第8章 图

知识要点

(1) 掌握图的定义和术语，熟悉图的各种存储结构；
(2) 熟练掌握图的两种遍历策略；
(3) 理解图的各种应用算法。

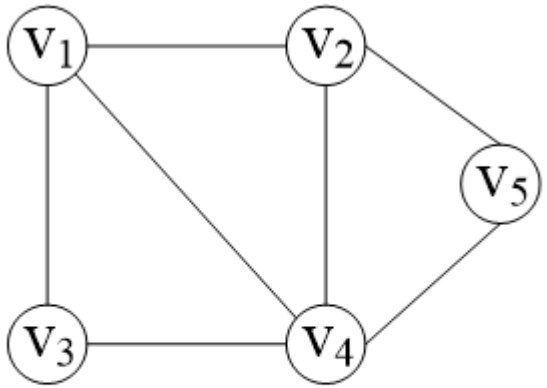
薄钧戈
2021年4月21日

6.1 图的定义和术语

- 图G是由表示数据元素的集合V和表示数据元素之间关系的集合E组成，记为G = (V, E)。
- 在图中，数据元素通常称作顶点，V是有限非空的顶点集合；
- 顶点的偶对称为边，E是两个顶点之间具有关系的边集合。
- 一个图可以形式化定义为：
  - G = (V, E)
  - V = {vi | vi ∈ 数据对象}
  - E = {<vi, vj> | vi, vj ∈ V ∧ P(vi, vj)}
  - 其中，vi, vj为数据元素，称为顶点，P(vi, vj)表示在顶点vi和顶点vj之间有一条边，即vi和vj之间存在一个关系。

6.1 图的定义和术语-无向图

- 若顶点v_i和v_j之间的边没有方向，则称这条边为无向边，表示为(v_i, v_j)。
- 如果图中任意两个顶点之间的边都是无向边，则称该图为无向图(Undigraph)。



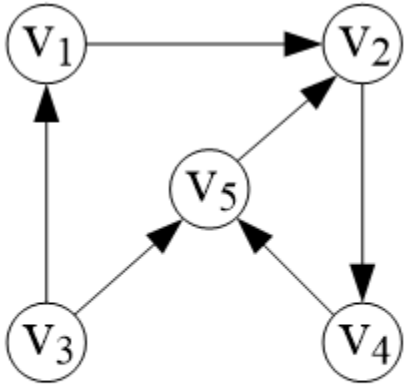
(a) 无向图 G_1

- 无向图G1，可形式化的表示为：
  - G1 = (V1, E1)
  - V(G1)= {v1, v2, v3, v4, v5}
  - E(G1)= {(v1, v2), (v1, v3), (v1, v4), (v2, v4), (v2, v5), (v3, v4), (v4, v5) }

在无向图中，对于任何$(v_i, v_j) \in E$，必有$(v_j, v_i) \in E$，即E是对称的(v_i, v_j)和(v_j, v_i)代表的是同一条边。

6.1 图的定义和术语-有向图

- 若顶点v_i和v_j之间的边有方向，则称这条边为有向边，表示为<v_i，v_j>。
- 如果图中任意两个顶点之间的边都是有向边，则称该图为有向图(Digraph)。在有向图中，<v_i, v_j>表示从v_i到v_j的一条弧，称v_i为弧尾或起点，v_j为弧头或终点。

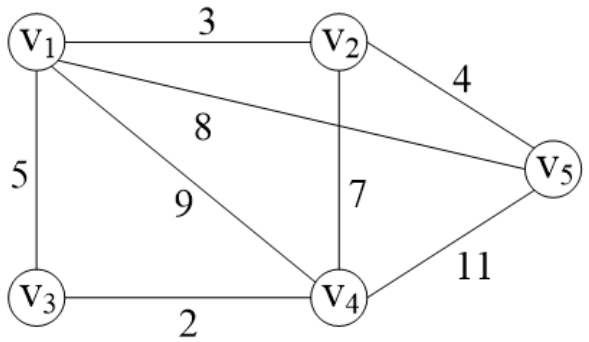


(b) 有向图 G_2

- 有向图G2，可形式化的表示为：
  - G2 = (V2, E2)
  - V(G2)= {v1, v2, v3, v4, v5}
  - E(G2) = {<v1, v2>, <v2, v4>, <v3, v1>,
  - <v3, v5>,<v4, v5>, <v5, v2> }

6.1 图的定义和术语-带权图

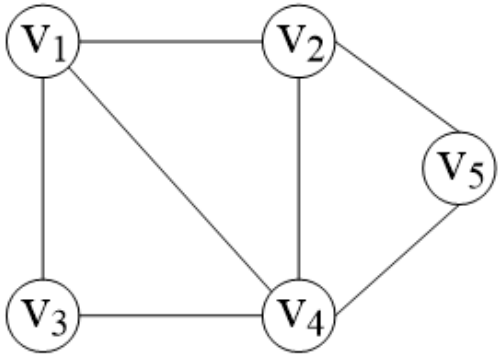
- 在无向图G = (V, E)中，若$(v_i, v_j) \in E$，则称顶点v_i和v_j互为邻接点，或称v_i和v_j相邻接，并称边(v_i, v_j)依附于顶点v_i和v_j，或者说边(v_i, v_j)与顶点相关联。
- 在有向图G = (V, E)中，若$<v_i, v_j> \in E$，则称顶点v_i邻接到顶点v_j，顶点v_j邻接自顶点v_i，并称弧$<v_i, v_j>$和顶点v_i和v_j相关联。
- 带权图(Weighted graph)又称为网，图中的边都有权值。



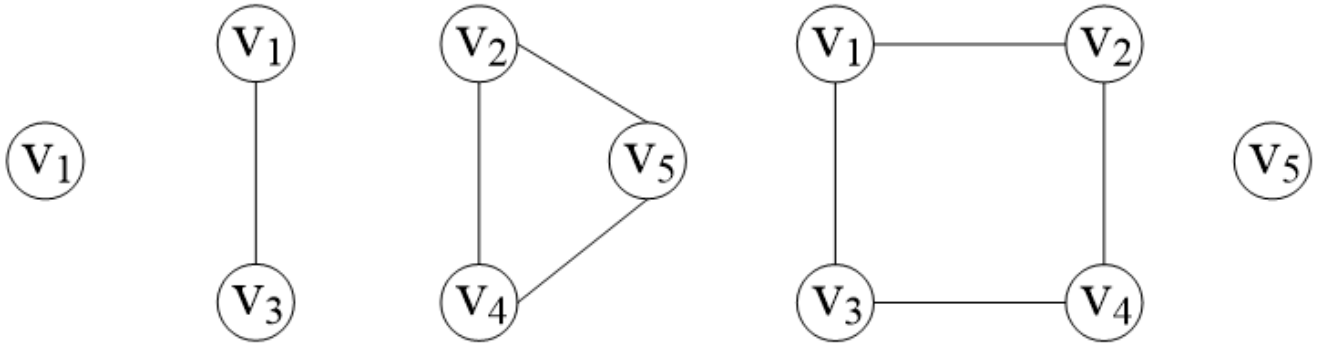
带权图

6.1 图的定义和术语-子图

- 设有两个图G = (V, E)和G′ = (V′, E′)，如果V′⊆V，E′⊆E，且E′中的边仅与V′中顶点相关联，则称G′为G的**子图**(Subgraph)。

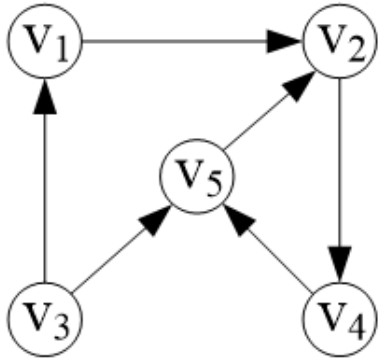


(a) 无向图 G_1

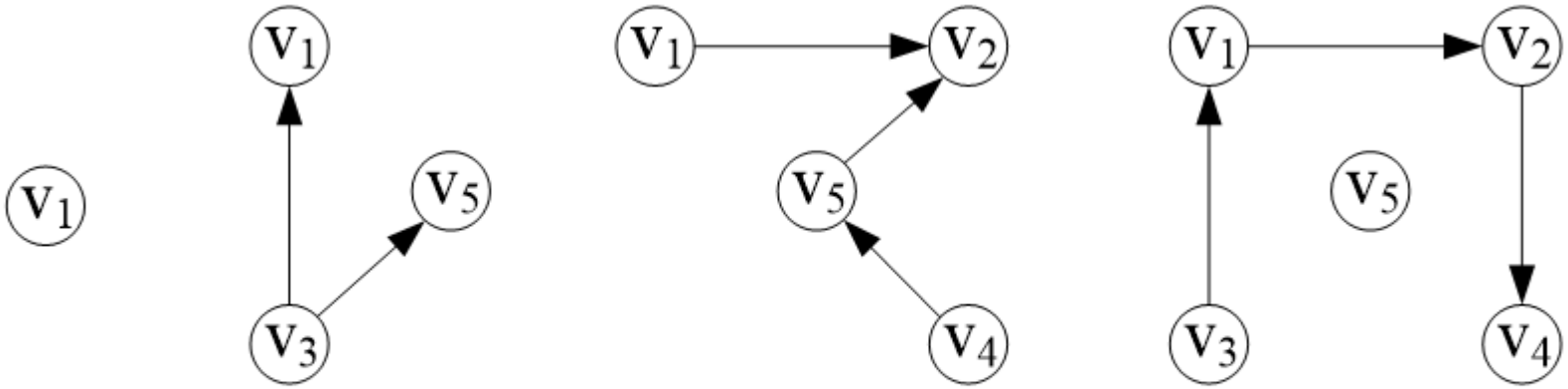


(a) 无向图 G_1 的若干子图

6.1 图的定义和术语-子图



(b) 有向图 G_2



(b) 有向图 G_2 的若干子图

6.1 图的定义和术语-路径

- 在无向图$G=(V,E)$中，若从顶点v_s到顶点v_t之间存在一个顶点序列$(v_s, v_{i0}, v_{i1}, v_{i2}, \cdots, v_{im}, v_t)$，并且$(v_s, v_{i0})$，$(v_{i0}, v_{i1})$，$\cdots$，$(v_{im}, v_t)$都是图G中的边，则称顶点$v_s$到$v_t$之间存在一条**路径**。
- 若G是有向图，则路径也是有向的，顶点序列满足$<v_s, v_{i0}>$，$<v_{i0}, v_{i1}>$，$\cdots$，$<v_{im}, v_t>$都是图G中的有向边。
- **路径的长度**定义为该路径上**边的数目**。
- 除了起点和终点之外，序列中顶点不重复出现的路径称为**简单路径**。
- 若路径的起点和终点相同，则称该路径为**回路或环**。

6.1 图的定义和术语-路径

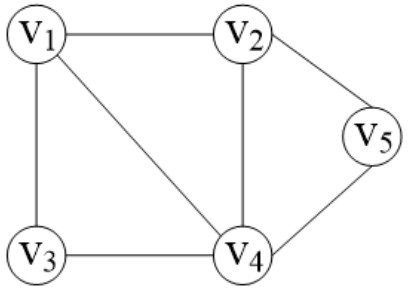
- 不带回路的图称为无环图。
- 不带回路的有向图则称为有向无环图(Directed Acyclic Graph, DAG)。
- 自由树(Free Tree)就是无回路的连通无向图。

- ★注：树和图的关系：
- 树是图的一种特殊情况，树是一种连通的无环图。当图称为树时，它的根不明确，所以将其称为自由树。

6.1 图的定义和术语-连通图

- 在无向图$G=(V,E)$中，如果从顶点v_i到顶点v_j有路径可达，则称v_i和v_j是连通的。
- 如果图中任意两个不同的顶点v_i和v_j都是连通的，则称G是连通图。
- 无向图G的极大连通子图称为G的连通分量。
- 任何连通图的连通分量只有一个，就是其本身，而非连通的无向图有多个连通分量，因此可以说连通分量是对无向图的一种划分。

6.1 图的定义和术语-连通图



(a) 无向图 G_1

无向图G_1就是一个连通图，
其连通分量就是其本身

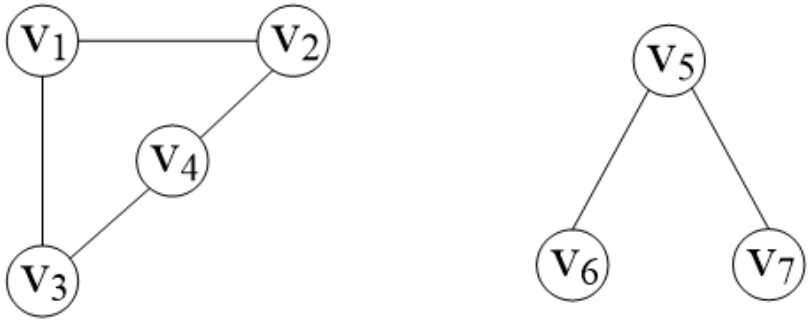


图 6-4 具有两个连通分量的非连通图 G_3

无向图G_3是非连通图，
它有两个连通分量。

6.1 图的定义和术语-连通图

- 在有向图G＝(V,E)中，若对于V中任意两个不同的顶点v_i和v_j都有有向路径可达，即存在从v_i到v_j和从v_j到v_i的路径，则称G是强连通图。
- 如果一个有向图不是强连通图，但是去掉弧上的方向形成的图为连通图，则称该有向图为弱连通图.

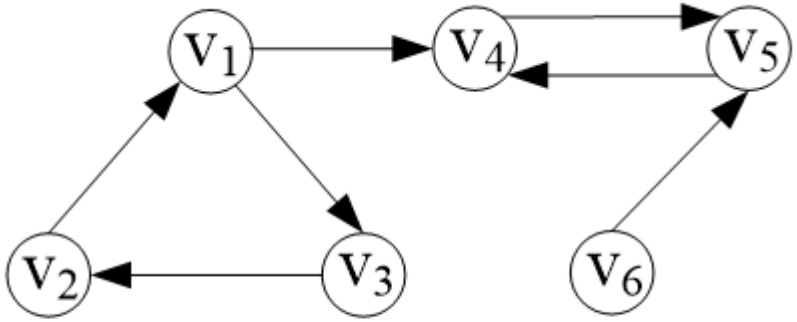
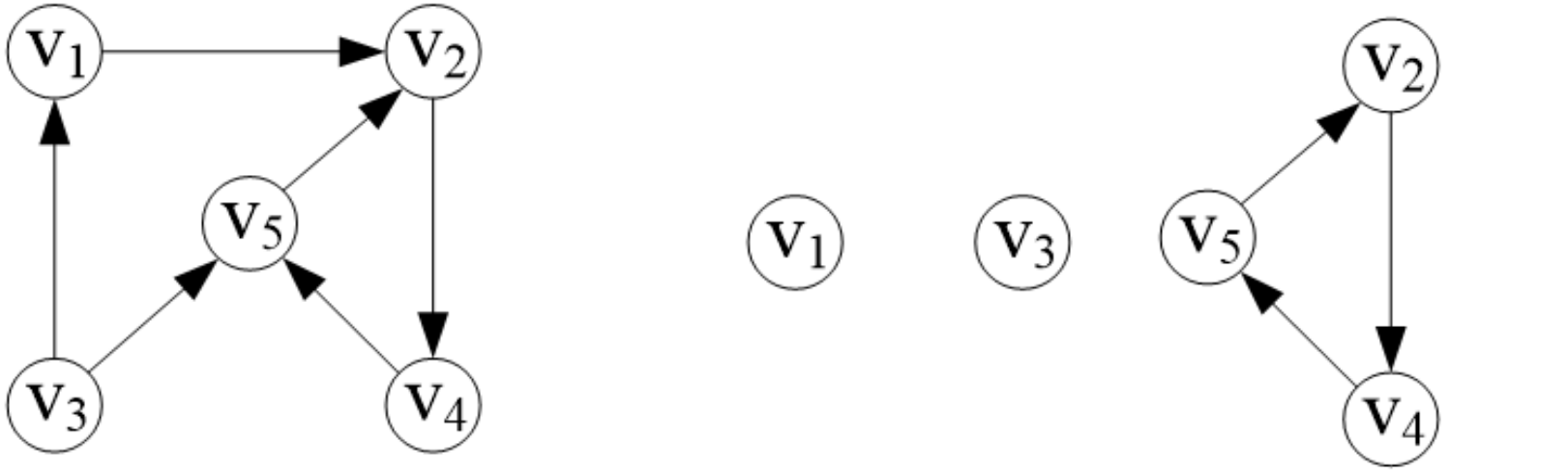


图 6-5 弱连通图

6.1 图的定义和术语-连通图

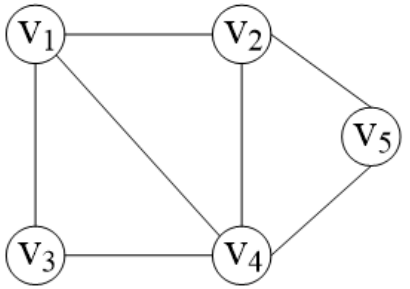
- 有向图的极大强连通子图称为G的强连通分量。显然，强连通图只有一个强连通分量，就是其本身，而非强连通的有向图有多个强连通分量。



(a) 有向图 G_2　　(b) G_2 的三个强连通分量

图 6-6 有向图 G_2 的三个强连通分量

6.1 图的定义和术语-出度、入度



(a) 无向图 G_1

- 顶点v的度是关联于该顶点的边的数目，记为D(v)。
- 顶点v的入度是指以顶点v为弧头的数目，记为ID(v)；
- 顶点v的出度是指以顶点v为弧尾的数目，记为OD(v)；
- 顶点v的度为D(v) = ID(v) + OD(v)。
- 无论是无向图还是有向图，一个有n个顶点、e条边的图，满足如下关系：

$$e = \frac{1}{2}\sum_{i=1}^{n} D(v_i)$$

- 无向图G_1中，各顶点v_1、v_2、v_3、v_4、v_5的度分别为3、3、2、4、2，图中的边数为：e = (3+3+2+4+2)/2 = 7。

6.1 图的定义和术语-出度、入度

- 图G的顶点数n和边数e满足以下关系：
  - ✓ 若G是无向图，则0 ≤ e ≤ n(n-1)/2；
  - ✓ 若G是有向图，则0 ≤ e ≤ n(n-1)。
  - ✓ 恰好有n(n-1)/2条边的无向图称为无向完全图；
  - ✓ 恰好有n(n-1)条弧的有向图称为有向完全图。
  - ✓ 完全图具有最多的边数，任意一对顶点间均有边相连。边数很少（如e ≤ nlogn）的图称为稀疏图，反之称为稠密图。

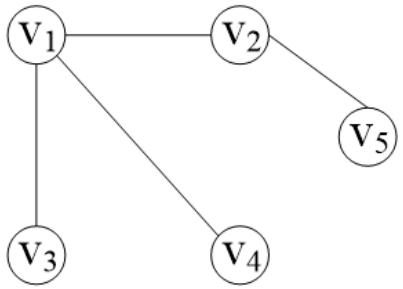
- 结论：

  (1) n个顶点的无向完全图，总的边数n(n-1)/2；

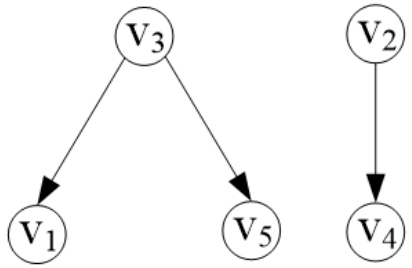
  (2) n个顶点的有向完全图，总的弧数n(n-1)。

6.1 图的定义和术语-生成树

- 一个具有n个顶点的连通图G的生成树是包含G中全部顶点的一个极小连通子图，它有且仅有n-1条边。如果在图G的生成树上添加一条边，则一定存在环，因为这使得依附于这条边的两个顶点之间有了第二条路径；如果生成树的边数少于n-1条，则该生成树必定是不连通的。
- 一个有向图的生成森林是由若干棵互不相交的有向树组成的，它含有图中全部顶点。有向树是恰有一个顶点的入度为0，其余顶点的入度均为1的有向图。



(a) 无向图 G_1 的一棵生成树



(b) 有向图 G_2 的生成森林

图 6-7 生成树和生成森林

★注：

(1) 图可转换为树，树可依照二叉树进行处理；非连通图可转换为森林；

(2) 任何树都可转换为二叉树，而二叉树的根无右子树时才能转换成树；

(3) 森林同二叉树之间可实现相互转换。

通过对图的生成树遍历可以完成图遍历。

6.1 图的定义和术语-代码

算法6.1：图的类定义

```
1.  class Graph{
2.  public:
3.      int numVertex;          //图中顶点的个数
4.      int numEdge;            //图中边的个数
5.      bool *Visited;          //Visited指针指向保存图的顶点的标志位数组
6.      int *InDegree;          //InDegree指针指向保存图的顶点的入度的数组
7.      Graph(int numVert);       //构造函数
8.      ~Graph();                   //析构函数
9.      virtual int FirstAdj(int oneVertex) = 0; //返回与顶点oneVertex相关联的第一个邻接点
10.     virtual int NextAdj(int oneVertex, int preVertex) = 0;
11.     int VerticesNum() {return numVertex;}     //返回图的顶点个数
12.     int EdgesNum() {return numEdge;}  //返回图的边数
13.     virtual int weight(int from,int to) = 0;
14.     virtual void setEdge(int from,int to,int weight) = 0;
15.     virtual void delEdge(int from,int to) = 0;
16. };
```

6.2 图的存储结构

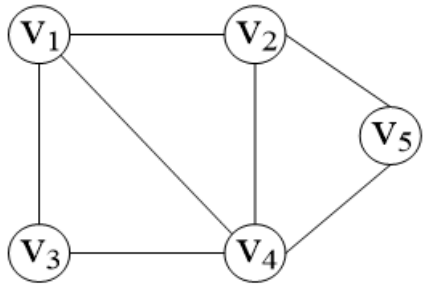
6.2.1 邻接矩阵存储方法

- 对于一个具有n个顶点的图G＝(V, E)，其中顶点集合V可以用一个一维数组来存放，顶点关系集合E可以使用n*n的二维数组AdjMatrix[n][n]来存储各顶点之间的邻接关系，该二维数组称为邻接矩阵(Adjacency Matrix)，其定义如下：

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

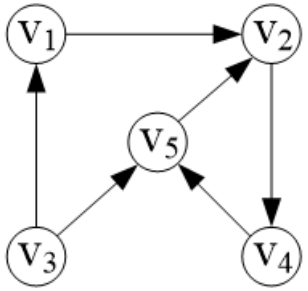
6.2 图的存储结构

- 图6-8所示的(a)无向图G_1和(b)有向图G_2所对应的邻接矩阵分别为A_1和A_2：



(a) 无向图 G_1

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



(b) 有向图 G_2

$$A_2=\begin{bmatrix}0&1&0&0&0\\0&0&0&1&0\\1&0&0&0&1\\0&0&0&0&1\\0&1&0&0&0\end{bmatrix}$$

图 6-8　G_1 和 G_2 的邻接矩阵

6.2 图的存储结构

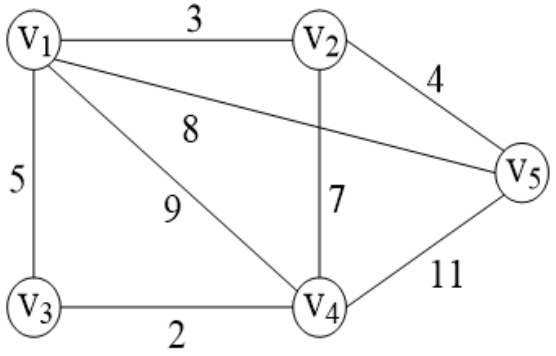
★注：

- 图的存储由二个部分组成，分别为顶点集合存储和边集合存储。
- 无向图的邻接矩阵是对称的；有向图的邻接矩阵可能是不对称的。
- 有向图的行只表示弧的发射关系，列表示接入关系。
- 完全图的邻接矩阵中，对角元素为0，其余全为1。

6.2 图的存储结构

- 带权图的邻接矩阵可定义为：

$$\text{AdjMatrix}[i][j]=\begin{cases} w_{ij} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\in E,\ w_{ij}\text{为边}(v_i,v_j)\text{或}<v_i,v_j>\text{的权} \\ \infty & \text{否则} \end{cases}$$



$$A_3=\begin{bmatrix} \infty & 3 & 5 & 9 & 8 \\ 3 & \infty & \infty & 7 & 4 \\ 5 & \infty & \infty & 2 & \infty \\ 9 & 7 & 2 & \infty & 11 \\ 8 & 4 & \infty & 11 & \infty \end{bmatrix}$$

图 6-9　带权图的邻接矩阵

6.2 图的存储结构

- 对于无向图，顶点v_i的度$D(v_i)$等于邻接矩阵中第i行（或第i列）的元素之和，即

$$D(v_i) = \sum_{j=0}^{n-1} AdjMatrix[i][j]$$

- 对于有向图，顶点v_i的出度$OD(v_i)$等于邻接矩阵中第i行的元素之和，顶点v_i的入度$ID(v_i)$等于邻接矩阵中第i列的元素之和，顶点v_i的度$D(v_i)$等于邻接矩阵中第i行的元素和第i列的元素之和，即

$$OD(v_i) = \sum_{j=0}^{n-1} AdjMatrix[i][j]$$

$$ID(v_i) = \sum_{j=0}^{n-1} AdjMatrix[j][i]$$

$$D(v_i) = OD(v_i) + ID(v_i)$$

6.2 图的存储结构

结论：

(1) 复杂度：

- 确定边关系的时间复杂度：O(1)；
- 加入或删除一条边的时间复杂度：O(1)；
- 寻找一个顶点的所有邻接点的时间复杂度：O(n)；
- 存储边的空间复杂度：$O(n^2)$，即顶点的平方。

(2) 困难：

- 增加或删除一个顶点需要重新建邻接矩阵，由此操作变得很复杂，时间复杂度变得很高。因此，邻接矩阵比较适合静态表，即表中的结点不发生变化。

6.2 图的存储结构-代码

算法6.2：邻接矩阵方式实现图的定义

```
1.  class Graphm: public Graph{
2.  private:
3.     int **AdjMatrix;  //指向邻接矩阵的指针
4.  public:
5.     Graphm(int numVert): Graph(numVert){       //构造函数
6.        AdjMatrix = new int*[numVertex];      //申请AdjMatrix指针数组
7.        for(int i = 0; i < numVertex; i++)
8.           AdjMatrix[i] = new int[numVertex];  //申请每行元素向量
9.  for(int i = 0; i < numVertex; i++)       //相邻矩阵的所有元素都初始化为0
10.       for(int j = 0; j < numVertex; j++)
11.             AdjMatrix[i][j] = 0;
12.    }
13.    ~Graphm(){   //析构函数，释放申请的空间
14.       for(int i = 0; i < numVertex; i++)
15.        delete [] AdjMatrix[i];
16.   delete [] AdjMatrix;
17. }
```

6.2 图的存储结构-代码

```
1.        int weight(int from, int to){
2.              return AdjMatrix[from][to];
3.      }
4.      void setEdge(int from,int to,int weight){ //为图设定一条边
5.        if(AdjMatrix[from][to] == 0){
6.                  numEdge++;
7.                  InDegree[to]++;
8.        }
9.        AdjMatrix[from][to] = weight;
10.   }
```

6.2 图的存储结构-代码

```
1.  void delEdge(int from,int to){      //删掉图的一条边
2.      if(AdjMatrix[from][to] != 0){
3.          numEdge--;
4.          InDegree[to]--;
5.      }
6.      AdjMatrix[from][to] = 0;
7.  }
8.  int FirstAdj(int oneVertex){
9.    for (int i = 0; i < numVertex; i++)
10.         if (AdjMatrix[oneVertex][i] != 0)  return i;
11.     return -1;
12. }
13. int NextAdj(int oneVertex, int preVertex){
14.   for (int i = preVertex+1; i < numVertex; i++)
15.         if (AdjMatrix[oneVertex][i] != 0)  return i;
16.     return -1;
17. }
18. };
```

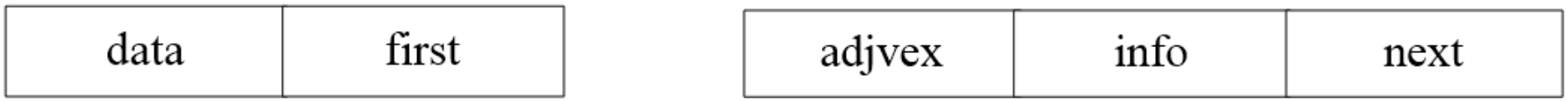
6.2 图的存储结构

6.2.2 邻接表存储方法

- 图的邻接表存储方法是一种链式存储结构，这种表示法类似于树的孩子单链表表示法。邻接点数的多少决定链表的长度。
- 在邻接表(Adjacency List)中，对图中的每个顶点建立一个单链表，第i个单链表中的结点表示所有依附于顶点v_i的边。每个单链表设置一个表头结点。
- 单链表中的结点称为表结点，如果这个表头结点所对应的顶点存在邻接点，则把邻接点依次存放于表头结点所指向的单链表中。

6.2 图的存储结构

- 在表头结点中包括两个域，其中data域存放顶点的数据信息，first域指向依附于该顶点的第一条边。表结点由三个域组成，其中adjvex域存放与顶点v_i邻接的点在图中的位置，next域指向下一个表结点，info域存储与边相关的信息，如权值等，可以省略此域。表头结点和表结点的结构如图6-10所示。

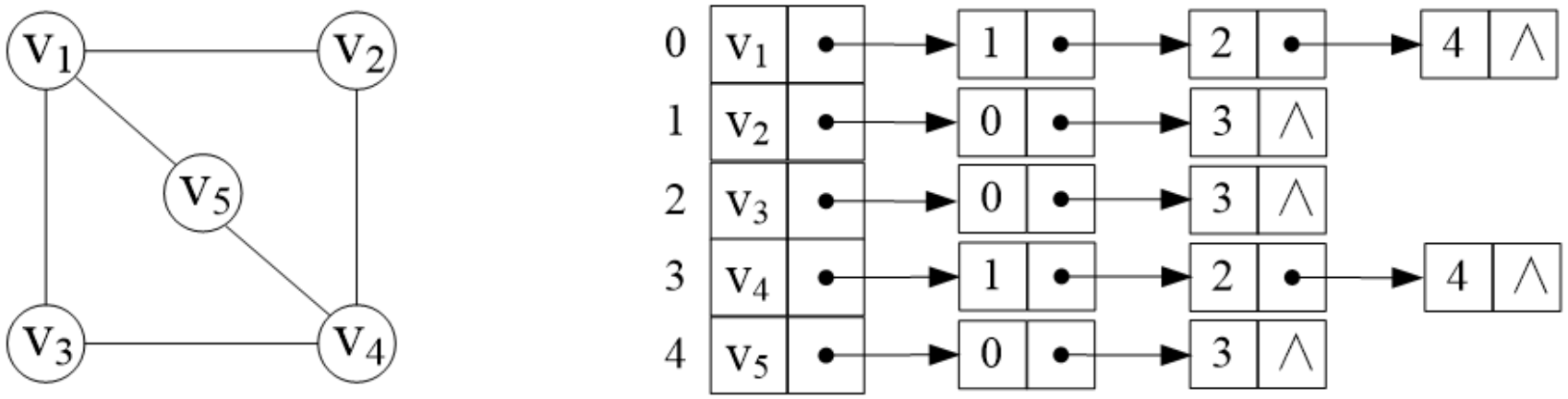


(a) 表头结点　　(b) 表结点

图 6-10 表头结点和表结点的结构图

6.2 图的存储结构

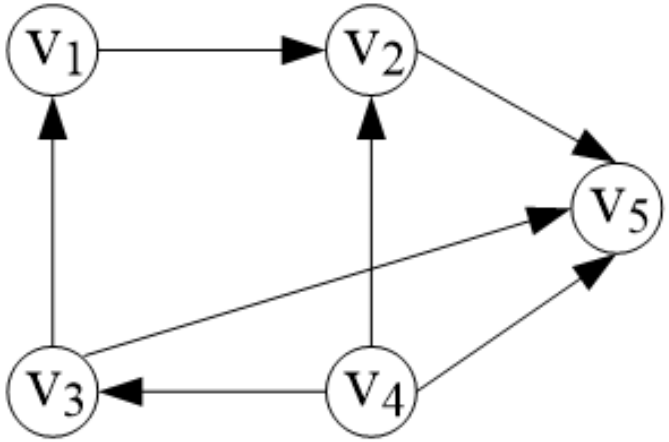
- 在图的邻接表表示方法中，可以将表头结点构造成一个链表，通常将所有表头结点用一个一维数组来存储，以方便随机访问任意结点的链表。图6-11(b)为图6-11（a）所示的无向图G_4的邻接表。



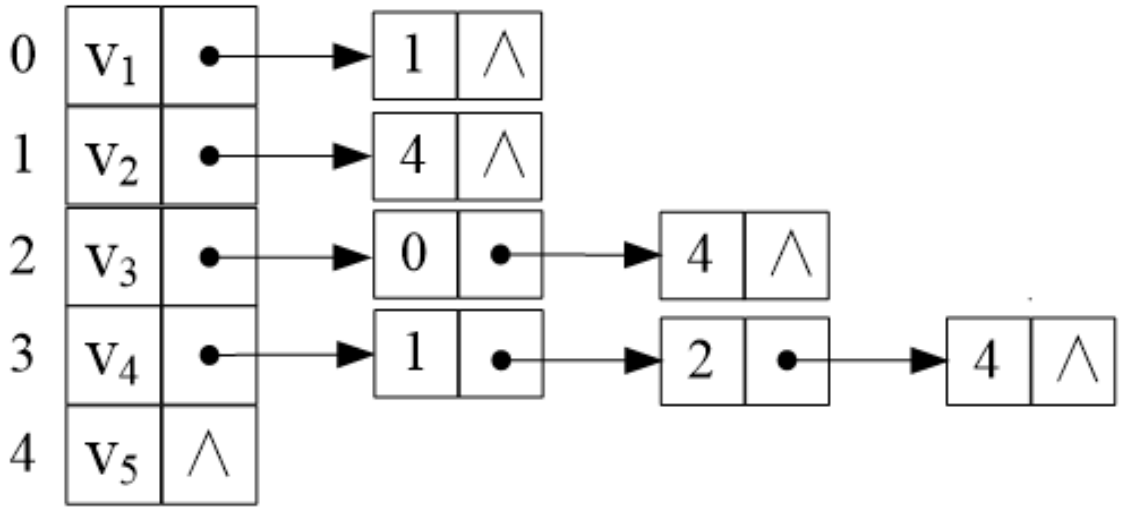
(a) 无向图 G_4　　(b) G_4 的邻接表

图 6-11 无向图 G_4 及其邻接表存储

6.2 图的存储结构



(a) 有向图 G_5

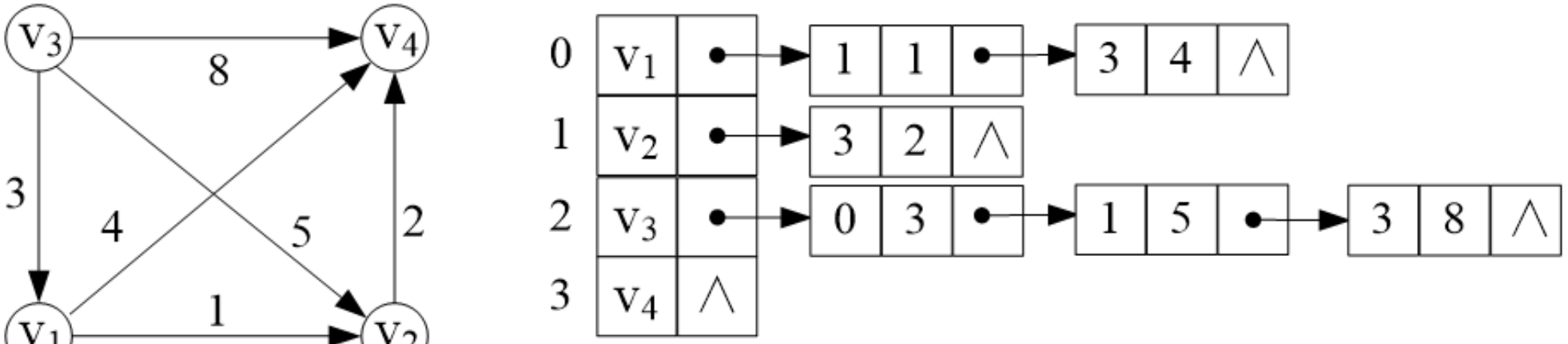


(b) G_5 的邻接表

图 6-12 有向图 G_5 及其邻接表存储

6.2 图的存储结构

- 图6-13(b)为图6-13(a)所示带权图G_6的邻接表。

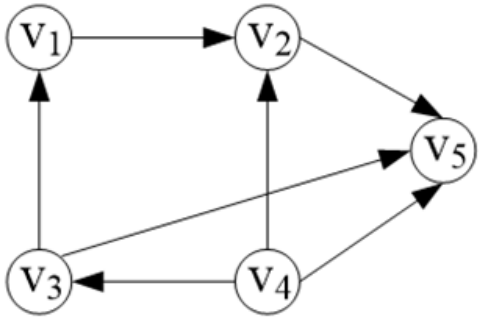


(a) 带权图 G_6　　(b) G_6 的邻接表

图 6-13　带权图 G_6 及其邻接表存储

6.2 图的存储结构

- 存放弧尾相邻的关系（以发出弧相邻）叫邻接表
- 存入弧头相邻的关系（以接入弧相邻）叫逆邻接表
- 如图6-14所示为6-12所示有向图G_5的逆邻接表。



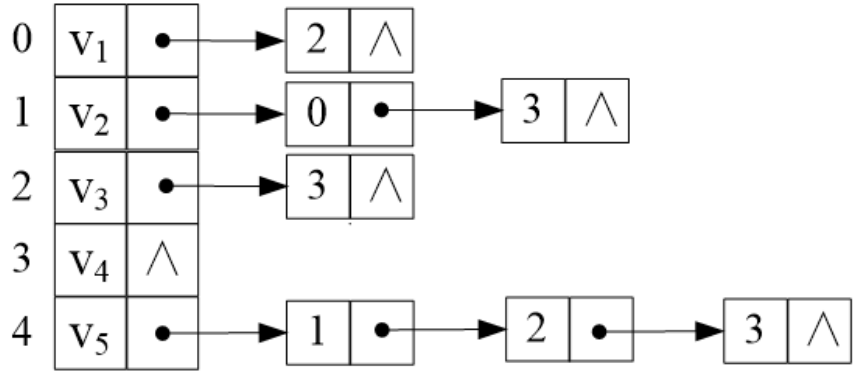


图 6-14 有向图 G_5 的逆邻接表

结论：

(1) 无向图中邻接点的总数为：2e；

(2)有向图中邻接点的总数为：弧的个数

6.2 图的存储结构-代码

算法6.3：利用邻接表方式实现图的定义

```
1.  struct listUnit{          //邻接表表目中数据部分的结构定义
2.  int vertex;          //边的终点
3.  int weight;          //边的权
4.  };
5.  class Graphl: public Graph{
6.  private:
7.  LList<listUnit> *vexList;   //vexList是保存所有边表的数组
8.  public:
9.  Graphl(int numVert):Graph(numVert){   //构造函数
10.         vexList = new LList<listUnit>[numVertex];
11. }
12. ~Graphl(){  delete [] vexList; }      //析构函数
13. int weight(int from, int to){
14.         Link<listUnit> *temp = vexList[from].head;
15.         while(temp->next != NULL && temp->next->element.vertex != to)
16.                 temp = temp->next;
17.         return temp->next->element.weight;
18. }
```

6.2 图的存储结构-代码

```
1.    int FirstAdj(int oneVertex){
2.        Link<listUnit> *temp = vexList[oneVertex].head;
3.        if(temp->next != NULL)
4.        return temp->next->element.vertex;
5.        return -1;
6.    }
7.  int NextAdj(int oneVertex, int preVertex){
8.        Link<listUnit> *temp = vexList[oneVertex].head;
9.        while(temp->next != NULL && temp->next->element.vertex <= preVertex)
10.       temp = temp->next;
11.       if(temp->next != NULL)
12.           return temp->next->element.vertex;
13.       return -1;
14.   }
```

6.2 图的存储结构-代码

```
1.  void setEdge(int from,int to,int weight){       //为图设定一条边
2.      Link<listUnit> *temp = vexList[from].head;
3.      while(temp->next != NULL && temp->next->element.vertex < to)
4.          temp = temp->next;
5.      if(temp->next == NULL){
6.          //边(from,to)或狐<from,to>在边表中不存在且其后无其它边
7.          temp->next = new Link<listUnit>;
8.          temp->next->element.vertex = to;
9.          temp->next->element.weight = weight;
10.         numEdge++;
11.         InDegree[to]++;
12.         return;
13.     }
14.     <续下页>
```

6.2 图的存储结构-代码

```
1.  if(temp->next->element.vertex == to){
2.  //边(from,to)或<from,to>在边表中已存在,故只需要改变边的权值
3.              temp->next->element.weight = weight;
4.          return;
5.  }
6.  if(temp->next->element.vertex > to){
7.          //边(from,to)或<from,to>在边表中不存在且其后存在其它边
8.          Link<listUnit> *other = temp->next;
9.          temp->next = new Link<listUnit>;
10.         temp->next->element.vertex = to;
11.         temp->next->element.weight = weight;
12.         temp->next->next = other;
13.         numEdge++;
14.         InDegree[to]++;
15. }
16. }
```

6.2 图的存储结构-代码

```
1.  void delEdge(int from,int to){    //删掉图的一条边
2.  Link<listUnit> *temp = vexList[from].head;
3.  while(temp->next != NULL && temp->next->element.vertex < to)
4.          temp = temp->next;
5.  if(temp->next == NULL) return; //边(from,to)或<from,to>在边表中不存在
6.  if(temp->next->element.vertex == to){ //边(from,to)或<from,to>在边表中存在
7.              Link<listUnit> *other = temp->next->next;
8.              delete temp->next;
9.              temp->next = other;
10.             numEdge--;
11.             InDegree[to]--;
12.         }
13. }
14. };
```

6.2 图的存储结构

结论

(1) 时间复杂度：

- ①确定边关系的时间复杂度为：O(n)；
- ②加入或删除一条边的时间复杂度为：O(n)；
- ③寻找一个顶点的所有邻接点的时间复杂度为：O(n)；
- ④无向图存储边的空间复杂度为：O(n + 2e)；
- ⑤有向图存储边的空间复杂度为：O(n + e)。

(2) 困难：

- 删除和增加结点同样相对困难，可通过将表头结点的一维数组改成一个链表结构，即将表头变为动态结构来解决此问题。

第8章 图

知识要点

(1) 掌握图的定义和术语，熟悉图的各种存储结构；
(2) 熟练掌握图的两种遍历策略；
(3) 理解图的各种应用算法。

薄钧戈
2021年4月21日

6.2 图的存储结构

- 除了邻接矩阵和邻接表两种常用的方法外，还有两种不常用的存储方法：十字链表和邻接多重表。
  - 十字链表(Orthogonal List)是有向图的另一种链式存储结构。
  - 可以看成是将有向图的邻接表和逆邻接表结合起来得到的一种链表。

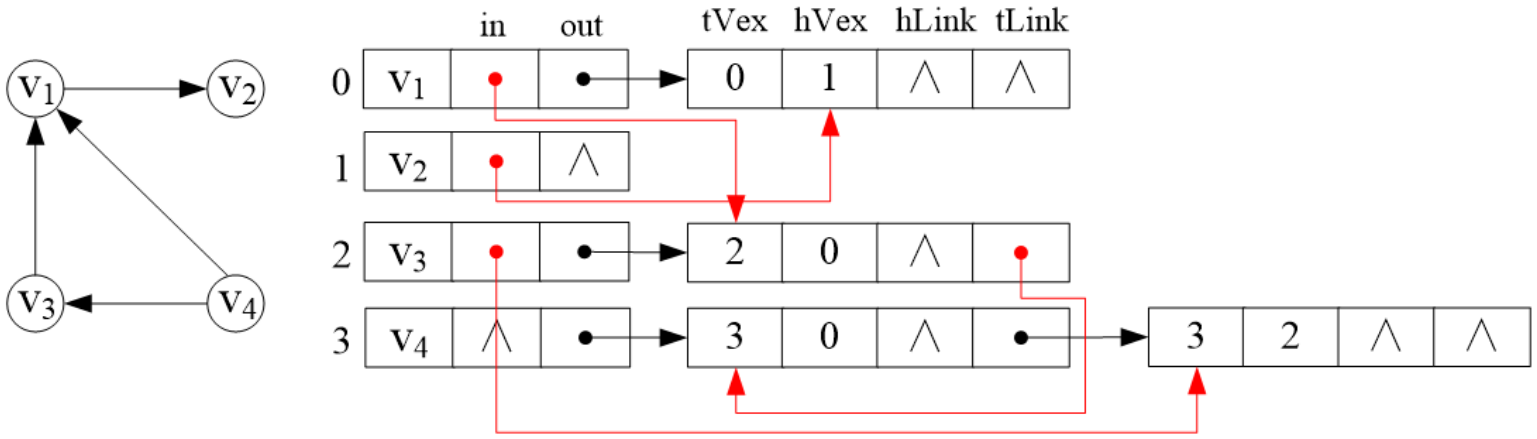


图 6-15 有向图及其十字链表存储

6.2 图的存储结构

- 无向图的存储可以使用邻接表，但在实际使用时，如果想对图中某顶点进行实操（修改或删除），由于邻接表中存储该顶点的节点有两个，因此需要操作两个节点。
- 为了提高在无向图中操作顶点的效率，本节学习一种新的适用于存储无向图的方法——邻接多重表。

图 1 邻接多重表各首元节点的结构示意图

图 2 邻接多重表中其他节点结构

data：
存储此顶点的数据；
firstedge：
指针域，用于指向同该顶点有直接关联的存储其他顶点的节点

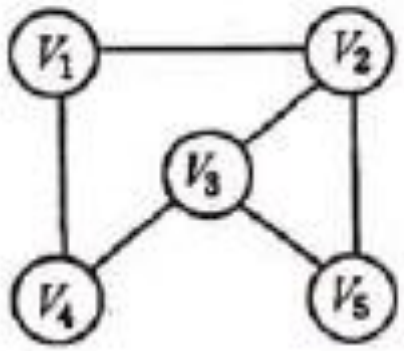
mark：标志域，用于标记此节点是否被操作过，例如在对图中顶点做遍历操作时，为了防止多次操作同一节点，mark 域为 0 表示还未被遍历；mark 为 1 表示该节点已被遍历；
ivex 和 jvex：数据域，分别存储图中各边两端的顶点所在数组中的位置下标；
ilink：指针域，指向下一个存储与 ivex 有直接关联顶点的节点；
jlink：指针域，指向下一个存储与 jvex 有直接关联顶点的节点；
info：指针域，用于存储与该顶点有关的其他信息，比如无向网中各边的权；

6.2 图的存储结构

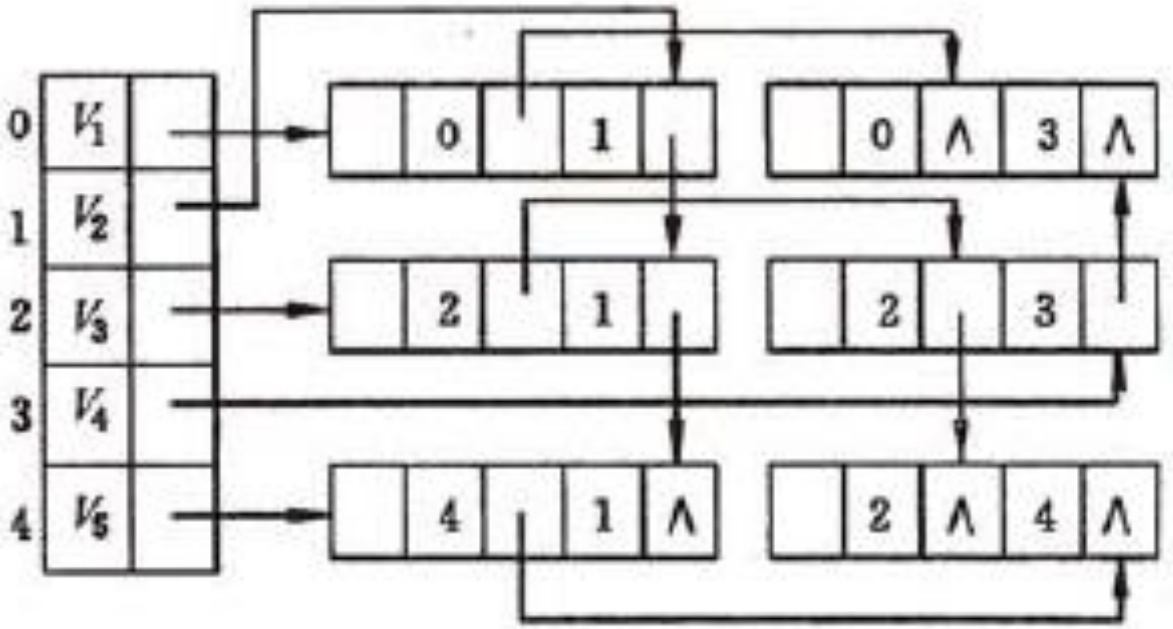
mark	ivex	ilink	jvex	jlink	info

图 2 邻接多重表中其他节点结构

无向图 G_2



无向图 G_2 的邻接多重表



6.3 图的遍历

- 图的遍历要比树的遍历复杂得多，在遍历过程中可能出现两个问题：
  - 图的任一顶点都可能和其余顶点相邻接，因此在访问了某顶点之后，可能顺着某条边又访问到了已访问过的顶点，即图中存在回路问题；
  - 如果图不是连通的，从起点出发可能到达不了所有其它顶点。

6.3 图的遍历

- 在图的遍历过程中，需设置一个标志位数组Visited[n]，用来标记每个顶点是否被访问过；
- 其初始值均为false；
- 一旦某个顶点i被访问，其Visited[i]标志被置为true，从而防止了该顶点被多次访问；
- 当遍历算法一趟结束时，就可以通过检查标志位数组Visited[n]来查看是否所有的顶点都已被访问；
- 如果还有顶点未被访问，则可从某个未被标记的顶点开始继续遍历。

6.3 图的遍历

- 图的遍历实质就是查找每个顶点的邻接点的过程。
- 通常有两种方式：
  - 深度优先搜索(DFS)
  - 广度优先搜索(BFS)
- 它们对无向图和有向图都适用。

结论：

- 图的遍历是将非线性结构数据进行线性化的过程，结点的访问顺序至关重要。

6.3 图的遍历-代码

算法6.4：图遍历---主程序

```
1.  void graph_traverse(Graph& G, bool useDFS){
2.  for(int i = 0; i < G.VerticesNum(); i++)  //所有顶点的标志位初始化
3.          G.Visited[i] = false;
4.  for(int i = 0; i < G.VerticesNum(); i++){
5.          if(G.Visited[i] == false){          //检查是否有未访问顶点?
6.                  if(useDFS) DFS(G,i);        //深度优先搜索
7.                  else BFS(G,i);              //广度优先搜索
8.          } // if
9.  } // for ...
10. }
```

6.3 图的遍历-深度优先搜索

- 深度优先搜索(Depth-First Search，DFS)遍历类似于树的先根遍历。其特点是按照顶点邻接关系，尽可能先对纵深方向进行搜索。

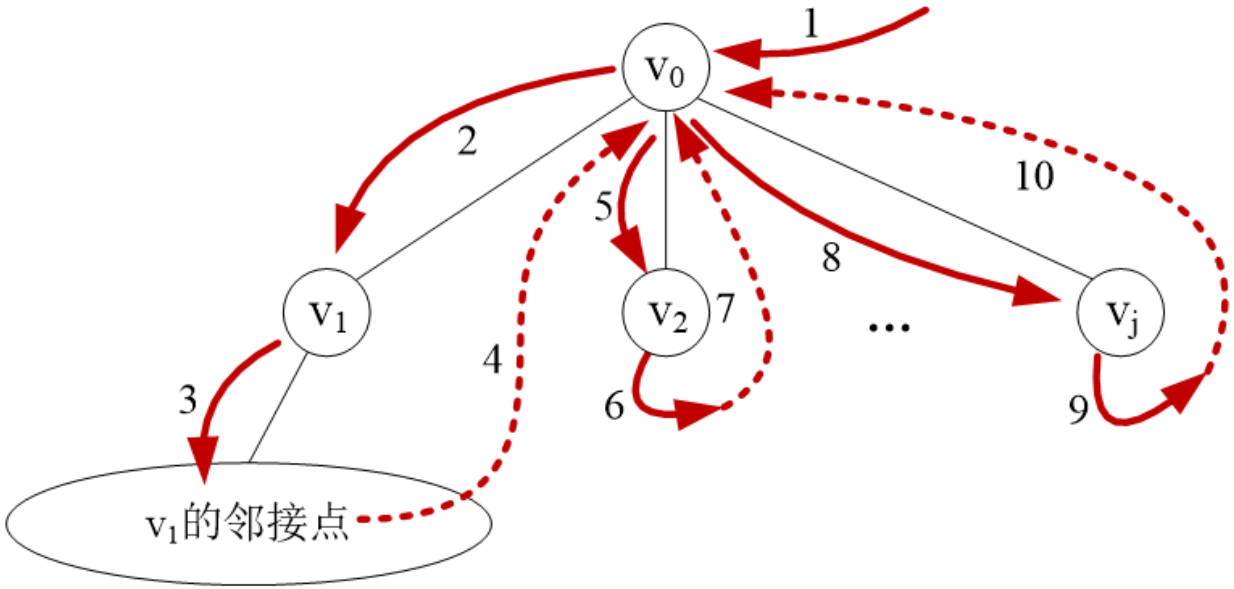


图 6-17 DFS 的基本思想

6.3 图的遍历-深度优先搜索

- 深度优先搜索DFS的基本思想如下：

（1）初始状态为图中所有的顶点都未被访问，即标志位Visited置为false。从图中选取一个顶点v_0开始搜索。设$v_0 \in V$是源点，访问顶点v_0并将其标志位Visited置为true，然后访问v_0的未被访问过的邻接点v_1；

（2）再从顶点v_1出发递归地按照深度优先搜索的方式遍历；

（3）当遇到一个所有邻接点都被访问过了的顶点v_i时，则退回到调用它的顶点v_j；

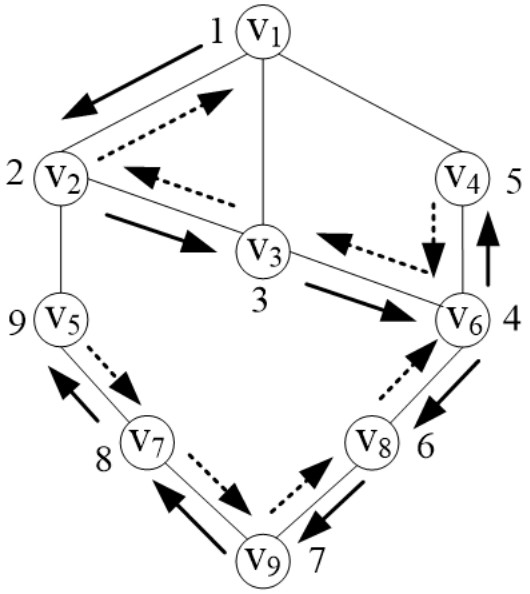
（4）再从顶点v_j继续递归地按照深度优先搜索的方式遍历；

（5）重复上述过程直到从v_0出发的所有边都已检测过为止。此时，图中所有与源点v_0有路径可达的顶点都已被访问过；

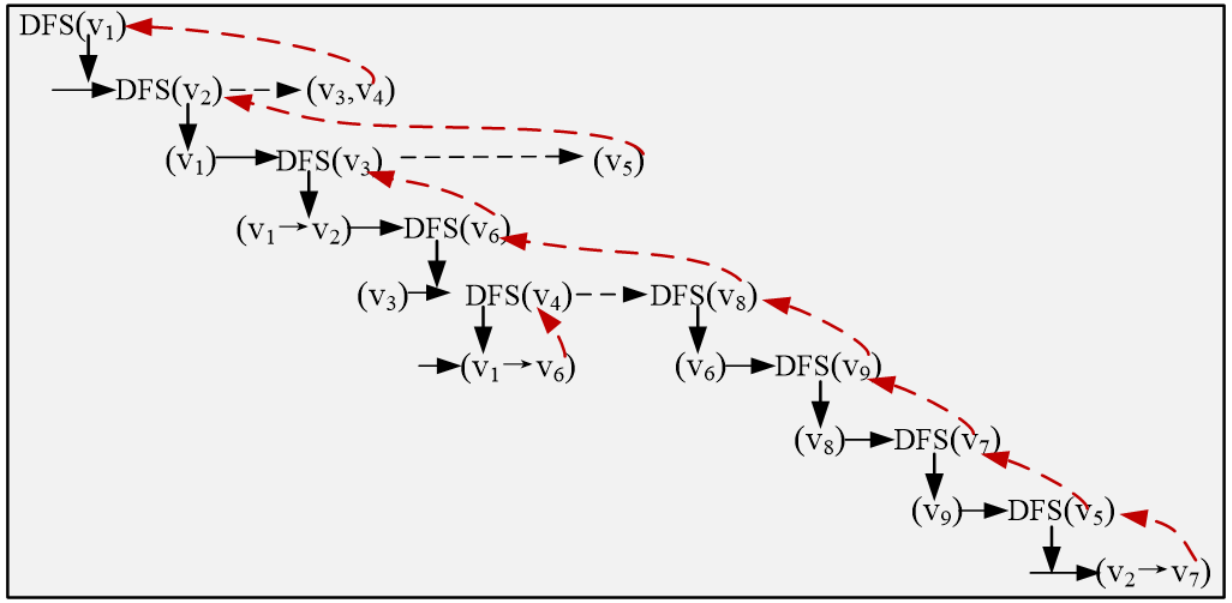
（6）若此时图中尚有顶点未被访问，则另选图中一个未曾被访问过的顶点作为新源点，重复上述过程，直至图中所有顶点都被访问到为止。

6.3 图的遍历-深度优先搜索

- 例子：无向图G_7为例说明深度优先搜索过程。



(a) 无向图 G_7 的 DFS 结果



(b) 无向图 G_7 的 DFS 遍历过程

图 6-19 无向图 G_7 深度优先搜索的遍历过程

结论：

深度优先搜索得到一棵树：n个结点，(n-1)条边，无回路，是一棵深度优先生成树。（访问过的点和用到的边）

6.3 图的遍历-深度优先搜索

- 深度优先搜索是一个递归的搜索过程。其递归算法实现如下：

算法6.5：深度优先搜索递归算法

```
1.  void DFS(Graph& G, int V){  //从顶点V开始一趟DFS遍历
2.      G.Visited[V] = true;  //标记顶点V
3.      cout<< V << "\t";//打印输出
4.      for(int w = G.FirstAdj(V); w != -1; w = G.NextAdj(V, w)) //对该点所有邻接点
        进行深度优先搜索
5.          if(G.Visited[w] == false) //检查该邻接点是否未被访问
6.              DFS(G, w);    //对未访问的邻接点递归调用DFS
7.  }
```

6.3 图的遍历-深度优先搜索

性能分析

- 对于具有n个顶点和e条边的图，深度优先搜索算法对图中每个顶点至多调用一次DFS函数，因为一旦某顶点被访问后，其标志位Visited就被置为true（已被访问），就不再从它出发进行搜索。
- 因此图的深度优先搜索时间主要耗费在从某顶点出发搜索它的所有邻接点上，这取决于所采用的存储结构。
- 当用邻接矩阵表示图时，查找每一个顶点的邻接点都要从头扫描该顶点所在行O(n)，因此，遍历全部顶点所需的时间为$O(n^2)$；
- 而当用邻接表表示图时，查找所有顶点的邻接点所需的时间为O(e)。因此，深度优先搜索图的时间复杂度为O(n+e)。

6.3 图的遍历-广度优先搜索

- 广度优先搜索(Breadth-First Search，BFS)遍历类似于树的按层次遍历。其特点是按路径长度搜索，尽可能先对横向进行搜索。

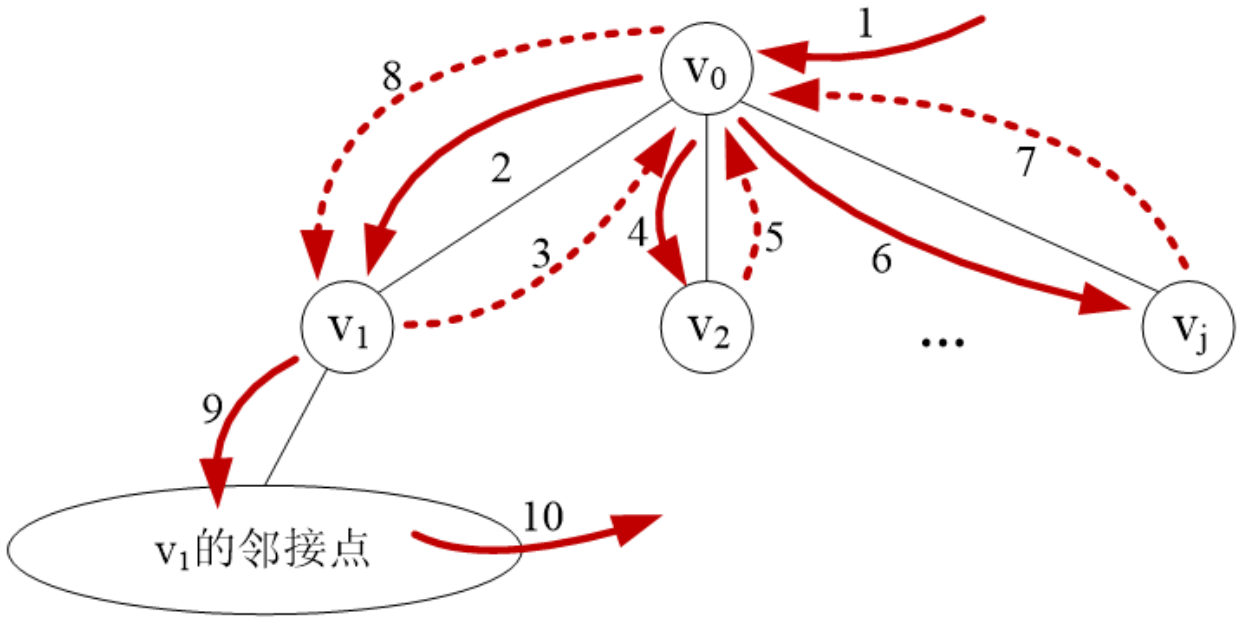


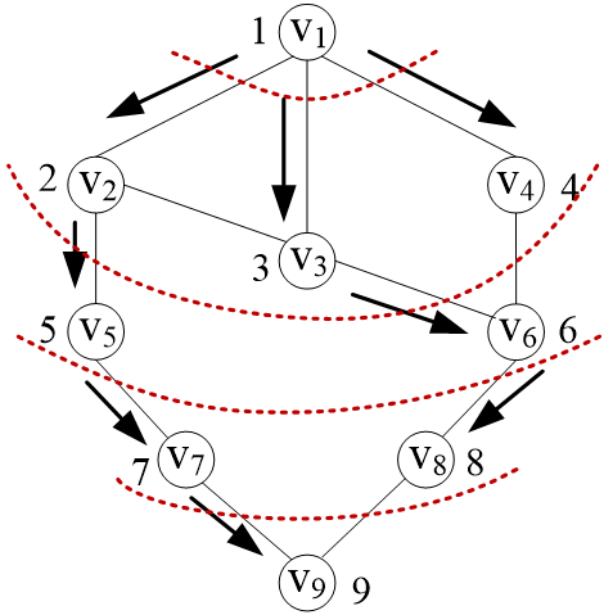
图 6-21 BFS 的基本思想

6.3 图的遍历-广度优先搜索

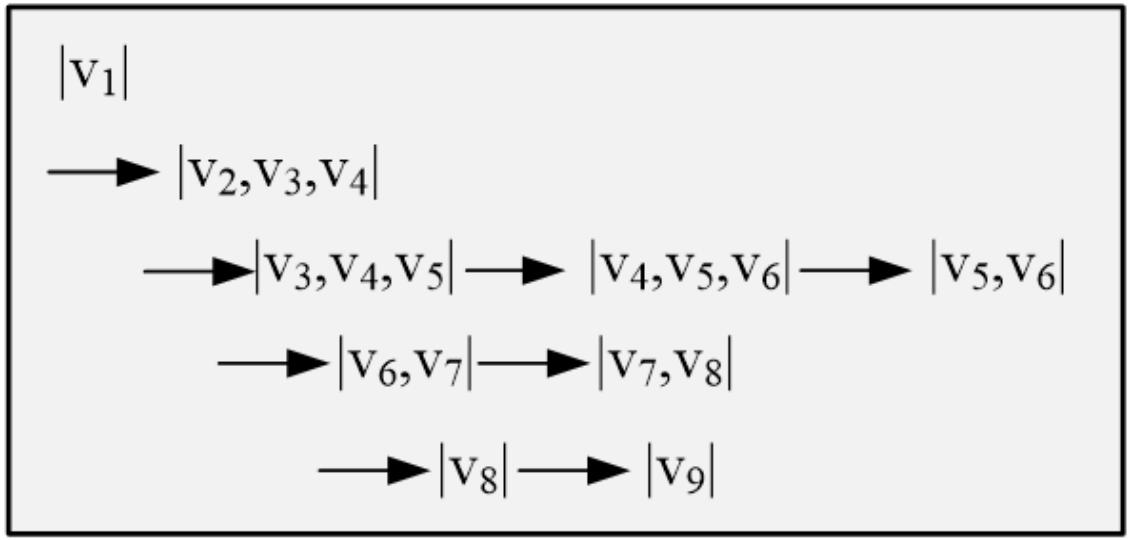
- 广度优先搜索基本思想如下：
  - 1）假设初始状态为图中所有的顶点都未被访问，即标志位Visited置为false。从图中的某个顶点v_0出发，访问顶点v_0并标记其Visited为true；
  - 2）横向搜索v_0的各个未被访问过的直接邻接点w_1，w_2，…，w_m；
  - 3）在依次访问完v_0的各个未被访问过的邻接点之后，再分别按照这些顶点被访问的先后次序，依次访问与它们邻接的所有未被访问过的顶点，直至图中所有已被访问的顶点的邻接点都被访问到；
  - 4）若此时图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止。
  - 广度优先遍历算法**采用队列结构保存按层遍历需要访问的顶点。**

6.3 图的遍历-广度优先搜索

- 无向图G_7为例说明广度优先搜索过程。



(a) 无向图 G_7 的 BFS 结果



(b) 无向图 G_7 的 BFS 遍历过程

图 6-22 无向图 G_7 广度优先搜索的遍历过程

6.3 图的遍历-广度优先搜索

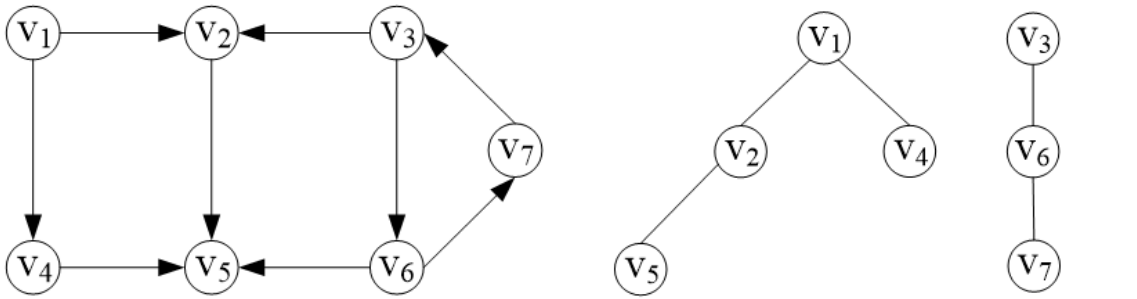
算法6.6：广度优先搜索非递归算法

```
1.  void BFS(Graph& G, int V){
2.  queue<int>  Q; //初始化广度优先搜索所用队列
3.  G.Visited[V] = true;
4.  Q.enqueue(V);          //V入队
5.  while( ! Q.empty() ){    //如果队列仍然有元素
6.          int v = Q.front();
7.          Q.dequeue();        //出队
8.          cout << v << "\t";  //访问输出对头顶点
9.          for(int w = G.FirstAdj(v); w != -1; w = G.NextAdj(v, w)){
10.             if(G.Visited[w] == false){ //所有未被访问过的顶点入队
11.                     G.Visited[w] = true;
12.                     Q.enqueue(w);
13.             }
14.         } //for(...)
15. } //while(...)
16. }
```

- 为了实现广度优先搜索，引入一个队列Q，将访问到的顶点依次入队，以便按顶点入队的先后顺序访问它们的邻接点。
- 广度优先搜索是一个非递归算法。

6.3 图的遍历-广度优先搜索

- 广度优先搜索实质上与深度优先搜索相同，只是对顶点的访问顺序不同而已。因此两者的时间复杂度也相同。
- 一般情况下，广度优先搜索和深度优先搜索的结果是不同的，且对有向图的遍历可能得到一个生成森林，每调用一次遍历程序就生成一棵生成树，如图6-24有向图G_8广度优先遍历得到的生成森林。



(a) 有向图 G_8　　(b) 有向图 G_8 的 BFS 生成森林

图 6-24 有向图 G_8 及其广度优先生成森林

6.4 图的应用1-拓扑排序

- 把子项目、工序或课程看成是图中的一个顶点，称之为活动。用图中的有向边来表示各活动之间的先后关系，如果从顶点v_i到v_j之间存在有向边$<v_i, v_j>$，则表示活动v_i必须优先于活动v_j进行。这样的DAG图称为顶点表示活动的网(Activity On Vertex Network)，简称AOV网。
- 在AOV网中不能存在回路，因为回路中隐含了相互冲突的条件，可能会出现某个活动将以自己为先决条件的情况，因此回路上的所有活动都将无法进行。

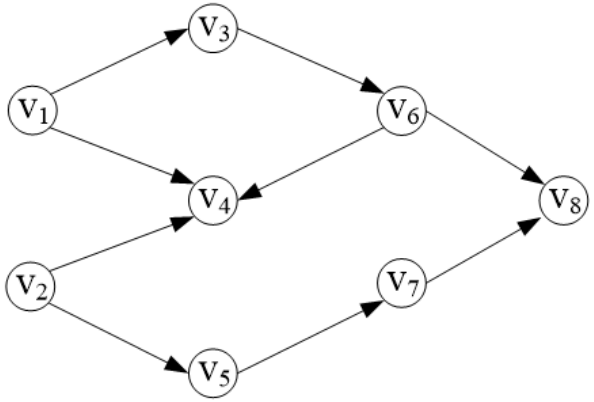


图 6-25 表示活动间先后关系的 AOV 网

6.4 图的应用1-拓扑排序

- 对于一个AOV网，常常需要将它的所有活动按照它们之间的先后关系排成一个线形序列，使得在这个线性序列中，任何一个活动不依赖于排在它后面的活动。这种序列称为拓扑序列(Topological Order)，构造拓扑序列的过程称为拓扑排序(Topological Sort)。

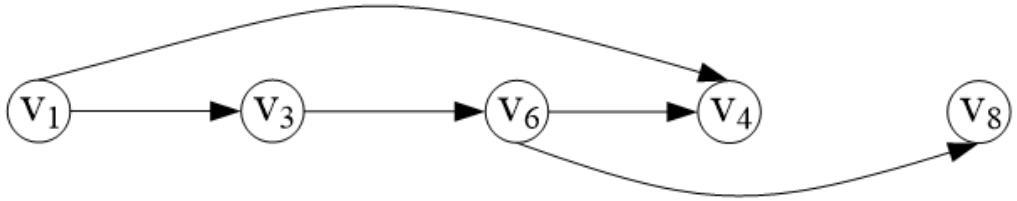


图 6-26 活动{v_1, v_3, v_6, v_4, v_8}间的先后关系

结论：拓扑排序必须满足以下要求：

(1) 有向图；(2) 图中没有回路。

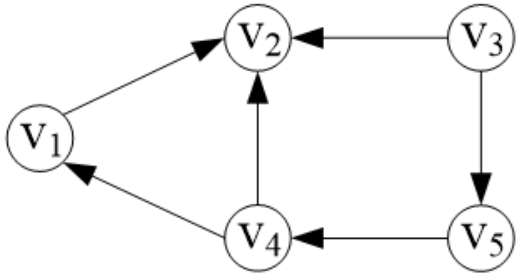
6.4 图的应用1-拓扑排序

算法6.7：深度优先拓扑排序

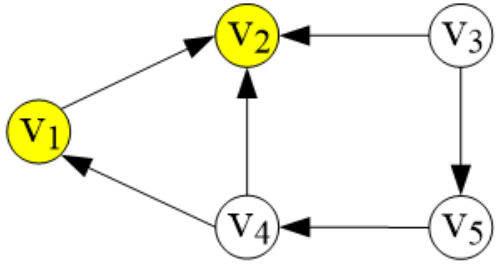
```
1.  void TopSort_DFS(Graph& G){  //深度优先拓扑序的主程序
2.      for(int i = 0; i < G.VerticesNum(); i++) //将图中所有顶点的标志位初始化为false
3.          G.Visited[i] = false;
4.      for(int i = 0; i < G.VerticesNum(); i++)
5.          if(G.Visited[i] == false)  Do_TopSort(G,i);  //深度优先搜索
6.  }
7.  void Do_TopSort(Graph& G, int V){
8.      G.Visited[V] = true;
9.      for(int w = G.FirstAdj(V); w != -1; w = G.NextAdj(V, w))
10.         if(G.Visited[w] == false)  Do_TopSort(G, w);
11.     cout << V << "\t";
12. }
```

6.4 图的应用1-拓扑排序

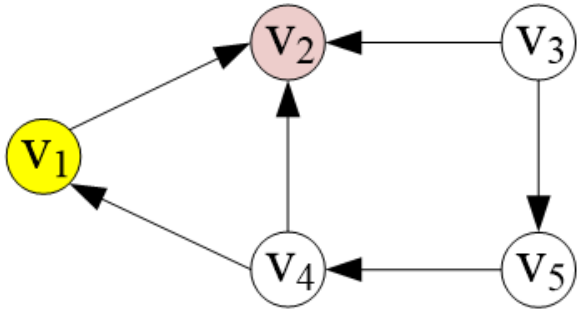
- AOV网深度优先拓扑排序的实现过程：



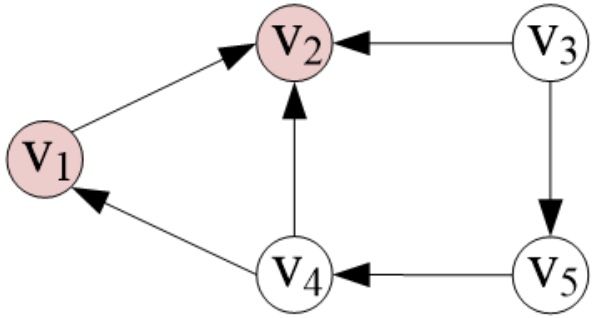
(a)



(b)



(c)



(d)

6.4 图的应用1-拓扑排序

- AOV网深度优先拓扑排序的实现过程：

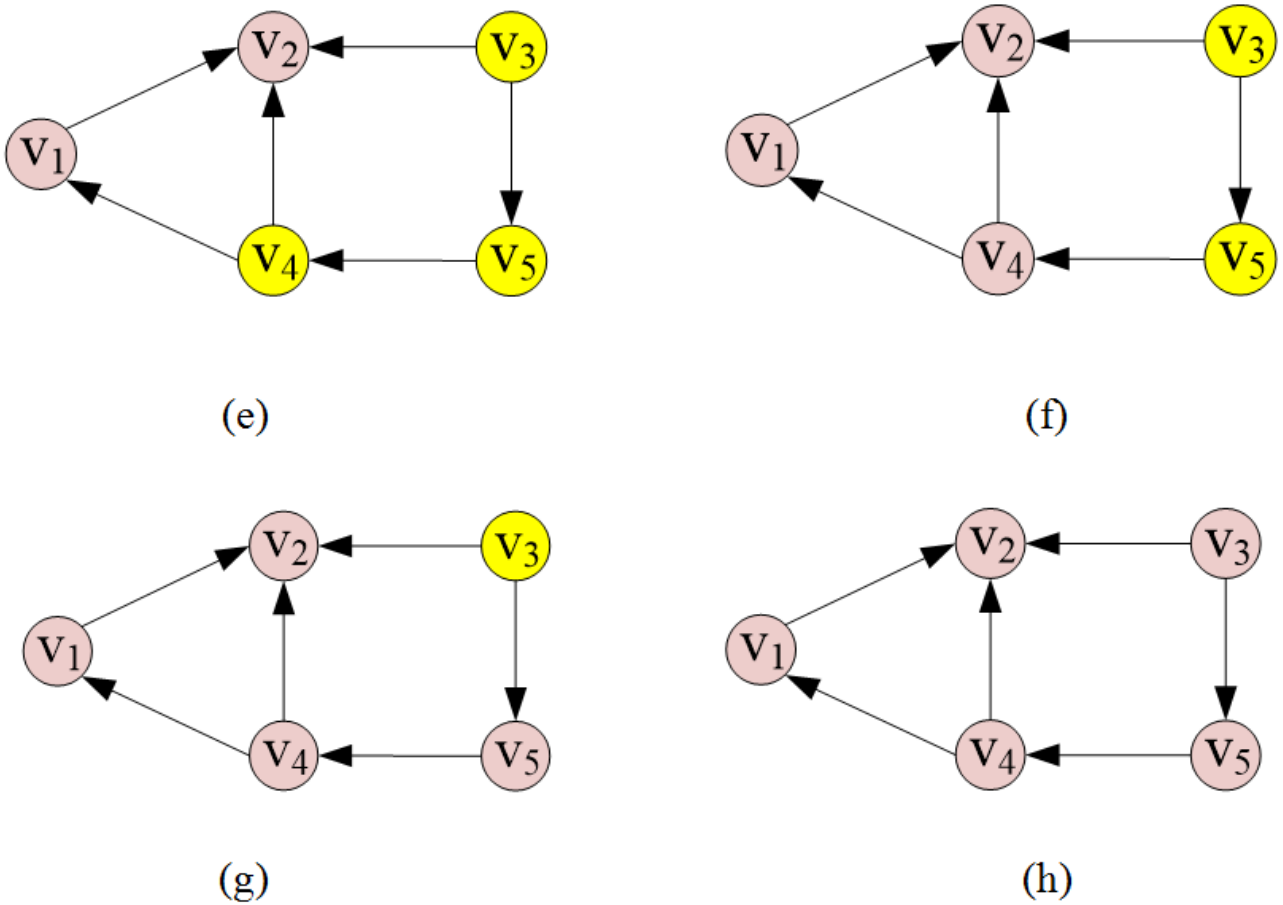


图 6-27 AOV 网深度优先拓扑排序的实现过程

利用深度优先搜索进行拓扑排序的方法只适用于DAG图，对于存在回路的图将给出错误的结果。（深度优先拓扑排序输出是一个逆拓扑序）

6.4 图的应用1-拓扑排序

- 第二种方法是广度优先拓扑排序，即使用队列代替递归来实现拓扑排序。具体步骤如下：

  (1) 计算各个结点的入度，邻接矩阵的行 为出度数，列为入度数（将1相加）；

  (2) 所有入度为0的结点放进一个队列；

  (3) 如果队列为空，则转到第(7)步；

  (4) 如队列非空，将队头元素从队列中删除，并输出队头元素；

  (5) 把它的每一个邻接点的入度减1，如果减1后入度变为0，则把这个顶点立刻入队；

  (6) 转到第(3)步，循环执行，直到队列为空；

  (7) 如果所有的顶点均输出，则给出一个广度优先拓扑序，如果有剩余的顶点未被输出，且当前队列已经为空，则认为存在回路。

6.4 图的应用1-拓扑排序

★注：拓扑排序也是检验回路的一种方法。

- 表6-1为图6-28所示AOV网广度优先拓扑排序的实现过程。

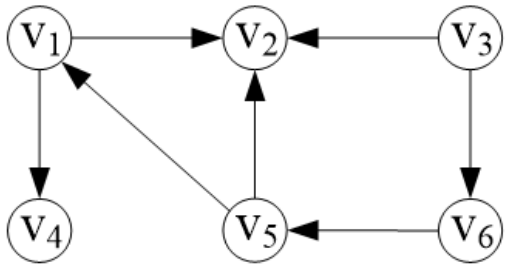


图 6-28 AOV 网

表 6-1 AOV 网广度优先拓扑排序的实现过程

步骤	Indegree[]						队列	输出结点
	v_1	v_2	v_3	v_4	v_5	v_6		
初态	1	3	**0**	1	1	1	v_3	
1	1	**2**	0	1	1	**0**	v_6	v_3
2	1	2	0	1	**0**	0	v_5	v_6
3	**0**	**1**	0	1	0	0	v_1	v_5
4	0	**0**	0	**0**	0	0	v_2, v_4	v_1
5	0	0	0	0	0	0	v_4	v_2
6	0	0	0	0	0	0		v_4

6.5 图的应用2-最短路径

- 路径代价为路径上所有边的权值之和。
- 假定两个顶点v_i和v_j，从v_i到v_j有多条路径存在时，权值最小（即路径代价最小）的路径称为这两个顶点之间的最短路径。
- 对于给定的两个顶点v_i和v_j，最短路径问题需要解决两个问题：
  - (1) 从v_i到v_j是否有路径可达？
  - (2) 如果存在多条路径时，哪个路径最短？

6.5 图的应用2-最短路径

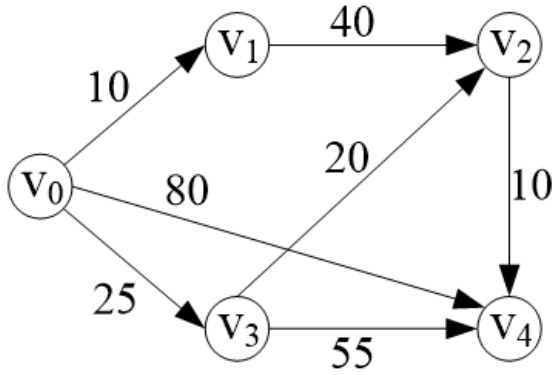


图 6-31 带权有向图

- 如图6-31所示的带权有向图，从顶点v_0到顶点v_4共有四条路径：

①路径(v_0, v_4)：路径代价为80；

②路径(v_0, v_3, v_4)：路径代价为80；

③路径(v_0, v_1, v_2, v_4)：路径代价为60；

④路径(v_0, v_3, v_2, v_4)：路径代价为55。

因此，从顶点v_0到顶点v_4的最短路径为(v_0, v_3, v_2, v_4)。

6.5 图的应用2-单源点最短路径问题

6.6.1 单源点最短路径问题

- 设给定一个带权有向图G = (V, E)，假设源点为v_s，求从v_s到G中其余各顶点之间的最短路径，这类问题就称为单源点最短路径问题。
- 解决单源点最短路径问题的一个常用算法是Dijkstra算法，它是由E.W.Dijkstra提出的一种按最短路径长度递增的次序产生最短路径的算法。

6.5 图的应用2-单源点最短路径问题

- Dijkstra算法的基本思想是：
  - 将图的顶点集划分为两个集合S和V-S，集合S存放已经确定最短路径的终点集合；
  - 初始状态时，S中只包含源点v_s，每求得一条最短路径，就将该路径的终点加入到集合S中，直到全部顶点都加入到集合S中，即S = V为止。

6.5 图的应用2-单源点最短路径问题

- 为了便于描述，引入一个辅助数组dist[]，它的每一个分量dist[i]表示当前所找到的从源点v_s到终点v_i的最短路径长度。
- 它的初始状态是：
  - 若$v_s = v_i$，dist[i] = 0；
  - 若从源点v_s到终点v_i存在弧$<v_s, v_i>$，则dist[i] = G.weight(s,i)；
  - 若不存在弧$<v_s, v_i>$，则dist[i] = ∞。
- 显然，长度为 $\text{dist}[j] = \min_i\{\text{dist}[i] \mid v_i \in V - S\}$ 的路径就是从源点v_s出发的长度最短的一条最短路径，即(v_s, v_j)，将顶点v_j加入集合S中。

6.5 图的应用2-单源点最短路径问题

注意：

- 如果求得一条从v_i到v_j的最短路径$(v_i, \cdots, v_k, v_j)$，v_k是v_j前面的一个顶点，那么$(v_i, \cdots, v_k)$也必定是从v_i到v_k的最短路径，且该路径上的顶点均在集合S中。
- 因此，在一般情况下，下一条长度次短的最短路径，设其终点为v_k，则或者是边(v_s, v_k)，或者是经过中间顶点$v_j(v_j \in S)$的路径$(v_s, \cdots, v_j, v_k)$。其长度为：

$$dist[k] = \min_i \{dist[i] \mid v_i \in V - S\}$$

- 其中，dist[i]或者等于dist[k]，或者等于G.weight(s, k)。

6.5 图的应用2-单源点最短路径问题

- 根据上面的分析，Dijkstra算法的基本步骤可描述如下：

(1) 令$S=\{v_s\}$，并对dist[i]数组按下面的公式赋初值：

$$\text{dist}[i]=\begin{cases}0 & v_s=v_i\\ \infty & \text{如果 } i\neq 0,\ <v_s,v_i>\notin E\\ G.\text{weight}(s,i) & \text{如果 } i\neq 0,\ <v_s,v_i>\in E,\ \text{weight}(s,i)\text{为权值}\end{cases}$$

(2) 选择顶点v_j，使得

$$\text{dist}[j]=\min_i\{\text{dist}[i]\mid v_i\notin S\}$$

(3) v_j即为所求下一条最短路径的终点，将v_j并入到集合S中，即

$$S=S\cup\{v_j\}$$

6.5 图的应用2-单源点最短路径问题

(4)对$v_i \notin S$，修改dist[i]的值。由于(3)中对S新增加了v_j，集合{V-S}中与v_j相连的顶点v_i的dist[i]值可能变小，因为可能存在一条从v_s通过v_j再到达v_i的路径$(v_s, \cdots, v_j, v_i)$，其长度小于原先的dist[i]，即

$$\text{dist[j]} + \text{G. weight(j, i)} < dist[i]$$

此时应修改dist[i]，使得

$$\text{dist[i]} = \text{dist[j]} + \text{G. weight(j, i)}$$

(5) 重复步骤(2)、(3)和(4)，直到$S = V$为止。这样就求出了图中从源点v_s到其它顶点的最短路径。

6.5 图的应用2-单源点最短路径问题

- 表6-4为图6-31所示带权有向图用Dijkstra算法求解最短路径过程中dist数组的变化情况。

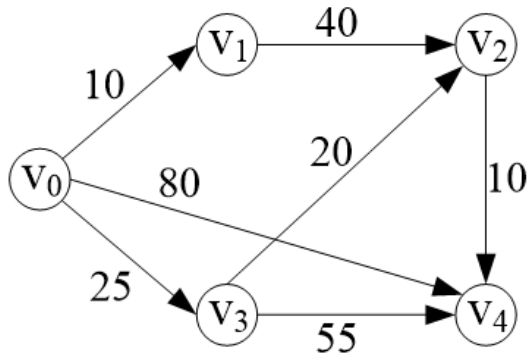


图 6-31 带权有向图

表 6-4 Dijkstra 求解过程中 dist 数组变化情况

步骤	S	dist[0]	dist[1]	dist[2]	dist[3]	dist[4]
初态	$\{v_0\}$	0	**10**	∞	25	80
1	$\{v_0, v_1\}$	0	10	50	**25**	80
2	$\{v_0, v_1, v_3\}$	0	10	**45**	25	80
3	$\{v_0, v_1, v_3, v_2\}$	0	10	45	25	**55**
4	$\{v_0, v_1, v_3, v_2, v_4\}$	0	10	45	25	55

6.5 图的应用2-单源点最短路径问题

为了存储最短路径，再引入一个辅助数组pre，pre[j]记录路径上的前一个顶点，即pre[j] = i。

- Dijkstra算法实现如下：
- //用Dijkstra算法求带权有向图G的顶点s到其余顶点的最短路径和路径长度

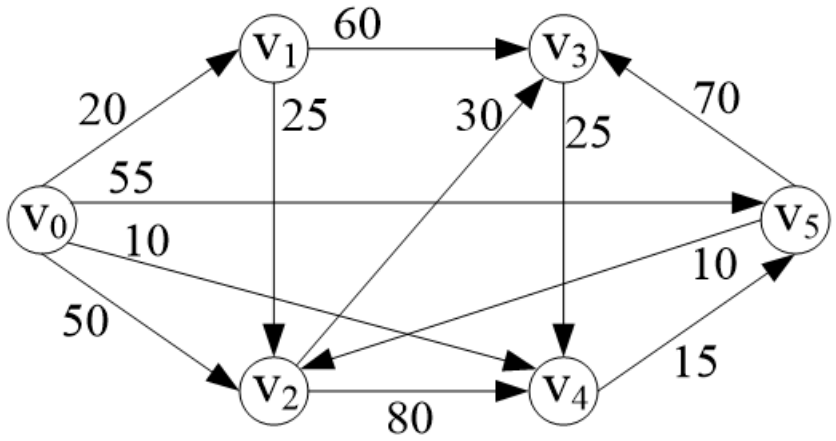
```
1.  void Dijkstra(Graph& G, int *dist, int *pre, int s){
2.      int i, j, n = G.VerticesNum();
3.      bool *S = new bool[n];          //最短路径终点集S
4.      for(i = 0; i < n; i++){         //初始化dist、pre、S
5.          S[i] = false;
6.          if(i != s && G.weight(s,i) == 0)
7.              dist[i] = maxValue;
8.          else
9.              dist[i] = G.weight(s,i);
10.         if(i != s && dist[i] < maxValue)
11.             pre[i] = s;
12.         else
13.             pre[i] = -1;
14.     }
```

6.5 图的应用2-单源点最短路径问题

```
1.    S[s] = true;
2.      int Min, v;
3.      for(i = 0; i < n-1; i++){
4.          Min = maxValue;
5.          v = s;
6.          for(j = 0; j < n; j++){
7.              if (S[j] == false && dist[j] < Min){
8.                  v = j;
9.                  Min = dist[j];
10.             }
11.         }
12.         S[v] = true;                    //将顶点v加入到集合S中
13.         for(j = 0; j < n; j++){         //修改
14.             if(S[j] == false && G.weight(v,j) != 0 && dist[v] + G.weight(v,j) < dist[j]){
15.                 dist[j] = dist[v] + G.weight(v,j);
16.                 pre[j] = v;
17.             }
18.         }
19.     }
20. }
```

6.5 图的应用2-单源点最短路径问题

- 对图6-32所示的带权有向图使用Dijkstra算法求从顶点v_0到其余顶点的最短路径，求解过程中辅助数组和的变化情况如表6-5所示。



$$
\begin{array}{c}
 \\
0 \\
1 \\
2 \\
3 \\
4 \\
5
\end{array}
\begin{bmatrix}
0 & 20 & 50 & \infty & 10 & 55 \\
\infty & 0 & 25 & 60 & \infty & \infty \\
\infty & \infty & 0 & 30 & 80 & \infty \\
\infty & \infty & \infty & 0 & 25 & \infty \\
\infty & \infty & \infty & \infty & 0 & 15 \\
\infty & \infty & 10 & 70 & \infty & 0
\end{bmatrix}
$$

图 6-32 带权有向图及其邻接矩阵

6.5 图的应用2-单源点最短路径问题

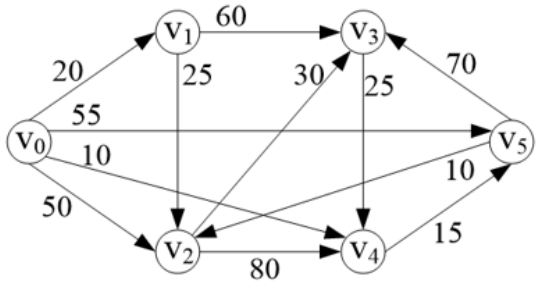


表 6-5 Dijkstra 算法中辅助数组 dist 和 pre 的变化情况

步骤 \ 顶点		v_1	v_2	v_3	v_4	v_5	S
初态	dist	20	50	∞	**10**	55	$\{v_0\}$
	pre	0	0	-1	**0**	0	
1	dist	**20**	50	∞	10	25	$\{v_0, v_4\}$
	pre	**0**	0	-1	0	4	
2	dist	20	45	80	10	**25**	$\{v_0, v_4, v_1\}$
	pre	0	1	1	0	**4**	
3	dist	20	**35**	80	10	25	$\{v_0, v_4, v_1, v_5\}$
	pre	0	**5**	1	0	4	
4	dist	20	35	**65**	10	25	$\{v_0, v_4, v_1, v_5, v_2\}$
	pre	0	5	**2**	0	4	
5	dist	20	35	65	10	25	$\{v_0, v_4, v_1, v_5, v_2, v_3\}$
	pre	0	5	2	0	4	

6.5 图的应用2-任意对顶点之间的最短路径

6.6.2 任意对顶点之间的最短路径

- 另一种求每对顶点之间最短路径的方法——Floyd算法。
- Floyd算法的基本思想是：从任意顶点v_i到任意顶点v_j的最短路径只有两种情况，一种是直接从顶点v_i到顶点v_j，另一种是从顶点v_i经过若干个中间顶点到顶点v_j。因此设一个顶点集合S表示路径上可能含的中间顶点的集合，其初始状态为空。

6.5 图的应用2-任意对顶点之间的最短路径

- 引入一个二维数组dist，其每个分量dist[i][j]存储求得的从顶点v_i只经过集合S中的中间顶点到达顶点v_j的最短路径的长度。初始状态时由于S为空，因此$dist_0[i][j]$为从v_i不经过任何顶点直接到达v_j的路径长度，即

$$dist_0[i][j] = \begin{cases} 0 & v_i = v_j \\ \infty & < v_i, v_j > \notin E \\ G.weight(i,j) & < v_i, v_j > \in E，\ weight(i,j)为权值 \end{cases}$$

- 将图中的所有顶点$v_0, v_1, \cdots, v_{n-1}$依次加入到集合S中，每加入一个新顶点就对dist[i][j]的值进行修改，直到所有顶点都加入到集合S中为止，$dist_n[i][j]$就是从顶点v_i到v_j的最短路径长度。$dist_k[i][j]$可以由$dist_{k-1}[i][j]$计算得到，

6.5 图的应用2-任意对顶点之间的最短路径

- 修改$dist_k[i][j]$的方法如图6-33所示。

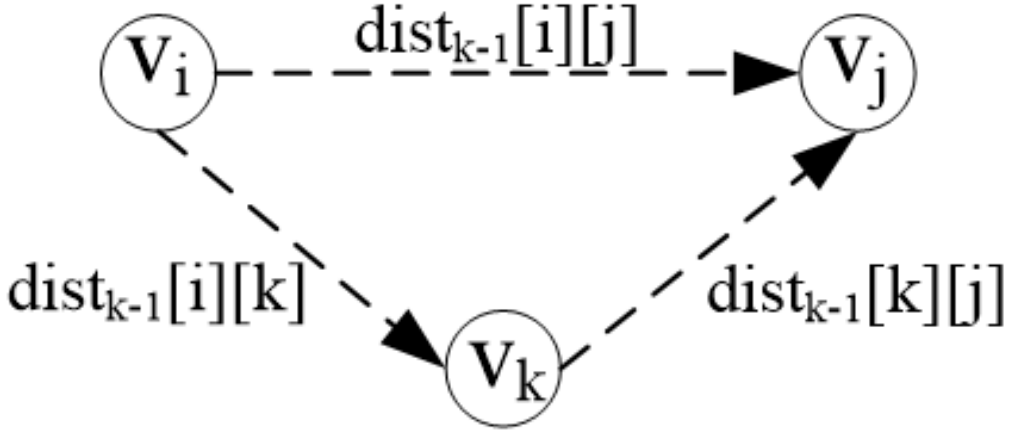


图 6-33 dist[i][j]的修改方法

6.5 图的应用2-任意对顶点之间的最短路径

- 假设当前的$S=\{v_0, v_1, \cdots, v_{k-2}\}$，此时$dist_{k-1}[i][j]$存放的是从$v_i$出发只经过中间顶点$\{v_0, v_1, \cdots, v_{k-2}\}$到达$v_j$的当前最短路径长度，向集合S中加入一个新顶点$v_{k-1}$后，从顶点$v_i$到$v_j$的最短路径有两种可能：

  (1) v_{k-1}为路径上的中间顶点，此时路径为$(v_i, \cdots, v_{k-1}, \cdots, v_j)$，路径长度为$dist_{k-1}[i][k-1] + dist_{k-1}[k-1][j]$；

  (2) v_{k-1}不为路径上的中间顶点，此时路径为$(v_i, \cdots, v_j)$，路径长度为$dist_{k-1}[i][j]$。

- 修改$dist_k[i][j]$的值为上述两种情况中路径长度较小的值，即：

$$dist_k[i][j] = \min\{dist_{k-1}[i][j], dist_{k-1}[i][k] + dist_{k-1}[k][j]\}$$

6.5 图的应用2-任意对顶点之间的最短路径

- 为了记录最短路径上经过的顶点，引入一个二维数组path，其每个分量path[i][j]用于存放从顶点v_i到v_j的最短路径所经过的某个顶点，即改变路径的中间点。
  - 若path[i][j] = k，说明从v_i到v_j的最短路径经过顶点v_k，设该最短路径为$(v_i, \cdots, v_k, \cdots, v_j)$，则由最短路径的最优子结构性质可知该最短路径的子序列$(v_i, \cdots, v_k)$和$(v_k, \cdots, v_j)$一定是从v_i到v_k和从v_k到v_j的最短路径，根据path[i][k]和path[k][j]的值可找到该最短路径上经过的其它顶点，此为一个递归过程。
  - 初始状态时，令path[i][j] = -1，表示从v_i到v_j不经过任何顶点。
  - 当某个顶点v_k加入S后使得dist[i][j]的值变小时，修改path[i][j] = k。

6.5 图的应用2-任意对顶点之间的最短路径

Floyd算法的具体实现如下：

算法6.11：Floyd算法

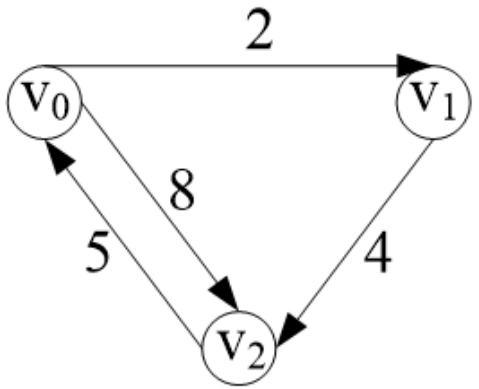
```
1.  void MidNode(int i, int j, int **path){ //输出中间过渡点
2.  if (path[i][j] != -1 && i != path[i][j]){
3.          MidNode(i, path[i][j], path);
4.          cout << path[i][j] << "->";
5.          MidNode(path[i][j], j, path);
6.  }
7.  }
8.  void Output_AllPaths(int n, int **path){ //输出任意对顶点之间的最短路径
9.  for (int i = 0; i < n; i++)
10.         for (int j = 0; j < n; j++)
11.                 if (i != j && path[i][j] != -1){
12.                         cout << i << "->";
13.                         MidNode(i, j, path);
14.                         cout << j << endl;
15.                 } else cout << i << "->"<< j << "之间无路径可达！" << endl;
16. }
```

6.5 图的应用2-任意对顶点之间的最短路径

```
//用Floyd算法求带权有向图G的任意对顶点之间的最短路径和路径长度
1.    void Floyd(Graph& G, int **dist, int **path){
2.       int i,j,k,n = G.VerticesNum();
3.       for(i = 0; i < n; i++)        //初始化dist、pre
4.          for(j = 0; j < n; j++){
5.             if(i != j && G.weight(i,j) == 0) dist[i][j] = maxValue;
6.             else dist[i][j] = G.weight(i,j);
7.             if(i != j && dist[i][j] < maxValue) path[i][j] = i;
8.             else path[i][j] = -1;
9.          }
10.      for(k = 0; k < n; k++){                    //k为中间过渡点
11.         for(i = 0; i < n; i++)
12.            for(j = 0; j < n; j++){
13.               if (dist[i][k] + dist[k][j] < dist[i][j]){
14.                  dist[i][j] = dist[i][k] + dist[k][j];
15.                  path[i][j] = k;
16.               }        }
17.      }
18.   Output_AllPaths(n, path);              // 输出任意对顶点之间的最短路径
19.   }
```

6.5 图的应用2-任意对顶点之间的最短路径

- 对于图6-34所示的带权有向图，利用Floyd算法求出每对顶点之间的最短路径及其路径长度，在求解过程中，二维数组dist和path中各个分量的变化情况如表6-6所示。



(a)

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array}\begin{bmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{bmatrix}$$

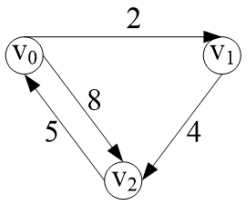
(b)

图 6-34 带权有向图及其邻接矩阵

6.5 图的应用2-任意对顶点之间的最短路径

表 6-6 Floyd 算法中二维数组 dist 和 path 的变化情况

步骤 / 数组	下标	初态	k=0	k=1	k=2
		0 1 2	0 1 2	0 1 2	0 1 2
dist	0	0 2 8	0 2 8	0 2 6	0 2 6
	1	∞ 0 4	∞ 0 4	∞ 0 4	9 0 4
	2	5 ∞ 0	5 7 0	5 7 0	5 7 0
path	0	-1 0 0	-1 0 0	-1 0 1	-1 0 1
	1	-1 −1 1	-1 −1 1	-1 −1 1	2 −1 1
	2	2 −1 −1	2 0 -1	2 0 −1	2 0 −1
S		{ }	{v_0}	{v_0, v_1}	{v_0, v_1, v_2}



$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \end{array}\begin{bmatrix} 0 & 2 & 8 \\ \infty & 0 & 4 \\ 5 & \infty & 0 \end{bmatrix}$$

(a) (b)

图 6-34 带权有向图及其邻接矩阵

6.5 图的应用2-任意对顶点之间的最短路径

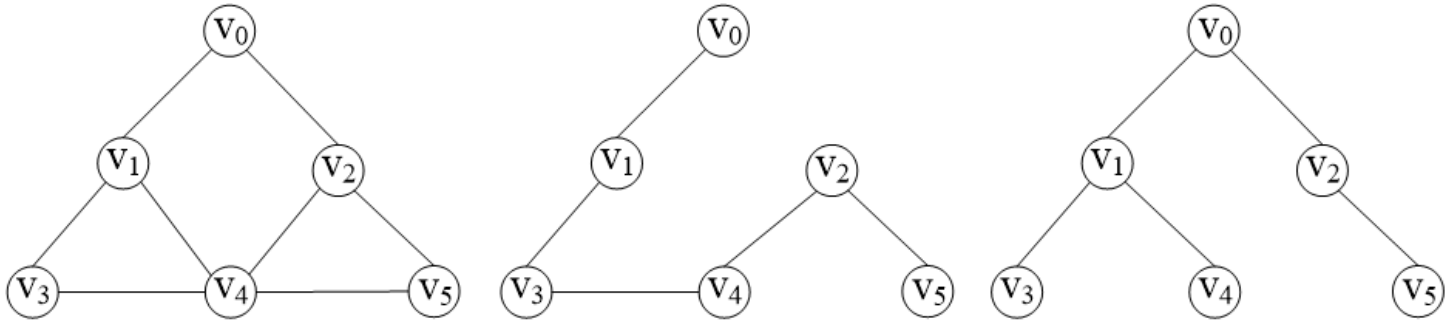
- 根据path数组，可以得到每对顶点的最短路径，图6-34(a)所示的带权有向图的最短路径如表6-7所示。

表 6-7 图 6-34(a)所示带权有向图的最短路径

源点	终点	最短路径	路径长度
v_0	v_1	(v_0, v_1)	2
v_0	v_2	(v_0, v_1, v_2)	6
v_1	v_0	(v_1, v_2, v_0)	9
v_1	v_2	(v_1, v_2)	4
v_2	v_1	(v_2, v_0)	5
v_2	v_0	(v_2, v_0, v_1)	7

6.6 图的应用4：图的最小生成树

- 一个连通无向图的生成树是不唯一的。
  - 构造生成树的方法有多种，如图的深度优先搜索和广度优先搜索等，按照不同的遍历方法，将得到不同的生成树；
  - 从不同的顶点出发，也可能得到不同的生成树；
  - 而且生成树有时还和图的存储结构中具体的结点顺序有关。



(a) 连通图　　(b) 生成树 1　　(c) 生成树 2

图 6-35 连通图及其两棵生成树

6.6 图的应用4：图的最小生成树

结论：

(1) 对连通图才构造最小生成树，非连通图无生成树。

(2) 对带权连通图构造生成树必须满足以下约束条件：

①必须且仅使用图中的n-1条边；

②不能使用产生回路的边。

6.6 图的应用4：图的最小生成树

- 一个生成树的代价为该生成树中所有边的权值之和。对于一个带权连通图，其不同生成树所对应的权值总和（即生成树的代价）也是不相同的，称代价最小的生成树为最小代价生成树(Minimum Cost Spanning Tree)，简称最小生成树(MST)。最小生成树是图的一种重要应用，在描述和求解城市交通道路规划、网络路由选择、城市通信网络架设、汽车导航等实际问题中应用广泛。

★注：同一个带权连通图可能有多棵最小生成树，比如当有两条边权值相等时就可能出现这种情况。

6.6 图的应用4：Prim算法---扩点法

- Prim算法和Dijkstra算法类似，都是以顶点来扩展的，每次找到一条边并加入一个顶点，同时保存从各顶点到当前生成树之间的距离。
- 设G＝(V, E)是一个带权连通图，T＝(U, TE)为欲构造的最小生成树，其中U和TE分别为G的最小生成树T中顶点和边的集合。

6.6 图的应用4：Prim算法---扩点法

Prim算法的基本思想：

(1) 初始化顶点集、边集，即开始时，最小生成树T中只包含一个顶点u_0而没有边；

(2) 在所有$u \in U$，$v \in V-U$的边$(u, v) \in E$中选择一条权值最小的边(u', v')加入到集合TE中，同时将顶点v'并入到U中。

- 重复上述过程直到图中所有的顶点都加入到集合U中，即$U = V$为止，此时TE中有n-1条边，$T = (U, TE)$就是G的最小生成树。

结论：

Prim算法也称扩点法。从一个顶点开始，每加入一条边则扩充一个顶点，加入的顶点一定是与已有生成树相关联的顶点。

6.6 图的应用4：Prim算法---扩点法

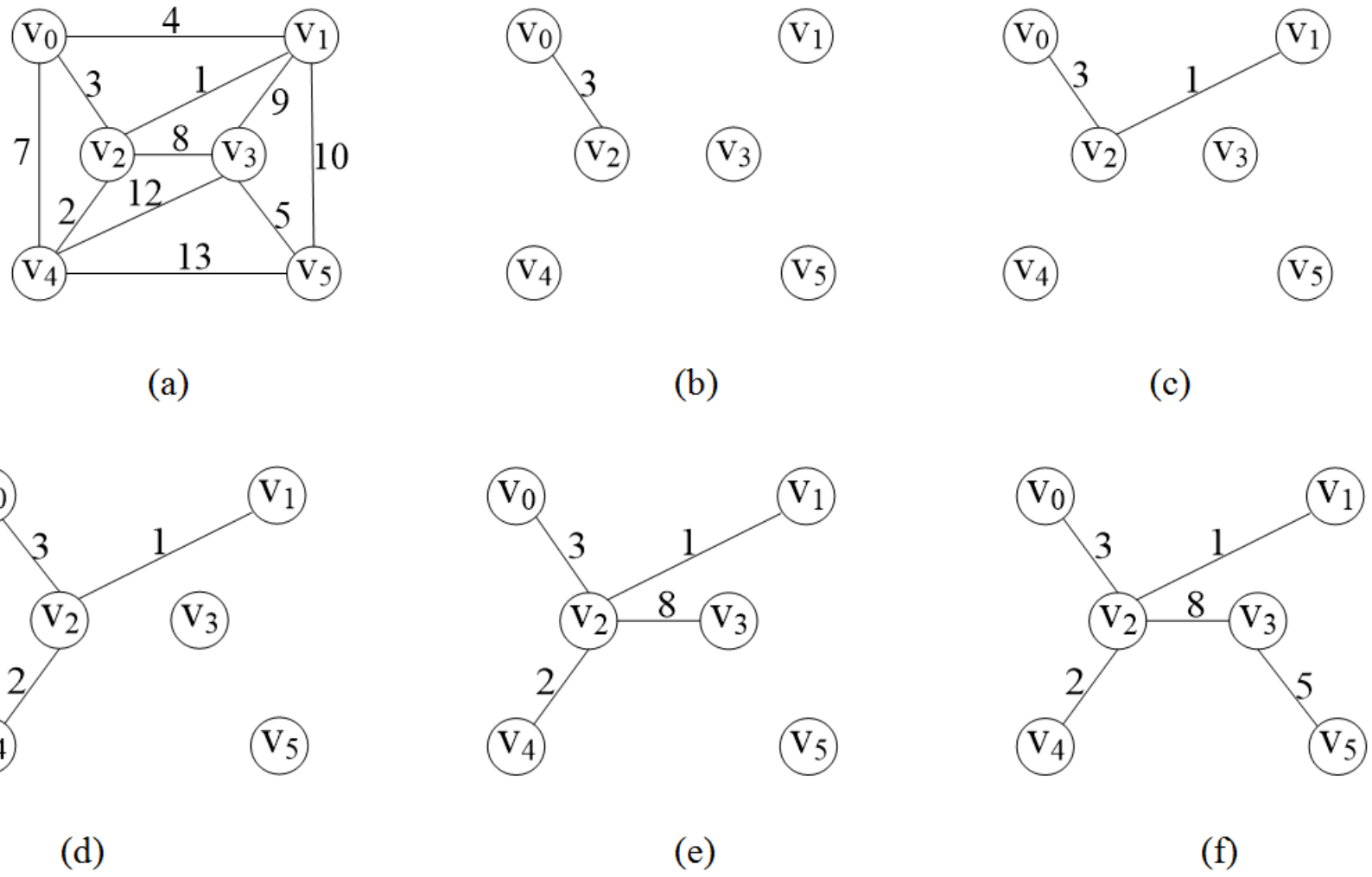


图 6-36 Prim 算法构造最小生成树的过程

6.6 图的应用4：Prim算法---扩点法

假设采用邻接矩阵来存储带权连通图，两个顶点之间不存在边的权值为∞。Prim算法的具体实现如下：

算法6.12：Prim算法

```
//利用Prim算法求带权连通图的最小生成树，用MST来存储最小生成树的边
1.  void Prim(Graph& G, int s, Edge* MST){
2.      int i, j, n = G.VerticesNum();
3.      Edge *MST_Edge = new Edge[n];     //辅助数组MST_Edge
4.      for(i = 0; i < n; i++){           //初始化MST_Edge数组
5.          MST_Edge[i].from = i;
6.          MST_Edge[i].to = s;
7.          if (i != s && G.weight(i,s) == 0)
8.              MST_Edge[i].weight = maxValue;
9.          else
10.             MST_Edge[i].weight = G.weight(i,s);
11.     }
```

6.6 图的应用4：Prim算法---扩点法

```
1.  int v, Min;
2.      for(i = 0; i < n-1; i++){
3.          Min = maxValue;  v = 0;//找到U中顶点到V-U中顶点权值最小的边，并记录顶点v
4.          for(j = 0; j < n; j++)
5.              if (MST_Edge[j].weight != 0 && MST_Edge[j].weight < Min){
6.                  Min = MST_Edge[j].weight;
7.                  v = j;
8.              }
```

6.6 图的应用4：Prim算法---扩点法

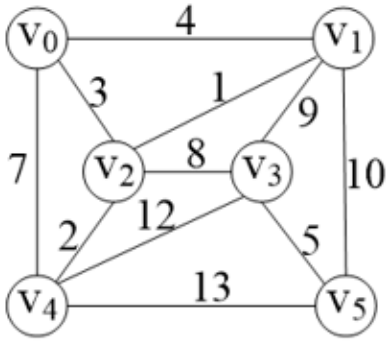
```
1.    //将顶点v加入集合U中
2.        MST[i].from = v;
3.        MST[i].to = MST_Edge[v].to;
4.        MST[i].weight = Min;
5.        MST_Edge[v].weight = 0;
6.        //修改辅助数组中与v关联的边的权值
7.        for(j = 0;j < n; j++)
8.            if (G.weight(v,j) != 0 && G.weight(v,j) < MST_Edge[j].weight){
9.                MST_Edge[j].to = v;
10.               MST_Edge[j].weight = G.weight(v,j);
11.           }
12.    }
13. }
```

6.6 图的应用4：Prim算法---扩点法

- 用Prim算法构造图6-36(a)所示带权连通图的最小生成树的过程中，MST_Edge数组中各分量的变化情况如表6-8所示。

表 6-8 Prim 算法中辅助数组中各分量的值

步骤 \ j / MST_Edge		1	2	3	4	5	U	v
初态	to	0	0	0	0	0	$\{v_0\}$	2
	weight	4	3	∞	7	∞		
1	to	2	0	2	2	0	$\{v_0, v_2\}$	1
	weight	1	0	8	2	∞		
2	to	2	0	2	2	1	$\{v_0, v_2, v_1\}$	4
	weight	0	0	8	2	10		
3	to	2	0	2	2	1	$\{v_0, v_2, v_1, v_4\}$	3
	weight	0	0	8	0	10		
4	to	2	0	2	2	3	$\{v_0, v_2, v_1, v_4, v_3\}$	5
	weight	0	0	0	0	5		
5	to	2	0	2	2	3	$\{v_0, v_2, v_1, v_4, v_3, v_5\}$	
	weight	0	0	0	0	0		



Prim算法的时间复杂度为$O(n^2)$

6.6 图的应用4：Kruskal算法---扩边法

- Kruskal算法是按照边的权值非递减的顺序来构造最小生成树的。设带权连通图为G = (V, E)，令G的最小生成树为T = (V, TE)。

结论：

Kruskal算法又称扩边法。一次性的把所有顶点都加进去，每次加入一条边，如加入的边没有产生环路则留下，如有回路则舍弃。

6.6 图的应用4：Kruskal算法---扩边法

- (1) 初始状态为，即开始时，最小生成树T中只包含了图中所有的顶点，而没有边，此时T为具有n个顶点的无边图，即T中的每个顶点各自构成一个连通分量。
- (2) 将E中的边按权值非递减的顺序排列，并按照这一顺序依次考察边集E中的各条边：若被考察的边关联的两个顶点分别位于T中不同的连通分量中，则将该边加入到T的边集TE中；若被考察的边所关联的两个顶点位于同一个连通分量中则舍弃该边（若将此边加入将使T中产生回路）。
- 依次类推，直至T中含有n-1条边为止，此时T中所有的顶点都在同一连通分量上，T便是G的一棵最小生成树。若算法结束，T少于n-1条边，则无生成树。

6.6 图的应用4：Kruskal算法---扩边法

- 利用Kruskal算法构造一个图的最小生成树的过程如图6-37所示。

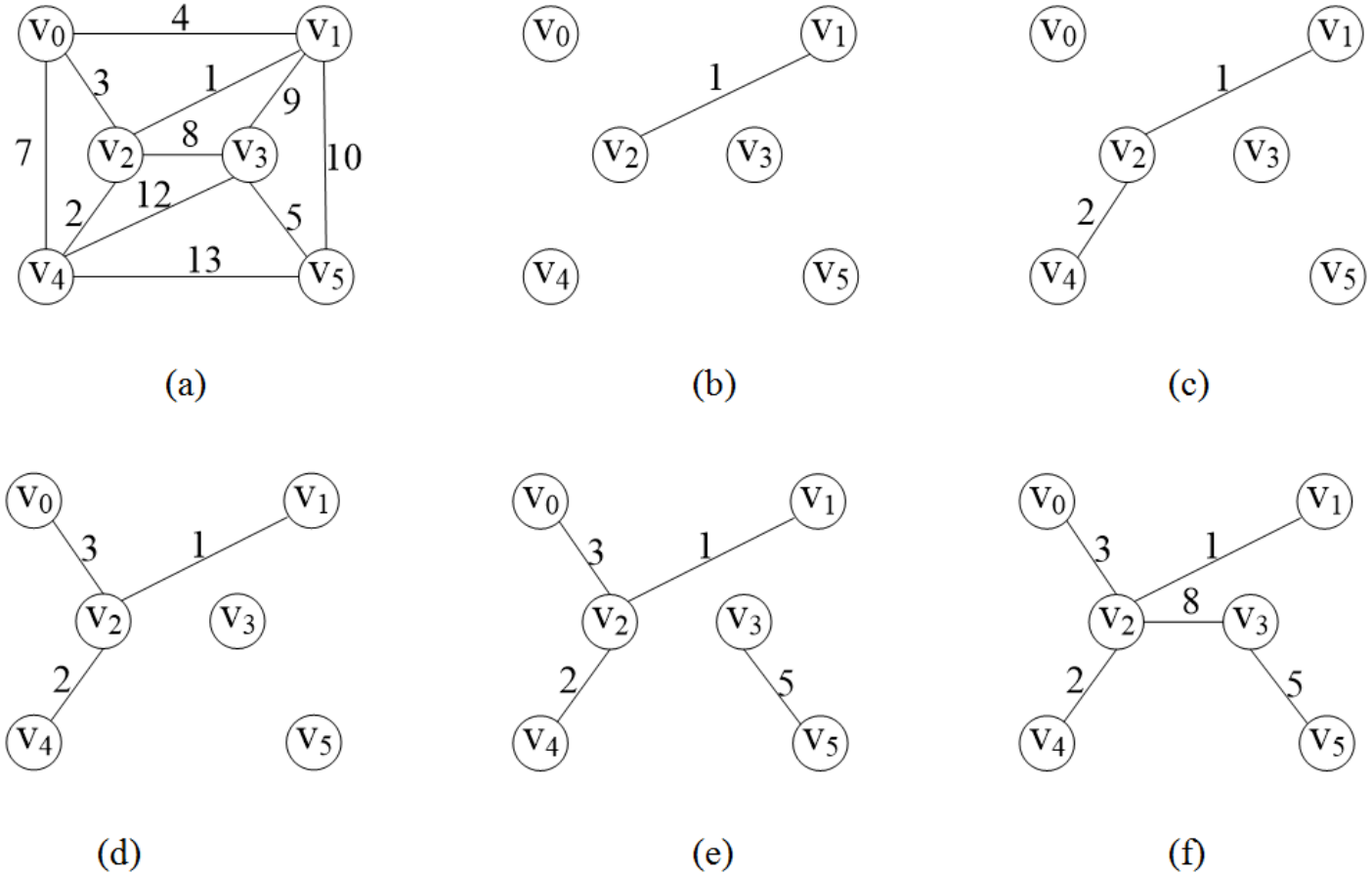


图 6-37 Kruskal 算法构造最小生成树的过程

6.6 图的应用4：Kruskal算法---扩边法

- 根据上述分析可知，实现Kruskal算法主要有两个关键问题：
- (1) 对G中的e条边按照权值进行排序；
- (2) 判断被考察的边的两个顶点是否属于同一个连通分量，即加入该边是否构成回路。

算法6.13： Kruskal算法

```
1.  int cmp(const void *a, const void *b){
2.      Edge *c = (Edge *)a;
3.      Edge *d = (Edge *)b;
4.      if(c->weight != d->weight) return c->weight - d->weight;
5.      else return d->from - c->from;
6.  }
```

6.6 图的应用4：Kruskal算法---扩边法

//用kruskal算法求带权连通图的最小生成树，用MST来存储最小生成树的边

```
1.  void Kruskal(Graph& G, Edge* MST){
2.      int n = G.VerticesNum();
3.      int *Vset = new int[n];
4.      Edge *E = new Edge[G.EdgesNum()];      //记录图的所有边
5.      int i,j;
6.      int EdgeCnt = 0;
7.      for (i = 0; i < n; i++){   //将图中所有的边存储在数组E中
8.          for (j = G.FirstAdj(i); j != -1; j = G.NextAdj(i,j))
9.              if (i < j){
10.                 E[EdgeCnt].from = i;
11.                 E[EdgeCnt].to = j;
12.                 E[EdgeCnt++].weight = G.weight(i,j);
13.             }
14.         Vset[i] = i;
15.     }
```

6.6 图的应用4：Kruskal算法---扩边法

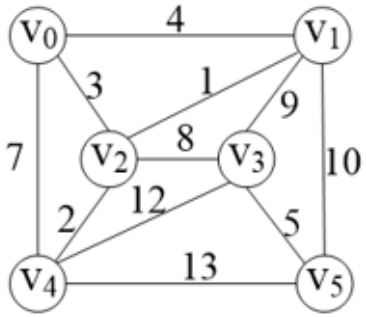
```
1.    qsort(E, EdgeCnt, sizeof(E[0]), cmp);  //使用快速排序按照权值从小到大排序
2.      int cnt = 0;     //生成的边数
3.      i = 0;         //E的下标
4.      int x, y;
5.      while (cnt < n - 1){
6.          x = Vset[E[i].from];
7.          y = Vset[E[i].to];
8.          if (x != y)  //不在同一个连通分量中
9.          {
10.             MST[cnt++] = E[i];
11.             for (j = 0; j < n; j++)
12.                 if (Vset[j] == y)  //合并
13.                     Vset[j] = x;
14.         }
15.         i++;
16.     }
17. }
```

6.6 图的应用4：Kruskal算法---扩边法

- 表6-9示意了在图6-37中，每加入一条边后数组Vset的变化情况。

表 6-9 构造最小生成树的过程中 Vset 的变化过程

Vset	v_0	v_1	v_2	v_3	v_4	v_5
初始值	0	1	2	3	4	5
加入边(v_1, v_2)后	0	1	**1**	3	4	5
加入边(v_2, v_4)后	0	1	1	3	**1**	5
加入边(v_0, v_2)后	0	**0**	**0**	3	**0**	5
加入边(v_3, v_5)后	0	0	0	3	0	**3**
加入边(v_2, v_3)后	0	0	0	**0**	0	**0**



Kruskal算法的时间复杂度主要由排序算法决定，当采用时间复杂度为O(eloge)的快速排序时，Kruskal算法的时间复杂度为O(eloge)，可见，Kruskal算法更适合于边稀疏的连通网络。

END