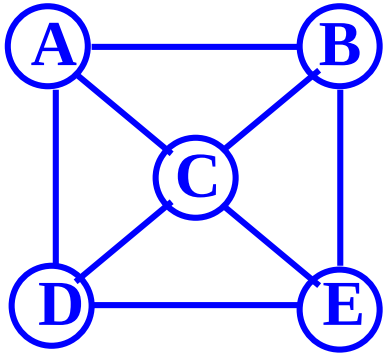
第七章　图



图是一种较线性表和树更为复杂的数据结构。

线性表：线性结构

树：层次结构

图：结点之间的关系可以是任意的，即图中任意两个数据元素之间都可能相关。

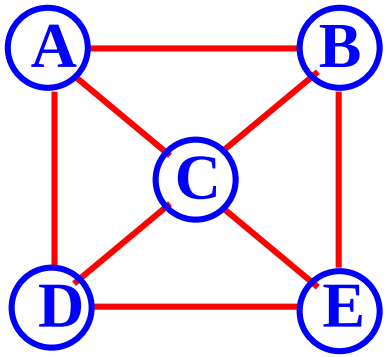
7.1 图的定义和基本术语

图 G 是由两个集合顶点集 V(G) 和边集 E(G) 组成的，记作G=(V(G)，E(G))，简称G=(V，E)。

(图的ADT定义 pp156)

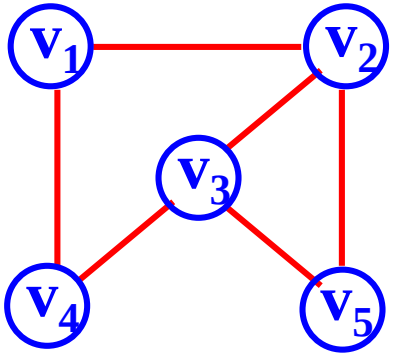
V是顶点的有穷非空集合

E是两个顶点之间的关系，即边的有穷集合



无向图和有向图

无向图：边是顶点的无序对，即边没有方向性。

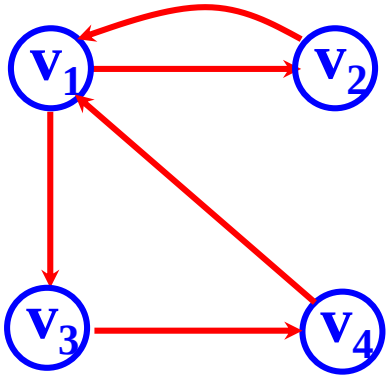


$V = \{ v_1, v_2, v_3, v_4, v_5 \}$

$E = \{ (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_5), (v_3, v_4), (v_3, v_5) \}$

(v_1, v_2)表示顶点 v_1 和 v_2 之间的边，$(v_1, v_2) = (v_2, v_1)$。

有向图：其边是顶点的有序对，即边有方向性。



$V = \{ v_1, v_2, v_3, v_4 \}$

$E = \{ \langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \langle v_3, v_4 \rangle, \langle v_4, v_1 \rangle \}$

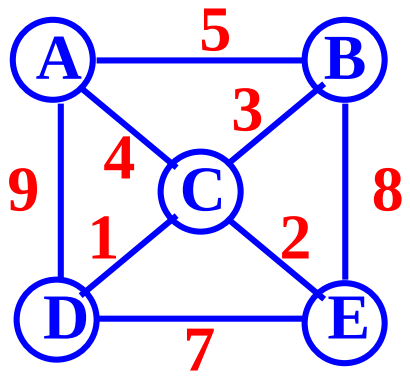
通常边称为弧，$\langle v_1, v_2 \rangle$表示顶点 v_1 到 v_2 的弧。

称 v_1 为弧尾，称 v_2 为弧头。

$\langle v_1, v_2 \rangle \neq \langle v_2, v_1 \rangle$

带权无向图(无向网) 和 带权有向图(有向网)

有时对图的边或弧赋予相关的数值，这种与图的边或弧相关的数值叫做权。



这些权可以表示从一个顶点到另一个顶点的距离。

或者，表示从一个顶点到另一个顶点的代价。

这种带权的图通常称为网。

带权的无向图称为无向网。

带权的有向图称为有向网。

完全图、稀疏图、稠密图

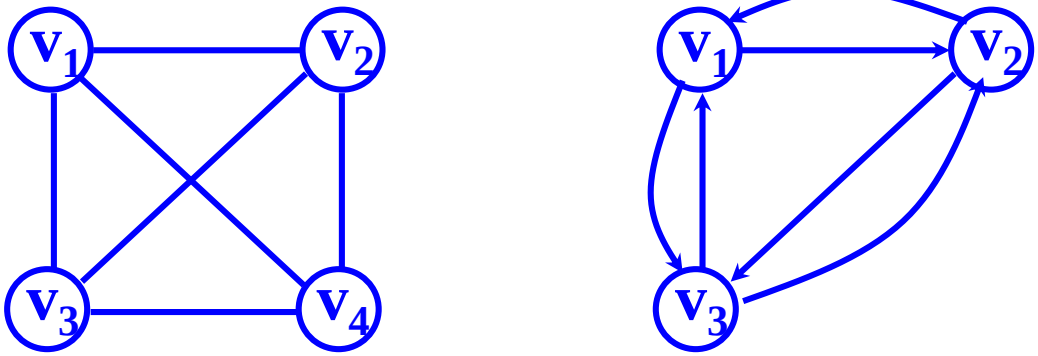
性质：若用 n 表示图中顶点数目，用 e 表示边或弧的数目，若在图中不存在顶点到自身的边或弧，则

对于无向图，$0 \le e \le \frac{1}{2} * n(n-1)$

对于有向图，$0 \le e \le n(n-1)$

有 $\frac{1}{2}n(n-1)$ 条边的无向图称为完全图。

有 n(n-1) 条弧的有向图称为有向完全图。

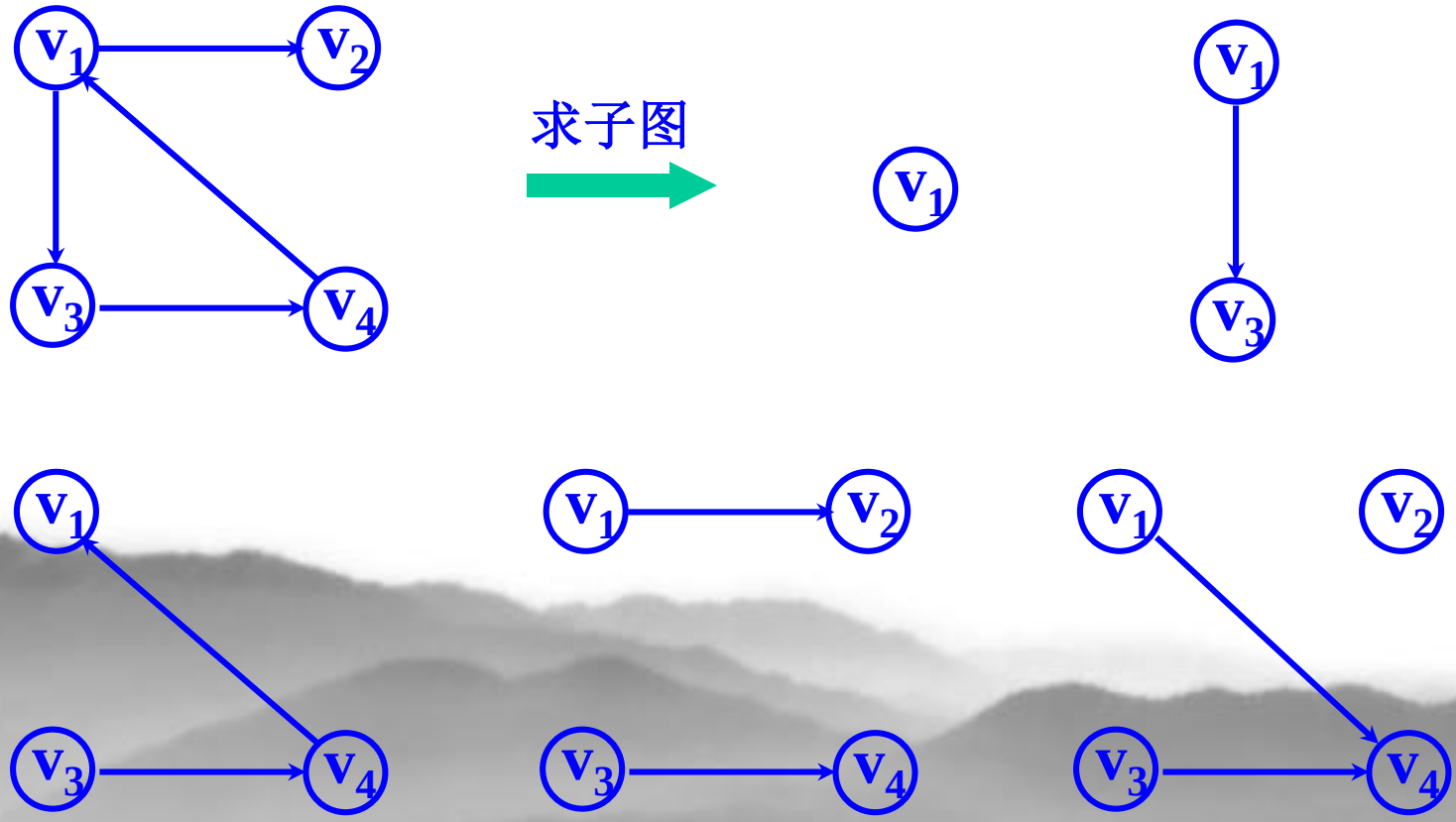


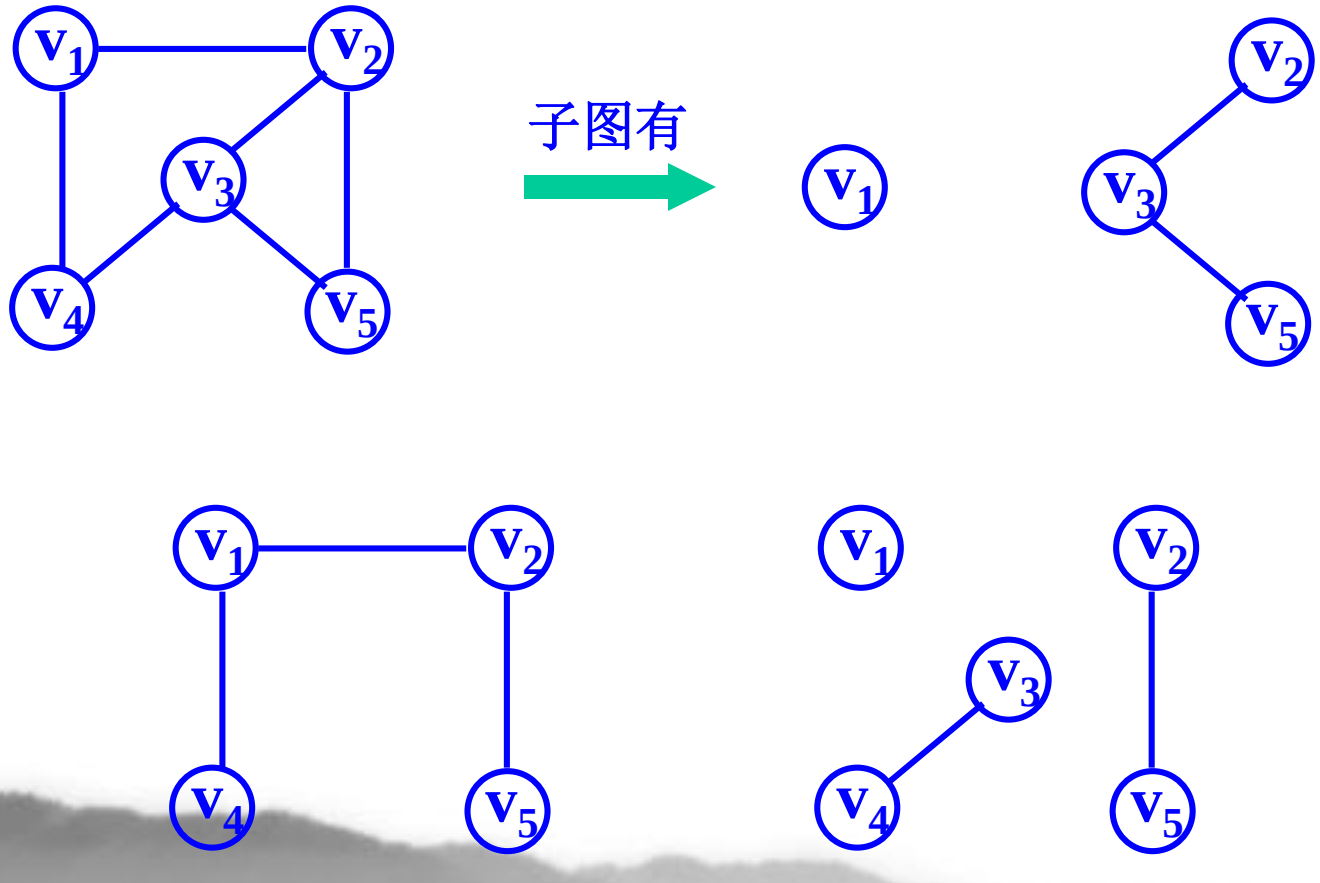
有很少($e<n\log n$)条边或弧的图称为稀疏图。

反之称为稠密图。

子图

假设有两个图 G=(V, E) 和 G'=(V', E')，如果 V' ⊆ V，且 E' ⊆ E，则称 G' 为 G 的子图。




v1
v2
v3
v4
v5
子图有
v1
v2
v3
v5
v1
v2
v4
v5
v1
v3
v4
v2
v5

邻接与关联

对于无向图 G=(V, E)，如果边 (v, v’) ∈ E，则称顶点 v 和 v’ 互为邻接点，即 v 和 v’ 相邻接。

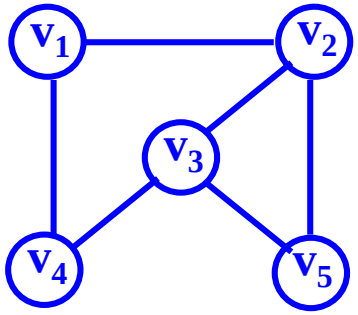
边 (v, v’) 依附于顶点 v 和 v’，或者说 (v, v’) 与顶点 v 和 v’ 相关联。

对于有向图 G=(V, E)，如果弧 <v, v’> ∈ E，则称顶点 v 邻接到顶点 v’，顶点 v’ 邻接自顶点 v 。

弧 <v, v’> 和顶点 v, v’ 相关联。

顶点的度

对于无向图，顶点 v 的度是和 v 相关联的边的数目，记做TD(v)。



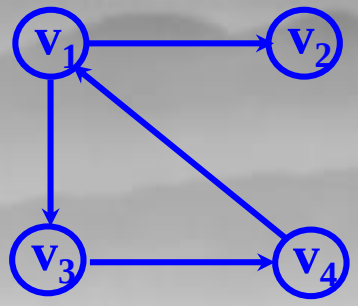
顶点 v_3 的度为 3

对于有向图，顶点 v 的度 TD(V) 分为两部分——出度、入度。

以顶点 v 为头的弧的数目称为 v 的入度，记为ID(v)；

以顶点 v 为尾的弧的数目称为 v 的出度，记为OD(v)；

顶点 v 的度为 TD(v) = ID(v) + OD(v)。



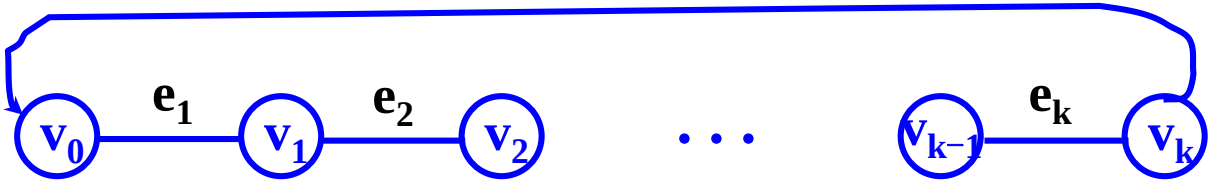
顶点 v_1 的出度为 2

顶点 v_1 的入度为 1

顶点 v_1 的度为 3

路径、链、简单路径、回路(环)、简单回路

*无向*图 G 中若存在一条有穷非空序列 $w = v_0\ e_1\ v_1\ e_2\ v_2 \cdots e_k\ v_k$，其中 v_i 和 e_i 分别为顶点和边，则称 w 是从顶点 v_0 到 v_k 的一条路径。



顶点 v_0 和 v_k 分别称为路径 w 的起点和终点。

路径的长度是路径上边的数目。

w 的长度为 k

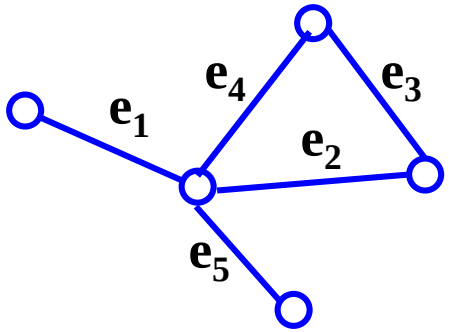
起点和终点相同的路径称为回路(环)。

若路径 w 的边 $e_1, e_2, \cdots, e_k$ 互不相同，则称 w 为链。

若路径 w 的顶点 $v_0, v_1, \cdots, v_k$ 互不相同，则称 w 为简单路径。

链是否为简单路径? 不一定

简单路径是否为链? 一定



仅起点和终点相同的简单路径称为简单回路(简单环)。

*有向*图 G 中若存在一条有穷非空序列 $w = v_0\ e_1\ v_1\ e_2\ v_2 \cdots e_k\ v_k$，其中 v_i 和 e_i 分别为顶点和弧，则称 w 是从顶点 v_0 到 v_k 的一条路径。

链

简单路径

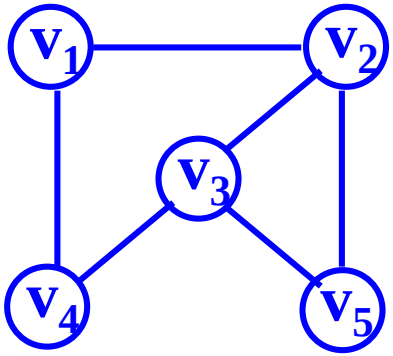
回路

简单回路

连通、连通图、连通分量、强连通图、强连通分量

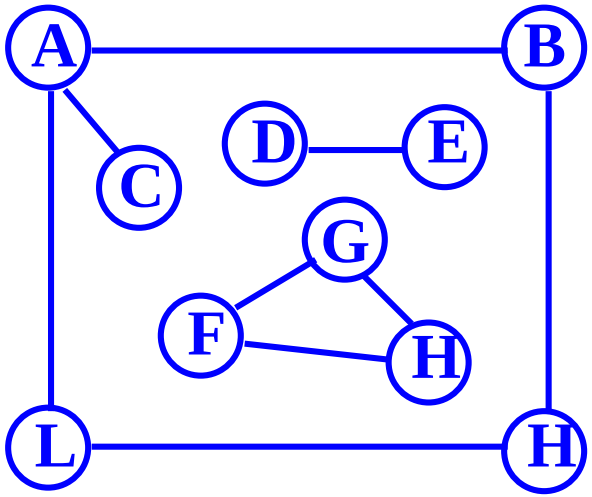
无向图G，如果从顶点 v 到顶点 v’ 有路径，则称 v 和 v’ 是连通的。

如果对于无向图 G 中任意两个顶点 v_i，$v_j \in V$，v_i 和 v_j 都是连通的，则称 G 是连通图。



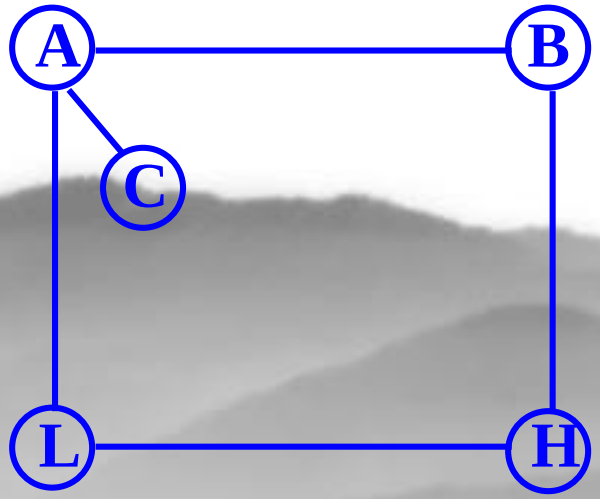
是否为连通图

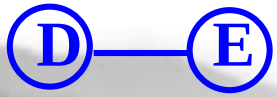
连通分量指的是无向图中的极大连通子图。

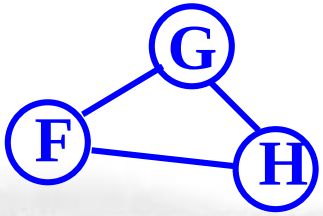


非连通图

连通分量为

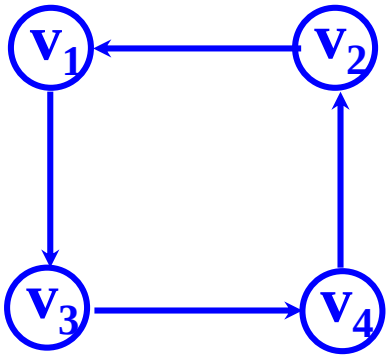






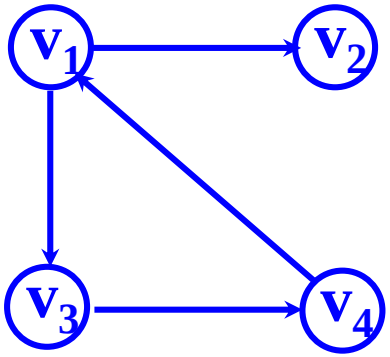
有向图G，如果从顶点 v 到顶点 v’ 有路径 或 从顶点 v’ 到顶点 v 有路径，则称 v 和 v’ 是连通的。

在有向图 G 中，如果对于每一对 $v_i, v_j \in V$，$v_i \neq v_j$，从 v_i 到 v_j 和 从 v_j 到 v_i 都存在路径，则称 G 是强连通图。

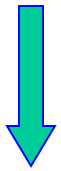


是否为强连通图

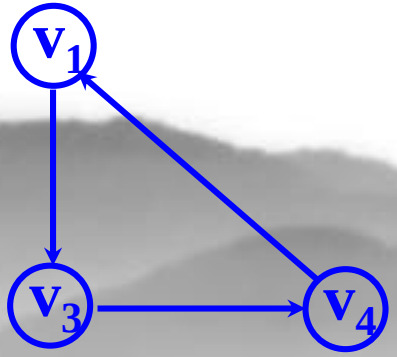
有向图中的极大强连通子图称作有向图的强连通分量。



非强连通图



强连通分量



生成树、生成森林 ——主要是对无向图而言

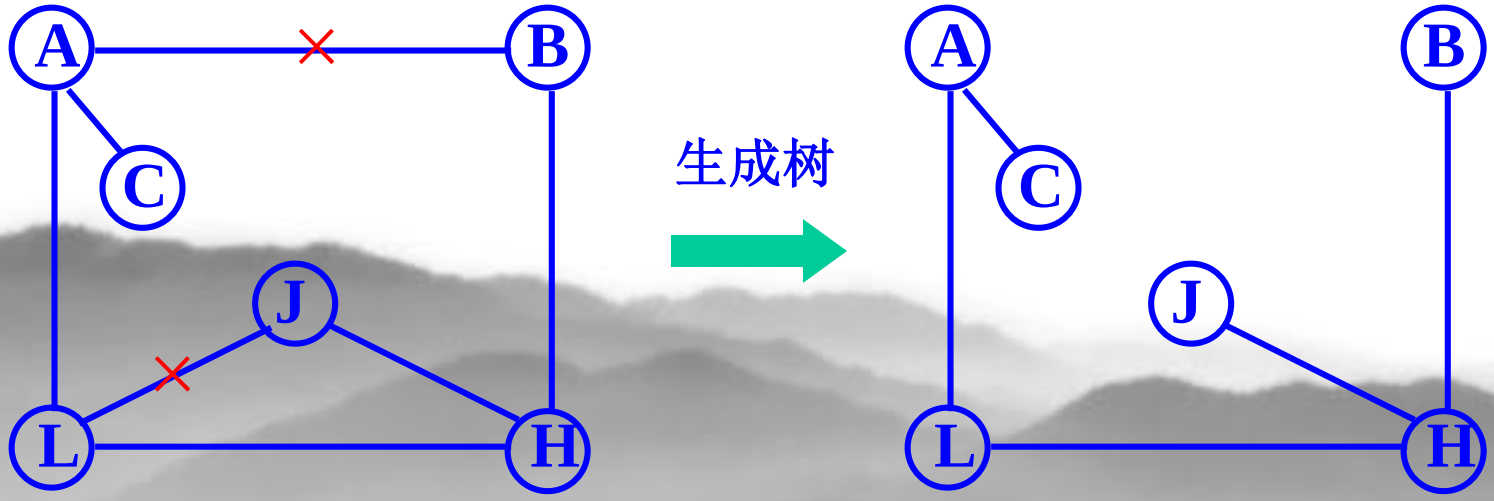
一个连通图 G 的一个包含所有顶点的极小连通子图 T 是

(1) T 包含 G 的所有顶点 n 个

(2) T 为连通子图

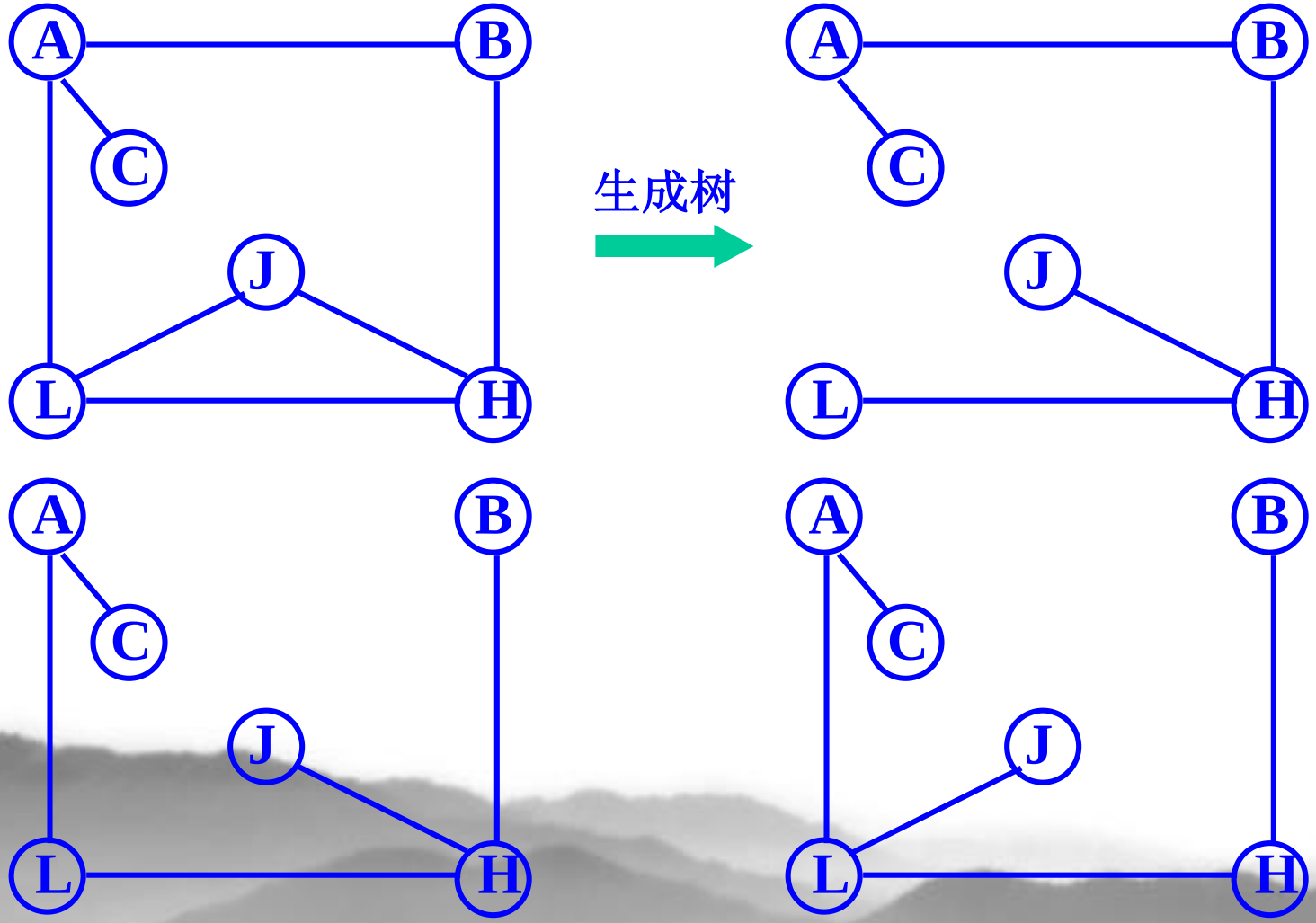
(3) T 包含的边数最少

T 是一棵有 n 个顶点，n−1 条边的生成树。



性质：一个有n个顶点的连通图的生成树有且仅有n−1条边。

性质：一个连通图的生成树并不唯一



删除环中的任一条边

7.2 图的存储结构

顺序存储　　如何表达顶点之间存在的联系？

邻接矩阵

链式存储　　多重链表，如何设计结点结构？

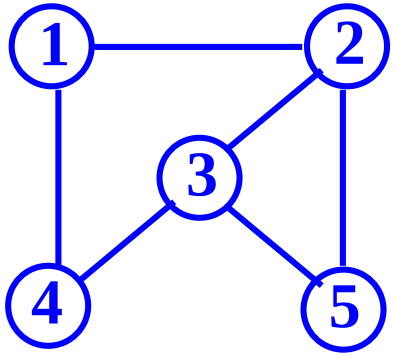
邻接表

十字链表

邻接多重表

习惯性约定

为每一个顶点设定一个序号，表示“顶点在图中的位置”。



7.2.1 数组表示法(邻接矩阵)

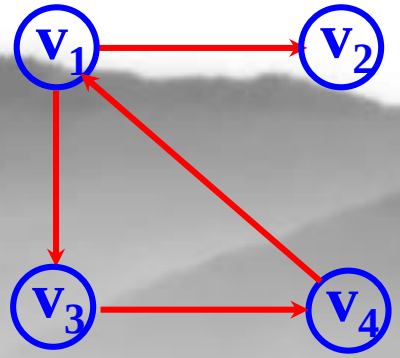
设图 G = (V ， E) 具有 n(n≥1) 个顶点 v_1 , v_2 ， ‥, v_n 和 m 条边或弧 e_1 , e_2 ， ‥, e_m ， 则 G 的邻接矩阵是 n×n 阶矩阵，记为 A(G) 。

邻接矩阵存放 n 个顶点信息和 n^2 条边或弧信息。

其每一个元素 a_{ij} 定义为:

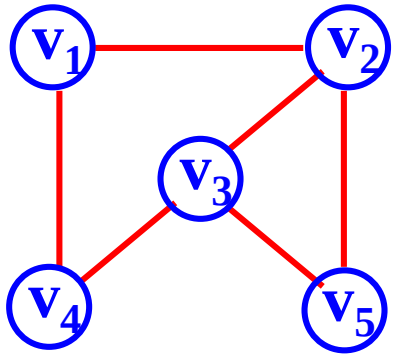
$$a_{ij} = \begin{cases} 0 & \text{顶点 } v_i \text{ 与 } v_j \text{ 不相邻接} \\ 1 & \text{顶点 } v_i \text{ 与 } v_j \text{ 相邻接} \end{cases}$$

例有向图 G



$$A(G) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

例无向图 G



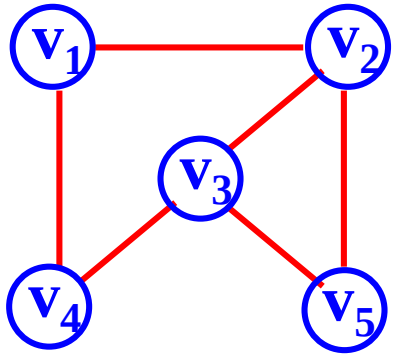
$$A(G)=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

优点：

1. 容易判断任意两个顶点之间是否有边或弧。

2. 容易求取各个顶点的度。

例无向图 G

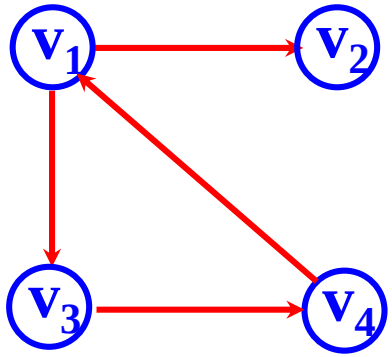


$$A(G)=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

无向图，顶点 v_i 的度是邻接矩阵中第 i 行或第 i 列的元素之和。

例 G_1中，v_1 的度为 2。

例有向图 G



$$A(G)=\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

有向图，顶点 v_i 的出度是邻接矩阵中第 i 行的元素之和。

顶点 v_i 的入度是邻接矩阵中第 i 列的元素之和。

例 v_1 的出度为 2；入度为 1。

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}\begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

无向图

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array}\begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

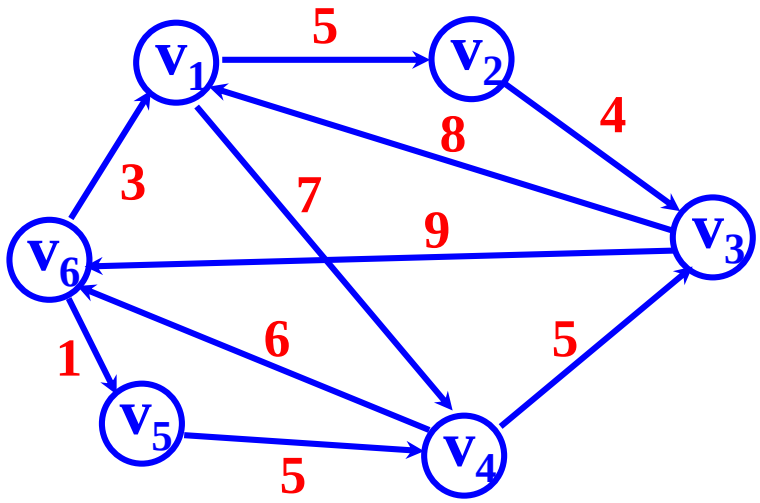
有向图

无向图的邻接矩阵都是对称矩阵。

有向图的邻接矩阵一般不对称。

故无向图可以采用压缩存储方式

带权图(网) 的邻接矩阵



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{bmatrix} \infty & 5 & \infty & 7 & \infty & \infty \\ \infty & \infty & 4 & \infty & \infty & \infty \\ 8 & \infty & \infty & \infty & \infty & 9 \\ \infty & \infty & 5 & \infty & \infty & 6 \\ \infty & \infty & \infty & 5 & \infty & \infty \\ 3 & \infty & \infty & \infty & 1 & \infty \end{bmatrix} \end{array}$$

$$a_{ij} = \begin{cases} w_{ij} & \text{顶点 } v_i \text{ 与 } v_j \text{ 相邻接} \\ \infty & \text{顶点 } v_i \text{ 与 } v_j \text{ 不相邻接} \end{cases}$$

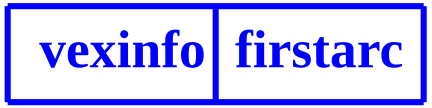
```
typedef struct ArcCell { // 弧的定义
    VRType  adj;    // VRType是顶点关系类型。对无权图，用1或0
                    // 表示相邻否；对带权图，则为权值类型。
    InfoType  *info;  // 该弧相关信息的指针
} ArcCell,   AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef struct { // 图的定义
    VertexType   vexs[MAX_VERTEX_NUM];     // 顶点向量
    AdjMatrix    arcs;          // 邻接矩阵
     int  vexnum, arcnum;   // 顶点数，弧数
     GraphKind   kind;       // 图的种类标志{DG / DN / UDG / UDN}
} MGraph;
```

7.2.2 邻接表

对图中每一个顶点建立一个单链表，指示与该顶点关联的边或出弧。

头结点



vexinfo ： 顶点的信息

firstarc ： 第一条关联边结点

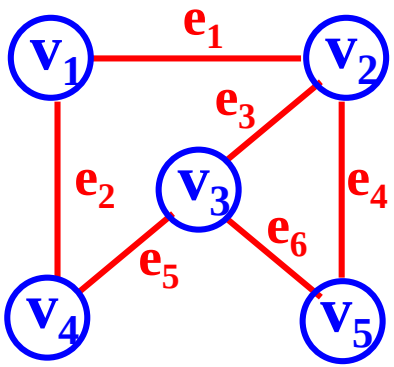
表结点

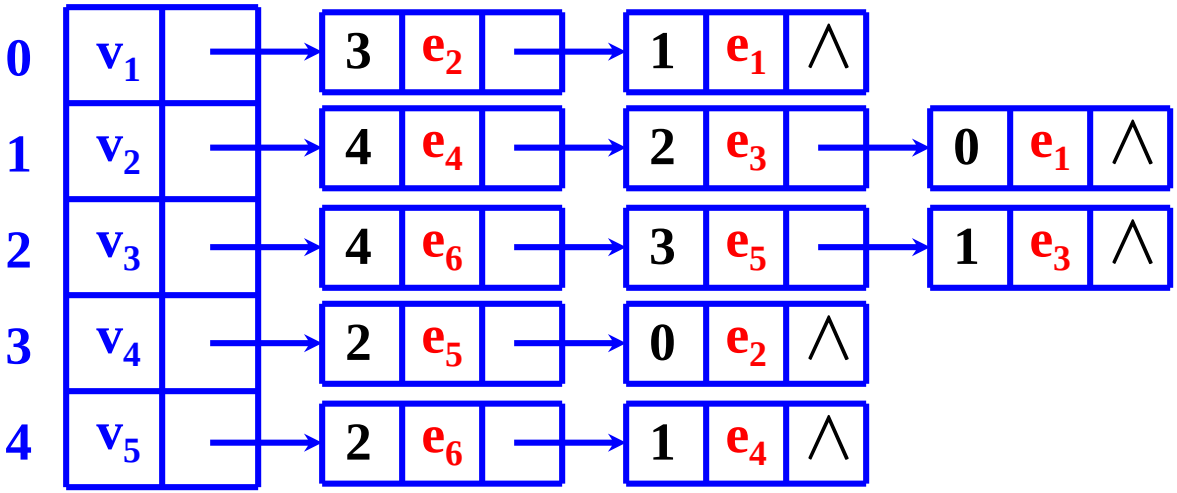
adjvex ： 邻接顶点位置

arcinfo ： 边的信息

nextarc ： 下一条关联边结点

例无向图 G





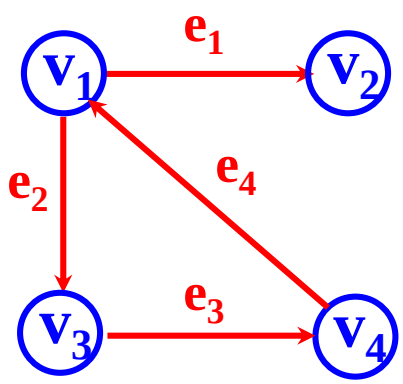
如何获取顶点的度？

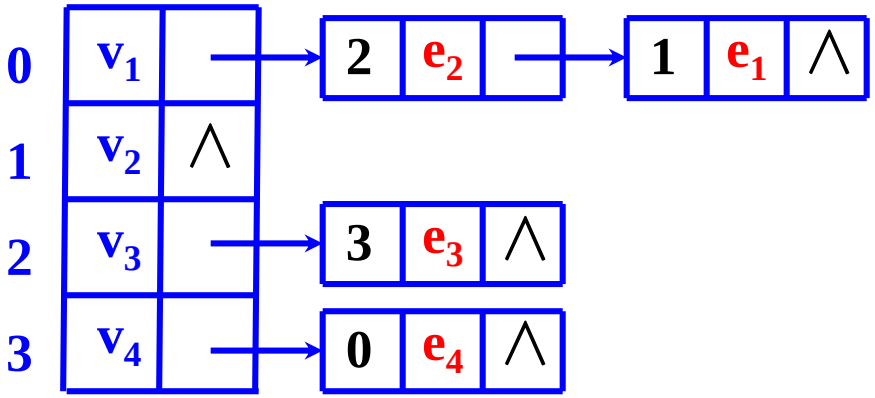
顶点 v_i 的度为第 i 条链表中的结点数。

需要多少存储空间？

n + 2e

例有向图 G





如何获取顶点的度？

顶点 v_i 的出度为第 i 条链表中的结点数。

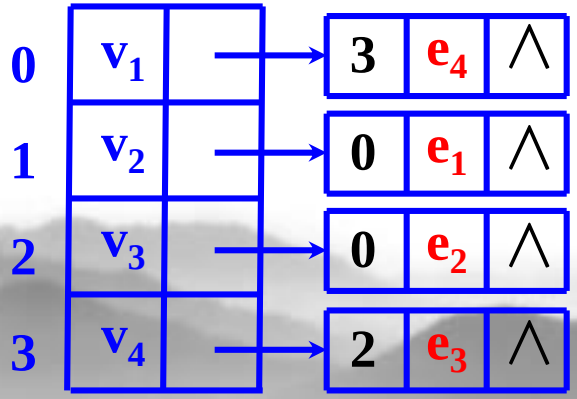
需要多少存储空间？

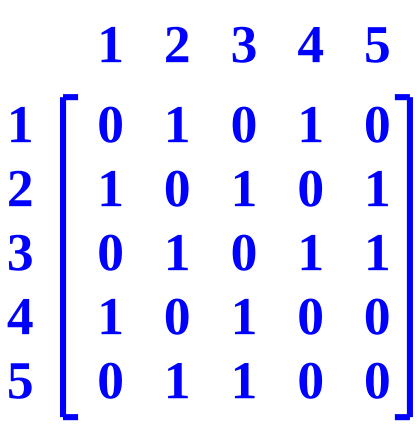
n + e

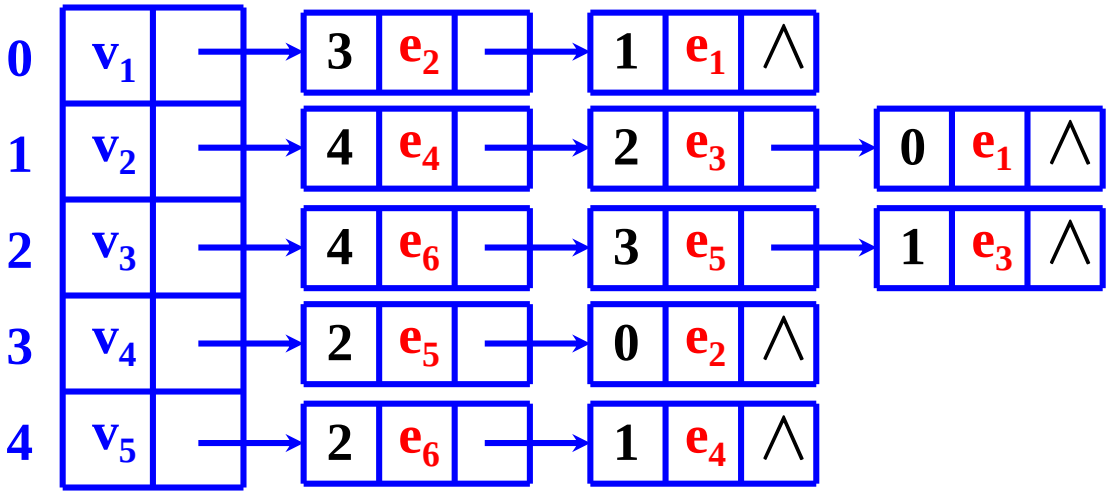
最终需要多少存储空间？

2n + 2e

为了方便求顶点的入度，引入逆邻接表







复杂度---邻接矩阵 vs. 邻接表

存储空间	邻接表
求顶点的度	一样
求顶点的邻接顶点	一样
判断两个顶点是否关联	邻接矩阵

表结点

adjvex	nextarc	info

头结点

data	firstarc

```
typedef struct ArcNode {
  int                adjvex;      // 该弧所指向的顶点的位置
  struct ArcNode  *nextarc;    // 指向下一条弧的指针
  InfoType           *info;       // 该弧相关信息的指针
} ArcNode;
typedef struct VNode {
  VertexType  data;       // 顶点信息
  ArcNode      *firstarc; // 指向第一条依附该顶点的弧
} VNode, AdjList[MAX_VERTEX_NUM];
typedef struct {
    AdjList  vertices;
    int         vexnum, arcnum;
    int         kind;          // 图的种类标志
  } ALGraph;
```

7.2.3 十字链表

将有向图的邻接表和逆邻接表结合在一起，就得到了有向图的另一种链式存储结构----十字链表。

顶点结点

data	firstin	firstout

弧结点

tailvex	headvex	info	hlink	tlink

data: 顶点的信息

firstin ： 第一条入弧的弧结点

firstout ： 第一条出弧的弧结点

tailvex ： 弧尾顶点位置

headvex ： 弧头顶点位置

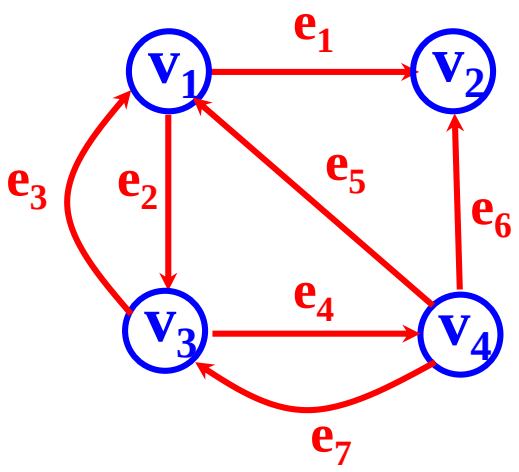
info ： 弧的信息

hlink ： 弧头相同的下一条弧

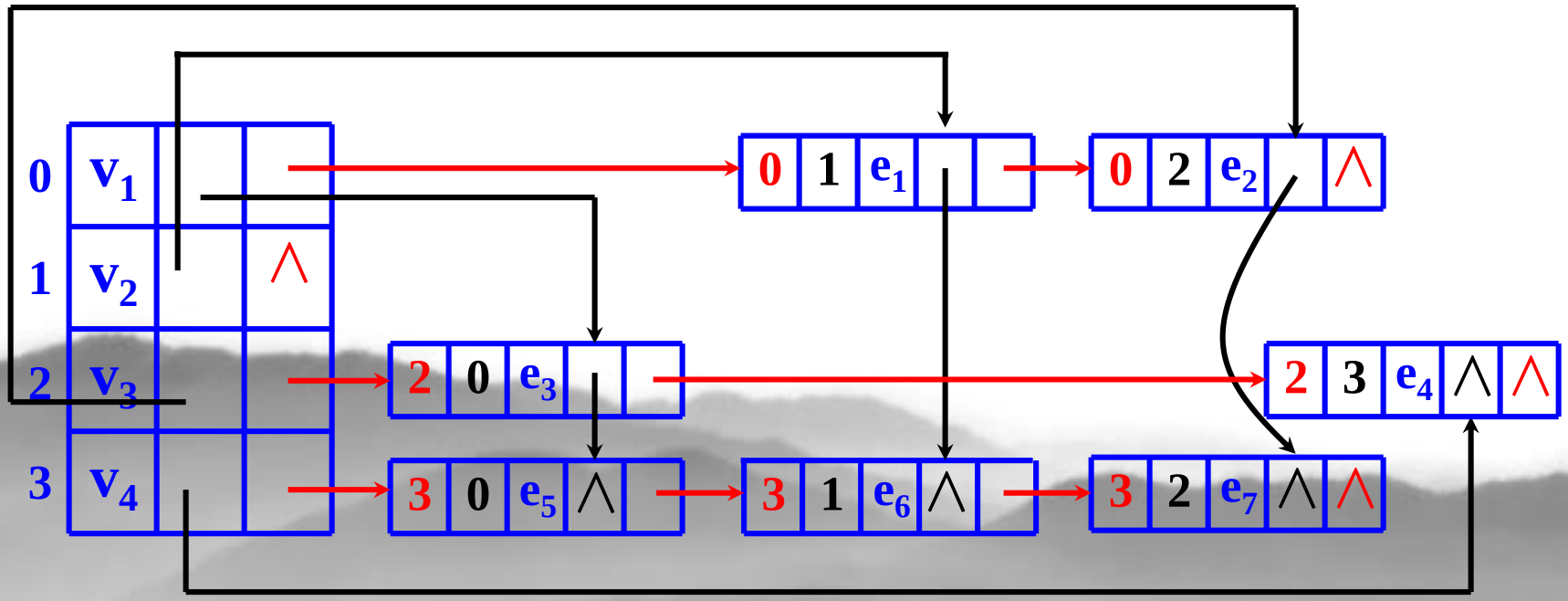
tlink ： 弧尾相同的下一条弧

入度

出度



	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	1	0	0	1
3	1	1	1	0



```
typedef struct ArcBox { // 弧的结构表示
    int                 tailvex, headvex; // 弧的尾和头顶点
    struct ArcBox  *hlink, *tlink;      // 弧头/弧尾相同的弧链
    InfoType           *info;
} ArcBox;
typedef struct VexNode { // 顶点的结构表示
    VertexType  data;
    ArcBox         *firstin, *firstout;    // 顶点的第一条入弧和出弧
} VexNode;
typedef struct {
  VexNode  xlist[MAX_VERTEX_NUM];  // 表头向量
  int              vexnum, arcnum;   //有向图的当前顶点数和弧数
} OLGraph;
```

7.2.4 邻接多重表

邻接表是无向图的一种很有效的存储结构，在邻接表中容易求得顶点和边的各种信息；

但在邻接表中，每一条边都有两个结点表示，因此在某些对边进行的操作(例如对搜索过的边做标记)中就需要对每一条边处理两遍；

故引入邻接多重表实现无向图的存储结构。

邻接多重表的结构与十字链表相似

顶点

data ： 顶点的信息

firstedge ： 第一条关联边

边

mark ： 标志域，是否遍历过

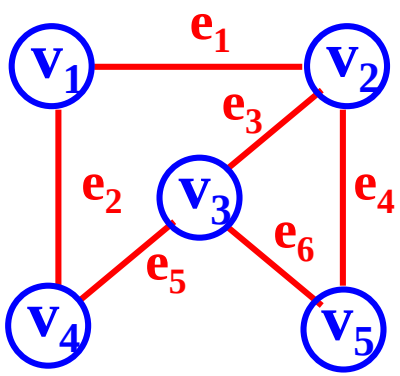
info ： 边的信息

ivex ： 边的一个顶点

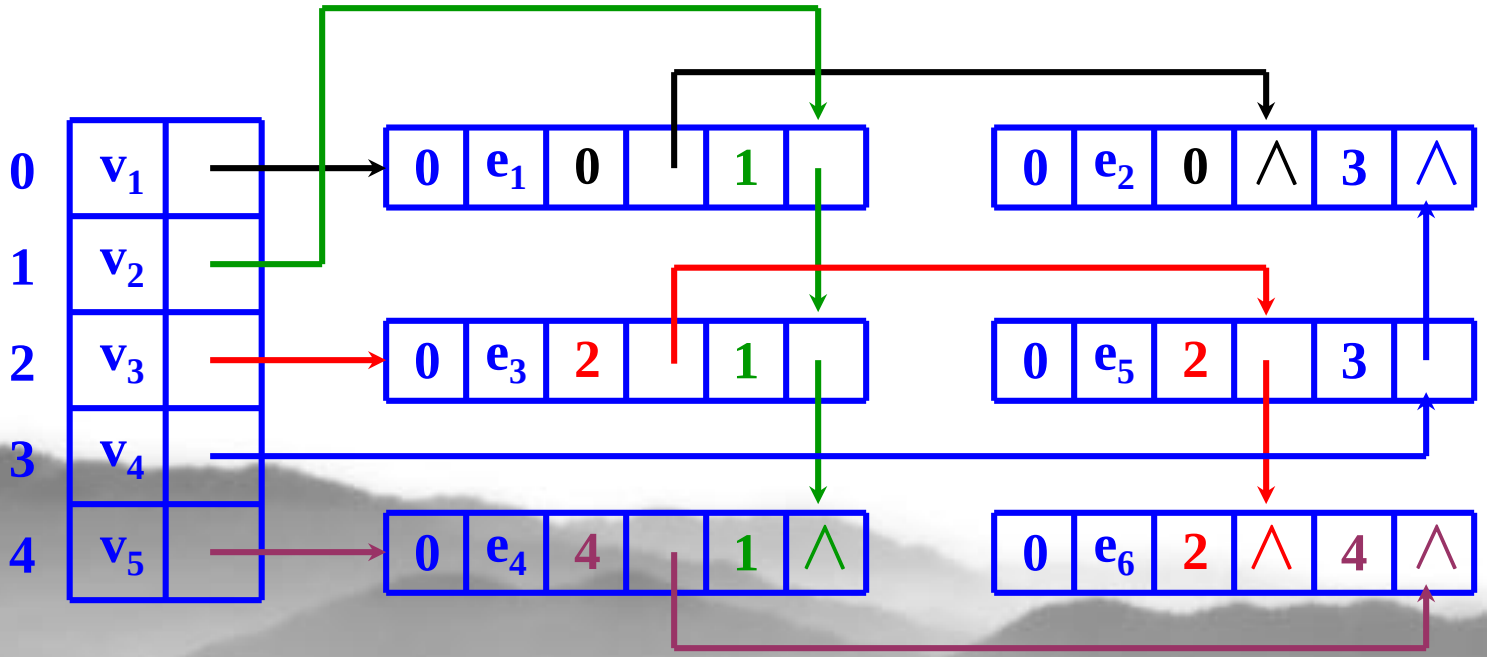
ilink ： 顶点 i 的下一条关联边

jvex ： 边的另一个顶点

jlink ： 顶点 j 的下一条关联边



	0	1	2	3	4
0	0	1	0	1	0
1	1	0	1	0	1
2	0	1	0	1	1
3	1	0	1	0	0
4	0	1	1	0	0



```
typedef struct Ebox {
    VisitIf          mark;          // 访问标记
    InfoType        *info;          // 边信息指针
    int              ivex, jvex;    // 边依附的两个顶点的位置
    struct EBox  *ilink, *jlink;
} EBox;
typedef struct VexBox {
  VertexType  data;
  EBox          *firstedge; // 指向第一条依附该顶点的边
} VexBox;
typedef struct {  // 邻接多重表
   VexBox  adjmulist[MAX_VERTEX_NUM];
    int          vexnum, edgenum;
} AMLGraph;
```

7.3 图的遍历

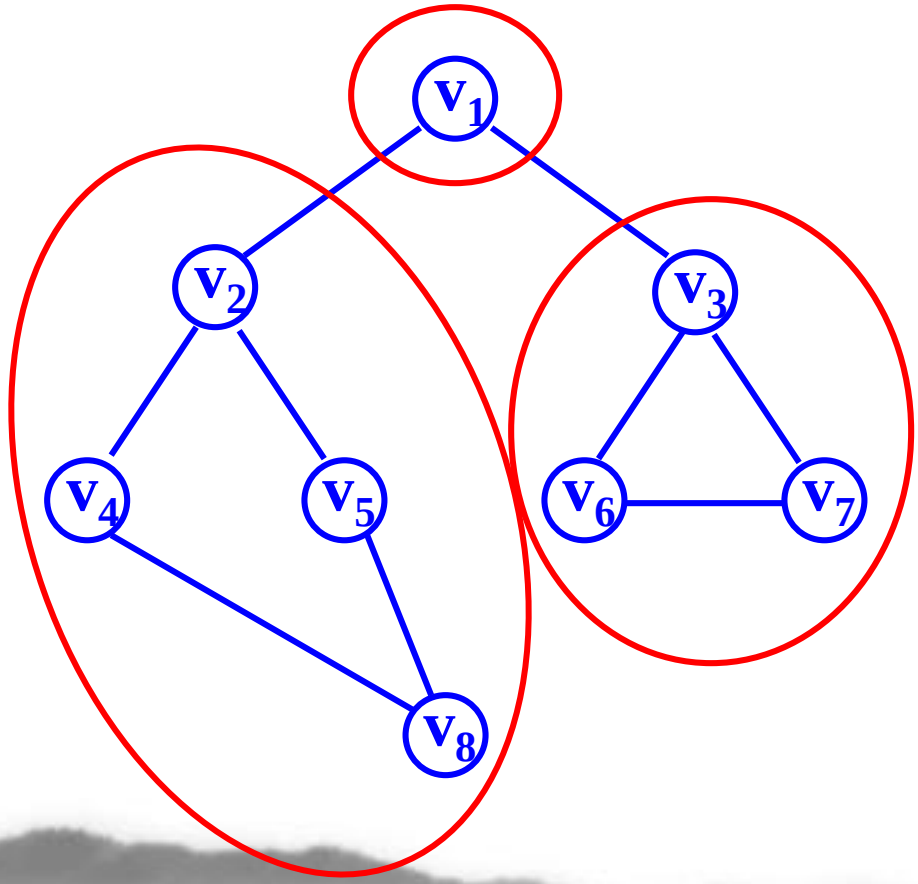
与树的遍历类似，如果从图中某一顶点出发访遍图中所有顶点，且使每一个顶点仅被访问一次，这一过程称为图的遍历。

图的遍历算法是求解图的连通性问题、拓扑排序和求关键路径等算法的基础。

通常有两条遍历图的路径：深度优先搜索、广度优先搜索。

图的遍历相对复杂，为了避免同一个顶点被访问多次，增设一个辅助的布尔数组 **visited[0 . . n−1]** 指示顶点是否已被访问过。

7.3.1 深度优先搜索



图可分为三部分:

基结点

第一个邻接结点导出的子图

其它邻接顶点导出的子图

深度优先搜索是类似于树的一种先序遍历

深度优先搜索顺序: v_1 v_2 v_4 v_8 v_5 v_3 v_6 v_7

算法描述:

1. 从图中某个顶点 v 出发，访问此顶点；

2. 然后依次从 v 的未被访问的邻接点出发进行深度优先遍历；

3. 直至图中所有和 v 有路径相通的顶点都被访问到。

4. 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做起始点，重复上述过程，直至图中所有顶点都被访问到。

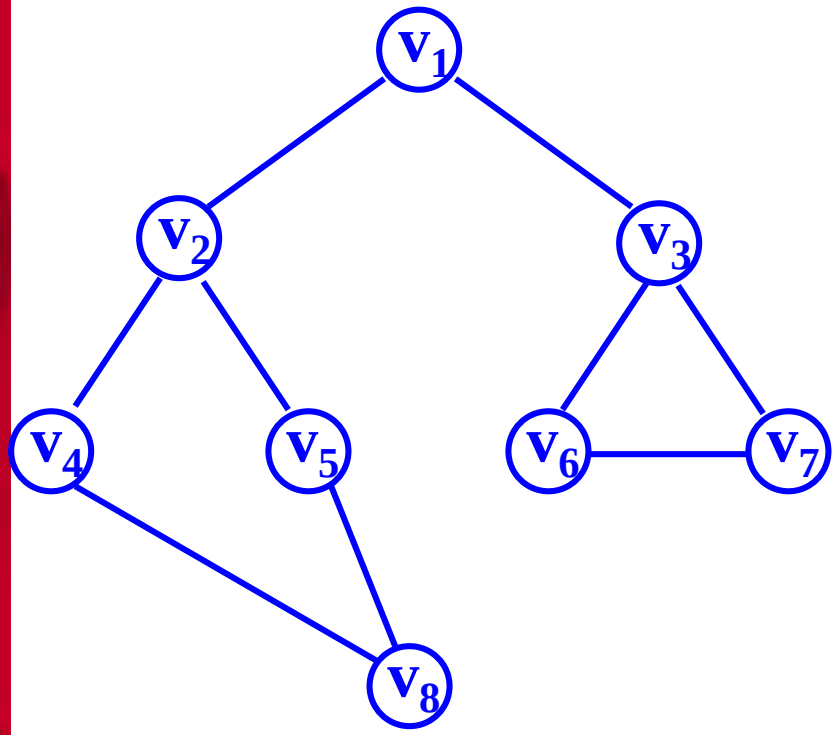
深度优先搜索算法是一个递归过程吗？

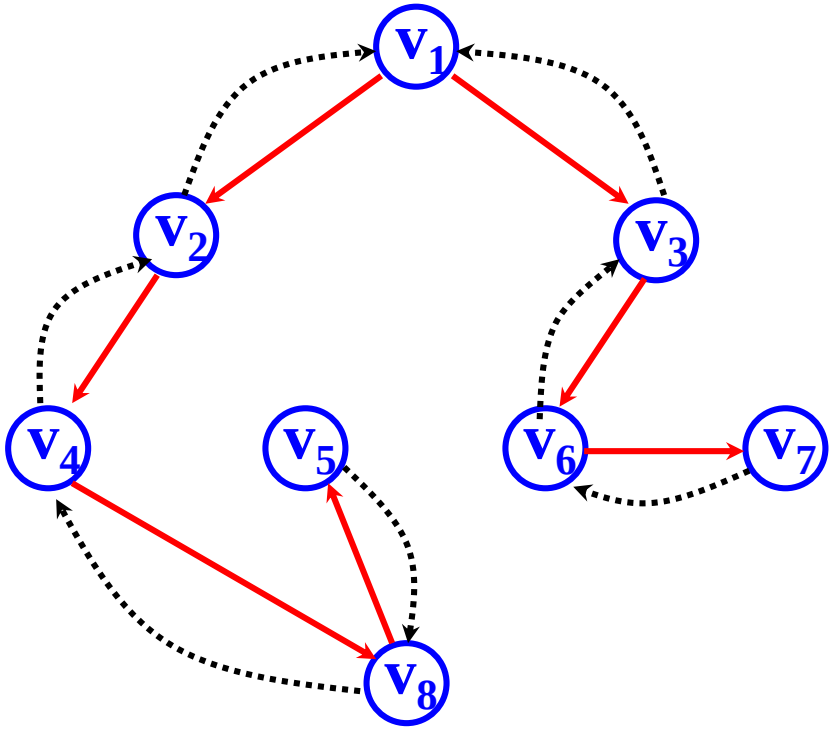
是否可以利用此算法来求解图的连通性问题？

```
Boolean visited[MAX];
Status (*VisitFunc)(int v);

void DFSTraverse(Graph G,  Status (*Visit)(int v)) {  // 图 G 深度优先遍历
  VisitFunc = Visit;
  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) {
   // 从顶点v出发，深度优先搜索遍历连通图 G
    visited[v] = TRUE;   VisitFunc(v);
    for(w=FirstAdjVex(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
} // DFS
```

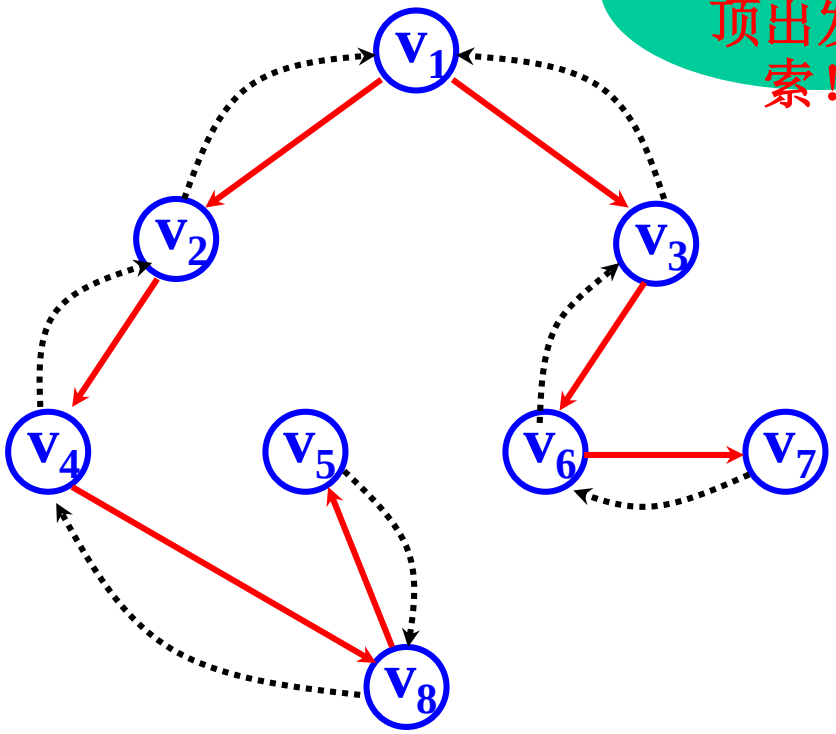
过程分析



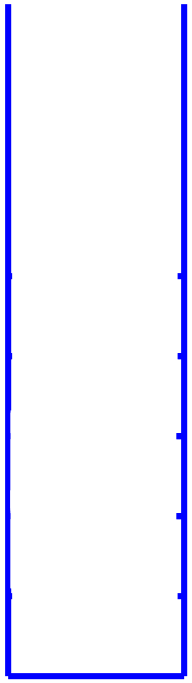


深度优先搜索顺序：　v_1 v_2 v_4 v_8 v_5 v_3 v_6 v_7

栈实现深度优先搜索



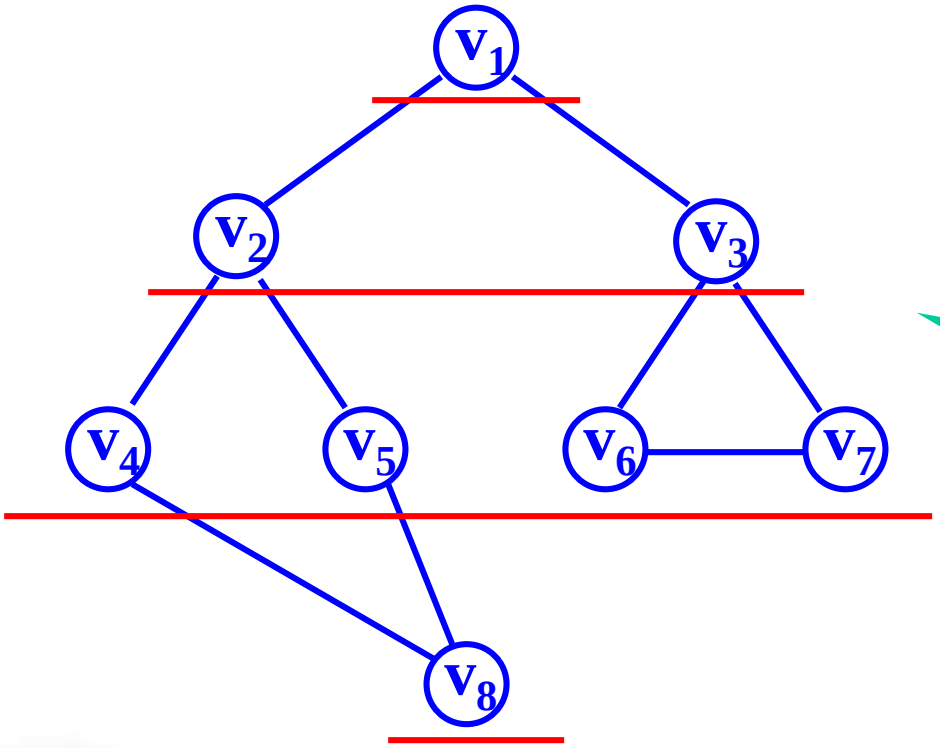
总是从栈顶出发搜索！



深度优先搜索顺序：　v_1 v_2 v_4 v_8 v_5 v_3 v_6 v_7

```
void   DFSTraverse(Graph  G, int  v)   {
    initstack(S); visited[v] = TRUE; Push(S , v); printf(v);
    while  ( !StackEmpty(S))    {
        Gettop(S , v) ;
        for  ( w=FirstAdjVex(G,v); w>=0; w=NextAdjVex(G, v, w) )
            if   ( !visited[w] )    {
                visited[w] = TRUE; Push(S , w); printf(w) ;
                break;
            }
        if (w<0) Pop(S);
    }
}
```

7.3.2 广度优先搜索



把图人为的分层，

按层遍历。

只有父辈结点
被访问后才会
访问子孙结点！

深度优先搜索类似于树的层次遍历，

广度优先搜索顺序：　v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8

算法描述：

1. 从图中某个顶点 v 出发，访问此顶点；

2. 然后依次访问 v 的各个未曾访问的邻接点；

3. 然后依次从这些邻接点出发再依次访问它们的邻接点；

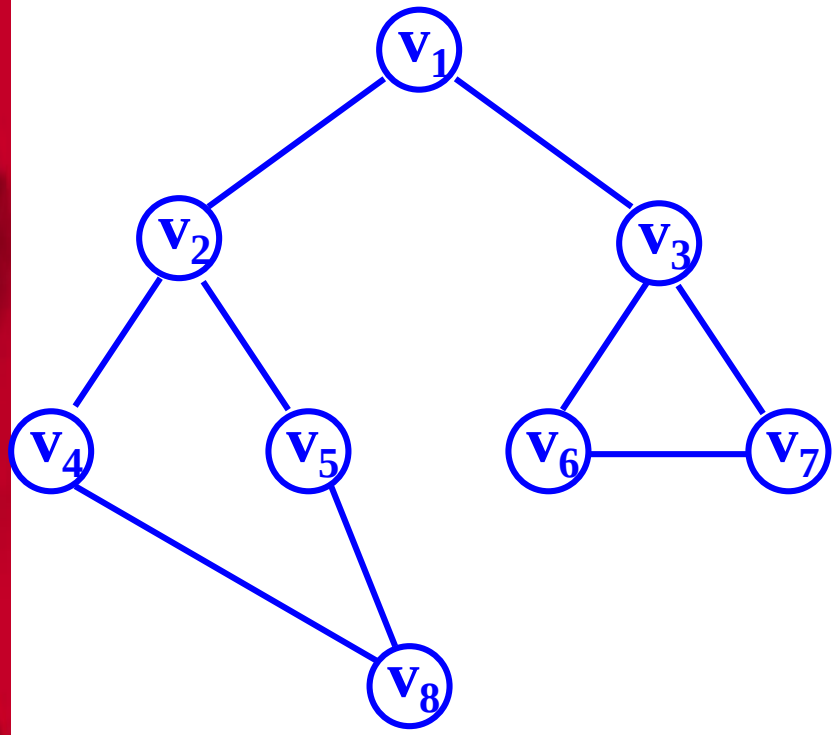
4. 直至图中所有和 v 有路径相通的顶点都被访问到。

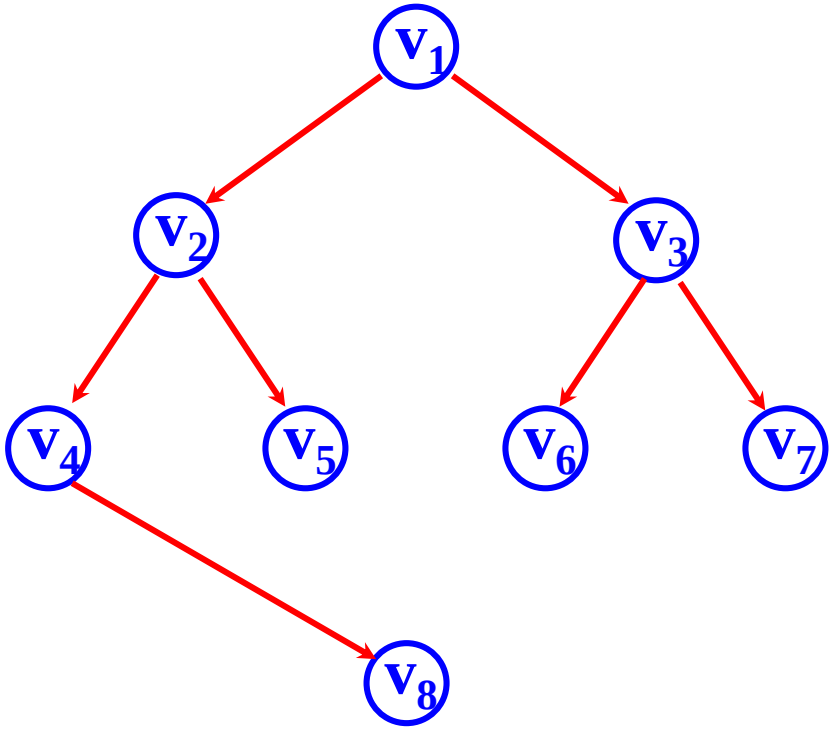
5. 若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点做起始点，重复上述过程，直至图中所有顶点都被访问到。

广度优先搜索算法是一个递归过程吗？

是否可以利用此算法来求解图的连通性问题？

过程分析





广度优先搜索顺序：　v_1　v_2　v_3　v_4　v_5　v_6　v_7　v_8

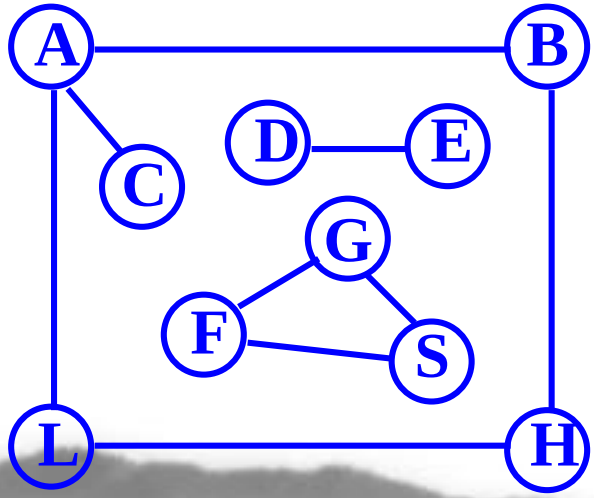
队列实现广度优先搜索算法

```
void BFSTraverse(Graph G, Status (*Visit)(int v))
{
  for (v=0; v<G.vexnum; ++v)   visited[v] = FALSE;  //初始化访问标志
  InitQueue(Q);       // 置空的辅助队列Q
  for ( v=0;  v<G.vexnum;  ++v )
    if (!visited[v]) {          // v 尚未访问
          visited[v] = TRUE;  Visit(v);    // 访问v
          EnQueue(Q, v);              // v入队列
          while (!QueueEmpty(Q))  {
            DeQueue(Q, u);          // 队头元素出队并置为u
            for(w=FirstAdjVex(G, u); w>=0;  w=NextAdjVex(G,u,w))
              if (!visited[w])  {
                visited[w]=TRUE;  Visit(w);
                EnQueue(Q, w); // 访问的顶点w入队列
              } // if
          } // while
   }  // if
 } // BFSTraverse
```

7.4 图的连通性问题

7.4.1 无向图的连通分量

利用 DFS 或 BFS 获取连通分量



DFS

A B H L C

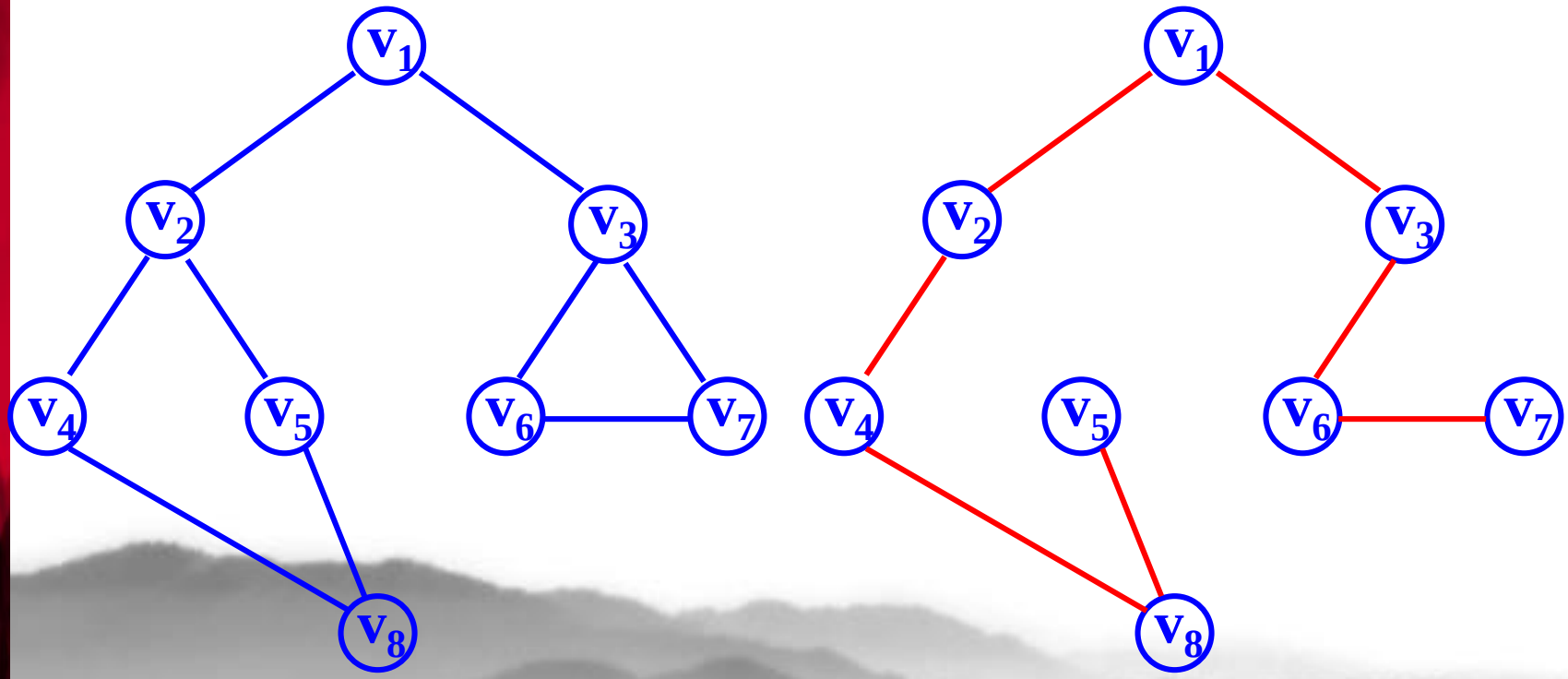
D E

F G S

7.4.2 无向图的生成树

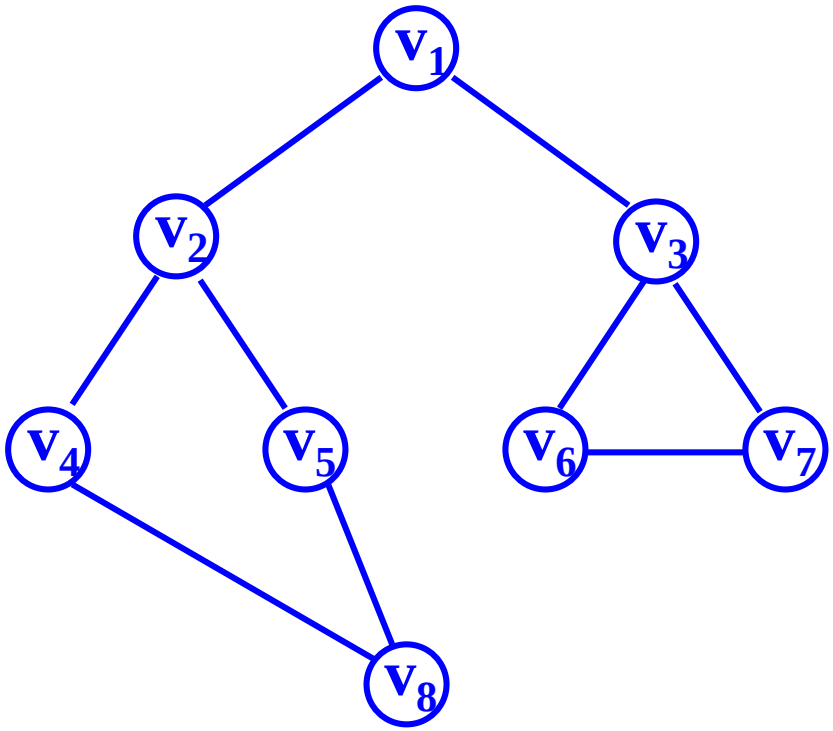
利用 DFS 或 BFS 获取生成树

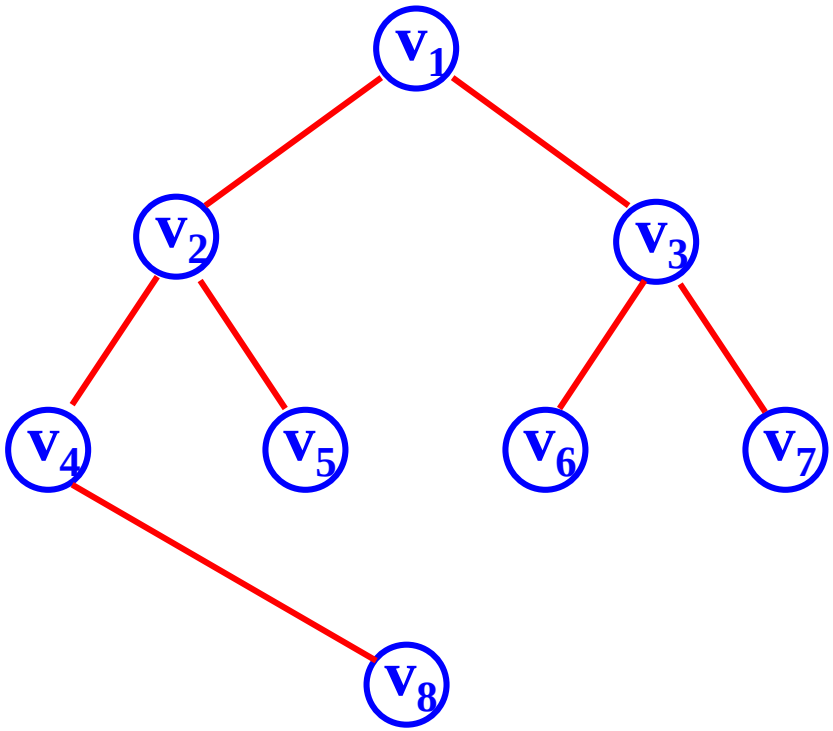
例，DFS



DFS生成树

例，BFS



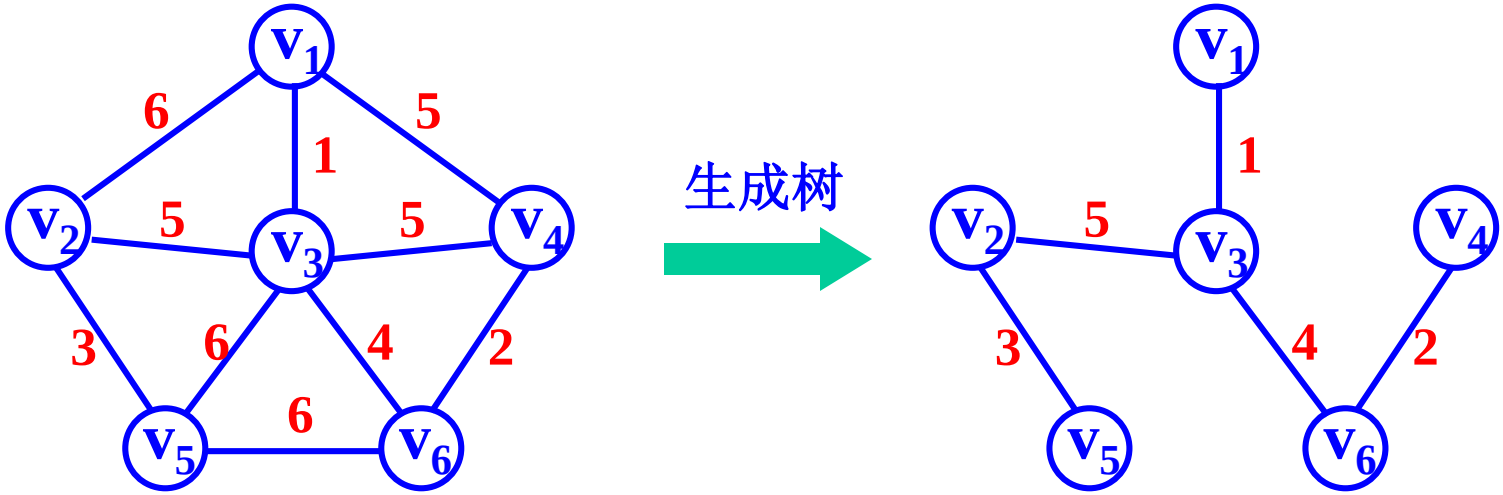


BFS生成树

7.4.3 最小生成树(Minimum Cost Spanning Tree, MST)

一个无向图可以对应多个生成树。

一个带权无向图(无向网)同样可以对应多个生成树。

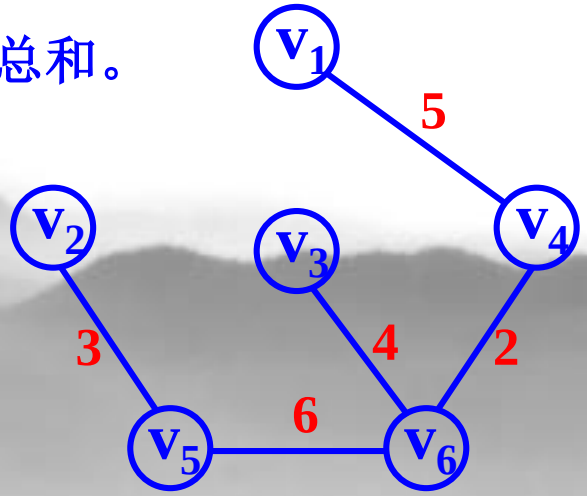


一棵生成树的代价定义为树上各边的权之总和。

代价最小的生成树称为最小生成树。

实际价值？

如何求取最小生成树？



讨论

- 如何获得一个图的最小生成树？

Prim 算法

思想：

N = (V , E) 是具有 n 个顶点的连通网，设 U 是最小生成树中顶点的集合，设 TE 是最小生成树中边的集合；

初始，U = { u_1 }，TE = { }，

重复执行:

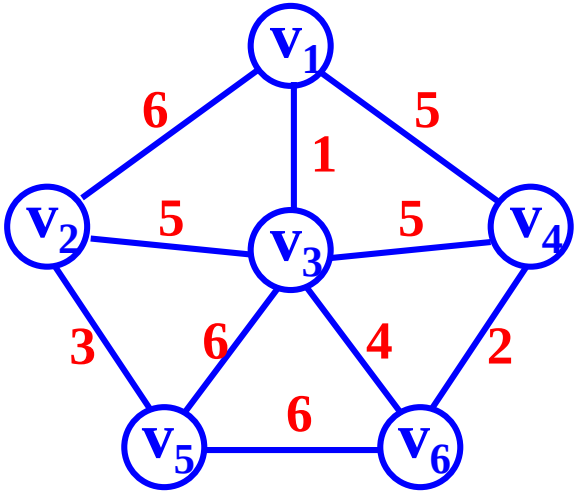
在所有 u∈U，v∈V−U 的边 (u , v) 中寻找代价最小的边(u', v')，并纳入集合 TE 中；

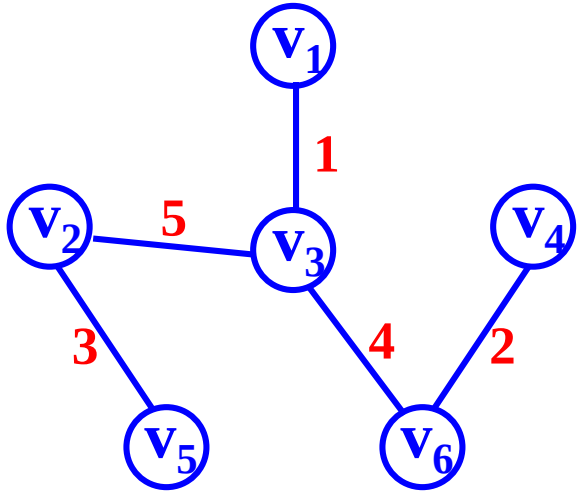
同时将 v' 纳入集合 U 中；

直至 U = V 为止。

集合 TE 中必有 n−1 条边。

例，





初始：$U=\{v_1\}$，$V-U=\{v_2, v_3, v_4, v_5, v_6\}$　　$TE=\{\}$

$U=\{v_1, v_3\}$，$V-U=\{v_2, v_4, v_5, v_6\}$　　$<v_1, v_3>$

$U=\{v_1, v_3, v_6\}$，$V-U=\{v_2, v_4, v_5\}$　　$<v_3, v_6>$

$U=\{v_1, v_3, v_4, v_6\}$，$V-U=\{v_2, v_5\}$　　$<v_6, v_4>$

$U=\{v_1, v_2, v_3, v_4, v_6\}$，$V-U=\{v_5\}$　　$<v_3, v_2>$

$U=\{v_1, v_2, v_3, v_4, v_5, v_6\}$，$V-U=\{\}$　　$<v_2, v_5>$

重点：边一定存在于 U 与 V−U 之间。

算法:

初始化，U = { v_1 }，TE = {} ；

记录 v_1 到其它各顶点的权值； //循环

```
while ( U != V )  {
    寻求权值最小边 ( u' , v' )，满足 u'∈U && v'∈V−U；//循环
    TE = TE + { < u' , v' > } ;
    U = U + { v' } ;
    记录新顶点 v' 到其它各顶点的权值；  //循环
}
return  OK ;
```

```
void MiniSpanTree_PRIM(MGraph G, VertexType u) {
  //用普里姆算法从顶点u出发构造网G的最小生成树
  //struct {
  //    VertexType  adjvex;  // U集中的顶点序号
  //    VRType     lowcost;  // 边的权值
  //} closedge[MAX_VERTEX_NUM];  //  顶点到U所形成的子树的距离/最短边
  k = LocateVex ( G, u );
  for ( j=0; j<G.vexnum; ++j )  // 辅助数组初始化
     if (j!=k)
        closedge[j] = { u, G.arcs[k][j].adj };  // 图存为邻接矩阵
  closedge[k].lowcost = 0;      // 初始，U = {u}
  for (i=0; i<G.vexnum; ++i) {
        k = minimum(closedge);   // 求出加入生成树的下一个顶点k,
                                 //  closedge[k].lowcost>0
        printf(closedge[k].adjvex, G.vexs[k]);  // 输出生成树上一条边
        closedge[k].lowcost = 0;              // 第k顶点并入U集，距离为0
        for (j=0; j<G.vexnum; ++j)        //修改其它顶点的最小边
          if (G.arcs[k][j].adj < closedge[j].lowcost)
            closedge[j] = { G.vexs[k], G.arcs[k][j].adj };
}
```

简单证明

- 分两步来证明：
- （1）Prim算法一定能得到一个生成树；
- （2）该生成树具有最小代价。

- Prim算法一定能得到一个生成树；

- （1）Prim算法每次引入一个新边，都恰好引入一个新节点：如果少于1个，则新加入的边的两个端点已经在树中，引入新边后就会形成回路；如果多于一个，即2个，则这条边的两个端点都不在树中，这条边与原来的树就独立了，不再构成一个新的树。
- 由于第一步中已直接引入了一个顶点，所以只需再引入n-1条边（即n-1个顶点）即可。
- 假设Prim算法引入边数小于n-1，就意味着还有剩余的点与已生成的树没有相连，且剩余的点中没有任何点可以和树中的点连通，而由于原图是连通的，所以不可能存在这种情况，因此Prim算法不可能在此之前结束。
- 因此Prim算法必能引入n-1条边，此时得到的树是原图的生成树。

- 该生成树具有最小代价：
  - 用prim算法得出的边分别为e1，e2，…, en；
    依次加入的点为p1，p2，…, pn；
    若不存在最小生成树包含e1，那么把e1加入任意一棵最小生成树，必然成环，并且在环上可以找到一条不小于e1的边，（因为成环了，所以环上的点必然至少链接了两条边，而e1是p1所链接的最小的边）删掉此边，得到一棵更优的生成树或者得到了一棵包含e1的最小生成树，矛盾。
  - 若包含e1的最小生成树都不包含e2，那么把e2加入其中一棵包含e1的最小生成树中，也会成环，并且在环中也能找到不小于e2的边（因为成环了，所以顶点1顶点2所形成的集合必然包含至少三条边，而e2是当中第二小的），同上也会产生矛盾。

Kruskal 克鲁斯卡尔算法

思想：

N = (V , E) 是 n 顶点的连通网，设 E 是连通网中边的集合；

构造最小生成树 N' = (V , TE)，TE 是最小生成树中边的集合，初始 TE = {} ；

重复执行:

选取 E 中权值最小的边 (u , v) ，

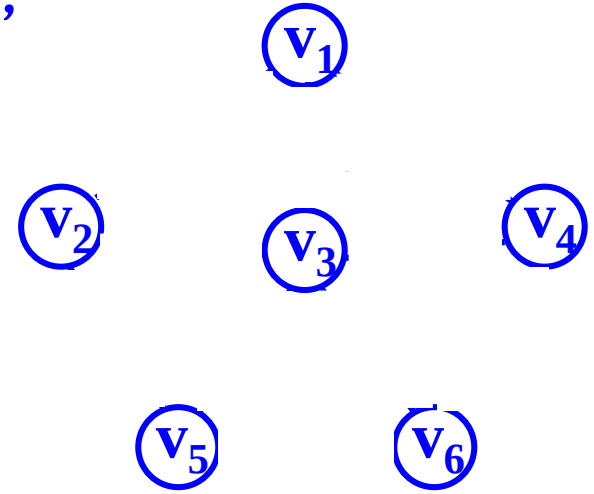
判断边 (u , v) 与 TE 中的边是否构成回路 ？

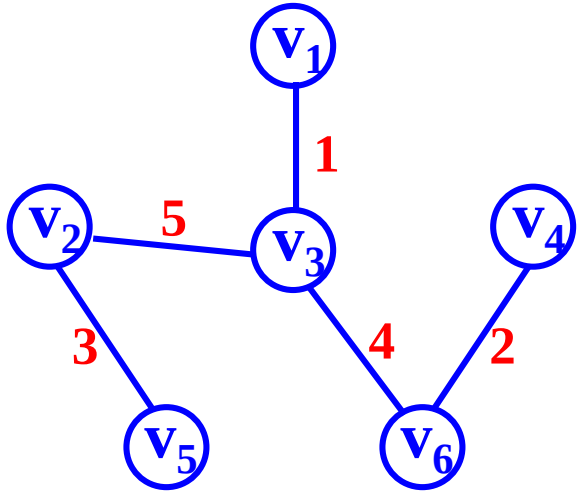
否，将边 (u , v) 纳入 TE 中，并从 E 中删除边 (u , v) ；

u 和 v 一定不在同一个连通分量中

直至 E 为空 ；

例，





当前权值最小边 (v_5 , v_6)

初始 TE = { }

< v_1 , v_3 >

< v_4 , v_6 >

< v_2 , v_5 >

< v_3 , v_6 >

< v_2 , v_3 >

证明：
归纳基础：
n=1，显然能够找到最小生成树。

归纳过程：
假设Kruskal算法对n≤k阶图适用，那么，在k+1阶图G中，我们把最短边的两个端点a和b做一个合并操作，即把u与v合为一个点v'，把原来接在u和v的边都接到v'上去，这样就能够得到一个k阶图G'(u,v的合并是k+1少一条边)，G'最小生成树T'可以用Kruskal算法得到。
我们证明T'+{<u,v>}是G的最小生成树。

用反证法，如果T'+{<u,v>}不是最小生成树，最小生成树是T，即W(T)<W(T'+{<u,v>})。显然T应该包含<u,v>，否则，可以用<u,v>加入到T中，形成一个环，删除环上原有的任意一条边，形成一棵更小权值的生成树。而T-{<u,v>}，是G'的生成树。所以W(T-{<u,v>})<=W(T')，也就是
W(T)<=W(T')+W(<u,v>)=W(T'+{<u,v>})，产生了矛盾。于是假设不成立，T'+{<u,v>}是G的最小生成树，Kruskal算法对k+1阶图也适用。
由数学归纳法，Kruskal算法得证。

已知一个图的顶点集V和边集G分别为：

V={0，1，2，3，4，5，6，7}；

G={(0,1)3,(0,3)5,(0,5)18,(1,3)7,(1,4)6,(2,4)10,
(2,7)20,(3,5)15,(3,6)12,(4,6)8,(4,7)12};

1.按照普里姆算法从顶点2出发得到最小生成树，试写出在最小生成树中依此得到的各条边和各顶点。

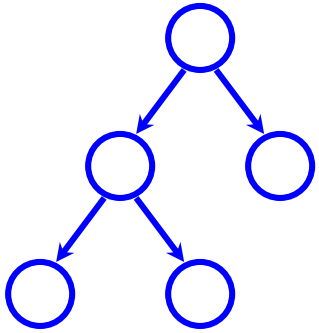
2.使用Kruskal算法得到最小生成树，写出依次加入的边和顶点

比较两种算法

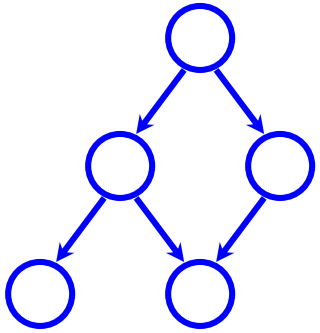
算法	Prim算法	Kruskal算法
时间复杂度	$O(n^2)$	$O(e \bullet \log e)$
邻接矩阵		
适应范围	稠密图	稀疏图

7.5 有向无环图及应用

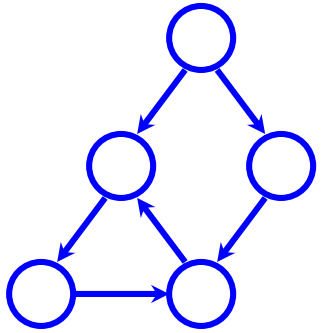
一个无环的有向图称为有向无环图，简称 DAG 图。



有向树



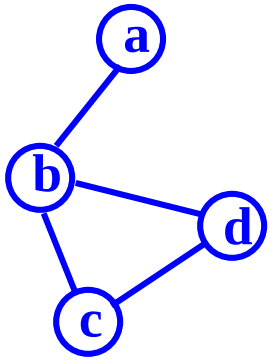
DAG 图



有向图

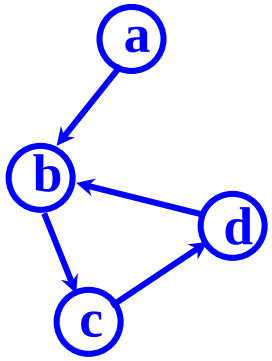
给定一个图，如何判断是否存在环？

利用深度优先搜索算法，若将要指向的顶点已被访问过，则存在环。



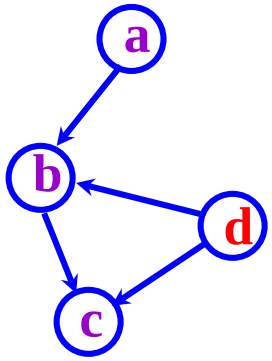
无向图

可正确判定



有向图

可正确判定



特例

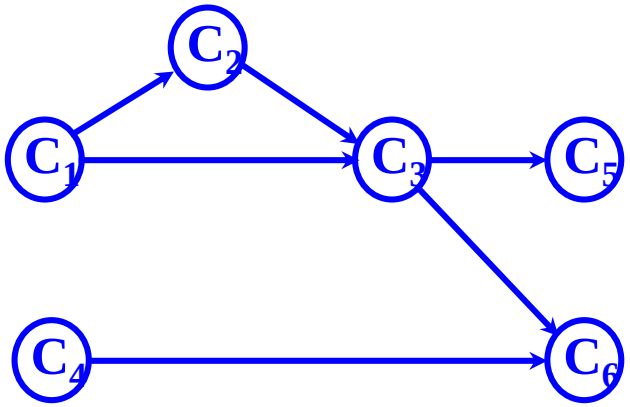
不可正确判定

如何判断一个有向图是否为 DAG 图？

方法1 在一个连通分量里寻找环。

方法2 —— 拓扑排序法

课程编号	课程名称	先决条件
C_1	程序语言基础	
C_2	离散数学	C_1
C_3	数据结构	C_1 , C_2
C_4	微机原理	
C_5	编译原理	C_3
C_6	操作系统	C_3 , C_4



表示课程间优先关系的有向图

这种用顶点表示活动，用弧表示活动之间的优先关系的有向图称为顶点表示活动的网 —— AOV网(Activity On Vertex Network) 。

在 AOV 网中不应该出现有向环，否则将存在某项活动以自己为先决条件。

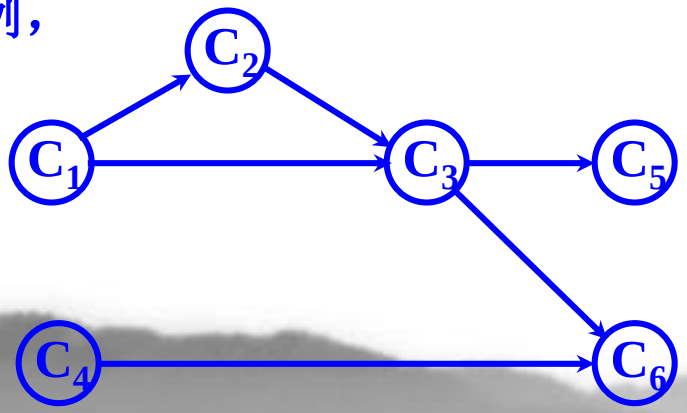
若AOV网中，< i ，j > 是一条弧，则称 i 为 j 的前驱，j 为 i 的后继。

对 AOV 网中的所有顶点进行一个排序，

如果满足: 若 i 是 j 的前驱顶点，则序列中 i 必在 j 之前

则称这样的排序为拓扑排序。

例，



表示课程之间优先关系的 AOV 网

拓扑排序可以为:

C_1 C_2 C_3 C_5 C_4 C_6

C_4 C_1 C_2 C_3 C_5 C_6

性质：若AOV 网中的所有顶点都在它的拓扑排序中，则该 AOV 网中必定不存在环；否则必存在环。

拓扑排序算法证明

算法思想：

1. 在有向图中选取一个没有前驱的顶点并输出；

2. 从图中删除该顶点及所有以此顶点为尾的弧；

3. 重复上述两步，直至全部顶点均已输出；或者当前图中不存在无前驱的顶点为止。

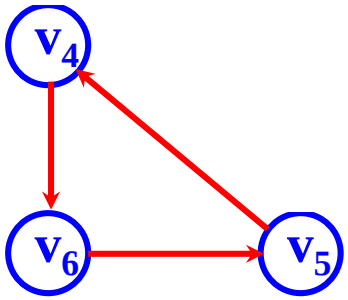
得到一个
拓扑排序

存在环

例，

拓扑排序　v_1 v_6 v_4 v_3 v_2 v_5

例，



拓扑排序　v_1　v_3　v_2

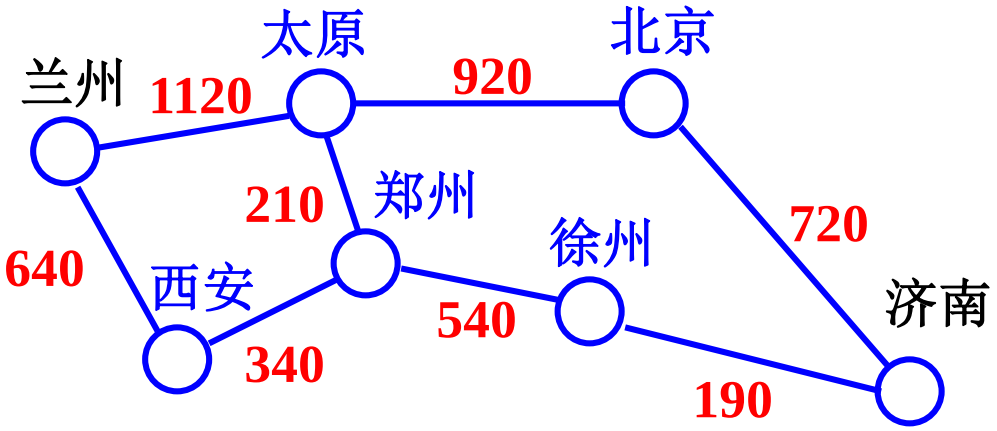
存在环

算法实现

1. 采用邻接表作为有向图的存储结构；
2. 数组 indegree[ ] 存放顶点的入度；
3. 入度为 0 的顶点即为没有前驱的顶点；
4. “删除顶点及以它为尾的弧的操作”换成以弧头顶点的入度减 1 来实现；
5. 为了避免重复检测入度为 0 的顶点，可另设一栈暂存所有入度为 0 的顶点。

```
Status TopologicalSort (ALGraph G) {
    // 有向图G采用邻接表存储
    FindInDegree(G, indegree);   //求各顶点的入度
    InitStack (S);
    for ( i=0; i<G.vexnum; ++i )
        if  ( ! indegree[i] )    Push (S, i);  //入度为 0 的顶点入栈
    count=0;  //对输出顶点计数
    while  ( ! StackEmpty(S) )    {
        Pop (S, i); printf (i, G.vertices[i].data); ++count; //输出顶点
        for  ( p=G.vertices[i].firstarc; p; p=p->nextarc )   {
            k=p->adjvex;
            if ( !(--indegree[k]))   Push (S, k);
        }  //依次处理邻接顶点，入度减 1 ，且入度为 0 的顶点入栈
    }
    if   (count<G.vexnum)    return ERROR;
    else   return OK;
}
```

7.6 最短路径



旅客希望停靠站越少越好，则应选择

济南——北京——太原——兰州

旅客考虑的是旅程越短越好，

济南——徐州——郑州——西安——兰州

带权图的最短路径计算问题

通常在实际中，航运、铁路、船行都具有有向性，故我们以带权有向图为例介绍最短路径算法。

带权无向图的最短路径算法也通用。

- 从单个源点到其余各顶点的最短路径算法。
- 每一对顶点之间的最短路径算法。

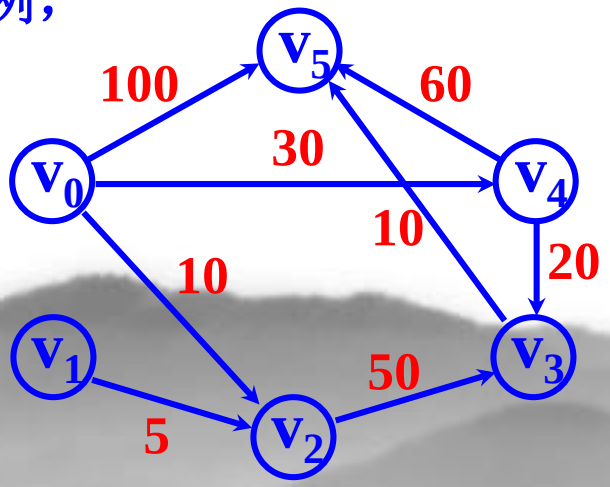
7.6.1 从单个源点到其余各顶点的最短路径算法
—— Dijkstra迪杰斯特拉 算法

思想： 贪心算法(局部最优)，按路径长度递增的次序产生最短路径。

贪心算法：利用局部最优来计算全局最优。

利用已得到的顶点的最短路径来计算其它顶点的最短路径。

例，



求从 v_0 到其余各顶点的最短路径。

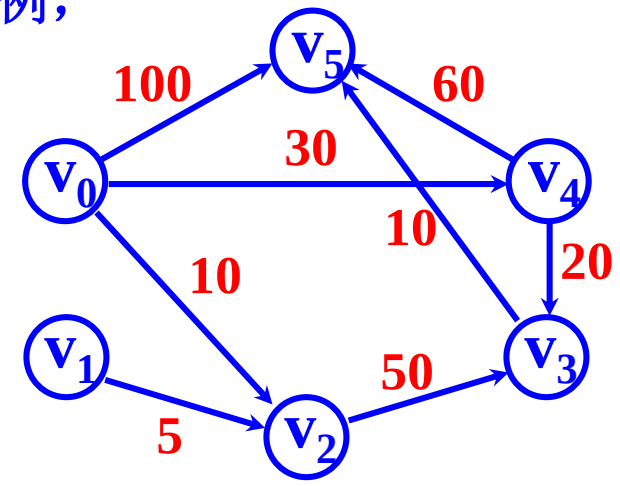
D[i] 表示 v_0 到 v_i 的最短路径的长度

Path[i]表示v_0 到 v_i 的最短路径

1. 初始，D[i] 的值为 v_0 到 v_i 的弧的权值

显然，D[i] 中的最小值 D[2] 便是 v_0到 v_2 的最短路径的长度，Path[2]=(v_0 , v_2)

例，



2. 寻找下一条最短路径

设下一条最短路径的终点是 v_j，则这条最短路径或者是 (v_0, v_j)、或者是 v_0 经过 v_2 到达 v_j 的路径；

其中取 D[i](D[2]除外) 中的最小值得到 v_4，Path[4]=(v_0, v_4)。

3. 继续寻找下一条最短路径

设下一条最短路径的终点是 v_k，则这条最短路径或者是 (v_0, v_k)、或者是 v_0 经过 v_2 或 v_4 到达 v_k 的路径；

取 D[i](D[2]、D[4] 除外) 中的最小值得到 v_3，Path[3]=(v_0, v_4, v_3)。

一般情况，假设 S 为已求得最短路径的终点的集合，则有：下一条最短路径(设终点为 x) 或者是弧 (v_0 , x)，或者是 v_0 出发中间只经过 S 中的顶点而最后到达顶点 x 的路径。

反证法:

假设下一条最短路径上有一个顶点不在 S 中，不妨设 v’；

则必存在一条终点为 v’ 的最短路径，其长度比该路径短；

可这是不可能的，因为我们是按照路径长度递增的次序来依次产生最短路径，即长度比该路径短的所有路径都已产生；

矛盾。

Dijkstra 算法描述

利用已得到的顶点的最短路径来修改得到其它顶点的更短路径。

假设用带权的邻接矩阵 arcs[i][j] 来表示带权有向图。

初始，D[i] 存放 v_0 到 v_i 各顶点的弧的权值，D[i]=arcs[0][i] ， S={}；

重复执行 n−1 遍，每遍求出一条新的最短路径

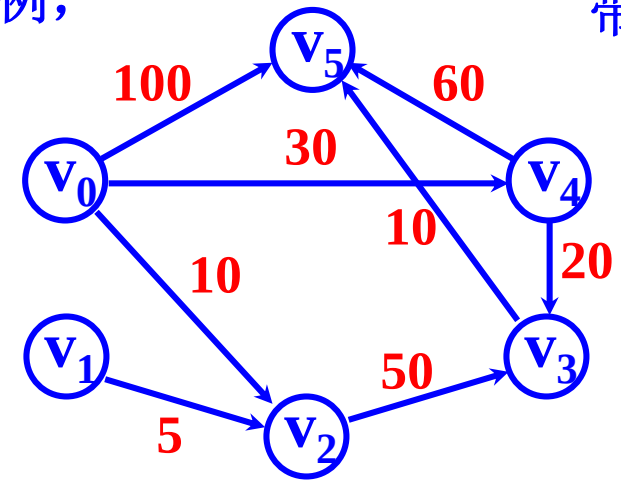
利用公式 D[j] = Min { D[i] ｜ $v_i \in V-S$ } 得到一条新的从 v_0 出发的最短路径及新的终点 v_j ，令 S = S + { v_j } ；

利用 v_j 修改从 v_0 出发到集合 V−S 中任一顶点 v_k 可达的路径的长度；

D[j] + arcs[j][k]　　vs.　　D[k]

算法时间复杂度：$O(n^2)$

例，



带权邻接矩阵

	0	1	2	3	4	5
0	∞	∞	10	∞	30	100
1	∞	∞	5	∞	∞	∞
2	∞	∞	∞	50	∞	∞
3	∞	∞	∞	∞	∞	10
4	∞	∞	∞	20	∞	60
5	∞	∞	∞	∞	∞	∞

顶点 \ S		$\{v_2\}$	$\{v_2, v_4\}$	$\{v_2, v_4, v_3\}$	
v_1	∞	∞	∞	∞	
v_2	10				
v_3	∞	60	50		
v_4	30	30			
v_5	100	100	90	60	
最短路径	v_0v_2	v_0v_4	$v_0v_4v_3$	$v_0v_4v_3v_5$	
新顶点	v_2	v_4	v_3	v_5	v_1
路径长度	10	30	50	60	∞

每次修改都用的是最新加入集合 S 的顶点

练习

- 图的广度优先搜索类似于树的(　)次序遍历。

  A．先根　B．中根　C．后根　D．层次

- 具有n个顶点的有向无环图最多可包含（　）条有向边。

  A. n-1　B. n　C. n(n-1)/2　D. n(n-1)

- 任何一个无向连通图的最小生成树（　）。

  A．只有一棵　B．有一棵或多棵

  C．一定有多棵　D．可能不存在

练习

- 设图G＝(V，E)，V＝{1，2，3，4}，E＝{<l，2>，<1，3>，<2，4>，<3，4>}，从顶点1出发，对图G进行广度优先搜索的序列有_____种。

- 有向图G用邻接矩阵A[1..n,1..n]存储，矩阵中元素值1代表有弧，0代表无弧，其第i行的所有元素之和等于顶点i的_____度。

练习

- 一个连通图的生成树是该图的_____连通子图。若这个连通图有n个顶点，则它的生成树有_____条边。
- 在用于表示有向图的邻接矩阵中（矩阵中元素值1代表有弧，0代表无弧），对第i行的元素进行累加，可得到第i个顶点的_____度。
- 设图G=（V，E），V={1，2，3，4，5，6}，E={<1，2>，<1，3>，<2，5>，<3，6>，<6，5>，<5，4>，<6，4>}。请写出图G中顶点的所有拓扑序列。

- 已知一个图的顶点集V和边集G分别为：

V={0，1，2，3，4，5，6，7}；

G={(0,1)3,(0,3)5,(0,5)18,(1,3)7,(1,4)6,(2,4)10,
(2,7)20,(3,5)15,(3,6)12,(4,6)8,(4,7)12};

按照普里姆算法从顶点2出发得到最小生成树，试写出在最小生成树中依此得到的各条边。

- 已知一个带权图的顶点集V和边集G分别为：

V={0，l，2，3，4，5，6}；

G ={(0，1)19，(0，2)10，(0，3)14，(1，2)6，
(1，5)5，(2，3)26，(2，4)15，(3，4)18，
(4，5)6，(4，6)6，(5，6)12}；

试根据迪克斯特拉(Dijkstra)算法求出从顶点0到其余各项点的最短路径，在下面填写对应的路径长度。

顶点：　0　1　2　3　4　5　6

路径长度：