第八章 图

- 图的基本概念
- 图的存储结构
- 图的遍历
- 最小生成树
- 最短路径
- 拓扑排序
- 关键路径

8. 1 图的基本概念

- **图定义** 图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构：

$$Graph = (V, E)$$

其中 $V = \{x \mid x \in$ 某个数据对象$\}$

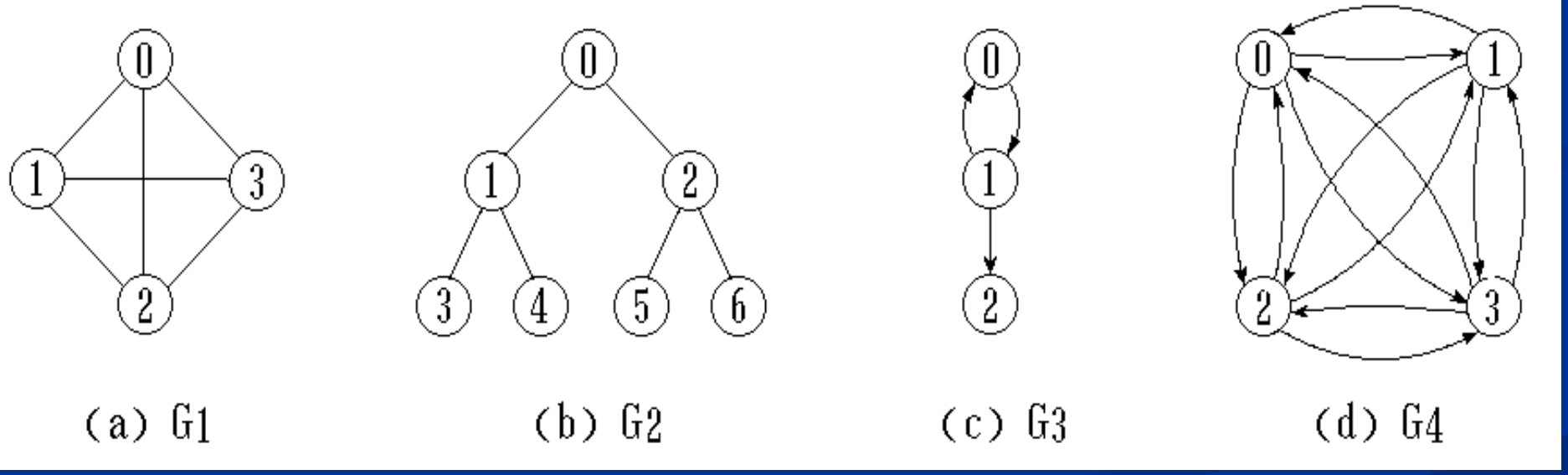
是顶点的有穷非空集合；

$$E = \{(x, y) \mid x, y \in V\}$$

或 $$E = \{<x, y> \mid x, y \in V \ \&\& \ Path(x, y)\}$$

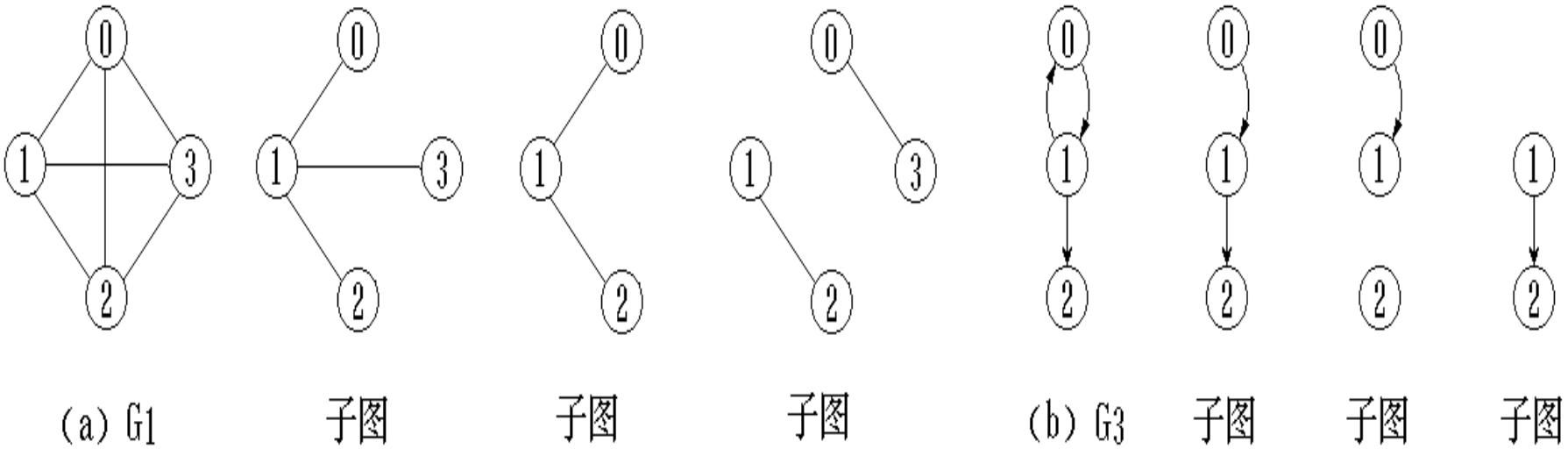
是顶点之间关系的有穷集合，也叫做边(edge)集合。

- **有向图与无向图** 在有向图中，顶点对<x, y>是有序的。在无向图中，顶点对(x, y)是无序的。
- **完全图** 若有 n 个顶点的无向图有 $n(n-1)/2$ 条边，则此图为完全无向图。有 n 个顶点的有向图有 $n(n-1)$ 条边，则此图为完全有向图。



邻接顶点 如果 (u, v) 是 $E(G)$ 中的一条边，则称 u 与 v 互为邻接顶点。

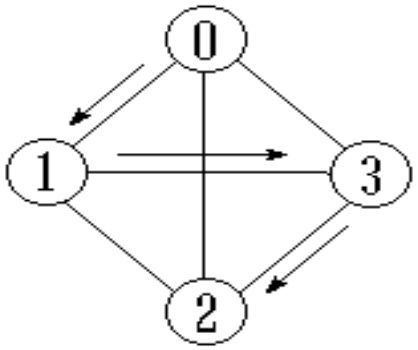
- **权**　某些图的边具有与它相关的数，称之为权。这种带权图叫做**网**。
- **子图**　设有两个图 $G=(V, E)$ 和 $G'=(V', E')$。若 $V'\subseteq V$ 且 $E'\subseteq E$，则称 图G' 是 图G 的子图。



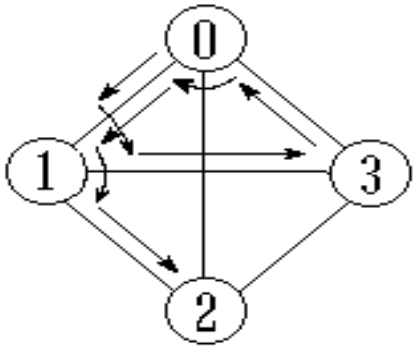
顶点的度　一个顶点v的度是与它相关联的边的条数。记作TD(v)。在有向图中，顶点的度等于该顶点的入度与出度之和。

- **顶点 v 的入度**是以 v 为终点的有向边的条数, 记作 ID(v); **顶点 v 的出度**是以 v 为始点的有向边的条数, 记作 OD(v)。
- **路径**　在图 G = (V, E) 中, 若从顶点 v_i 出发, 沿一些边经过一些顶点 v_{p1}, v_{p2}, …, v_{pm}, 到达顶点v_j。则称顶点序列 (v_i v_{p1} v_{p2} … v_{pm} v_j) 为从顶点v_i 到顶点 v_j 的路径。它经过的边<v_i, v_{p1}>、 <v_{p1}, v_{p2}> 、 … 、 < v_{pm}, v_j >应是属于E的边。
- **路径长度**
  - 非带权图的路径长度是指此路径上边的条数。
  - 带权图的路径长度是指路径上各边的权之和。

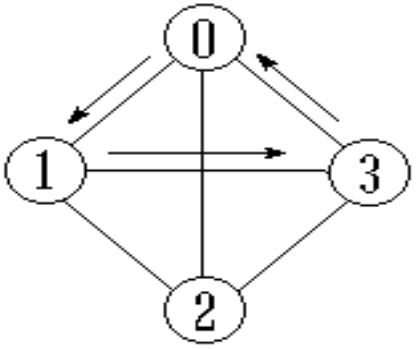
- **简单路径**　若路径上各顶点 $v_1,v_2,\cdots,v_m$ 均不互相重复，则称这样的路径为简单路径。
- **回路**　若路径上第一个顶点 v_1 与最后一个顶点 v_m 重合，则称这样的路径为回路或环。



（a）简单路径



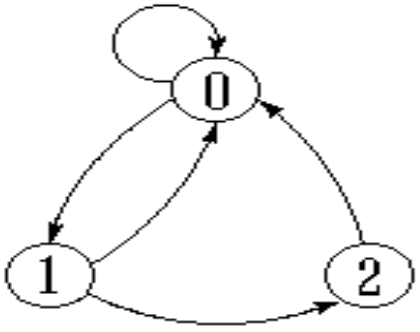
（b）非简单路径



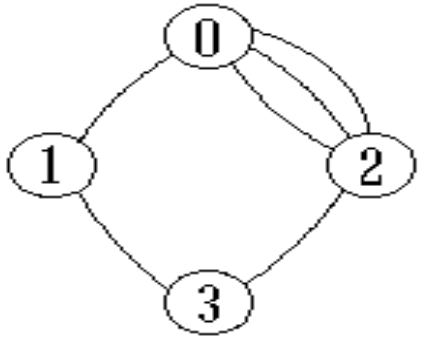
（c）　回路

连通图与连通分量　在无向图中，若从顶点 v_1 到顶点 v_2 有路径，则称顶点 v_1 与 v_2 是连通的。如果图中任意一对顶点都是连通的，则称此图是连通图。非连通图的**极大连通子图**叫做连通分量。

- **强连通图与强连通分量**　在有向图中，若对于每一对顶点v_i和v_j，都存在一条从v_i到v_j和从v_j到v_i的路径，则称此图是强连通图。非强连通图的**极大强连通子图**叫做强连通分量。
- **生成树**　一个连通图的生成树是它的**极小连通子图**，在n个顶点的情形下，有n-1条边。但有向图则可能得到它的由若干有向树组成的生成森林。
- **本章不予讨论的图**：



（a）带自身环的图



（b）多重图

8.2 图的存储结构

一. 图的数组表示法 (邻接矩阵表示法)

基本思想：

- ✓ 图需要存储的信息：顶点和弧。
- ✓ **引入两个数组，一个记录各个顶点信息的顶点表，还有一个表示各个顶点之间关系的邻接矩阵。**

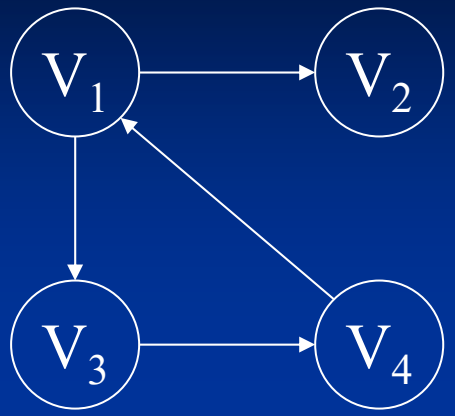
- **设图 A = (*V*, *E*)是一个有 *n* 个顶点的图，则图的邻接矩阵是一个二维数组 *edges*[*n*][*n*]，定义：**

$$edges[i][j] = \begin{cases} 1 & \text{若} <v_i, v_j> \text{或} (v_i, v_j) \in E \\ 0 & \text{反之} \end{cases}$$

下面两个图的邻接矩阵分别为：

edges =

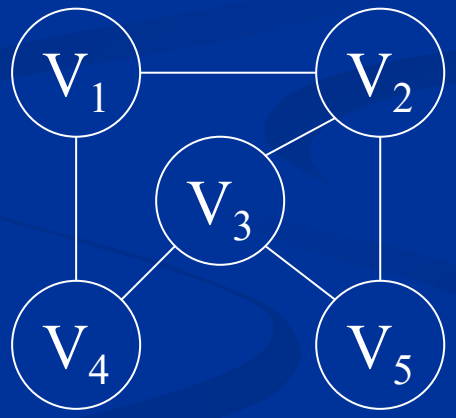
$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$



有向图G1

edges =

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

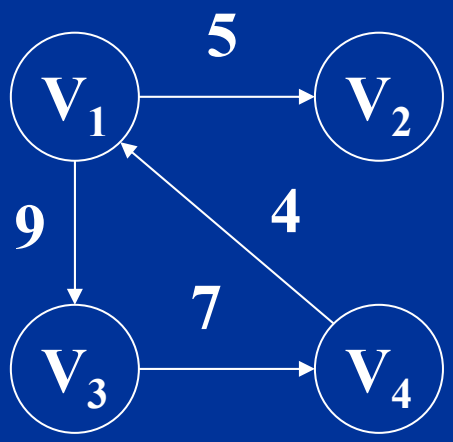


无向图G2

网的邻接矩阵可以定义为：

$$edges[i][j] = \begin{cases} W_{i,j} & \text{若} <v_i, v_j> \text{或} (v_i, v_j) \in E \\ 0\text{或}\infty & \text{反之} \end{cases}$$

有向网G1的邻接矩阵为：



有向图G1

$$edges = \begin{pmatrix} \infty & 5 & 9 & \infty \\ \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 7 \\ 4 & \infty & \infty & \infty \end{pmatrix}$$

邻接矩阵的特点：

- ✓ 无向图的邻接矩阵一定是一个对称矩阵。
- ✓ 无向图的邻接矩阵的第i行(或第i列)非零元素(或非∞元素)个数为第i个顶点的度$D(v_i)$。
- ✓ 有向图的邻接矩阵的第i行非零元素(或非∞元素)个数为第i个顶点的出度$OD(v_i)$，第i列非零元素(或非∞元素)个数就是第i个顶点的入度$ID(v_i)$。

图的数组(邻接矩阵)存储表示：

```
enum graphType{undigraph, digraph,
undinetwork, dinetwork};

template<class T>
struct EdgeType{
    T head, tail;
     int cost;
     EdgeType(T h, T t, int c)
    {
        head=h; tail=t; cost=c;
    }
};
```

```
template<class T>
class MGraph{
    int vexnum, edgenum;   // 图中的顶点数、边数
    graphType  kind;  //图的类型标记
    vector <vector <int>> edges;  //邻接矩阵
    vector <T> vexs;   // 顶点表
    void DFS(int v, bool* visited);  //连通图的深度优先遍历
public:
    MGraph(graphType t, T v[], int n, int e);
    ~MGraph( ){};   //析构函数
    int vertexNum();  //返回图中的顶点数量
    int edgeNum();  //返回图中的边数量
    void DFSTraverse();  //深度优先遍历图
    void BFSTraverse ();  //广度优先遍历图
};
```

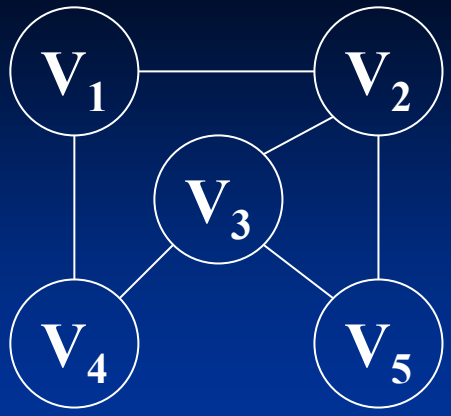
无向网的邻接矩阵建立算法：

```
template<class T>
MGraph<T>::MGraph( graphType t, T v[], int n, int e) {
    vexnum=n;
    edgenum=e;
    kind=t;
    vexs.resize(vexnum);
    edges.resize(vexnum);
    for(i=0;i<vexnum;i++)      //初始化顶点表
        vexs[i]=v[i];
    for(i=0;i<vexnum;i++)  //根据图中的顶点数预置邻接矩阵的大小
        edges[i].resize(vexnum);
    for(i=0;i<n; i++)     //初始化邻接矩阵
        for(j=0;j<n;j++)
            edges[i][j]= INFINITY;
```

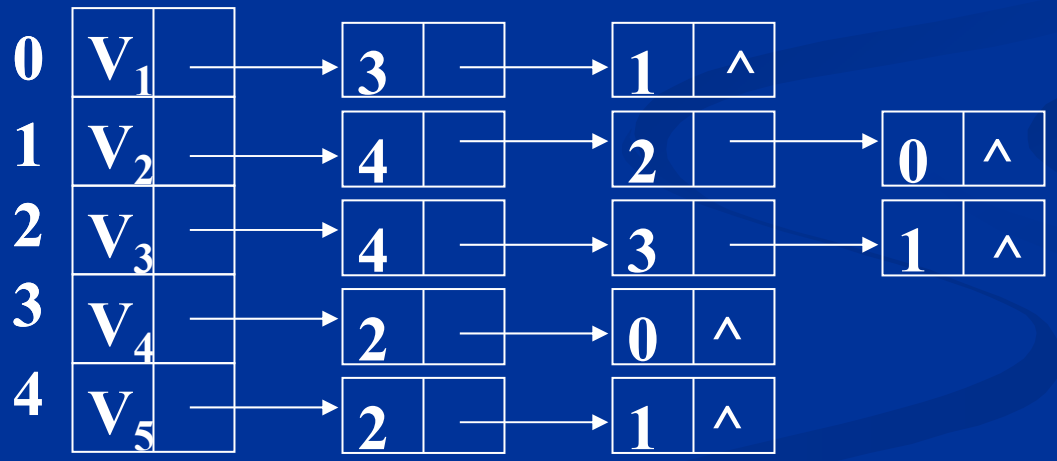
```
        for(i=0;i<e;i++) {        //依次输入所有的边的信息
            cin>>va>>vb;
            cin>>w;
            edges[va][vb]=edges[vb][va]= w;
        }
}
```

二. 邻接表 (Adjacency List)

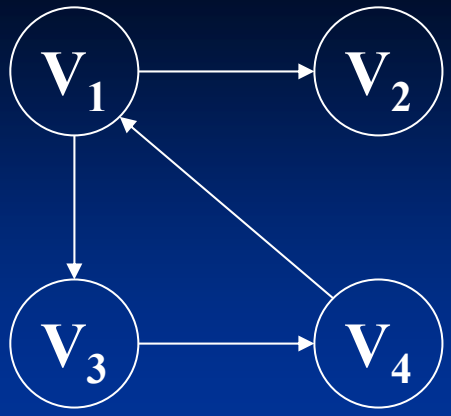
- 基本思想：
  - 如果对图中的每个顶点都建立一个单链表（边表）来存储所有依附于该顶点的弧或边，就可以把图中所有已有的弧或边的信息保存下来。
  - 对于图中所有顶点还是使用一个一维数组来存放。
- 在邻接表表示法中，对于顶点单元i，需要存放的内容有顶点信息以及指向依附于该顶点的所有的边组成的单链表（边表）。
- 对于边表中的每个结点，需要存放该边指向的顶点的位置（也就是该边依附的另一个顶点的位置）和指向下一条边的指针。



无向图G1

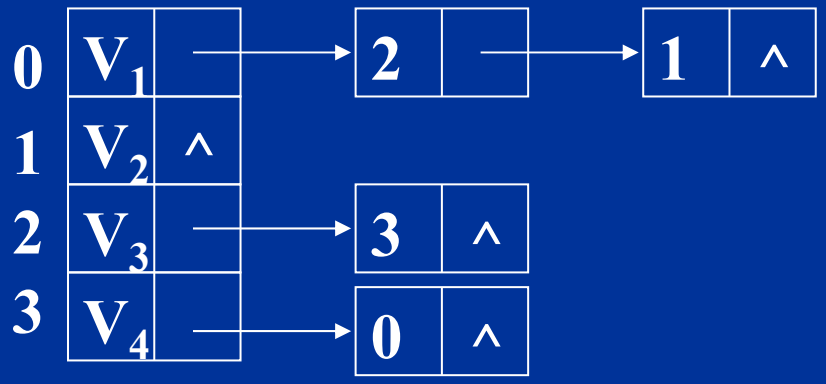


无向图G2的邻接表



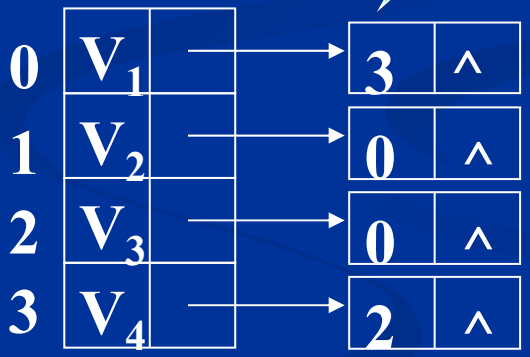
有向图G2

出边表



有向图G2的邻接表

入边表



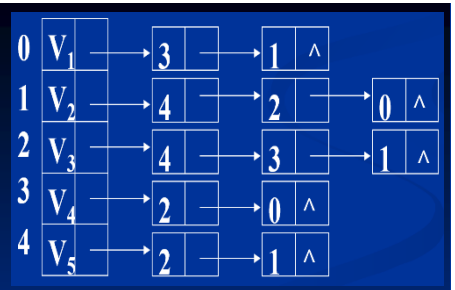
有向图G2的逆邻接表

图的邻接表存储表示：

```
 struct EdgeNode{      // 边表的结点结构类型
        int adjvex;        // 该边的终点位置
        EdgeNode* nextedge;   // 指向下一条边的指针
};

template<class T>
struct VexNode{   //顶点表的元素结构类型
        T data;
        EdgeNode* firstedge;
};
```

```
template<class T>
class ALGraph{
    int vexnum, edgenum;   //图中的顶点数、边数
    vector<VexNode<T>>  adjlist;    //顶点表
    graphType  kind;  //图的类型标记
    void DFS(int v, bool* visited);  //连通图的深度优先遍历
public:
    ALGraph(graphType t, T vexs[], int n, int e);  //构造函数
    ~ALGraph( );   //析构函数
    EdgeNode* firstEdge(int v);  //返回第v个顶点对应的边表的头指针
    void DFSTraverse();  //深度优先遍历图
    void BFSTraverse ();  //广度优先遍历图
     ……
};
```



☞ （无向图）邻接表的生成算法：

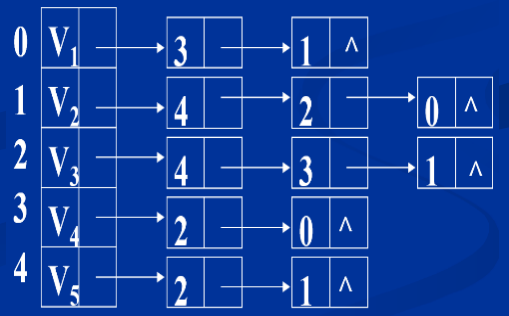
算法思想：

（1）依次读入图的各顶点数据并将其存入到头结点数组中，并将表头结点数组的链域均置“空”；

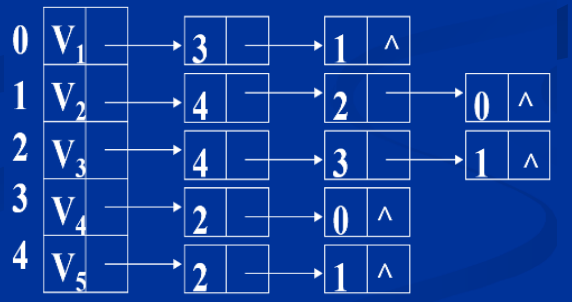
（2）逐个输入表示边的顶点序号（i，j）；约定i>0&&j>0是图的有效顶点对。

- 对于每一个有效的顶点对（i，j），动态生成两个结点，它们的顶点域 分别为j，i，并分别插入到第i个和第j个链表（为简便起见，插入的位置为第一个结点之前）中。

```
template<class T>
ALGraph<T>::ALGraph(GraphType t, T vexs[], int n, int e){
        EdgeNode* p;
        vexnum=n;          //确定图的顶点个数和边数
        edgenum=e;
        kind=t;
        adjlist.resize(vexnum);
        for(i=0;i< vexnum;++i)        //初始化顶点表
        {
                adjlist[i].data= vexs[i];
                adjlist[i].firstedge=0;
        }
```



```
for(j=0;j< edgenum; j++ )    //依次输入所有的边的信息
{
        cin>>va>>vb;      // 输入一条边邻接的两个顶点的序号
        p=new EdgeNode;  //产生第一个表结点
        p->adjvex=vb;
        p->nextedge=adjlist[va].firstedge; // 插在表头
        adjlist[va].firstedge=p;
        p=new EdgeNode;   // 产生第二个表结点
        p->adjvex=va;
        p->nextedge=adjlist[vb].firstedge; // 插在表头
        adjlist[vb].firstedge=p;
}
}
```



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

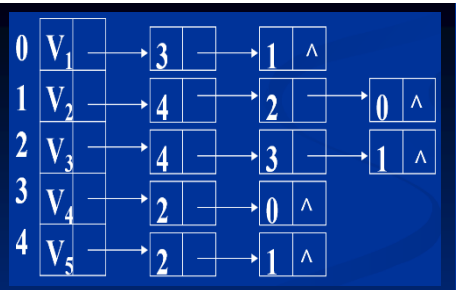
邻接矩阵表示法的优点：

- ✓ 邻接矩阵表示图，很容易确定图中任意两个顶点之间是否有边相连。
- ✓ 容易判断两个顶点之间是否有长度为m的路径相连。

邻接矩阵表示法的缺点：

- 邻接矩阵占用的存储单元个数只与图中的结点个数有关，而与边的数目无关。一个n个结点的图，若其边数比 n^2 少得多，那么邻接矩阵中会存在大量的无用单元。

邻接表的优缺点：



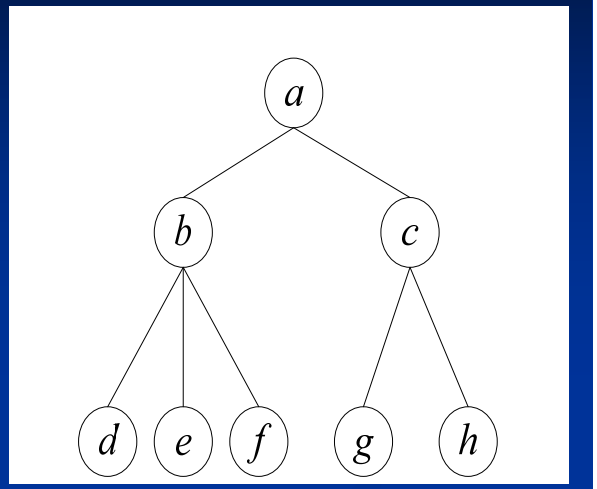
优点：

- ✓ 在边稀疏的情况下，节省存储空间
- ✓ 易于确定图中任一顶点的度数和它的所有邻接点。

缺点：

- ➢ 判定任意两个顶点之间是否有边或弧不方便。

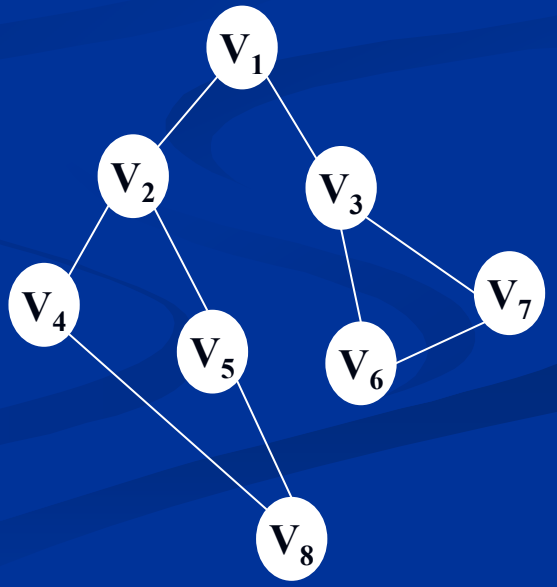
8.4 图的遍历



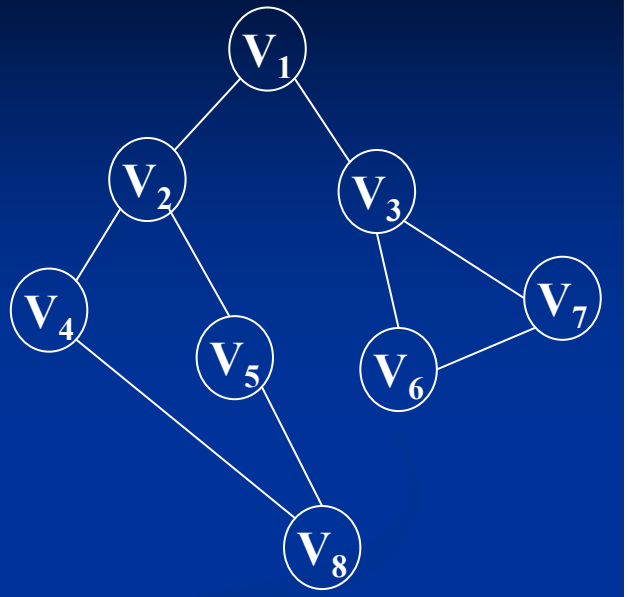
- 图的遍历

  从图中某一顶点出发，沿着某条搜索路径访问图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的遍历。

- 相对于树或二叉树结构，图遍历的复杂性是什么？
  - 可能存在结点被重复访问的问题
    - ✓ 图的任一顶点都可能与其它顶点相通，即图中可能存在回路，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。



8.4 图的遍历

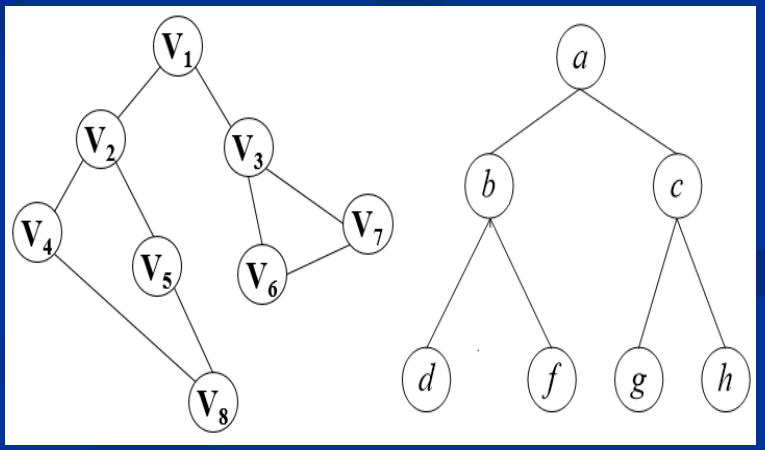


- 相对于树或二叉树结构，图遍历的复杂性是什么？
  - 可能存在顶点被重复访问的问题
  - 如何解决?
    - 为了避免顶点重复访问，需设置一个标志顶点是否被访问过的辅助数组 *visited*[ ]，它的初始状态为 0，
    - 在图的遍历过程中，一旦某一个顶点 *i* 被访问，就立即修改 *visited*[*i*] 为 1，防止它被多次访问。
  - 搜索的规律如何如何组织?

8.3.1 深度优先搜索(Depth First Search)

- DFS的基本过程
  - ✓ 假设从图中某个顶点v出发，首先访问顶点v；
  - ✓ 然后依次从v的各个未曾访问过的邻接点出发执行深度优先遍历，直至图中所有已被访问的顶点的邻接点都被访问到。
  - ✓ 若此时图中还有未被访问的顶点，则任选其中之一作为新的出发点，重新开始上述过程，直至图中所有顶点都被访问到。

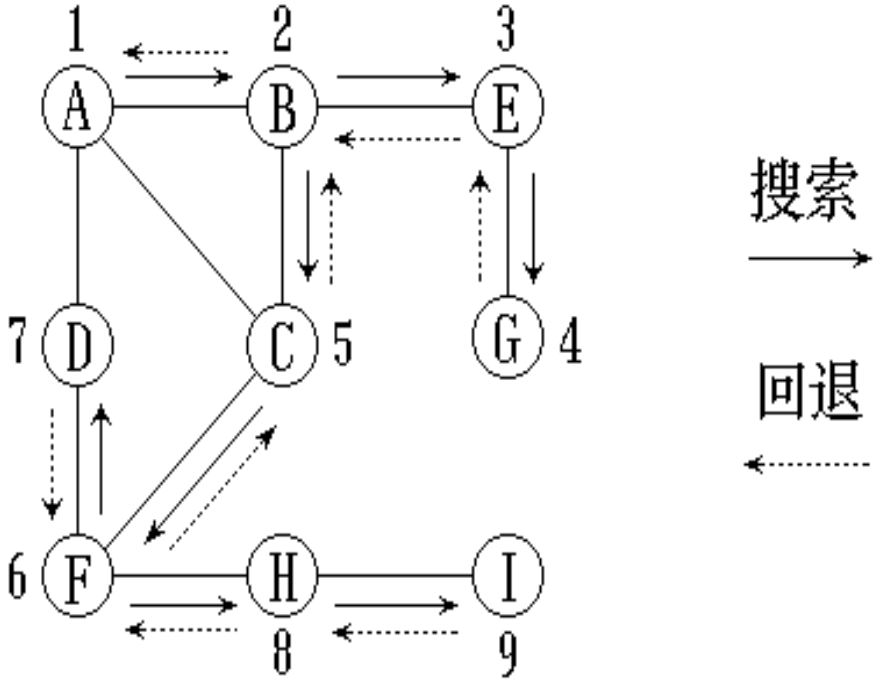
- 树的先根遍历过程
  - ✓ 首先访问根结点；
  - ✓ 对根的各个子树依次执行先根遍历；



- *DFS* 在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一邻接顶点 w_1；再从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2；然后再从 w_2 出发，进行类似的访问，… 如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 u 为止。接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点。如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；如果没有，就再退回一步进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

深度优先搜索*DFS* (Depth First Search)

- 深度优先搜索的示例



图的深度优先搜索算法：

以邻接矩阵表示法作为图的存储结构

连通图：

```
template<class T>
void MGraph<T>::DFS( int v, bool* visited )
{
    cout<<vexs[v];
    visited[v]=true;
    for(i=0; i<vexnum; i++)
        if( edges[v][i] == 1 && !visited[i] )
            DFS( i, visited );
}
```

一般图：

```
template<class T>
void MGraph<T>::DFSTraverse( )
{
    bool* visited=new bool[vexnum];
    for(v=0; v<vexnum; v++)
        visited[v]=false;
    for(v=0;v<vexnum;v++)
        if(!visited[v])
            DFS(v, visited);
    delete []visited;
}
```

问题：

1. 如何判断一个无向图是否是连通的？
2. 如何求出一个非连通图中的连通分量个数？

算法分析：

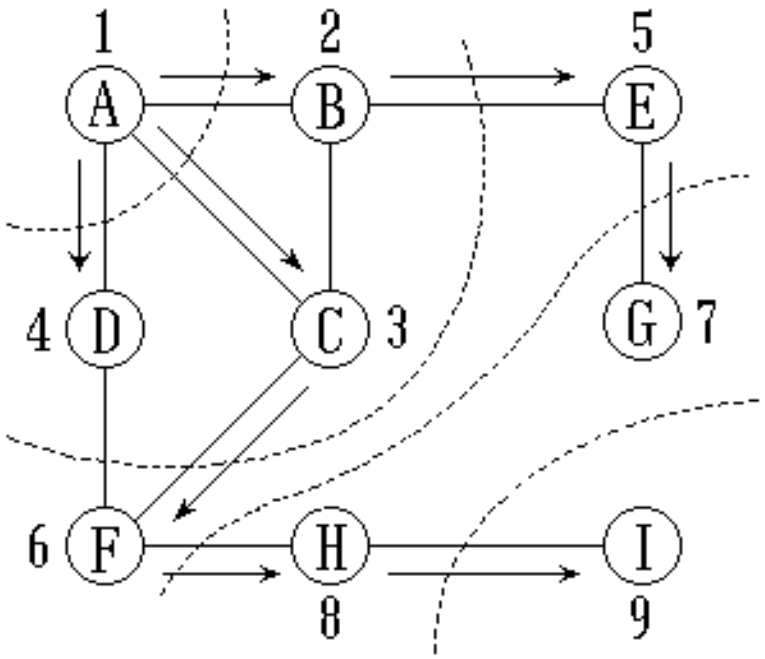
- 设图中有 n 个顶点，e 条边。
- 如果用邻接矩阵表示图，查找每一个顶点的所有的边，所需时间为$O(n)$，则遍历图中所有的顶点所需的时间为$O(n^2)$。
- 如果用邻接表表示图，沿 firstedge链可以找到某个顶点 v 的所有邻接顶点 w。由于总共有 $2e$ 个边结点，所以扫描边的时间为$O(e)$。而且对所有顶点递归访问1次，所以遍历图的时间复杂性为$O(n+e)$。

8.3.2 广度优先搜索 (Breadth First Search)

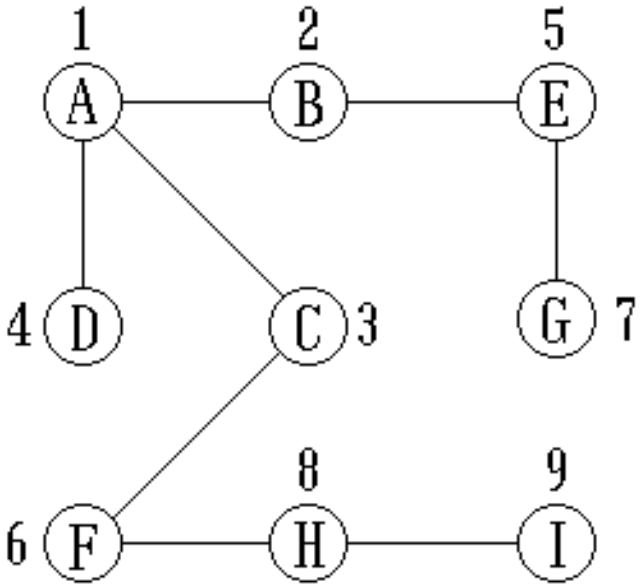
- ✓ 类似于树的层次遍历。
- ✓ 假设从图中某个顶点v出发，在访问了v之后，依次访问v的各个未曾访问过的邻接点，并保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问。直至图中所有已被访问的顶点的邻接点都被访问到。
  - ✓若此时图中还有未被访问的顶点，则任选其中之一作为起点，重新开始上述过程，直至图中所有顶点都被访问到。

广度优先搜索*BFS* (Breadth First Search)

- 广度优先搜索的示例



搜索



广度优先搜索过程　　广度优先生成树

- **广度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有往回退的情况。因此，广度优先搜索不是一个递归的过程，其算法也不是递归的。**

✓对于广度优先遍历：

- **其关键之处在于怎么保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问，也就是先到先被访问。**
- **为了实现逐层访问，算法需要使用一个*队列*，以便按序逐层访问。**
- **为避免重复访问，需要一个辅助数组 visited[ ]，给被访问过的顶点加标记。**

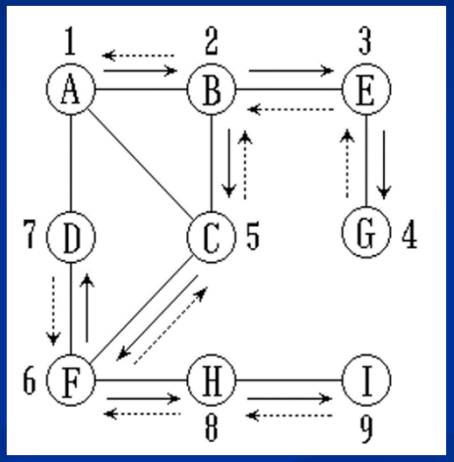
广度优先遍历算法（邻接矩阵表示法）：

```
template<class T>
void MGraph<T>::BFSTraverse()   //v为遍历出发点序号
{
    SeqQueue Q;
    bool* visited=new bool[vexnum];
    for(i=0; i<vexnum; i++)
        visited[i]=0; // 置初值
    for(v=0;v<vexnum;v++)
        if(!visited[v]){
            cout<<vexs[v];
            visited[v]=1;  //设置出发点访问标志为已访问
```

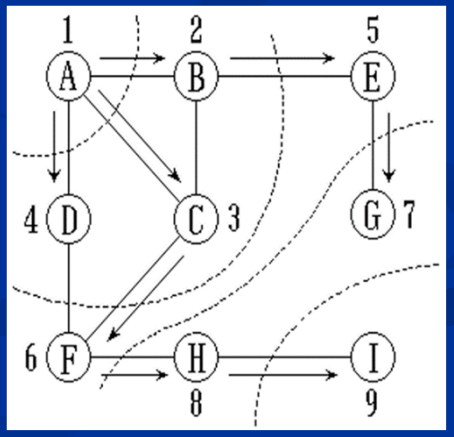
```
            Q.EnQueue(v);
             while( ! Q. Empty() ) {
                  u=Q.DeQueue();
                  for(j=0; j<vexnum; j++ )
                        if(edges [u][j]==1 && !visited[j]){
                             cout<<vexs[j];
                             visited[j]=1;
                             Q.EnQueue( j);
                        }
                  }
            }
            delete [] visited;
}
```

DFS与BFS算法特性的比较

- 对每个顶点的处理机会？
  - DFS: 2
    - ✓ At discovering
    - ✓ At finishing
  - BFS: only 1, when dequeued



DFS



BFS

8.3.4 DFS与BFS算法的应用场景

- **BFS**
  - 适合用来求从一个源点到另一个顶点的最短距离（指路径上的边数最少）的路径。
- **DFS**
  - 相对于BFS，DFS因为有2次处理时机，因而有非常广泛的应用
  - 例如：搜索从一个源点到另一个顶点的所有路径、判别有向图的有向环、有向图强连通分量的划分、无向图双连通分量的划分等。

8.3.4 图遍历算法的应用

- 图遍历算法是很多图处理算法的核心与基础，因为很多基于图的问题求解往往是在遍历过程中完成的。
- 图的遍历算法包含深度优先遍历和广度优先遍历两种算法，在不同的应用场合可选择不同的遍历算法。

- 例8-1：设计一个算法，找出图G中从顶点u到顶点v的所有简单路径。假设图G采用邻接表存储结构。

  问题1：采用哪种搜索方法？

  - 深度优先搜索

  问题2：如何找出图中从顶点u到顶点v的所有简单路径？

  - 允许顶点能重复到达

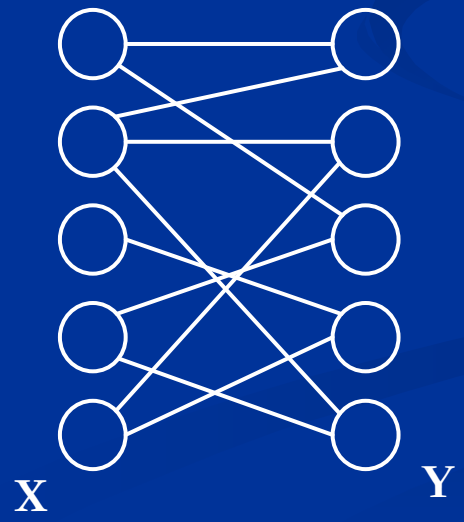
```
void Find_All_Path(ALGraph<T> &G, int u, int v, int k)
// k表示当前路径长度
{
    path[k]=u;   //加入当前路径中
    visited[u]=true;
    if( u==v ) {    //找到了一条简单路径
          cout<<"Found one path!\n";
          for(int i=0; i<=k; i++)
             cout<<path[i]<<endl;   //输出路径
    }
    else
          for(p=G.firstEdge(u); p; p=p->nextedge){
             w=p->adjvex;
             if(!visited[w])
                   Find_All_Path(G, w, v, k+1);  //继续寻找
          }
    visited[u]=false;
     path[k]=0;       //回溯
}
```

- 例8-2：设计一个算法，找出图G中从顶点u到顶点v长度为n的所有简单路径。假设图G采用邻接表存储结构。
  - 在例8-1算法的基础上增加一个长度参数n。

```
void Find_All_Path(ALGraph<T> &G, int u, int v, int k, int n){
    path[k]=u; //加入当前路径中
    visited[u]=true;
    if(u==v && k==n){ //找到了一条简单路径
        cout<<"Found one path!\n";
        for(int i=0; i<=n; i++)
            cout<<path[i]<<endl; //输出路径
    }
    else
        for(p=G.firstEdge(u); p; p=p->nextedge){
            w=p->adjvex;
            if(!visited[w])
                Find_All_Path(G, w, v, n, k+1);
        }
    visited[u]=false;
    path[k]=0;      //回溯
}
```

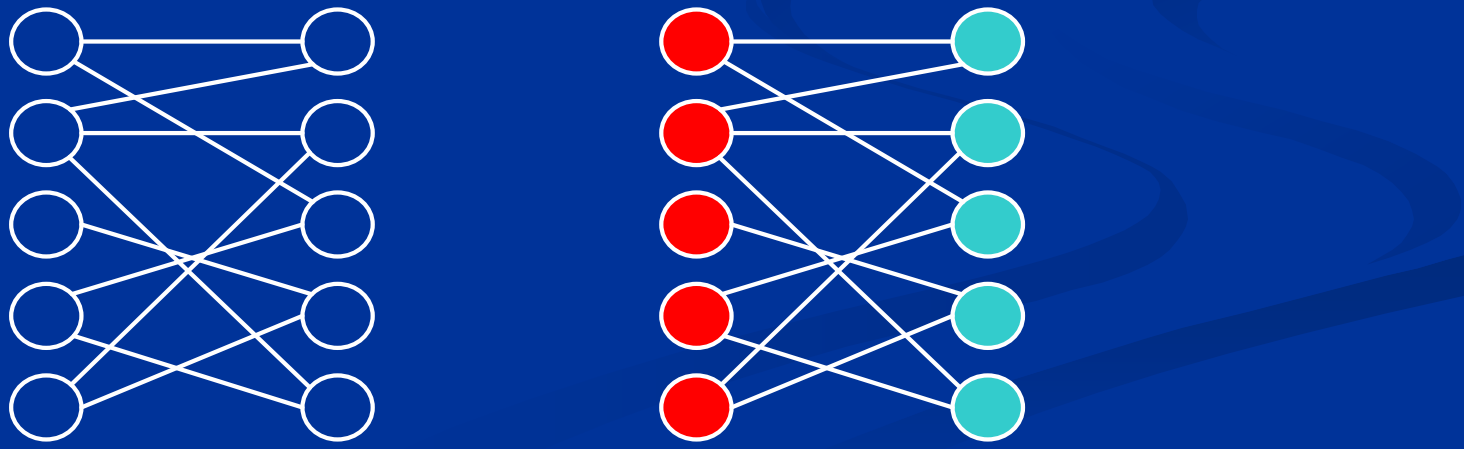
- 例8-3：设计一个算法，判断一个给定的连通图G是否为二部图（Bipartite）。假设图G采用邻接表存储结构。

首先给出二部图的定义：一个二部图G=(V, E)是一个无向图，它的顶点集合可以被划分成X和Y，并具有下述性质：每条边有一个端点在X中且另一个端点在Y中。下图给出了一个二部图的示例。

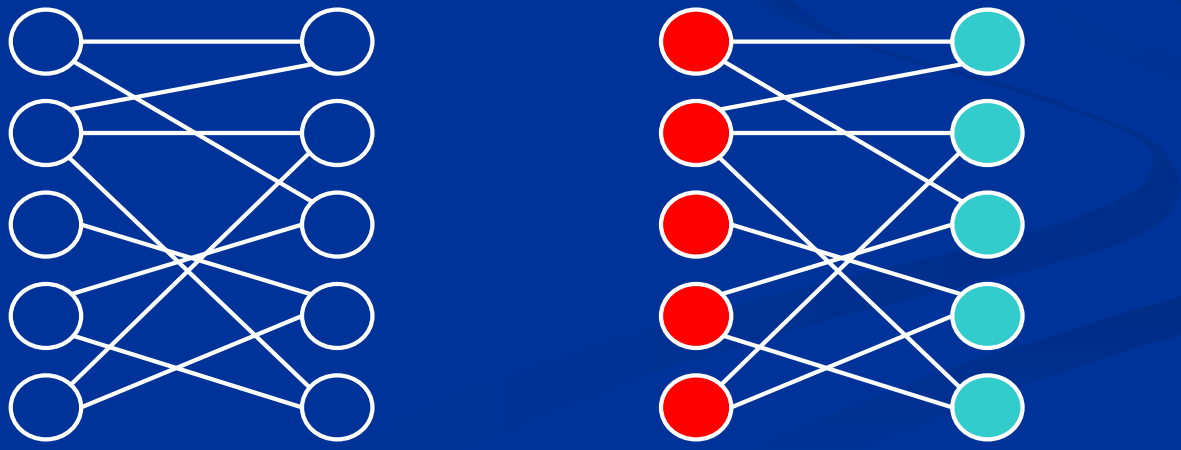


算法设计思路：

- 问题1: 判断的基本方法？
  - 为了便于实现二部图的判定，可以想象将集合X中的顶点着红色，集合Y中的顶点着蓝色。
  - 这样将判断问题转换成着色问题。
  - 则若发现某条边的两个端点颜色相同，该图为非二部图。



- 问题2: 如何实现顶点的着色？
  - 任取一个顶点s并且将它着红色。接着，找出顶点s 的所有邻接点，并将这些顶点都着为蓝色；接下来再分别将这些顶点的所有邻接点都着为红色，…，如此间替对顶点进行着色，直到图中所有顶点都被着色为止。
  - 可以在广度优先遍历算法的实现中再加上一个记录所有顶点颜色的数组Color[]，并在遍历过程中相应地记录每个顶点的颜色。



```
bool check_Bipartite( ALGraph<T> &G ) {
    int* visited=new int[G.vexterNum()];
    int* color=new int[G.vexterNum()];
    SeqQueue Q;
    for(int i=0;i<G.vexterNum();i++)   visited[i]=0;
    visited[0]=1;  //从第0个顶点开始处理
    color[0]=1;    //1表示红色，-1表示蓝色
    Q.EnQueue(0);
    while(! Q.Empty() ){
        u=DeQueue( Q );
        for(p=G.firstEdge(u); p; p=p->nextedge ){
            k=p->adjvex;
            if(!visited[k]){
                visited[k]=1;
                color[k]= -color[u];
                Q.EnQueue( k );
            }
        }
    }
```

```
for( i=0; i<G.vexterNum(); i++)
    for(p=G.firstEdge(i); p; p=p->nextedge){
        k=p->adjvex;
        if(color[i]==color[k]){
            delete []visited;
            delete []color;
            return false;
        }
    }
 delete []visited;
 delete []color;
 return true;
}
```

8.4 最小生成树

- ✓ 生成树的定义：*n* 个顶点的连通网络的生成树有 *n* 个顶点、*n*-1 条边。生成树是连通图的极小连通子图。
- ✓ 问题：对于给定的连通网络，如何求得其**生成树**？
  - ➢ 对含有n个顶点的连通图G，从任一顶点出发，作一次深度优先或广度优先遍历，将遍历过程中经过的n-1条边和图中的n个顶点连接起来构成一个极小连通子图，就是图G的一棵生成树。
  - ➢ 用不同的遍历图的方法，可以得到不同的生成树；从不同的顶点出发，也可能得到不同的生成树。

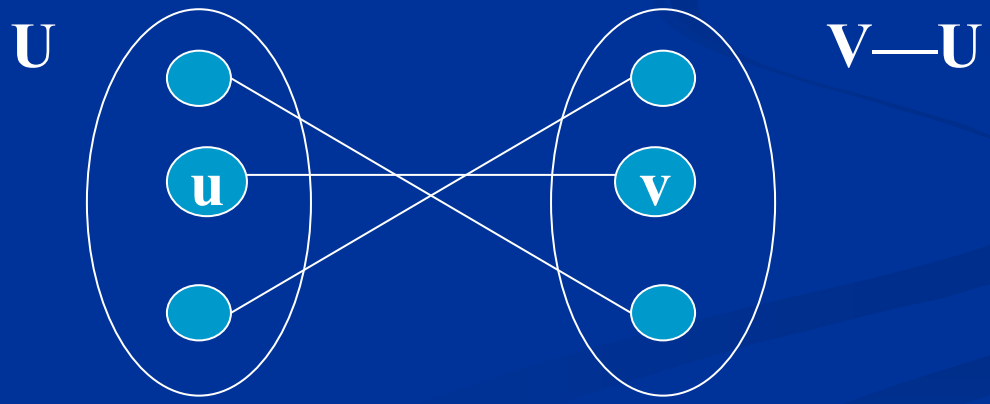
最小生成树

- ✓ 生成树各边的权值总和称为**生成树的权**。
- ✓ 权最小的生成树称为**最小生成树**。
- ✓ 最小生成树的应用实例：
  - ➢ 假设要在n个城市之间建立通信网。令图G的顶点表示城市，边表示连接两个城市的通信线路，边的权值表示通信线路的长度或代价。在n个城市间构造通信网最少需要n-1条线路，问：如何选择这n-1条边，使得构造这个通信网总的代价最小?
- ✓ 构造最小生成树的准则：
  - ➢ 必须只使用该网络中的边来构造最小生成树；
  - ➢ 必须使用且仅使用 n-1 条边来联结网络中的 n 个顶点；
  - ➢ 不能使用产生回路的边。

最小生成树的构造算法：

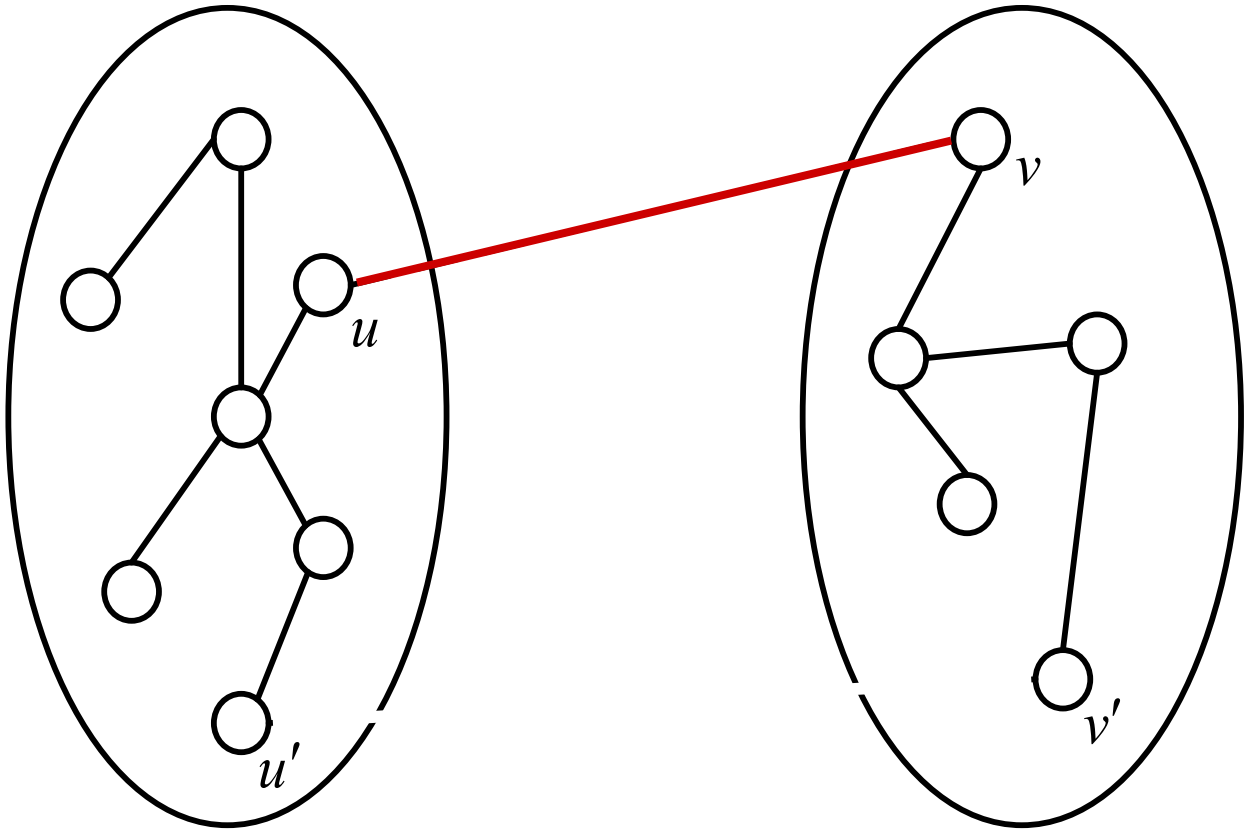
MST 性质：

设 G = （V，E）是一个连通网络，U 是顶点集 V 的一个真子集。若（u，v）是 G 中所有的一个顶点在 U，另一个顶点不在 U 的边中，具有最小权值的一条边，则一定存在 G 的一棵最小生成树包括此边。



证明（反证法）：

- ✓ 假设 G 中任何一棵最小生成树都不包含(u，v)。
- ✓ 设T是一棵最小生成树但不包含(u，v)。由于T是最小生成树，所以 T 是连通的，因此有一条从u到v的路径，且该路径上必有一条连接两个顶点集 U、V-U 的边(u',v')，其中u'∈U，v'∈V-U。当把边(u，v)加入到T中后，得到一个含有边(u，v)的回路。删除边(u',v')，上述回路即被消除。由此得到另一棵生成树 T'， T和T'的区别仅在于用边(u，v)代替了(u',v')。由于(u，v)的权<= (u',v')的权，所以， T'的权<= T的权，与假设矛盾。


u
u′
v
v′
顶点集合U
顶点集合V-U

Prim算法:

输入：一个含有n个顶点的无向连通网G=(V,E)

输出：一棵最小生成树T=(U,TE)

Prim算法的基本思想：

- 初始时从V中任取一个顶点（如v_1），将其放入U中，使U={v_1}，这样集合U与集合V-U就构成图G的一个**割（cut）**。
- 只要U是V真子集，就不断进行如下的贪心选择：选择一条**权值最小的交叉边**（v_i, v_j），其中$v_i \in U$, $v_j \in V-U$，并把该边（v_i, v_j）和其不在最小生成树中的顶点v_j分别加入T的顶点集U和边集TE中。

Prim算法:

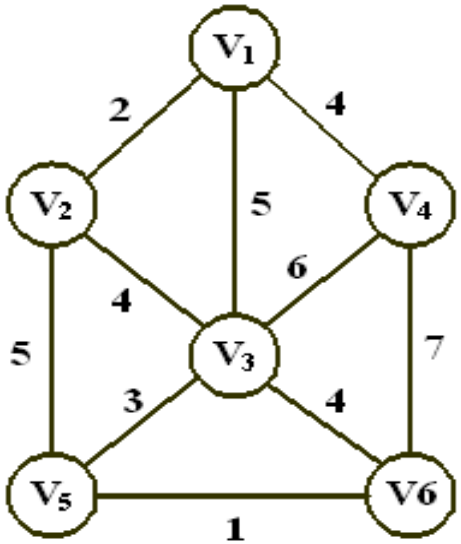
输入：一个含有n个顶点的无向连通网G=(V,E)

输出：一棵最小生成树T=(U,TE)

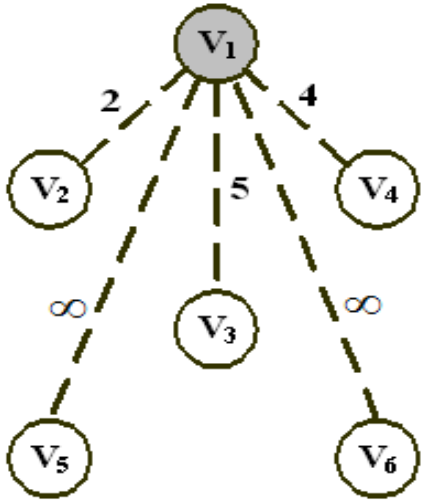
Prim算法中的关键问题：

- 每次如何从图的割中选择一条**权值最小的交叉边**（v_i, v_j）？
  - 蛮力（Brute-force）法效率低！
  - 注意增加一个个顶点到最小生成树中的过程实际上是一种增量式的变化。
  - **我们感兴趣的是从生成树T外的每个顶点到生成树T中顶点的最短距离（即最小权值）。**

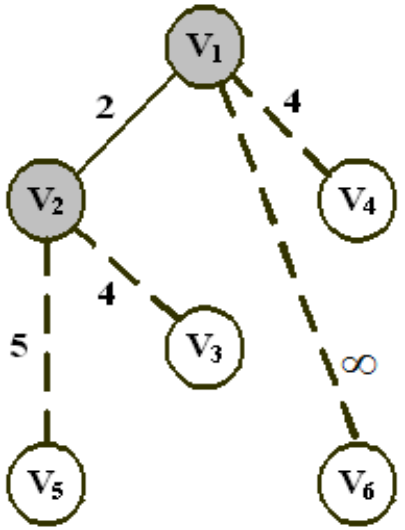
用Prim算法构造最小生成树



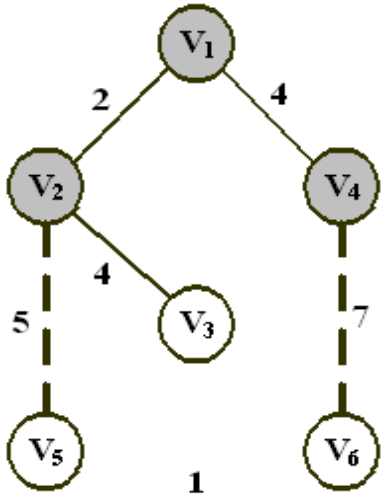
(a) 无向连通网N_2



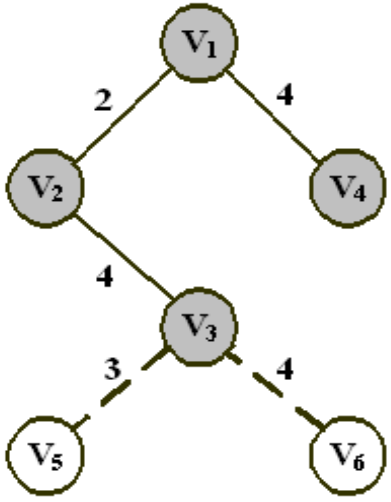
(b)



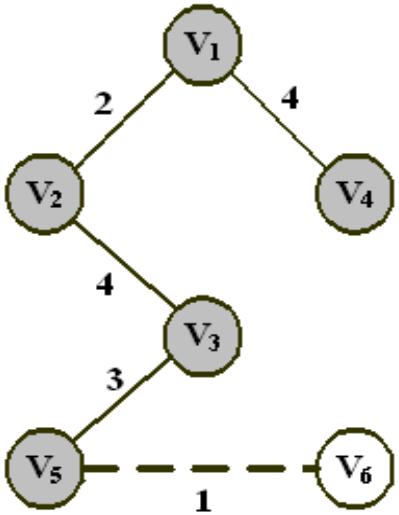
(c)



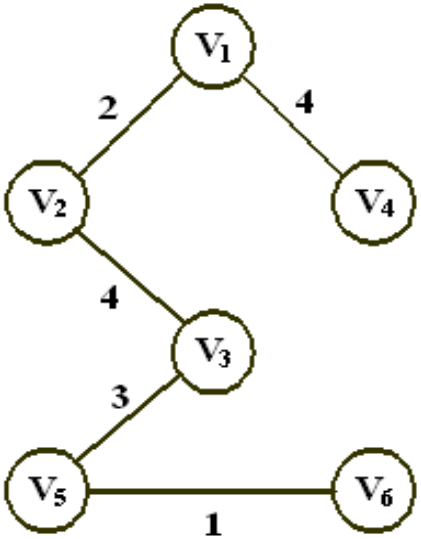
(d)



(e)



(f)



(g)

Prim 算法框架描述：

1. 从连通网络 $N=\{V, E\}$ 中的选择任一顶点v_1加入到生成树 T 的顶点集合 U中，并为集合 $V-U$中的各顶点置最小权值边.

2. **while (生成树T中顶点数目 < n) {**

   从集合 $V-U$中各顶点对应的最小权值边中选取最小权值边(u, v);

   将边(u, v) 及其在集合 $V-U$中的顶点 v，加到生成树 T 中；

   调整集合 $V-U$中各顶点对应的最小权值；

   }

- 为实现上述思想，需要引入数据结构来记录下列信息：
  - 从生成树T外的每个顶点（即集合V-U中的每个顶点）到生成树T中顶点（即集合U中顶点）的最小交叉边；
  - 这些最小交叉边相应的权值。
- 因此需要引入一个辅助数组miniedges[]，用于存放集合V-U中的各顶点到集合U中顶点的最小交叉边及其权值。

miniedges数组的元素类型具体定义如下：

```
template<class T>
struct edgeNode{
    T adjvex;
    float lowcost;
};
```

对于每个顶点$v_i \in V-U$,在数组miniedges中对应一个相应元素miniedges[i]，且：

miniedges[i].lowcost =
Min{ cost(v_i,u)|u∈U }

Prim 算法（邻接矩阵表示法）：

```
void Prim( MGraph<T> &G, int v ) {
   edgeNode<T>* miniedges=new edgeNode<T>[G.vexterNum()];
   for(i=0; i<G.vexterNum(); ++i){ //辅助数组初始化
         miniedges[i].adjvex = G.getVexValue(v);
         miniedges[i].lowcost=G.getEdgeValue(v, i);
   }
   miniedges[v].lowcost = 0;   //将顶点v加入到集合U中
```

```
    for(i=1; i<G.vexterNum(); ++i){  //选择其余G.vexnum-1个顶点
        k = minimum( miniedges );
        cout<< miniedges[k].adjvex<<"-->“
             << G.getVexValue(k)<<endl;
        miniedges[k].lowcost = 0;    //将第k个顶点并入集合U
        for(j=0; j<G.vexterNum(); ++j)
            if( G.getEdgeValue(k, j) < miniedges[j].lowcost ){
                miniedges[j].adjvex = G.getVexValue(k) ;
                miniedges[j].lowcost = G.getEdgeValue(k, j);
            }
    }
    delete []miniedges;
}
```

算法的时间复杂度为$O(n^2)$

Kruskal 算法

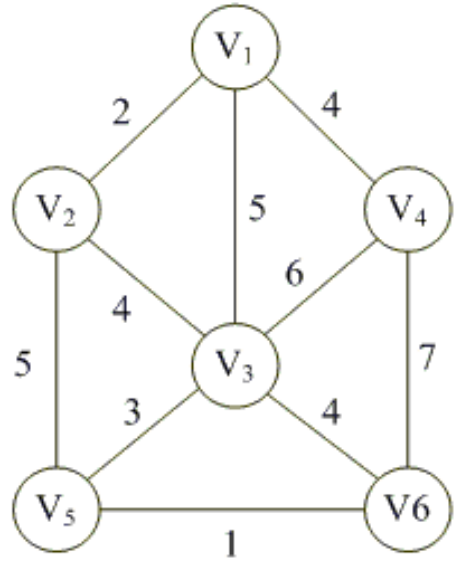
基本思想：

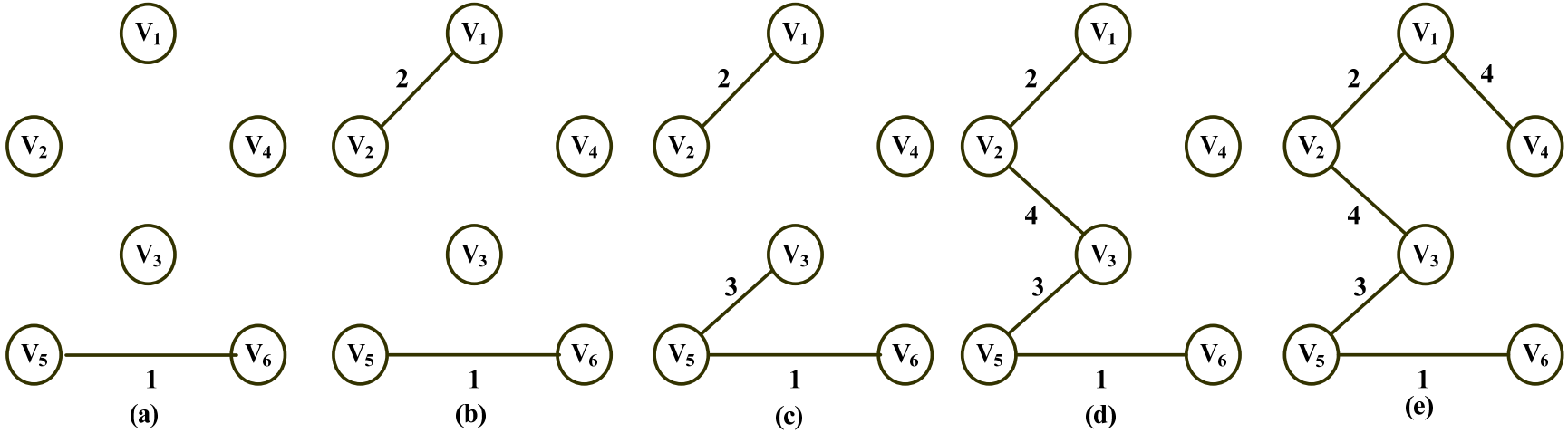
初始化：

设有一个有 n 个顶点的连通网络 $N=\{V,E\}$，最初先构造一个只有 n 个顶点，没有边的非连通图 $T=\{V,\varnothing\}$，图中每个顶点自成一个连通分量。

循环执行，直到所有顶点在同一个连通分量上为止：

当在 E 中选到一条具有最小权值的边时,若该边的两个顶点落在不同的连通分量上，则将此边加入到 T 中；否则将此边舍去，重新选择一条权值最小的边。


V1
V2
V3
V4
V5
V6
2
4
5
6
4
5
7
3
4
1


V1
V2
V3
V4
V5
V6
2
4
3
1
(a)
(b)
(c)
(d)
(e)

Kruskal算法的描述:

```
T = (V, φ);
while ( T中所含边数 < n-1 )
{
    从E中选取当前最小权值边 (u,v);
    从E中删除边(u,v);
    if ( (u,v) 并入T之后不产生回路 )
        将边 (u,v) 并入T中;
}
```

怎么有效地实现？

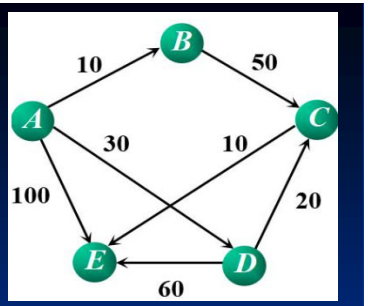
```
Template <class T>
void Kruskal(MGraph<T> &G, vector<EdgeType> &tree ) {
    vector<EdgeType> graph;
    getGraph(G, graph)  //将边集按权值cost从小到大放入graph数组
    MFset<EdgeType> set(G);    // 并查集
    k = 0; j=0;
    while( k<vexnum - 1 ){
        h1=graph[j].head;  t1=graph[j].tail;
        h2=set.Find(h1);     t2=set.Find(t1);
        if(h2 != t2){
            tree[k].head=h1; tree[k].tail=t1;
            tree[k].cost=graph[j].cost;
            set.Merge(h2, t2);
            k++;
        }
        j++;
    }
}
```

算法的时间复杂度为 $O(e\log_2 e)$

Prim算法 与 Kruskal算法比较

- 共同点?
  - 都是基于贪心策略的算法
  - 基于MST性质
- 不同点?
  - Prim算法的特点是生成树中的边总是形成单棵树，而在Kruskal算法中生成树中的边是一个不断生长的森林。
  - 相比较Prim算法而言，Kruskal算法更适用于求解稀疏网的最小生成树。

8. 5 最短路径 (Shortest Path)



- **问题的提出：**

  假如要在计算机上建立一个交通咨询系统，可用图或网的结构来表示实际的交通网络，用顶点表示城市，边表示城市之间的交通联系。这个交通咨询系统可以回答旅客提出的各种问题。如：

  一位旅客想从A城到B城，希望选择中转次数最少的路线。

  一位司机想选择一条从A城到B城所需距离最短的路径？

- **最短路径问题：**

  如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。

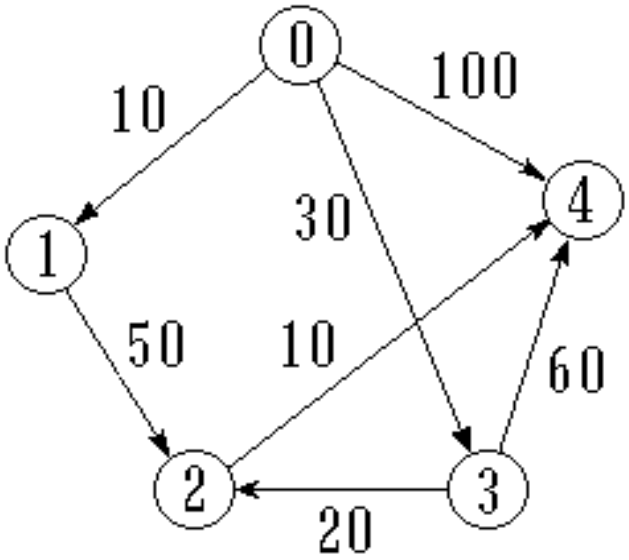
问题解法：

- 边上权值非负情形的单源最短路径问题
  - — Dijkstra算法
- 所有顶点之间的最短路径
  - — Floyd算法

单源最短路径问题

- **问题的提法：** 给定一个带权有向图G与源点v，求从v到G中其它顶点的最短路径。限定各边上的权值大于或等于0。

- 举例说明



（a）带权有向图

	0	1	2	3	4	
	0	10	∞	30	100	0
	∞	0	50	∞	∞	1
	∞	∞	0	∞	10	2
	∞	∞	20	0	60	3
	∞	∞	∞	∞	0	4

（b）邻接矩阵

从顶点v0到其余各顶点的最短路径

源点	终点	最短路径	路径长度
v_0	v_1	(v_0, v_1)	10
	v_3	(v_0, v_3)	30
	v_2	(v_0, v_3, v_2)	50
	v_4	(v_0, v_3, v_2, v_4)	60

为求得这些最短路径，Dijkstra提出按路径长度的递增次序，逐步产生最短路径的算法：

- 假设从源点v_0到各终点v_1，v_2，…,v_k间存在最短路径，其路径长度分别为：$l_1, l_2, \ldots, l_k$，并且l_p为其中的最小值。即从源点v_0到终点v_p的路径最短。显然这条路径上只有一条弧，否则它就不可能是所有最短路径中长度最短者。
- 第二条长度次短的（设从源点v_0到终点v_q）最短路径只可能产生于下列两种情况：
  - 一是从源点到该点有弧$<v_0, v_q>$存在；
  - 另一是从已求得最短路径的点v_p到该点有弧$<v_p, v_q>$存在，且弧$<v_0, v_p>$和$<v_p, v_q>$上的权值之和小于弧$<v_0, v_q>$的权值。

- 类推到一般情况：假设已求得最短路径的的顶点集合S为$\{ v_{p1}, v_{p2}, \cdots, v_{pk} \}$，则下一条最短路径（设终点为x）或者为弧$< v_0, x >$，或者为一条**只经过S中的顶点**而最后到达终点x的路径。

证明：假设此路径上存在一个不在集合S中出现的顶点，则说明存在终点不在S而路径长度比此更短的路径，这是不可能的。因为我们是按长度递增的次序来产生各最短路径的，故长度比此路径短的所有路径均已产生，它们的终点必在集合S 中。

Dijkstra算法的实现:

- 设置并逐步扩充一个集合S，存放已求出的最短路径的顶点，则尚未确定最短路径的顶点集合是V-S。

引入一个辅助数组dist[]。它的每一个分量dist[i]表示当前确定的从源点v_0到终点v_i的最短路径的长度（当前最短距离）。

初始状态：

- 若从源点v_0到顶点v_i有边，则dist[i]为该边上的权值；
- 若从源点v_0到顶点v_i没有边，则dist[i]为+∞。

Dijkstra算法框架：

S = { v };

置集合V-S 中各顶点的初始距离值；

while （集合S中顶点数 < n)

{

在集合V-S中选择距离值最小的顶点v_j；

S = S + { v_j };

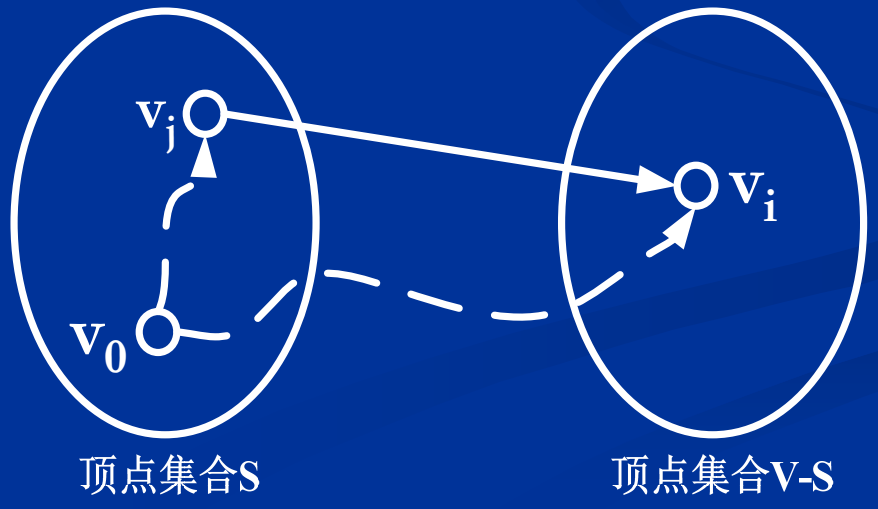
调整集合V-S中剩余顶点的当前最短距离值；

}

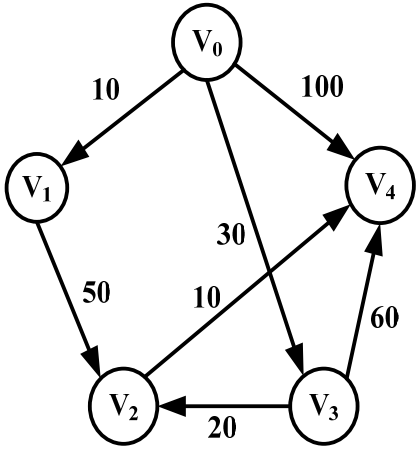
当前最短距离值的调整方法

- 算法执行过程中数组dist的元素值是一个增量的变化过程。
- 从集合V-S中选择一个当前距离最小的顶点v_j加入集合S后，需要调整V-S中剩余顶点的当前最短距离值。
- 对于V-S中的顶点v_i，其最短距离值的调整方法是：

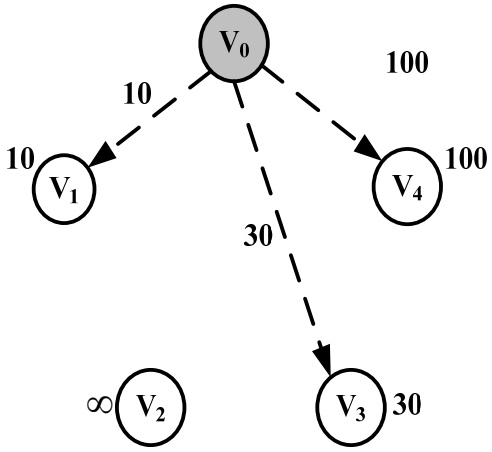
$$dist[i]=Min\{dist[i], dist[j] + cost(j, i)\}$$



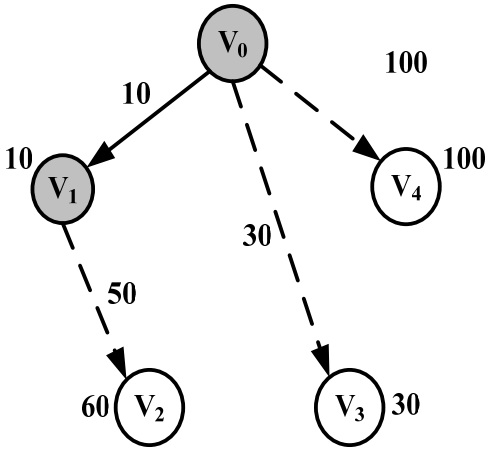
Dijkstra算法求单源最短路径示例



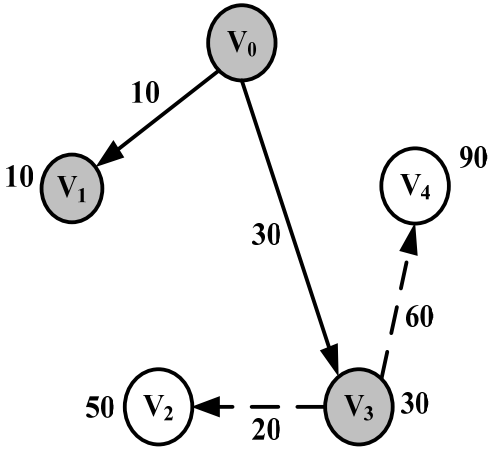
(a) 带权有向图G_9



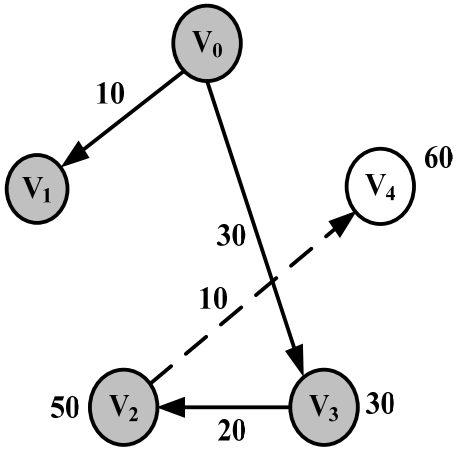
(b)



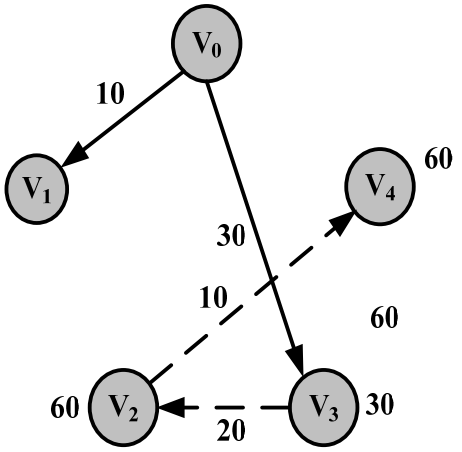
(c)



(d)



(e)



(f)

Dijkstra算法的具体实现

- 图的存储结构？
  - 邻接矩阵方式
- 如何标识图中各顶点在算法执行过程中是否已被加入到集合S中？
  - 设置一个一维数组S[]，并规定：
    - S[i]=0　顶点v_i未加入集合S
    - S[i]=1　顶点v_i已加入集合S
- 如何记录Dijkstra算法所求出的从源点到各顶点的最短路径？
  - 引入一个数组path[]，其中，path[i]中保存了从源点到终点v_i的最短路径上该顶点的前驱顶点的序号。

```
void Dijkstra (MGraph<T> &G, int v0, int path[]){
        bool* S=new bool[G.vexterNum()];
        float *dist=new float[G.vexterNum()];
        for(v=0; v<G.vexterNum(); ++v){
                S[v]=0;    //集合S置空
                dist[v]=G.getEdgeValue(v0,v);   //距离初始化
                if(dist[v]<INFINITY)    //路径初始化
                        path[v]=v0;
                else
                        path[v]=-1;   //表示顶点v无前驱顶点
        }
        dist[v0] = 0;
        S[v0] = 1;   // 将顶点v0加入S集
```

```
    for(i=1; i<G.vexterNum(); ++i){
        min=INFINITY;
        for(j=0; j<G.vexterNum(); ++j)
            if( S[j]==0 && dist[j]<min ){
                    k=j;  min=dist[j];
            }
        S[k]=1;   //将离源点v0最近的顶点k加入S集
        for(w=0; w<G.vexterNum();++w) {
            if( S[w]==0 && dist[k]+G.getEdgeValue(k,w)
                                                <dist[w] ){
                    dist[w]=dist[k]+G.getEdgeValue(k,w);
                    path[w]=k;
            }
        }
    }
    outputPath(path); //输出最短路径
    delete []S;  delete []dist;
}
```

算法的时间复杂度为$O(n^2)$

所有顶点之间的最短路径

问题的提法： 已知一个各边权值均大于0的带权有向图，对每一对顶点 $v_i \neq v_j$，要求求出v_i与v_j之间的最短路径和最短路径长度。

两种解决方法：

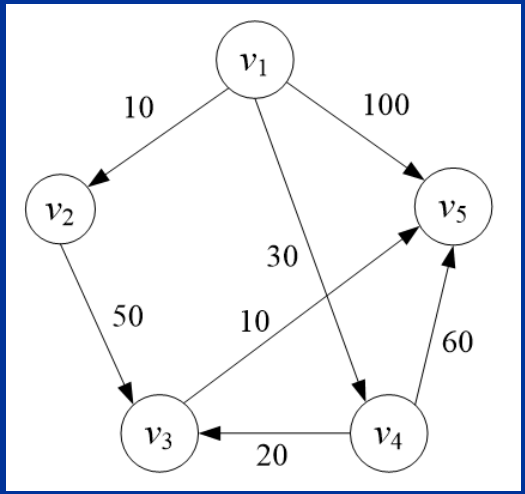
可以依次把有向网络中的每个顶点作为源点，重复执行DIJKSTRA算法n次，即可求出每对顶点之间的最短路径。

弗洛伊德（Floyd）算法

直观的Floyd算法的基本思想：

假设求从顶点v_i到v_j的最短路径。如果从v_i到v_j有弧，则从v_i到v_j存在一条长度为edges[i][j]的路径，但该路径不一定是最短路径。因为可能存在一条从v_i到v_j，但包含有其它顶点为中间点的路径。

因此，尚需进行**n次试探**，测试从v_i到v_j有能否有以顶点v_1,v_2，…,v_n为中间点的更短路径.



$$\begin{bmatrix} 0 & 10 & \infty & 30 & 100 \\ \infty & 0 & 50 & \infty & \infty \\ \infty & \infty & 0 & \infty & 10 \\ \infty & \infty & 20 & 0 & 60 \\ \infty & \infty & \infty & \infty & 0 \end{bmatrix}$$

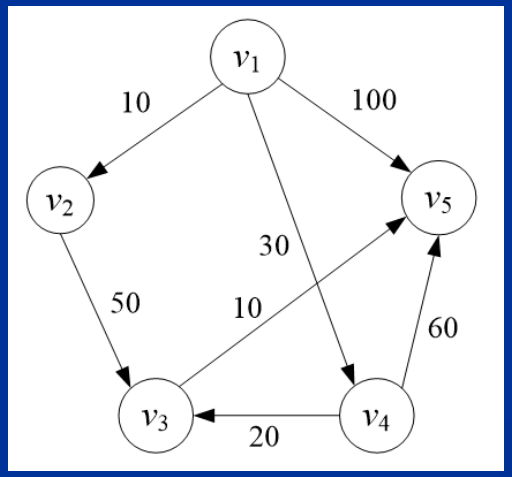
Floyd算法的基本思想：多阶段决策的过程

$D^{(k)}[i][j]$是从顶点v_i到v_j，中间顶点的序号不大于k的最短路径的长度

阶段1 → 阶段2 → 阶段3 → 阶段4 → 阶段5

$$D^{(1)}=\begin{bmatrix}0&10&\infty&30&100\\\infty&0&50&\infty&\infty\\\infty&\infty&0&\infty&10\\\infty&\infty&20&0&60\\\infty&\infty&\infty&\infty&0\end{bmatrix}\quad D^{(2)}=\begin{bmatrix}0&10&60&30&100\\\infty&0&50&\infty&\infty\\\infty&\infty&0&\infty&10\\\infty&\infty&20&0&60\\\infty&\infty&\infty&\infty&0\end{bmatrix}\quad D^{(3)}=\begin{bmatrix}0&10&60&30&70\\\infty&0&50&\infty&60\\\infty&\infty&0&\infty&10\\\infty&\infty&20&0&30\\\infty&\infty&\infty&\infty&0\end{bmatrix}$$

$$D^{(4)}=\begin{bmatrix}0&10&60&30&70\\\infty&0&50&\infty&60\\\infty&\infty&0&\infty&10\\\infty&\infty&20&0&30\\\infty&\infty&\infty&\infty&0\end{bmatrix}\quad D^{(5)}=\begin{bmatrix}0&10&50&30&60\\\infty&0&50&\infty&60\\\infty&\infty&0&\infty&10\\\infty&\infty&20&0&30\\\infty&\infty&\infty&\infty&0\end{bmatrix}$$



$$D^{(0)}=\begin{bmatrix}0&10&\infty&30&100\\\infty&0&50&\infty&\infty\\\infty&\infty&0&\infty&10\\\infty&\infty&20&0&60\\\infty&\infty&\infty&\infty&0\end{bmatrix}$$

Floyd算法的基本思想：多阶段决策的过程

Floyd算法主要基于以下观察：

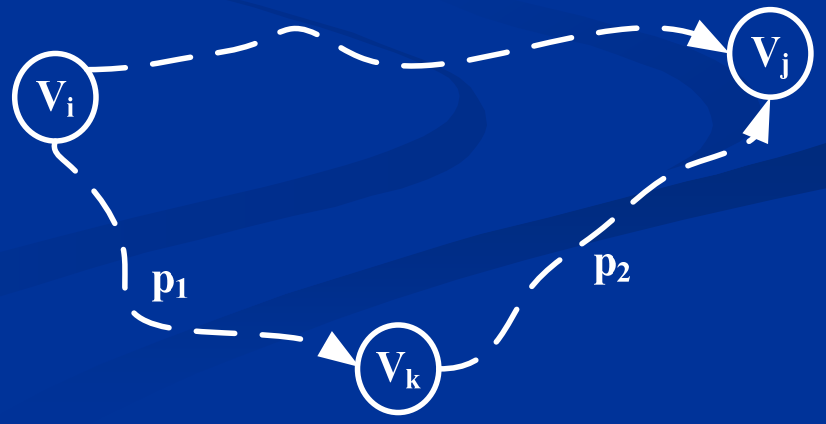
设图G的顶点集为$V=\{v_1,v_2,\ldots,v_n\}$，对某个序号k考虑顶点的一个子集$\{v_1,v_2,\ldots,v_k\}$。

对任意一对顶点v_i和v_j，考察从v_i到v_j且中间顶点皆属于集合$\{v_1,v_2,\ldots,v_k\}$的所有路径，即**中间顶点序号不大于k**的所有路径，设p是其中的一条最小的权值路径。

Floyd算法利用了**路径p**与另一条**中间顶点序号不大于k-1的路径**（即顶点v_i与v_j之间的所有中间顶点都属于$\{v_1,v_2,\ldots,v_{k-1}\}$的路径）之间的联系。这一联系依赖于顶点$v_k$是否是路径p上的一个中间顶点。

Floyd算法的基本思想：

- 如果v_k不是路径p上的中间顶点，则p的所有中间顶点皆在集合$\{v_1,v_2,\cdots,v_{k-1}\}$中。
- 如果v_k是路径p上的中间顶点，则可将路径p分解为如下图所示的两端p1和p2：
  - p1是一条从v_i到v_k且所有中间顶点皆属于集合$\{v_1,v_2,\cdots,v_{k-1}\}$的最短路径；
  - p2是一条从v_k到v_j且所有中间顶点皆属于集合$\{v_1,v_2,\cdots,v_{k-1}\}$的最短路径。



Floyd算法的实现：

- ✓ 实现Floyd算法的关键，是保留每一步所求得的所有顶点对之间的当前最短路径长度。
- ✓ 为此，定义一个n阶方阵序列：

$$D^{(0)}, D^{(1)}, \ldots, D^{(n)}.$$

其中 $D^{(0)}[i][j] = edges[i][j]$；

$$D^{(k)}[i][j] = \min\{ D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \}, k = 1,2,\ldots, n$$

$D^{(0)}[i][j]$是从顶点v_i到v_j，中间顶点是v_0的最短路径的长度，$D^{(k)}[i][j]$是从顶点v_i到v_j，中间顶点的序号不大于k的最短路径的长度，$D^{(n)}[i][j]$是从顶点v_i到v_j的最短路径长度。

```
template<class T>
void MGraph<T>::Floyd(int path[][MAXV],int D[][MAXV]){
    for( i=0;i<vexnum;i++)
      for( j=0;j<vexnum;j++){
          if(i==j) D[i][j]=0;
          else D[i][j]=GetEdgeValue(i,j);
              if(D[i][j]<INFINITY) path[i][j]=j;
              else path[i][j]=-1;
      }
      for(k=0;k<vexnum;k++)
         for(i=0;i<vexnum;i++)
            for(j=0;j<vexnum;j++)
                if(D[i][k]+D[k][j]<D[i][j]){
                    D[i][j]=D[i][k]+D[k][j];    //修改最短路径长度
                    path[i][j]=path[i][k];      //修改最短路径
                }
}
```

算法的时间复杂度为$O(n^3)$