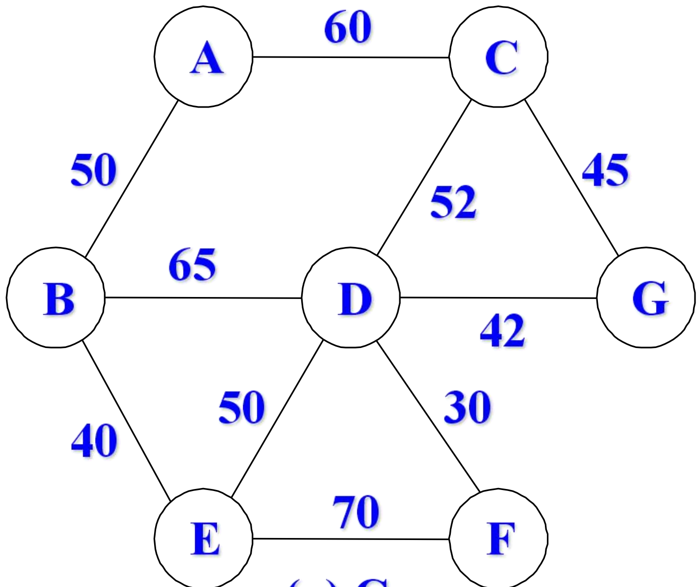
```
template<class T, class E>void
DFS(Graph<T, E>& G,int v,bool visited[])
{ cout << G.getValue(v) << ' '; //访问顶点v
   visited[v] = true;             //作访问标记
   int w = G.getFirstNeighbor (v);//第一个邻居w
   while (w != -1)        //若邻居w存在
   {  if ( !visited[w] )//若w未访问过, 从w开始递归深遍
           DFS(G, w, visited);
      w = G.getNextNeighbor (v, w); //取v的下一邻居
   }
};
```

思考2：图深度优先前序递归遍历算法与树深度优先前序递归遍历算法的异同？

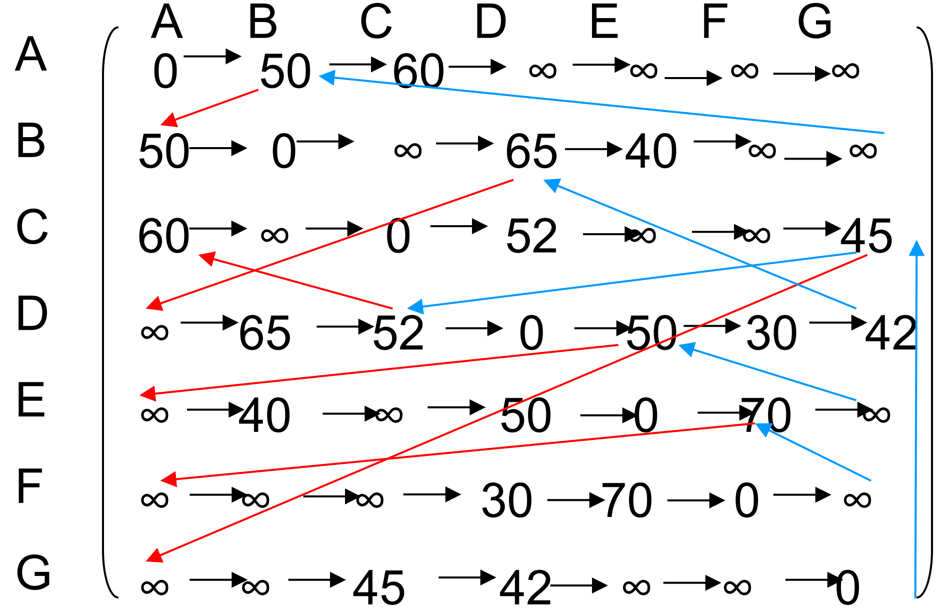
	A	B	C	D	E	F	G
A	0	50	60	∞	∞	∞	∞
B	50	0	∞	65	40	∞	∞
C	60	∞	0	52	∞	∞	45
D	∞	65	52	0	50	30	42
E	∞	40	∞	50	0	70	∞
F	∞	∞	∞	30	70	0	∞
G	∞	∞	45	42	∞	∞	0

思考3：图深度优先搜索前序、后序递归遍历算法的异同？

思考4：DFS(G, w, visited)的递归调用次数?

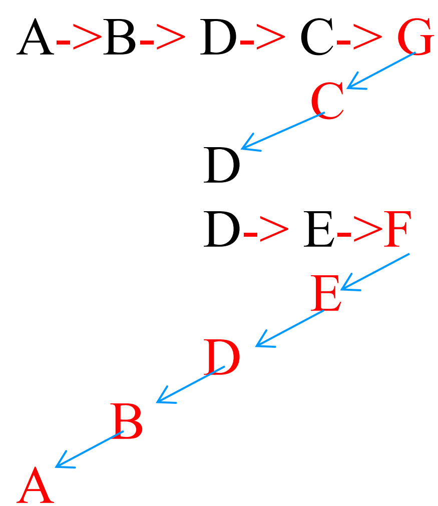


(a) G



(c) 邻接表深度优先搜索过程

```
template<class T, class E>
void DFS (Graph<T, E>& G, int v, bool visited[]) {
   cout << G.getValue(v) << ' ';          //访问v顶点
   visited[v] = true;                     //作访问标记
   int w = G.getFirstNeighbor (v);        //第一个邻接顶点
    while (w != −1) {                     //若邻接顶点w存在
      if ( !visited[w] ) DFS(G, w, visited);
      //若w未访问过, 递归调用：从顶点w开始深遍
     w = G.getNextNeighbor (v, w); //v顶点的下一个邻接顶点
   }
};
```



前序访问序列：A B D C G E F

后序访问序列：G C F E D B A

(d) 邻接矩阵扫描过程

深度优先搜索递归算法分析

时间开销=访问n个顶点+得到所有顶点的邻居+访问检查

访问n个顶点=n次

边的判断<2e次(e<<n^2)

得到一个顶点的邻居n次，w取值<n次。得到所有顶点的邻居n^2次，w取值<n^2次

```
template<class T, class E>void DFS(Graph<T, E>& G,int v,bool visited[])
{ cout << G.getValue(v) << ' ';          //访问顶点v(例如输出)
  visited[v] = true;                     //作访问标记
  int w = G.getFirstNeighbor (v);        //第一个邻接顶点w
  while (w != -1)      //若邻接顶点w存在
  { if ( !visited[w] )    /*若w未访问过,递归调用从顶点w开始深遍*/
        DFS(G, w, visited);
    w = G.getNextNeighbor (v, w); //v顶点的下一个邻接点
  }
};
```

	A	B	C	D	E	F	G
A	0	50	60	∞	∞	∞	∞
B	50	0	∞	65	40	∞	∞
C	60	∞	0	52	∞	∞	45
D	∞	65	52	0	50	30	42
E	∞	40	∞	50	0	70	∞
F	∞	∞	∞	30	70	0	∞
G	∞	∞	45	42	∞	∞	0

假设图G采用 邻接矩阵 存储结构

$O(n+2e+n^2)=O(n^2)$

深度优先搜索递归算法分析

时间开销=访问n个顶点+得到所有顶点的邻居+访问检查

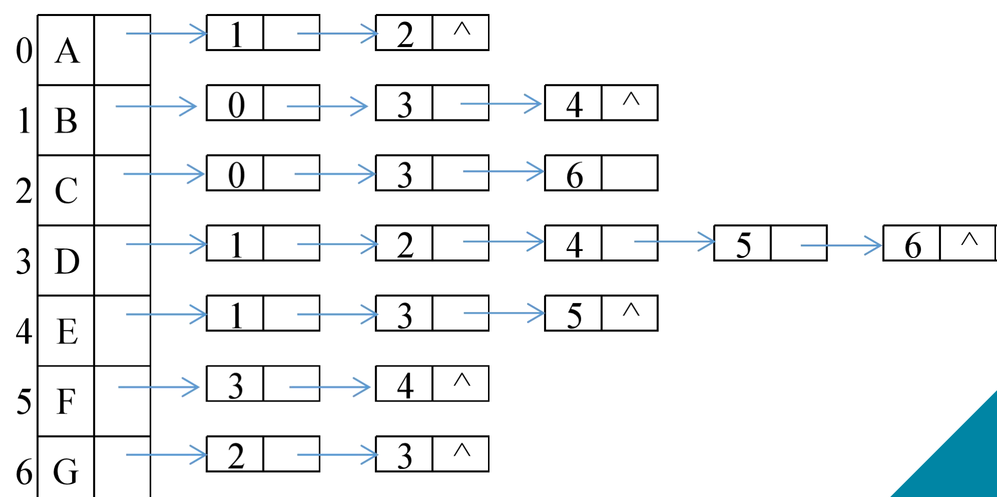
访问n个顶点=n次

```
template<class T, class E>void DFS(Graph<T, E>& G,int v,bool visited[])
{ cout << G.getValue(v) << ' ';        //访问顶点v(例如输出)
  visited[v] = true;                   //作访问标记
  int w = G.getFirstNeighbor (v);      //第一个邻接顶点w
  while (w != -1)     //若邻接顶点w存在
   { if ( !visited[w] )   /*若w未访问过,递归调用从顶点w开始深遍*/
         DFS(G, w, visited);
     w = G.getNextNeighbor (v, w); //v顶点的下一个邻接点
   }
};
```

边的判断<2e次

得到一个顶点的邻居d次，w取值<d次。得到所有顶点的邻居2e次，w取值<2e次

假设图G采用 邻接表 存储结构



=O(n+2e+2e)=O(n+e)

◆ 深度优先搜索算法分析

设图G有n个顶点、e条边。

DFS对每一条逻辑边处理两次，每个顶点访问一次。

以邻接矩阵作存储结构：处理所有的边需$O(n^2)$的时间，故总代价为$O(n+2e+n^2)=O(n^2)$。

以邻接表作存储结构：由于对邻接表中的每个边结点仅检测一次，而边结点共有2e个，所以处理所有边的时间可记为O(e)，故总代价为O(n+2e+2e)=O(n+e)。

空间开销：O(n)，使用visited[n]数组。

图存储结构	解决方案	时间效率	DFS空间效率
邻接表	DFS	0(n+e)	0(n)
邻接矩阵	DFS	$0(n^2)$	0(n)

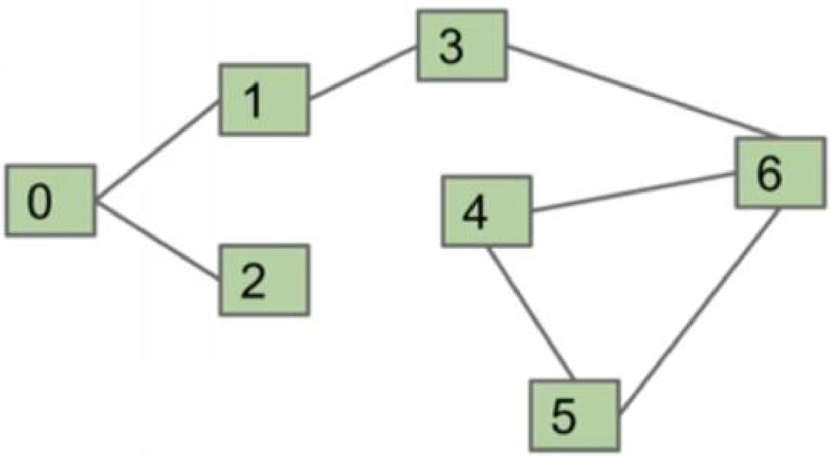
问题：s-t路径

- 问题描述
  - 求一个顶点s到另一个顶点t的路径
- 解决方案
  - 从顶点s开始进行DFS，直至v==t
- 辅助变量
  - path[n]：存放路径
- 时间效率是O(n+e)（邻接表）
  - 代价开销：O(n)次访问顶点+O(e)次visited(w)判断+O(e)次获得所有顶点的邻居
- 额外空间开销是O(n)
  - 需要n个长度的数组存放路径信息

问题	图存储结构	解决方案	时间效率	空间效率
s-t路径	邻接表	DFS	O(n+e)	O(n)

问题：判断一个图是否有环

- 解决方案
  - DFS
- 具体方法
  - 从出发顶点开始进行DFS，直至visited[i]==true
- 潜在的危险
  - 例如：1的邻接顶点0的visited[i]==true
  - 解决风险方法：不计父结点
- 时间开销
  - 最坏O(n+e)（邻接表）



问题	图存储结构	解决方案	时间效率	空间效率
判断一个图是否有环	邻接表	DFS	O(n+e)	O(n)

■ 图的广度优先搜索

- breadth-first search, **BFS**

◆ 步骤

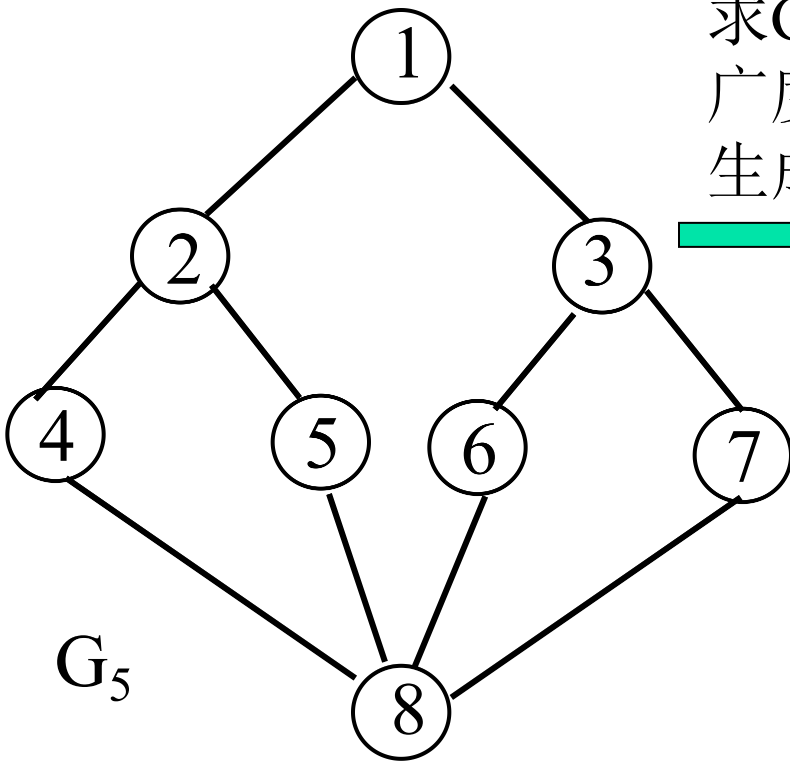
① 访问起始顶点v后，依次访问与v相邻接的所有顶点$w_1, w_2, \ldots, w_t$；

② 再按$w_1, w_2, \ldots, w_t$的顺序，访问其中每一个顶点的所有未被访问过的邻接顶点；对w_1为：$w_{11}, w_{12}, \ldots, w_{1m1}$；…；对$w_t$为：$w_{t1}, w_{t2}, \ldots, w_{tmt}$等；

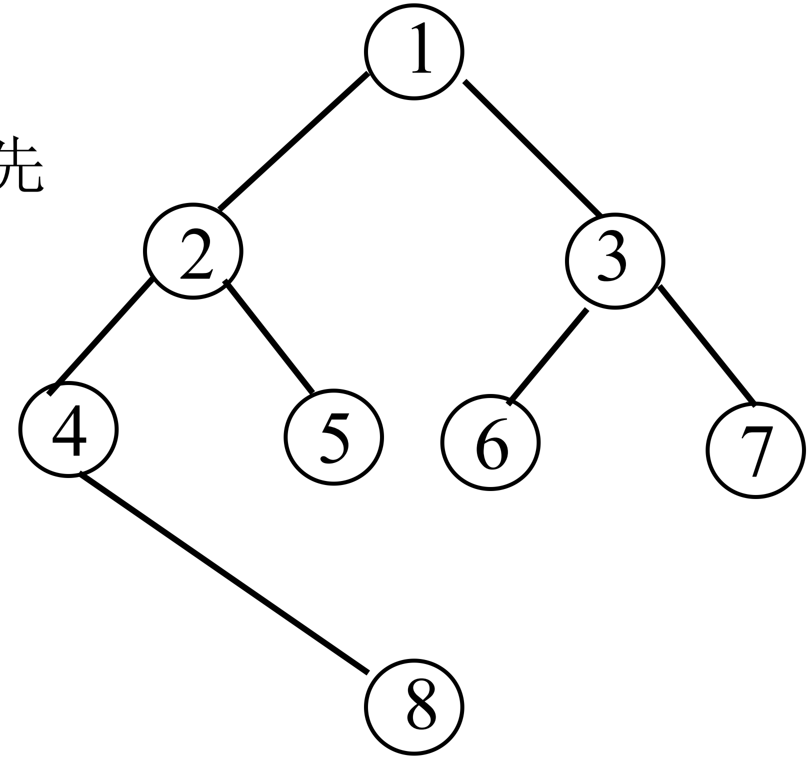
③ 再按$w_{11}, w_{12}, \ldots, w_{1m1}, w_{21}, \ldots, w_{2m2}, \ldots, w_{t1}, \ldots, w_{tmt}$的顺序，去访问它们各自的未被访问过的邻接顶点。依次类推，直到v所在连通分量中所有被访问过的顶点的邻接顶点都被访问过为止。

◆ 广度优先搜索树（depth-first search tree）

例：



求G_5的广度优先生成树



G_5的广度优先生成树

广度优先搜索序列为：

$$V_1 \rightarrow V_2 \rightarrow V_3 \rightarrow V_4 \rightarrow V_5 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8$$

或 $$V_1 \rightarrow V_3 \rightarrow V_2 \rightarrow V_6 \rightarrow V_7 \rightarrow V_5 \rightarrow V_4 \rightarrow V_8$$

图的广度优先搜索遍历算法

由于先被访问的结点其邻居将先被访问，因此使用一个队列存放需要做的工作

- (1) 初始化队列
- (2) 将出发顶点指针入队列，访问标记赋值为true
- (3) 当队列不空时，循环

  3.1) 从队列删除顶点v

  3.2) 对刚出队列的v的每一个未访问的邻居w

  3.2.1) 访问w

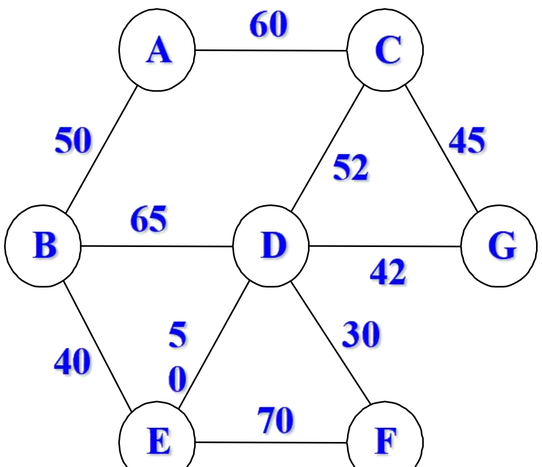
  3.2.2）w的访问标记赋值为true

  3.3.3) w入队列

【图广度优先搜索算法】

```
template <class T, class E>
void BFS (Graph<T, E>& G, const T& v) {
    int i, w, n = G.NumberOfVertices();      //图中顶点个数
    bool *visited = new bool[n];
    for (i = 0; i < n; i++) visited[i] = false;
    int loc = G.getVertexPos (v);              //取顶点号
    SeqQueue<int> Q;
    for (i=loc; i<=(i+n-1)%n; i=(i+1)%n) //增加处理非连通图
    {   if (!visited [i])   //增加处理非连通图
        {    cout << G.getValue (i) << ' '; //访问loc顶点
             visited[i] = true;              //做已访问标记
             Q.EnQueue (i); //顶点进队列, 实现分层访问
```

```
while (!Q.IsEmpty() )   //循环，访问所有结点
{   Q.DeQueue (loc);    //出队列
    w = G.getFirstNeighbor (loc);  //获得loc顶点的第一个邻接点
    while (w != -1)       //若邻居w存在， 双重循环
    {   if (!visited[w])  //若顶点w未访问过
        {   cout << G.getValue (w) << ' ';   //访问顶点w(例如输出)
            visited[w] = true;               //做已访问标记
            Q.EnQueue (w);                   //顶点w进队列
        }
        w = G.getNextNeighbor (loc, w); //找loc顶点的下一个邻接点
    }
}//外层循环end of while (!Q.IsEmpty() )，  每次循环删除一个插入若干个
 }              //end of if  增加
}               //end of for(i)  增加
delete [] visited;
};//思考5：如何计算图连通分量的个数？
//思考6：图广度优先算法与树广度优先算法的异同？
```



(a) G

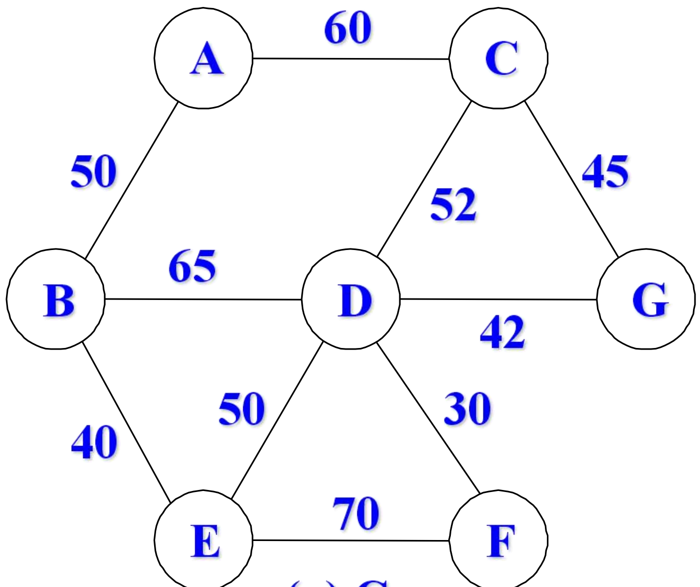
循环次数	visited[]							队列Q	访问顶点
	A	B	C	D	E	F	G		

$$
\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \end{array}
\begin{array}{c}
\begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\
\left(\begin{array}{ccccccc}
0 \rightarrow & 50 \rightarrow & 60 \rightarrow & \infty \rightarrow & \infty \rightarrow & \infty \rightarrow & \infty \\
50 \rightarrow & 0 \rightarrow & \infty \rightarrow & 65 \rightarrow & 40 \rightarrow & \infty \rightarrow & \infty \\
60 \rightarrow & \infty \rightarrow & 0 \rightarrow & 52 \rightarrow & \infty \rightarrow & \infty \rightarrow & 45 \\
\infty \rightarrow & 65 \rightarrow & 52 \rightarrow & 0 \rightarrow & 50 \rightarrow & 30 \rightarrow & 42 \\
\infty \rightarrow & 40 \rightarrow & \infty \rightarrow & 50 \rightarrow & 0 \rightarrow & 70 \rightarrow & \infty \\
\infty \rightarrow & \infty \rightarrow & \infty \rightarrow & 30 \rightarrow & 70 \rightarrow & 0 \rightarrow & \infty \\
\infty \rightarrow & \infty \rightarrow & 45 \rightarrow & 42 \rightarrow & \infty \rightarrow & \infty \rightarrow & 0
\end{array}\right)
\end{array}
$$

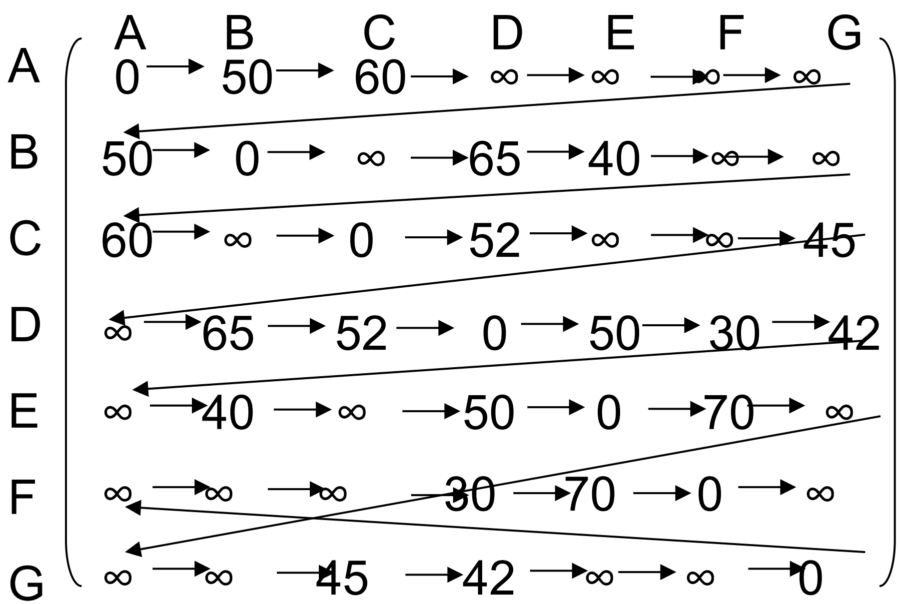
(b) 广度优先搜索遍历过程

A B C D E G F

```
template <class T, class E>
void BFS (Graph<T, E>& G, const T& v
{int i, w, n = G.NumberOfVertices();
  bool *visited = new bool[n];
  for(i=0;i<n;i++)visited[i]= false;
  int loc = G.getVertexPos (v);
  cout << G.getValue (loc) << ' ';
  visited[loc] = true;
  Queue<int> Q; Q.EnQueue (loc);
  while (!Q.IsEmpty() )
  {  Q.DeQueue (loc);
     w = G.getFirstNeighbor (loc);
     while (w != -1) {
       if (!visited[w]) {
          cout<<G.getValue(w)<< ' ';
          visited[w] = true;
          Q.EnQueue (w);
       }
       w=G.getNextNeighbor(loc, w);
     }
  }
  delete [] visited;
};
```

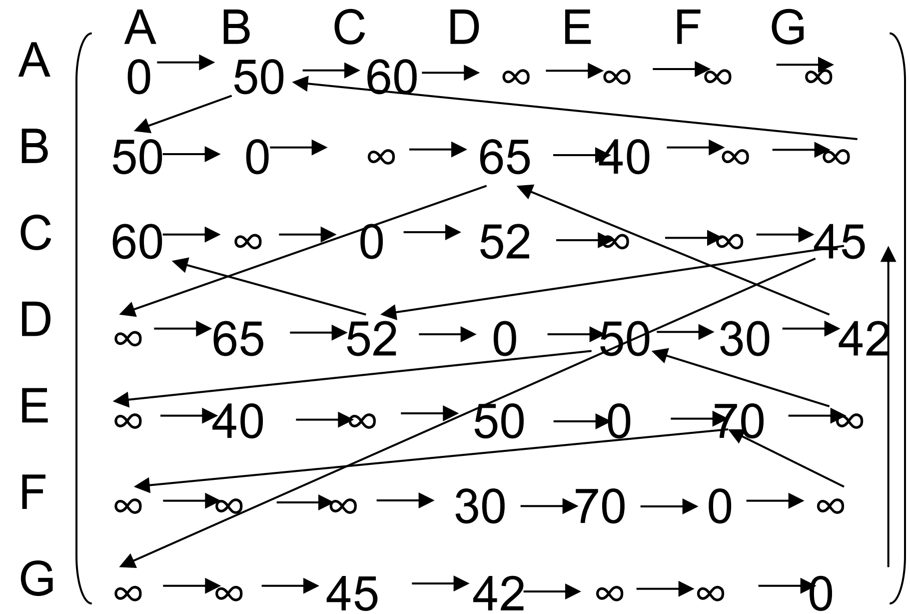


(a) G



(b) 广度优先搜索遍历过程

访问序列：**A B C D E G F**



(c) 深度优先搜索遍历过程

前序访问序列：**A B D C G E F**

后序访问序列：**G C F E D B A**

广度优先搜索遍历图的时间复杂度和深度优先搜索遍历相同，两者不同之处仅在于对顶点访问的顺序不同。

广度优先搜索算法分析

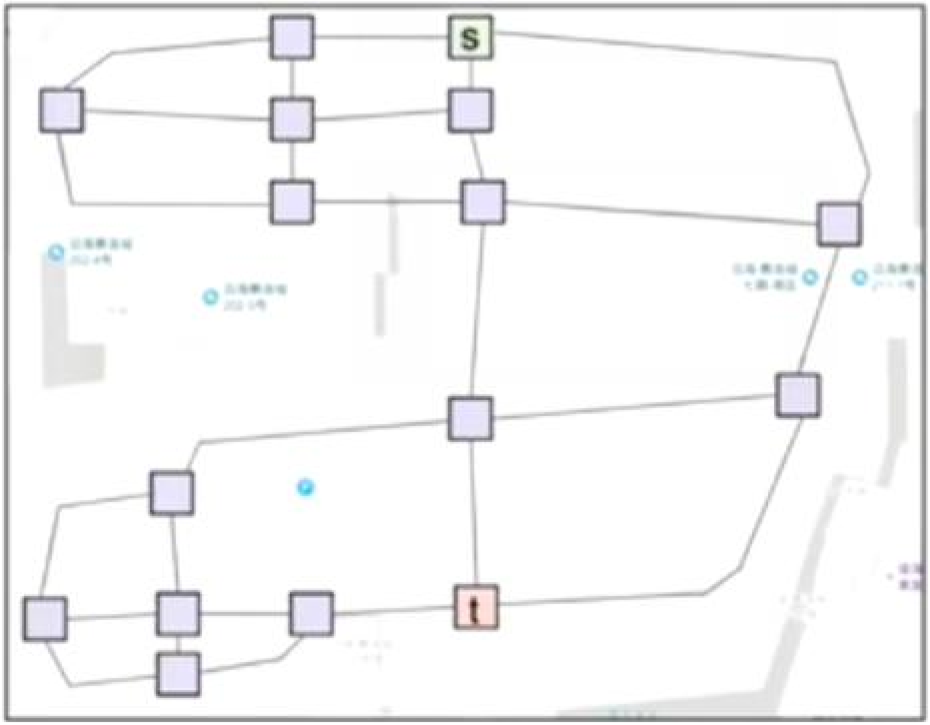
分析上述过程，每个顶点至多进一次队列。遍历图的过程实质上是通过边或弧找邻接点的过程。

设图G有n个顶点、e条边。BFS对每一条边处理一次，每个顶点访问一次。

以邻接矩阵作存储结构：处理所有的边需O(n^2)的时间，故总代价为O(n^2)。

以邻接表作存储结构：由于对邻接表中的每个边结点仅检测一次，而边结点共有2e个，所以处理所有边的时间可记为O(e)，故总代价为O(n+e)。

问题：最短路径（路径长度最小）



问题1：求顶点s到**顶点t**的最短路径（路径长度最小）

- 解决方案：BFS
- 辅助变量：path[n]存放路径
- 时间效率：O(n+e)（邻接表）
- 额外空间开销：O(n)

问题2：求顶点s到**每个顶点**的最短路径（路径长度最小）

提示：需要按BFS序列访问各顶点

- 解决方案：BFS
- 辅助变量：
  - path[n]存放n条路径，dist[n]存放s到各个顶点最短路径的长度
- 时间效率：O(n+e)（邻接表）
- 额外空间开销：O(n)

基于邻接矩阵的图问题

问题	解决方案	时间开销	空间开销
s-t路径	DFS	$O(n^2)$	$O(n)$
s-t最短路径	BFS	$O(n^2)$	$O(n)$

- 如果使用邻接矩阵，BFS和DFS的时间效率都是$O(n^2)$
  - 对于稀疏矩阵(e<<n)，时间开销非常大
  - 因此我们一般使用邻接表存储结构，除非有特殊情况
- 图的存储结构的选择会对DFS、BFS、输出图等其它用户程序产生很大影响
  - 时间
  - 空间

寻径方法BFS和DFS的比较

- 正确性
  - 两种方法对任何图都可以正确工作
- 输出质量
  - BFS一箭双雕，不仅得到路径，而且得到最短路径
- 时间效率
  - 两种方法基本类似，都需要考虑所有边
- 空间效率
  - 对细长的图DFS性能差
    - 调用栈的深度将非常深
    - 调用栈需要O(n)的内存开销
  - 对特别浓密的图BFS性能差
    - 队列很大，最坏情况下O(n)
    - 例如1,000,000的顶点全连接，一次999，999个顶点入队列

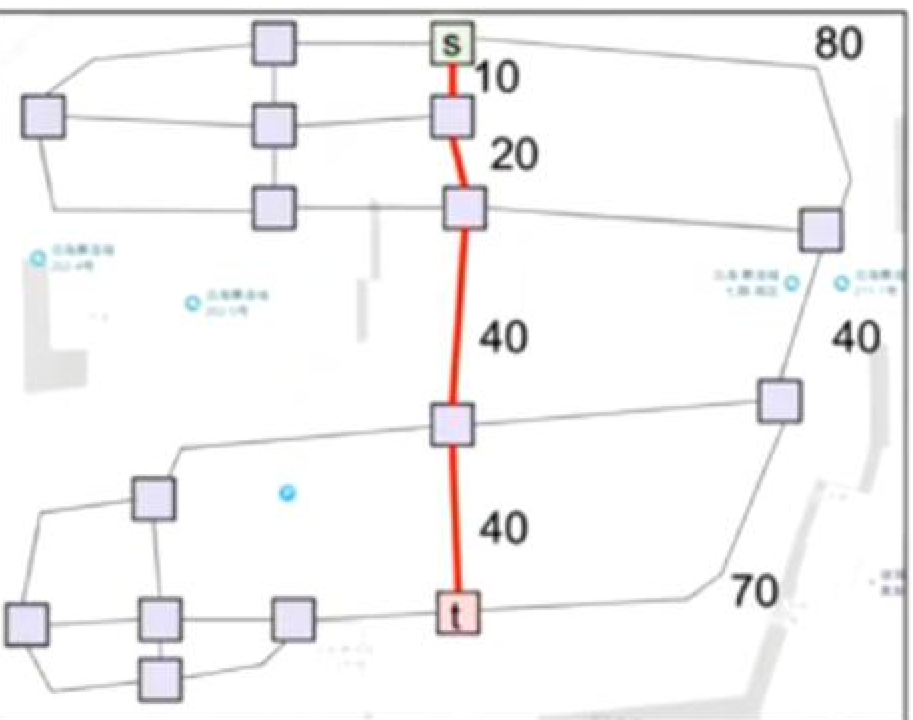
带权图的最短路径

BFS产生错误结果

产生的路径有三条边，距离是190（而不是110）

我们需要考虑边的权值（如距离）的最短路径算法

正确结果



BFS结果

自 学

1、P366-368 8.3.3章节 连通分量 程序8.11

作业17——图的遍历与应用

- **概念题**
  - 1、P392 8.10
  - 2、补充题：对P392图8.32
    - 1）写出该图的邻接表结构（要求邻接顶点按下标从小到大顺序）；
    - 2）分别写出邻接表的扫描过程、递归算法的递归过程、栈或队列的变化情况以及访问序列：a、从顶点1出发深度优先遍历；b、从顶点2出发广度优先遍历。
- **程序设计题**
  - 3、编写Path(v_0)函数，实现输出图中第v_0顶点到其余各顶点的路径。

    要求：采用遍历实现

    提示：要考虑路径的表现形式和存储结构。