第8章 图

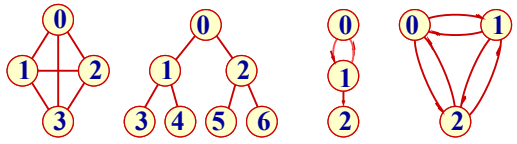
- 图的基本概念
- 图的存储表示
- 图的遍历与连通性
- 最小生成树
- 最短路径
- 活动网络

图的基本概念

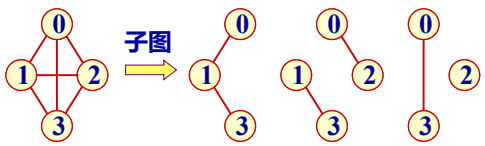
- 图定义
  - 图是由顶点集合(vertex)及顶点间的关系集合组成的一种数据结构：

    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。$Path(x, y)$表示从 x 到 y 的一条单向通路，它是有方向的。

- **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。



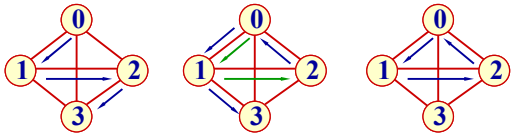
- **邻接顶点** 如果 (u, v) 是 $E(G)$ 中的一条边，则称 u 与 v 互为邻接顶点。
- **子图** 设有两个图 $G = (V, E)$ 和 $G' = (V', E')$。若 $V' \subseteq V$ 且 $E' \subseteq E$，则称 图G' 是 图G 的子图。



- **权** 某些图的边具有与它相关的数，称之为权。这种带权图叫做网络。

- **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 $v_{p1}, v_{p2}, \ldots, v_{pm}$，到达顶点 v_j。则称顶点序列 $(v_i\ v_{p1}\ v_{p2} \ldots v_{pm}\ v_j)$ 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 $v_1,v_2,...,v_m$ 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。



149-7

- **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其极小连通子图，在 n 个顶点的情形下，有 $n-1$ 条边。

149-8

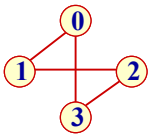
图的存储表示

邻接矩阵 (Adjacency Matrix)

- 在图的邻接矩阵表示中，有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，图的邻接矩阵是一个二维数组 A.edge[n][n]，定义：

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

149-9



$$\text{A.edge}=\begin{bmatrix}0&1&0&1\\1&0&1&0\\0&1&0&1\\1&0&1&0\end{bmatrix}$$

$$\text{A.edge}=\begin{bmatrix}0&1&0\\1&0&1\\0&0&0\end{bmatrix}$$

- 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。

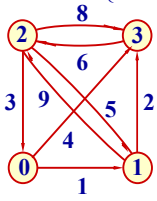
149-10

- 在有向图中，统计第 i 行 1 的个数可得顶点 i 的**出度**，统计第 j 列 1 的个数可得顶点 j 的**入度**。
- 在无向图中，统计第 i 行 (列) 1 的个数可得顶点 i 的**度**。
- 若图中有 n 个顶点，e 条边，邻接矩阵有 n^2 个矩阵元素，对于无向图，其中有 $2e$ 个非零元素，对于有向图，只有 e 个非零元素。
- 如果 e 远远小于 n^2，则称该图为**稀疏图**，其邻接矩阵将是稀疏矩阵；否则该图为**稠密图**，邻接矩阵适用于稠密图。
- 对于一个图来说，**邻接矩阵表示是惟一的**。

149-11

网络的邻接矩阵

$$\text{A.edge}[i][j]=\begin{cases}W(i,j), & \text{若} i\neq j \text{且} <i,j>\in E \text{或} (i,j)\in E\\ \infty, & \text{若} i\neq j \text{且} <i,j>\notin E \text{或} (i,j)\notin E\\ 0, & \text{若} i==j\end{cases}$$



$$\text{A.edge}=\begin{bmatrix}0&1&\infty&4\\\infty&0&9&2\\3&5&0&8\\\infty&\infty&6&0\end{bmatrix}$$

149-12

用邻接矩阵表示的图结构的定义

```
#include <stdio.h>
#include <stdlib.h>
#define maxVertices 30          //图中顶点数最大值
#define maxEdges 900            //最大边数
#define maxWeight 32767         //最大权值
#define impossibleValue '#'
#define impossibleWeight -1

typedef char Type;              //顶点数据的数据类型
typedef int Weight;             //带权图中边权值数据类型
typedef struct {
   int numVertices;             //图中实际顶点数
```



```
   int numEdges;                //图中实际边数
   Type VerticesList [maxVertices];          //顶点表
   Weight Edge[maxVertices][maxVertices];    //邻接矩阵
} MGraph;

int getVertexPos ( MGraph& G, Type x ) {
//从顶点的数据值 x 找出该顶点的顶点号，如果查找
//失败，函数返回-1
   for ( int i = 0; i < G.numVertices; i++ )
      if ( G.VerticesList[i] == x ) return i;     //找到
   return -1;                   //没找到, 没有顶点号
}
```



```
Type getValue ( MGraph& G, int v ) {
                //取顶点v的值
   if ( v != -1 ) return G.VerticesList[v];
   else return impossibleValue;      //在MGraph.h定义
}

Weight getWeight ( MGraph& G, int v, int w ) {
//取边(v, w)上的权值，v、w是顶点号
   if ( v != -1 && w != -1 ) return G.Edge[v][w];
   else return impossibleWeight;    //在MGraph.h定义
}
```



```
int firstNeighbor ( MGraph& G, int v ) {
//函数返回顶点v的第一个邻接顶点的顶点号，如果
//找不到, 则函数返回-1
   if ( v != -1 )
      for ( int j = 0; j < G.numVertices; j++ )
         if ( G.Edge[v][j] > 0 &&
            G.Edge[v][j] < maxWeight ) return j;
   return -1;
}

//在矩阵第 v 行顺序查找，一旦找到就返回。
```

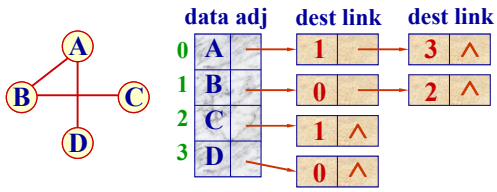


```
int nextNeighbor ( MGraph& G, int v, int w ) {
//函数返回顶点 v 的排在某邻接顶点 w 后面的下一个
//邻接顶点，若没有找到，则函数返回-1
   if ( v != -1 && w != -1 )      //v, w顶点号合法
      for ( int j = w+1; j < G.numVertices; j++ )
         if ( G.Edge[v][j] > 0 &&
            G.Edge[v][j] < maxWeight ) return j;
   return -1;
}

//从矩阵第 v 行的第 w+1 列开始顺序向后查找，找到
//即返回。
```

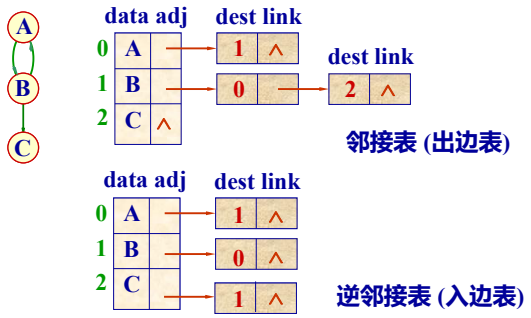

邻接表 (Adjacency List)

- 无向图的邻接表

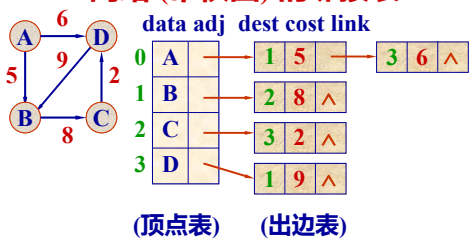


- 同一个顶点发出的边链接在同一个边链表中，每一个链结点代表一条边（边结点），结点中有另一顶点的下标 dest 和指针 link。

有向图的邻接表和逆邻接表



网络 (带权图) 的邻接表



- 对于一个图来说，由于各边链入的顺序不同，邻接表表示是不惟一的。它适用于稀疏图。

- 带权图的边结点中保存该边上的权值 cost。
- 顶点 *i* 的边链表的表头指针 adj 在顶点表的下标为 *i* 的顶点记录中，该记录还保存了该顶点的其它信息。
- 在邻接表的边链表中，各个边结点的链入顺序任意，视边结点输入次序而定。
- 设图中有 *n* 个顶点，*e* 条边，则用邻接表表示无向图时，需要 *n* 个顶点结点，2*e* 个边结点；用邻接表表示有向图时，若不考虑逆邻接表，只需 *n* 个顶点结点，*e* 个边结点。

邻接表表示的图的定义

```
#include<stdio.h>
#include<stdlib.h>
#define maxVertices 30          //图中顶点数最大值
#define maxEdges 450            //图中边数最大值
#define impossibleValue '#'
#define impossibleWeight -1
typedef char Type;              //顶点数据的数据类型
typedef int Weight;             //带权图边权数据类型
typedef struct Enode {          //边结点的定义
   int dest;                    //边的另一顶点的顶点号
   Weight cost;                 //边上的权值
```



```
   struct Enode *link;          //下一条边链指针
} EdgeNode;
typedef struct Vnode {          //顶点的定义
   Type data;                   //顶点数据
   struct Enode *adj;           //出边表的头指针
} VertexNode;
typedef struct {                //图的定义
   VertexNode VerticesList[maxVertices];  //顶点表
   int numVertices;             //实际顶点数
   int numEdges;                //实际边数
} ALGraph;
```



```
int getVertexPos ( ALGraph& G, Type v ) {
//函数返回从顶点数据 v 取得的顶点号，若没有，则
//函数返回-1
   int i = 0;
   while ( i < G.numVertices &&
      G.VerticesList[i].data != v ) i++;     //查顶点表
   if ( i < G.numVertices ) return i;
                                  //找到，返回顶点号
   else return -1;                //未找到，返回-1
}
```

```
int firstNeighbor ( ALGraph& G, int v ) {
//找顶点v的第一个邻接顶点
  if ( v != -1 ) {              //若顶点 v 存在
    EdgeNode *p = G.VerticesList[v].adj;
                                //取 v 边链表第一个边结点
    if ( p != NULL ) return p->dest;
                                //存在, 返回第一个邻接顶点
  }
  return -1;                    //第一个邻接顶点不存在
}
```



```
int nextNeighbor ( ALGraph& G, int v, int w ) {
//函数返回顶点 v 的排在邻接顶点 w 后面的下一个
//邻接顶点, 如果找不到, 则函数返回-1
  if ( v != -1 ) {              //若顶点 v 存在
    EdgeNode *p = G.VerticesList[v].adj;
    while ( p != NULL && p->dest != w ) p = p->link;
    if ( p != NULL && p->link != NULL )
      return p->link->dest;     //返回下一邻接顶点
  }
  return -1;                    //下一邻接顶点不存在
}
```



```
Type getValue ( ALGraph& G, int v ) {  //取顶点v数据
  if ( v != -1 ) return G.VerticesList[v].data;
  else return impossibleValue;
}
Weight getWeight ( ALGraph& G, int v, int w ) {
//取边(v, w)上的权值
  EdgeNode *p = G.VerticesList[v].adj;  //找边(v, w)
  while ( p != NULL && p->dest != w ) p = p->link;
  if ( p != NULL ) return p->cost;   //找到, 返回权值
  else return impossibleWeight;
}
```


邻接多重表 (Adjacency Multilist)

- 在解决以边的处理为主的问题中，无论是邻接矩阵或邻接表，都有重复处理的麻烦。改进的办法就是使用邻接多重表，每一条边仅被表示一次。
- 无向图的情形
  - 边结点的结构

mark	vertex1	vertex2	path1	path2

  - 其中, mark 是处理标记; vertex1和vertex2是该边两顶点位置。Path1 指向下一条依附 vertex1 的边；path2 指向下一条依附 vertex2 的边。

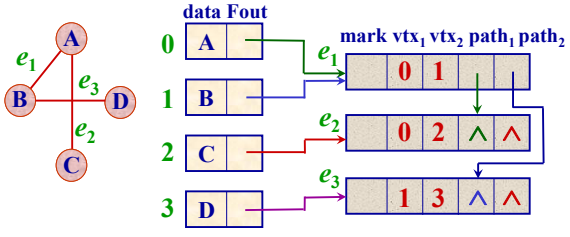
- 对于带权图还需设置一个存放与该边相关的权值的域 cost。
- 顶点结点的结构

data	Firstout

- 存储顶点信息的结点表以顺序表方式组织，每一个顶点结点有两个数据成员：其中，data 存放与该顶点相关的信息，Firstout 是指示第一条依附该顶点的边的指针。
- 在邻接多重表中，所有依附同一个顶点的边都链接在同一个单链表中。

- 从顶点 *i* 出发, 可以循链找到所有依附于该顶点的边，也可以找到它的所有邻接顶点。

邻接多重表结构图示



- 有向图的情形
  - 在用邻接表表示有向图时，有时需要同时使用邻接表和逆邻接表。用有向图的邻接多重表（十字链表）可把两个表结合起来表示。
  - 边结点的结构

mark	vertex1	vertex2	nextout	nextin

  - 其中，mark 是处理标记；vertex1 和 vertex2指明该有向边始顶点和终顶点的位置。nextout指向同一顶点发出的下一条边的边结点；nextin 指向进入同一顶点的下一条边的边结点。
  - 需要时还可有权值域 cost。

149-31

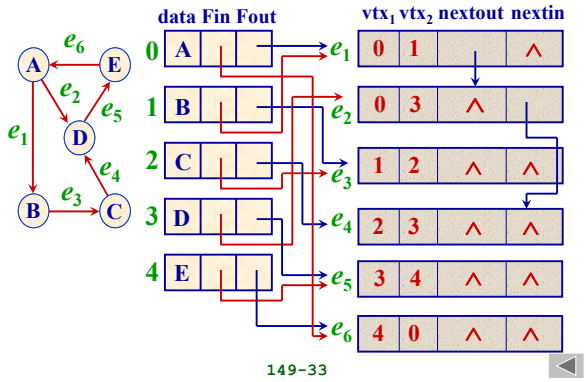
- 顶点结点的结构

data	Firstin	Firstout

- 每个顶点有一个结点，它相当于出边表和入边表的头结点：其中，数据成员 data 存放与该顶点相关的信息，指针 Firstout 指示以该顶点为始顶点的出边表的第一条边，Firstin 指示以该顶点为终顶点的入边表的第一条边。

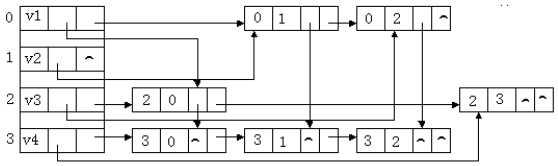
149-32

邻接多重表结构的图示



149-33

画出在下图所表示的有向图中删除顶点V3后的十字链表存储结构图。



146-34

图的遍历与连通性

- 从已给的连通图中某一顶点出发，沿着一 些边访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的遍历 (Graph Traversal)。
- 北京大学版本的教材称遍历为“周游”。
- 图中可能存在回路，且图的任一顶点都可能与其它顶点相通，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。
- 为了避免重复访问，可设置一个标志顶点是否被访问过的辅助数组 visited [ ]。

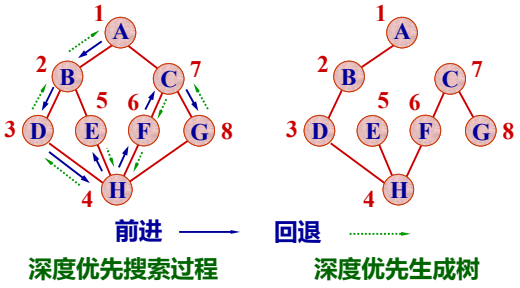
149-35

- 辅助数组 visited [ ] 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, 就让 visited [*i*] 为 1，防止它被多次访问。
- 图的遍历的分类:
  - 深度优先搜索
    DFS (Depth First Search)
  - 广度优先搜索
    BFS (Breadth First Search)

149-36

深度优先搜索DFS (Depth First Search)

- 深度优先搜索的示例



深度优先搜索过程　　深度优先生成树

深度优先搜索的基本思路

- 深度优先搜索算法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。

图的深度优先搜索算法

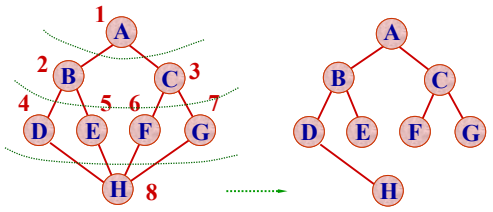
```
void DFS_Traversal ( MGraph& G, int v ) {
//从顶点 v 出发，对图G进行深度优先遍历的主过程
   int i, n = numberOfVertices ( G );     //取图中顶点数
   int visited[maxVertices];              //访问标记数组
   for ( i = 0; i < n; i++ ) visited[i] = 0;
   DFS_recur ( G, v, visited );
   printf ( "\n" );
}
```

- 每调用一次 DFS 就遍历了图的一个连通分量。


```
void DFS_recur ( MGraph& G, int v, int visited[ ] )
//从顶点 v 出发, 以深度优先次序访问所有尚未访问
//过的顶点。算法中用到一个辅助数组 visited, 对已
//访问过的顶点作访问标记
   printf ( "%c ", getValue(G, v) ); //访问顶点 v
   visited[v] = 1;                   //作访问标记
   int w = firstNeighbor ( G, v );   //递归遍历邻接顶点
   while ( w != -1 ) {
      if ( ! visited[w] ) DFS_recur ( G, w, visited );
      w = nextNeighbor ( G, v, w ) );
   }
}
```


广度优先搜索BFS (Breadth First Search)

- 广度优先搜索的示例



广度优先搜索过程　　广度优先生成树

广度优先搜索的基本思路

- BFS在访问了起始顶点 v 之后，由 v 出发，依次访问 v 的各个未被访问过的邻接顶点 $w_1, w_2, \ldots, w_t$，然后再顺序访问 $w_1, w_2, \ldots, w_t$ 的所有还未被访问过的邻接顶点。再从这些访问过的顶点出发，再访问它们的所有还未被访问过的邻接顶点，… 如此做下去，直到图中所有顶点都被访问到为止。
- 深度优先搜索是一种回溯的算法，而广度优先搜索不是，它是一种分层的顺序搜索过程，每向前走一步可能访问一批顶点。因此，广度优先搜索不是一个递归的过程。

- 为了实现逐层访问，广度优先搜索算法使用了一个队列，以记忆正在访问的这一层和下一层的顶点，以便于向下一层访问。
- 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。
- 为了实现和控制广度优先搜索算法的执行，在图的遍历的主程序

  void Graph_Traverse (ALGraph& G)

  中把调用 DFS (G, i, visited) 改为调用BFS (G, i, visited) 即可。

图的广度优先搜索算法

```
void BFS_Traversal ( ALGraph& G ) {
//算法实现图 G 的广度优先搜索。其中使用了一个队
//列Q，队头和队尾指针分别为front和rear
  int i, j, w, n = numberOfVertices(G);
  int visited[maxVertices];        //访问标志数组
  for ( i = 0 ; i < n; i++ ) visited[i] = 0;
  int Q[maxVertices]; int front = rear = 0; //队列置空
  for ( i = 0 ; i < n; i++ )          //顺序扫描所有顶点
    if ( ! visited[i] ) {             //若顶点 i 未访问过
      printf ( "%c ", getValue(G, i) );      //访问
      visited[i] = 1; Q[rear++] = i;         //进队列
```



```
      while ( front < rear ) {        //队列不空时执行
        j = Q[front++];               //队头 j 出队
        w = firstNeighbor ( G, j );
        while ( w != -1 ) {           //若顶点w存在
          if ( ! visited[w] ) {       //且该顶点未访问过
            printf ( "%c ", getValue(G, w) );   //访问
            visited[w] = 1; Q[rear++] = w;      //进栈
          }
          w = nextNeighbor ( G, j, w ) );
        }
      }
    }
}
```


连通分量 (Connected component)

- 当无向图为非连通图时，从图中某一顶点出发，利用深度优先搜索算法或广度优先搜索算法不可能遍历到图中的所有顶点，只能访问到该顶点所在的最大连通子图（连通分量）的所有顶点。
- 若从无向图的每一个连通分量中的一个顶点出发进行遍历，可求得无向图的所有连通分量。
- 求连通分量的算法需要对图的每一个顶点进行检测：若已被访问过，则该顶点一定是落在图中已求得的连通分量上；若还未被访问，则从该顶点出发遍历图，可求得图的另一个连通分量。

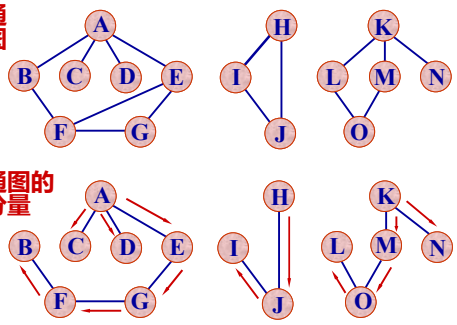

```
#include "DFS_Traversal.cpp"
void calcComponents ( MGraph& G ) {
  int i, k, n = numberOfVertices (G);  //图顶点个数
  int *visited = (int*) malloc ( n*sizeof (int));
  for ( i = 0; i < n; i++ ) visited[i] = 0;
  k = 0;                               //连通分量计数
  for ( i = 0; i < n; i++ )            //顺序扫描所有顶点
    if ( ! visited[i] ) {              //若未访问过, 访问
      printf ( "输出第%d个连通分量的边：\n", ++k );
      DFS_recur ( G, i, visited ); //遍历一个连通分量
      printf ( "\n" );
    }
```



```
  free ( visited );
}
```

- 对于非连通的无向图，遍历每一个连通分量，得到一棵生成树，所有连通分量的生成树组成了非连通图的生成森林。
- 算法中用一个循环，检查各顶点是否访问过。每当遇到一个 visited[i] = 0 的顶点 i，就从该顶点出发做深度优先遍历，此顶点即为生成树的根。
- 做一次遍历可遍访该顶点所在连通分量的所有顶点，然后继续扫描 visited 数组，若遇到 visited[i] = 0 的顶点，即可开始下一个连通分量的遍历。

非连通无向图

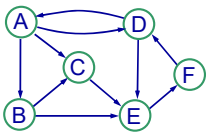


非连通图的连通分量

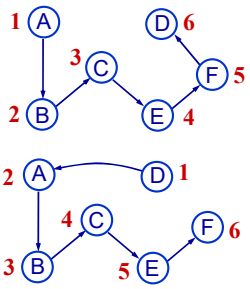
149-49

- 从不同顶点出发，通过强连通（有向）图的遍历，可以得到不同的生成树。

强连通图



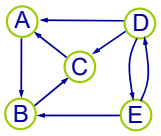
深度优先生成树



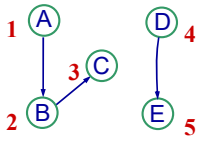
149-50

- 有向图的遍历一般得到的是生成森林。

非强连通图



深度优先生成森林

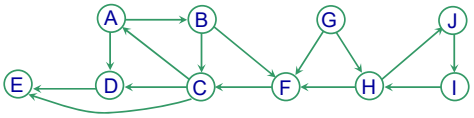


- 问题是通过 DFS 遍历，从某个顶点出发可以遍历强连通图的所有顶点 。但对于非强连通图，从某个顶点出发，只能遍历一个强连通分量。那么，如何通过 DFS 遍历，求出它的所有强连通分量?

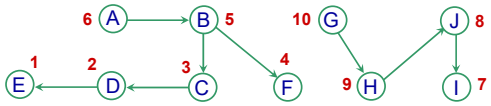
149-51

寻找强连通分量的Kosaraju算法

- 以下图为例，说明算法的思路。

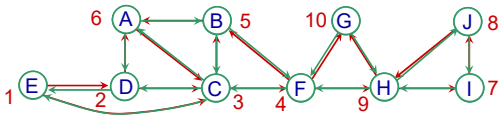


1. 首先对图进行一次DFS，在回退时记忆对结点回溯的顺序：EDCFBAIJHG

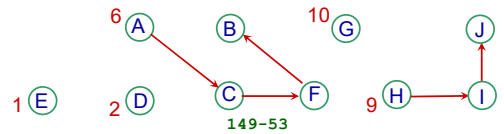


149-52

2. 把图的所有的有向边逆转：

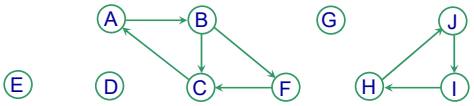


3. 再对得到的图沿回溯顺序 EDCFBAIJHG 从编号最高的 G 开始，再进行一次 DFS，所得到的深度优先森林（树）即为强连通分量的划分。



149-53

- 对应到原图，可得到 5 个强连通分量：



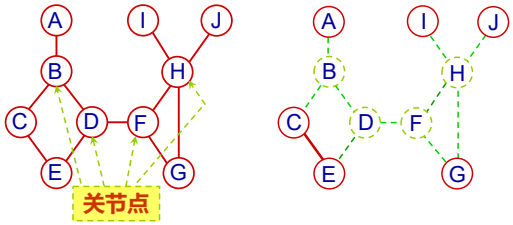
双连通分量

(Biconnected Component)

- 在无向连通图 G 中，当且仅当删去 G 中的顶点 v 及所有依附于 v 的所有边后，可将图分割成两个或两个以上的连通分量，则称顶点 v 为关节点。
- 没有关节点的连通图叫做双连通图（重连通图）。

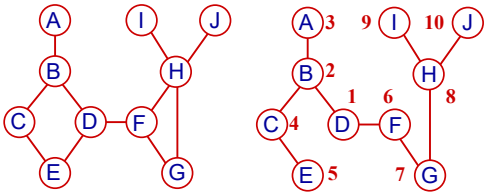
149-54

- 左图是一个连通图，有 4 个关节点，删除其中任一关节点，就会导致图不连通。



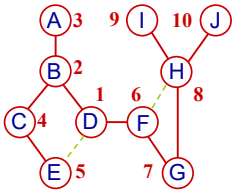
- 如何判断一个连通图中是否有关节点？步骤是：

1. 从图中某一顶点（如 D）出发，做 DFS 遍历。
2. 在生成树的每一结点旁边按深度优先遍历的顺序注明深度优先数。

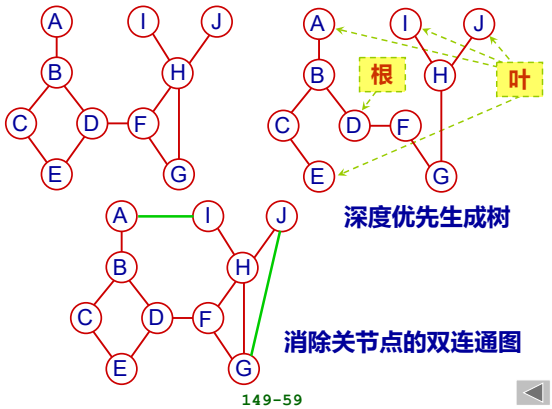


连通图　　深度优先生成树

4. 在深度优先生成树上找关节点的原则是：
   a) 深度优先生成树的根是关节点的充要条件是它至少有两个子女。（根 D 是关节点）
   b) 其它顶点 *u* 是关节点的充要条件是它至少有一个子女 *w*, 从 *w* 出发, 不能通过 *w*、*w* 的子孙及一条回边所组成的路径到达 *u* 的祖先。
   c) 叶结点不是关节点。



5. 如何消除关节点，构造双连通图：
   - 在双连通图上，任何一对顶点之间至少存在有两条路径，在删去某个顶点及与该顶点相关联的边时，也不破坏图的连通性。
   - 因此，在非双连通图上，找到其深度优先生成树中的叶结点（其度为 1），用最少的连线把叶结点与其祖先的某结点（兄弟不可）连接起来，就能消除非双连通图中的关节点，形成双连通图。
- 一个连通图如果不是双连通图，那么它可以包括几个双连通分量。



最小生成树
(minimum cost spanning tree)

- 使用不同的遍历方法，可得到不同的生成树；从不同的顶点出发，也可能得到不同的生成树。
- 按照生成树的定义，*n* 个顶点的连通带权图的生成树有 *n* 个顶点、*n*−1 条边。
- 构造最小生成树的准则
  - 必须使用且仅使用该带权图中的*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；
  - 各边上的权值的总和达到最小。

克鲁斯卡尔 (Kruskal) 算法

- 克鲁斯卡尔算法的基本思想：
  - 设有一个有n个顶点的连通带权图N = { V, E }，先构造一个只有n个顶点，没有边的非连通图 T = { V, ∅ }，图中每个顶点自成一个连通分量。
  - 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。
  - 如此重复下去，直到所有顶点在同一个连通分量上为止。

算法的设计思路

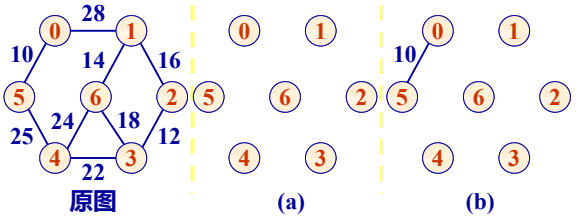
- 首先，将 E 中所有的边按权值存放在小根堆中，每次选择权值最小的边出堆，每个边结点的格式为

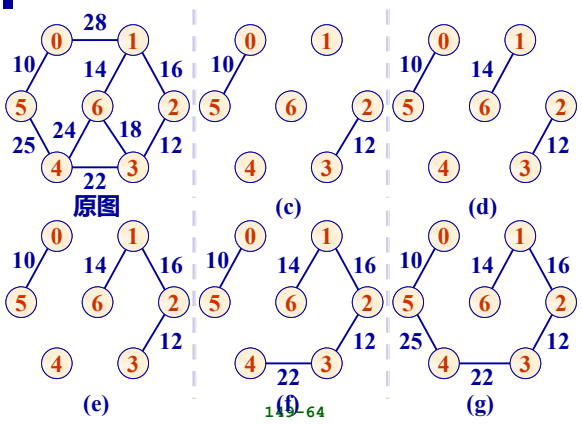
v1	v2	key
边的两个顶点位置		边的权值

- 在构造最小生成树过程中，利用并查集的运算检查依附一条边的两顶点 v1、v2 是否在同一连通分量（即并查集的同一个子集合）上，是则舍去这条边；否则将此边加入 T，同时将这两个顶点放在同一个连通分量上。

- 随着各边逐步加入到最小生成树的边集合中，各连通分量也在逐步合并，直到形成一个连通分量为止。

应用Kruskal算法构造最小生成树的过程





最小生成树的结构定义

```
#define maxSize 20              //边值数组默认大小
#define maxValue 32767          //大数
typedef int WeightType;         //权值的数据类型
typedef struct {                //最小生成树边结点
   int v1, v2;                  //两顶点的顶点号
   WeightType key;              //边上的权值
} MSTEdgeNode;
typedef struct {                //最小生成树的定义
   MSTEdgeNode edgeValue[maxSize];  //边值数组
   int n;                       //数组当前元素个数
} MinSpanTree;
```



```
void initMinSpanTree ( MinSpanTree& T ) {
//最小生成树初始化：构造一棵空树
   T.n = 0;
}

void printMinSpanTree ( MinSpanTree& T ) {
//输出最小生成树
   for ( int i = 0; i < T.n; i++ )
      printf ( "(%d, %d, %d) ", T.edgeValue[i].v1,
         T.edgeValue[i].v2, T.edgeValue[i].key );
   printf ( " \n" );
}
```

利用Kruskal算法建立最小生成树

```
#include "UFSets.h"
#include "MinSpanTree.h"
#include "minHeap.h"
#include "ALGraph.cpp"
void Kruskal_MST ( ALGraph& G, MinSpanTree& T ) {
   int i, j, k, u, v;  Edge w;
   UFSets Uset;  Initial ( Uset );    //建立并查集Uset
   InitMinSpanTree ( T );             //最小生成树置空
   minHeap H;  initMinHeap ( H );     //建立小根堆
   int n = numberOfVertices(G) , e = numberOfEdges(G);
                                      //图的顶点数和边数
```



```
for ( i = 0; i < n; i++ )      //将所有边插入小根堆
   for ( j = firstNeighbor ( G, i ); j != -1;
         j = nextNeighbor ( G, i, j ) ) {
      if ( i < j ) {
         w.v1 = i;  w.v2 = j;
         w.key = getWeight (G, w.v1, w.v2 );
         Insert ( H, w );
      }
   }
j = 0;  k = 0;
while ( k < e ) {              //大循环，选生成树的边
```

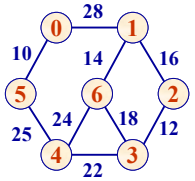


```
   if ( ! heapEmpty(H) ) Remove ( H, w );
   else break;                 //从小根堆退出一条最小边 w
   u = Find ( Uset, w.v1 );  v = Find ( Uset, w.v2 );
   if ( u != v ) {             //若不在同一连通分量
      T.edgeValue[T.n].v1 = w.v1;   //加入生成树
      T.edgeValue[T.n].v2 = w.v2;
      T.edgeValue[T.n++].key = w.key;
      Merge ( Uset, u, v );          //合并连通分量
      j++;                    //加入生成树的边计数
   }
   k++;                       //选过的边计数
}
```



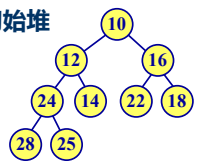
```
   if ( j < n-1 )             //若生成树的边不足
      printf ( "该图不连通，无最小生成树！\n" );
}
```

- 使用小根堆存放图的各条边，可以逐步选出当前权值最小的边，无需对所有的边按权值排序。
- 使用并查集判断两个顶点是否在同一个连通分量中，用以判断候选边是否应加入最小生成树。
- 总的计算时间$O(e\log_2 e+n)$
  - 采用邻接表存储，若图中有 e 条边，检测图的邻接表耗时$O(n+e)$；
  - 建成初始最小堆，需要$O(e\log_2 e)$时间；
  - 构成最小生成树的过程中，需要 $O(e)$ 次出堆操作，耗时也是 $O(e\log_2 e)$。

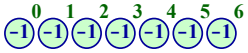


$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

- 初始堆

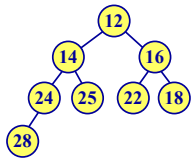


并查集

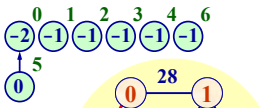


1. 边 (0,5,10) 出堆，两端点0, 5不在同一连通分量

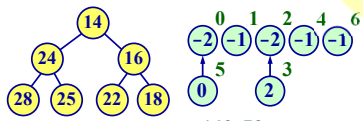
调整后的小根堆

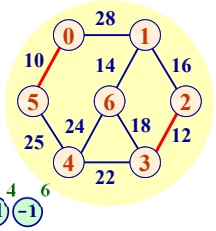


合并后的并查集

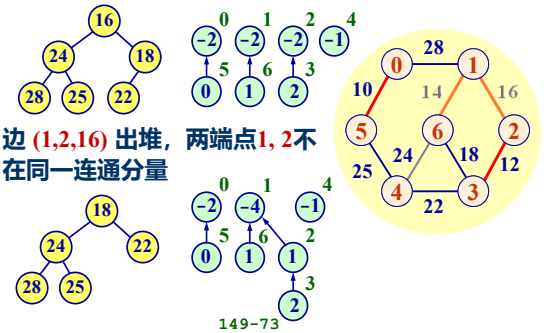


2. 边 (2,3,12) 出堆，两端点2, 3不在同一连通分量





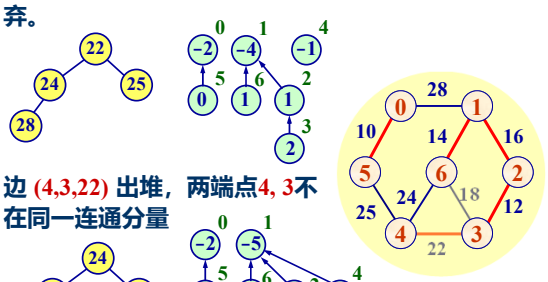
3. 边 (1,6,14) 出堆，两端点1, 6不在同一连通分量
调整后的小根堆　合并后的并查集



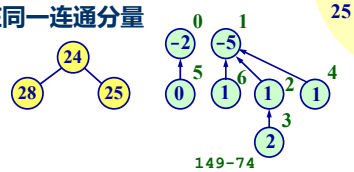
4. 边 (1,2,16) 出堆，两端点1, 2不在同一连通分量

149-73

5. 边 (6,3,18) 出堆，两端点6, 3在同一连通分量，舍弃。

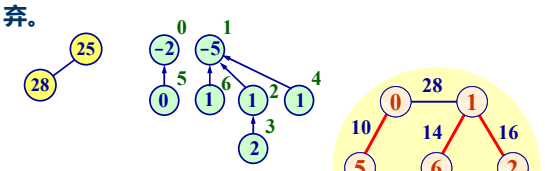


6. 边 (4,3,22) 出堆，两端点4, 3不在同一连通分量

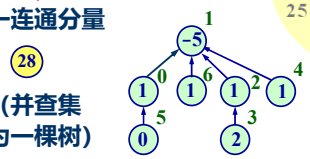


149-74

7. 边 (6,4,24) 出堆，两端点6, 4在同一连通分量，舍弃。



8. 边 (5,4,25) 出堆，两端点5, 4不在同一连通分量



- 完成（并查集合并为一棵树）

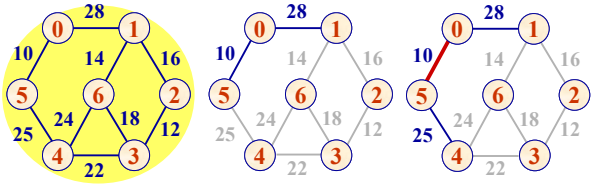
149-75

普里姆（Prim）算法

- 普里姆算法的基本思想：
  - 从连通带权图N = { V, E }中的某一顶点 u_0 出发，选择与它关联的具有最小权值的边 (u_0, v)，将其顶点加入到生成树顶点集合U中。
  - 以后每一步从一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边(u, v)，把它的顶点加入到集合U 中。如此继续下去，直到带权图中的所有顶点都加入到生成树顶点集合 U中为止。
- 采用邻接矩阵作为带权图的存储表示。

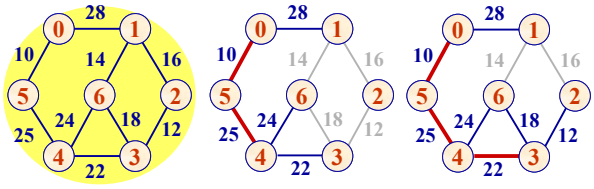
149-76

例子：设从顶点 0 出发构造最小生成树



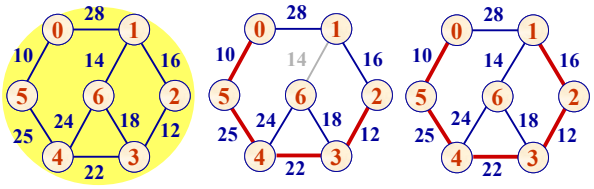
1. 与顶点 0 关联的候选边有两条，权值小的边是 (0, 5), 10，选取它及顶点 5 加入最小生成树。
2. 与顶点 0、5 关联的候选边有两条，权值小的是 (5, 4), 25，选取它及顶点 4 加入最小生成树。

149-77

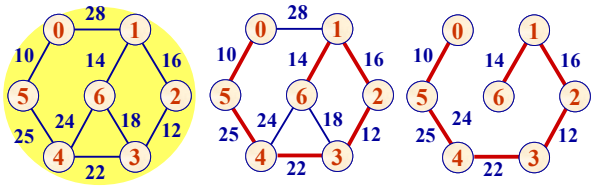


3. 与顶点 0、5、4 关联的候选边有三条，权值小的是 (4, 3), 22，选取它及顶点 3 加入最小生成树。
4. 与顶点 0、5、4、3 关联的候选边有四条，权值小的是 (3, 2), 12，选取它及顶点 2 加入最小生成树。

149-78



5. 与顶点 0、5、4 、3、2 关联的边有四条，权值小的 (2, 1), 16，选取它及顶点 1 加入生成树。
6. 与顶点 0、5、4、3、2、1 关联的候选边有四条，权值小的是 (1, 6), 14，选取它及顶点 6 加入最小生成树。



- 所有顶点 0、5、4 、3、2、1 均已加入最小生成树，算法完成，7 个顶点选取了 6 次。
- 每次为了在候选边中选取权值最小的边，算法使用了一个小根堆，把当前的候选边放在堆内，每次从堆中退出的一定是候选边权值最小的。

用Prim算法求最小生成树

```
#include "MinSpanTree.h"
#include "minHeap.h"
#include "MGraph.cpp"
void Prim ( MGraph& G, int u0, MinSpanTree& T ) {
//从带权连通图G的顶点u0出发，构造最小生成树T
   Edge w;  int i, u, v, count;
   int n = numberOfVertices(G);    //顶点数
   int e = numberOfEdges(G);       //边数
   minHeap H;  initMinHeap(H);     //小根堆
   bool *Vmst = (bool *) malloc ( n*sizeof ( bool ));
                                   //最小生成树顶点集合
```



```
   for ( i = 0; i < n; i++ ) Vmst[i] = false;
   Vmst[u0] = true;  u = u0;       //u0加入生成树
   count = 1;  T.n = -1;
   do {                            //迭代, 逐条边加入生成树
      v = firstNeighbor ( G, u );  //取 u第一个邻接顶点
      while ( v != -1 ) {          //当邻接顶点存在
         if ( !Vmst[v] ) {         //若 v 不在生成树, 加入堆
            w.v1 = u;  w.v2 = v;
            w.key = getWeight ( G, u, v );
            Insert ( H, w );
         }
         v = nextNeighbor(G, u, v) );  //下一个邻接顶点
```



```
      }
      while ( ! heapEmpty(H) && count < n ) {
         Remove ( H, w );  //从堆中退出具最小权重的边
         printf ( "(%d, %d, %d)\n", w.v1, w.v2, w.key );
         if ( ! Vmst[w.v2] ) {       //若不会构成回路, 则
            T.edgeValue[++T.n].v1 = w.v1;   //加入最小
            T.edgeValue[T.n].v2 = w.v2;     //生成树
            T.edgeValue[T.n].key = w.key;
            u = w.v2;  Vmst[u] = true;      //u加入Vmst
            count++;  break;
         }
      }
```



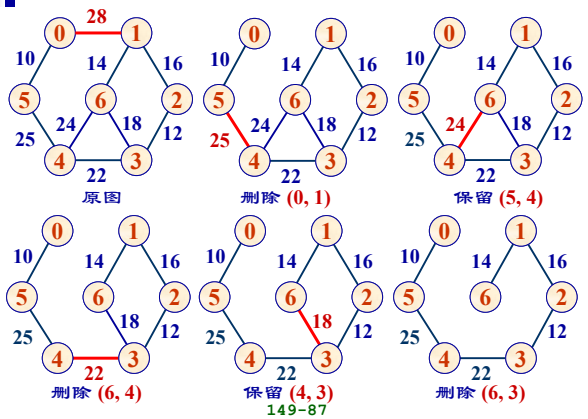
```
   } while ( count < n );
}
```

- Kruskal 算法通过选边来构造最小生成树，Prim 算法通过选顶点来构造最小生成树。
- Prim 算法需 $O(n)$ 次迭代，每次将平均 $2e/n$ 条边插入堆中，总共 e 条边出堆，每次堆的插入和删除都需耗时$O(\log_2 e)$ 。总计算时间$O(e \log_2 e)$ 。

- 当带权图各边的权值不相同时，产生的最小生成树一定是唯一的。
- 当带权图各边的权值有部分相同时，如果基于邻接矩阵，由于选边的顺序惟一，产生的最小生成树也是惟一的；如果基于邻接表，由于边链表可以不惟一，选边的顺序也可能不惟一。

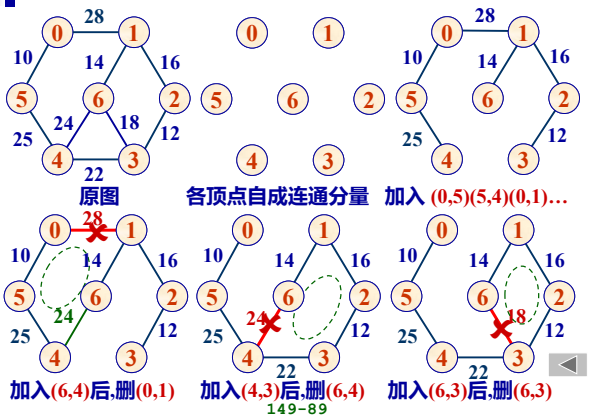
其他：破圈法

- 对一个有 n 个顶点的连通带权图，按其权值从大到小顺序逐个删除各边，直到剩下n-1 条边为止。删除的原则是：要确保删除该边后各个顶点之间还是连通的。
- 为了判断图的连通性，一种方案是每删除一条权值最大的边，就调用 DFS 遍历图，如果能够访遍图中所有顶点则表明删除此边没有破坏图的连通性，此边可删，否则撤销删除，恢复此边。
- 若采用邻接矩阵存储图，DFS的时间代价$O(n^2)$，若采用邻接表，DFS的时间代价O(n+e)。



其他：迪杰斯特拉(Dijkstra)算法

- 最初将带权图中每个顶点视为一个独立的连通分量，然后逐个检查各条边：
  - 如果该边的两个端点在不同的连通分量上，直接把它加入生成树；
  - 如果该边的两个端点在同一个连通分量上，加入它后会形成一个环，把该环上权值最大的边删除。
- 重复以上操作，直到所有边都检查完为止。
- 在此算法中，边的检查顺序没有限制，只要出现圈就破圈。为此又用到了并查集。



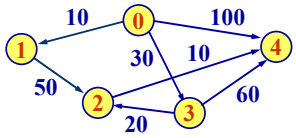
最短路径 (Shortest Path)

- 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
  - 边上权值非负的单源最短路径问题
    - — Dijkstra算法
  - 边上权值任意的单源最短路径问题
    - — Bellman和Ford算法
  - 所有顶点之间的最短路径
    - — Floyd算法

边上权值非负情形的单源最短路径问题

- 问题的提法：给定一个带权有向图 D 与源点 v，求从 v 到 D 中其它顶点的最短路径。限定各边上的权值大于或等于0。
- 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。首先求出长度最短的一条最短路径，再参照它求出长度次短的一条最短路径，依次类推，直到从顶点v到其它各顶点的最短路径全部求出为止。
- 举例说明

Dijkstra算法的逐步求解的过程



源	终	最短路径 1	2	3	4	路径长度 1	2	3	4
v_0	v_1	(v_0,v_1)				10			
	v_2	——	(v_0,v_1,v_2)	(v_0,v_3,v_2)		∞	60	50	
	v_3	(v_0,v_3)				30			
	v_4	(v_0,v_4)		(v_0,v_3,v_4)	(v_0,v_3,v_2,v_4)	100		90	60

- 引入辅助数组dist。它的每一个分量dist[i]表示当前找到的从源点v_0到终点 v_i 的最短路径的长度。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 无边，则dist[i]为∞ 。
- 假设 S 是已求得最短路径的终点的集合，则可证明：下一条最短路径必然是从 v_0 出发，中间只经过 S 中的顶点便可到达的那些顶点 v_x ($v_x \in V-S$) 的路径中的一条。
- 每次求得一条最短路径后，其终点v_k 加入集合S，然后对所有的$v_i \in V-S$，修改其 dist[i]值。

Dijkstra算法可描述如下：

① 初始化： $S \leftarrow \{ v_0 \}$;
dist[j] ← Edge[0][j]， j = 1, 2, …, n-1;
// n为图中顶点个数

② 求出最短路径的长度：
dist[k] ← min { dist[i] }， $i \in V-S$;
$S \leftarrow S \cup \{ k \}$;

③ 修改：
dist[i] ← min{ dist[i], dist[k] + Edge[k][i] },
对于每一个 $i \in V-S$;

④ 判断：若 $S = V$, 则算法结束，否则转 ②。

求解单源最短路径的Dikstra算法

```
#include "MGraph.cpp"
void ShortestPath ( MGraph& G, int v,
      Weight dist[ ], int path[ ] ) {
//求带权有向图G中从源点v到其他顶点的最短路径,
//算法返回两个数组/，dist[n] 存放当前求到的从顶点
// v 到其他顶点的最短路径长度, path[n]存放短路径
   int n = numberOfVertices(G);   //顶点个数
   int S[maxVertices];            //最短路径顶点集
   int i, j, k;  Weight w, min;
   for ( i = 0; i < n; i++ ) {    //初始化
      dist[i] = G.Edge[v][i];  S[i] = 0;
```



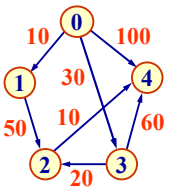
```
      if ( i != v && dist[i] < maxWeight ) path[i] = v;
      else path[i] = -1;
   }
   S[v] = 1;  dist[v] = 0;          //顶点v加入S集合
   for ( i = 0; i < n-1; i++ ) {    //求到其他点最短路径
      min = maxWeight;  int u = v;
      for ( j = 0; j < n; j++ )     //选不在S中具有
         if ( !S[j] && dist[j] < min )  //最短路径顶点u
            { u = j;  min = dist[j]; }
      S[u] = 1;                     //将顶点u加入集合S
      for ( k = 0; k < n; k++ ) {
                  //修改经过u到其他顶点的路径长度
```

```
            w = G.Edge[u][k];
            if ( !S[k] && w < maxWeight &&
                 dist[u]+w < dist[k] ) {
                 //顶点k未加入S, 且绕过u可以缩短路径
               dist[k] = dist[u]+w;
               path[k] = u;       //修改到k的最短路径
            }  //if 的结束
         }  //for_k的结束
      }  //for_i的结束
   }
```

- 算法的时间代价为$O(n^2)$，n 是图顶点数。

Dijkstra算法中各辅助数组的最终结果

	顶点1	顶点2	顶点3	顶点4
Dist	10	50	30	60
Path	0	3	0	2



- 从表中读取源点0到终点v的最短路径的方法：举顶点4为例
  path[4] = 2 ⇨ path[2] = 3 ⇨
  ⇨ path[3] = 0
- 反向读，得到源点0到终点4的最短路径0, 3, 2, 4。

输出最短路径的算法

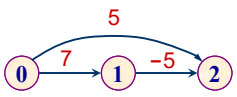
```
void printShortestPath ( MGraph& G, int v,
     Weight dist[ ], int path[ ] ) {
   printf ( "从顶点[%c]到其他各顶点的最短路径为:
     \n", G.VerticesList[v] );     //打印标题
   int i, j, k, n = G.numVertices;
   int d[maxVertices];
   for ( i = 0; i < n; i++ )  //输出到各顶点的最短路径
      if ( i != v ) {
         j = i;  k = 0;
         while ( j != v ) { d[k++] = j;  j = path[j]; }
         d[k++] = v;
```


输出最短路径的算法

```
         printf ( "到顶点[%c]的最短路径为: ",
            G.VerticesList[i] );
         while ( k > 0 )
            printf ( "%c ", G.VerticesList[d[--k]] );
         printf ( "\n最短路径长度为: %d\n", dist[i] );
      }
}
```

- 带权有向图D的某些边或所有边的长度可能为负值。利用Dijkstra算法不一定能得到正确结果。

- 看右图。若设源点$v = 0$，使用Dijkstra算法所得结果如下：

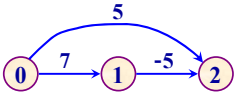


顶点	S[0]	D[0]	P[0]	S[1]	D[1]	P[1]	S[2]	D[2]	P[2]
初始	1	0	-1	0	7	0	0	5	0
2							1	5	0
1				1	7	0			

- 源点0到终点2的最短路径应是0, 1, 2，其长度为2，它小于算法中计算出来的dist[2]的值。

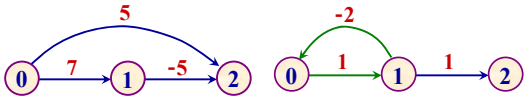
边上权值为任意值的单源最短路径问题

- 带权有向图的某几条边或所有边的长度可能为负值。用Dijkstra算法不一定能得到正确的结果。
- 如右图，若设源点$v = 0$，使用 Dijkstra算法所得结果如下表：



选取顶点	顶点 0			顶点 1			顶点 2		
	S[0]	d[0]	p[0]	S[1]	d[1]	p[1]	S[2]	d[2]	p[2]
0	1	0	-1	0	7	0	0	5	0
2	1	0	-1	0	7	0	1	5	0
1	1	0	-1	1	7	0	1	5	0

- 源点 0 到终点 2 的最短路径应是0, 1, 2，其长度为2，它小于算法中计算出来的dist[2]的值。
- Bellman和Ford提出了从源点逐次绕过其他顶点，以缩短到达终点的最短路径长度的方法。该方法有一个限制条件，即要求图中不能包含有由带负权值的边组成的回路。



149-103

- 当图中没有由带负权值的边组成的回路时，有 n 个顶点的图中任意两个顶点之间如果存在最短路径，此路径最多有 $n-1$ 条边。
- 我们将以此为依据考虑计算从源点 v 到其他顶点 u 的最短路径长度dist[u]。
- Bellman-Ford 方法构造一个最短路径长度数组序列 $dist^1[u]$, $dist^2[u]$, …, $dist^{n-1}[u]$。其中，
  - $dist^1[u]$是从源点 v 到终点 u 的只经过一条边的最短路径的长度，

    $dist^1[u] = Edge[v][u]$

149-104

  - $dist^2[u]$是从源点 v 出发最多经过两条边到达终点 u 的最短路径长度;
  - $dist^3[u]$是从源点 v 出发最多经过不构成带负长度边回路的三条边到达终点 u 的最短路径长度，…，
  - $dist^{n-1}[u]$是从源点 v 出发最多经过不构成带负长度边回路的 $n-1$ 条边到达终点 u 的最短路径长度。
- 算法的最终目的是计算出 $dist^{n-1}[u]$。
- 可以用递推方式计算 $dist^k[u]$。

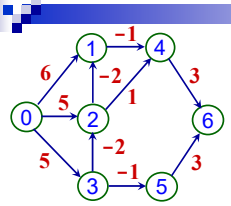
149-105

$dist^1[u] = Edge[v][u]$;

$dist^k[u] = \min\{ dist^{k-1}[u], \min\{ dist^{k-1}[j] + Edge[j][u] \} \}$

- 设已经求出 $dist^{k-1}[j]$, $j = 0, 1, \ldots, n-1$, 此即从源点 v 出发最多经过不构成带负长度边回路的 $k-1$ 条边到达终点 j 的最短路径长度。
- 计算 $\min\{ dist^{k-1}[j] + Edge[j][u] \}$，可得从源点 v 绕过各顶点 j，最多经过不构成带负长度边回路的 k 条边到达终点 u 的最短路径长度。用它与 $dist^{k-1}[u]$比较，取小者作为 $dist^k[u]$ 的值。

149-106



- 下表横排是源点和目标顶点编号；纵列是计算$dist^k[i]$值（简记d^k）。
- 采用邻接矩阵，时间复杂度为$O(n^3)$，n是图顶点个数。

k	[0]	[1]	[2]	[3]	[4]	[5]	[6]
d^1	0	6	5	5	∞	∞	∞
d^2	0	5-2=3	5-2=3	5	6-1=5	5-1=4	∞
d^3	0	5-2-2=1	3	5	5-2-1=2	4	5-1+3=7
d^4	0	1	3	5	5-2-2-1=0	4	5-2-1+3=5
d^5	0	1	3	5	0	4	5-2-2-1+3=3
d^6	0	1	3	5	0	4	3

计算最短路径的Bellman和Ford算法

```
void Bellman_Ford ( MGraph& G, int v, Weight dist[],
  int path[] ) {
//在有向带权图中有的边具有负的权值。从顶点 v 找
//到所有其它顶点的最短路径。
  int i, k, u;  Weight w;
  for ( i = 0; i < G.numVertices; i++ ) {
    dist[i] = G.Edge[v][i];                  //初始化
    if ( i != v && dist[i] < maxWeight ) path[i] = v;
    else path[i] = -1;
  }
```

149-108

```
    for ( k = 2; k < G.numVertices; k++ )
                              //计算dist²[i]到distⁿ⁻¹[i]
      for ( u = 0; u < G.numVertices; u++ )
        if ( u != v )
          for ( i = 0; i < G.numVertices; i++ ) {
            w = G.Edge[i][u];
            if ( w > 0 && w < maxWeight &&
                 dist[u] > dist[i]+w )
              { dist[u] = dist[i]+w;  path[u] = i; }
          }
}
```


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。
- Floyd算法的基本思想：

定义一个 n 阶方阵序列：

$A^{(0)}$, $A^{(1)}$, …, $A^{(n)}$.

其中 $A^{(0)}[i][j]$ = G.Edge[i][j];

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

- $A^{(1)}[i][j]$ 是从顶点 v_i 到 v_j，中间可能绕过顶点 v_1 的最短路径长度;
- $A^{(k)}[i][j]$ 是从顶点 v_i 到 v_j，中间可能绕过顶点 v_1, v_2, …, v_k 的最短路径的长度;
- $A^{(n)}[i][j]$是从顶点v_i到v_j的最短路径长度。

求各对顶点间最短路径的算法

```
void Floyd ( MGraph& G, Weight a[][maxVertices],
     int path[ ][maxVertices] ) {
//a[i][j] 是顶点 i 和 j 间的最短路径长度。path[i][j] 是
//相应路径上顶点 j 的前一顶点的顶点号。算法计算
//每一对顶点间最短路径及最短路径长度。
```



```
int i, j, k, n = G.numVertices;
for ( i = 0; i < n; i++ )          //矩阵a与path初始化
  for ( j = 0; j < n; j++ ) {
    a[i][j] = G.Edge[i][j];
    if ( i == j ) path[i][j] = 0;
    else if ( a[i][j] < maxWeight ) path[i][j] = i;
    else path[i][j] = -1;
  }
for ( k = 0; k < n; k++ ) {        //对每一个 k 计算
  for ( i = 0; i < n; i++ )
    for ( j = 0; j < n; j++ )
      if ( a[i][k] + a[k][j] < a[i][j] ) {  //绕过 k 到 j
```



```
          a[i][j] = a[i][k] + a[k][j];
          path[i][j] = path[k][j];      //缩短路径长度
        }
  }
}
```

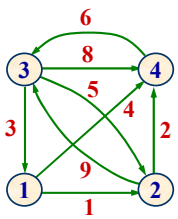
- Floyd算法允许图中有带负权值的边，但不许有包含带负权值的边组成的回路。
- Floyd算法的时间代价为$O(n^3)$，因为有一个三重循环。

$A^{(0)}$

	1	2	3	4	1	2	3	4
1	0	1	∞	4				
2	∞	0	9	2				
3	3	5	0	8				
4	∞	∞	6	0				

$Path^{(0)}$

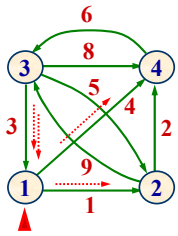
	1	2	3	4	1	2	3	4
1	0	1	0	1				
2	0	0	2	2				
3	3	3	0	3				
4	0	0	4	0				



陆续绕过顶点①、②、③、④，改善各顶点间路径长度

	$A^{(0)}$				$A^{(1)}$			
	1	2	3	4	1	2	3	4
1	0	1	∞	4	0	1	∞	4
2	∞	0	9	2	∞	0	9	2
3	3	5	0	8	3	4	0	7
4	∞	∞	6	0	∞	∞	6	0

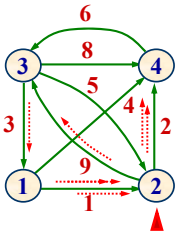
	$Path^{(0)}$				$Path^{(1)}$			
	1	2	3	4	1	2	3	4
1	0	1	0	1	0	1	0	1
2	0	0	2	2	0	0	2	2
3	3	3	0	3	3	1	0	1
4	0	0	4	0	0	0	4	0



绕过顶点①，改善各顶点间的路径长度

	$A^{(1)}$				$A^{(2)}$			
	1	2	3	4	1	2	3	4
1	0	1	∞	4	0	1	10	3
2	∞	0	9	2	∞	0	9	2
3	3	4	0	7	3	4	0	6
4	∞	∞	6	0	∞	∞	6	0

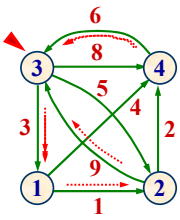
	$Path^{(1)}$				$Path^{(2)}$			
	1	2	3	4	1	2	3	4
1	0	1	0	1	0	1	2	2
2	0	0	2	2	0	0	2	2
3	3	1	0	1	3	1	0	2
4	0	0	4	0	0	0	4	0



绕过顶点②，改善各顶点间的路径长度

	$A^{(2)}$				$A^{(3)}$			
	1	2	3	4	1	2	3	4
1	0	1	10	3	0	1	10	3
2	∞	0	9	2	12	0	9	2
3	3	4	0	6	3	4	0	6
4	∞	∞	6	0	9	10	6	0

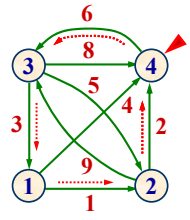
	$Path^{(2)}$				$Path^{(3)}$			
	1	2	3	4	1	2	3	4
1	0	1	2	2	0	1	2	2
2	0	0	2	2	3	0	2	2
3	3	1	0	2	3	1	0	2
4	0	0	4	0	3	1	4	0



绕过顶点③，改善各顶点间的路径长度

	$A^{(3)}$				$A^{(4)}$			
	1	2	3	4	1	2	3	4
1	0	1	10	3	0	1	9	3
2	12	0	9	2	11	0	8	2
3	3	4	0	6	3	4	0	6
4	9	10	6	0	9	10	6	0

	$Path^{(3)}$				$Path^{(4)}$			
	1	2	3	4	1	2	3	4
1	0	1	2	2	0	1	4	2
2	3	0	2	2	3	0	4	2
3	3	1	0	2	3	1	0	2
4	3	1	4	0	3	1	4	0



绕过顶点④，改善各顶点间的路径长度

- 以 $Path^{(4)}$为例，对最短路径读法加以说明。从$A^{(4)}$知，顶点2到1的最短路径长度为$a[2][1] = 11$，其最短路径看path[2][1]=3，path[2][3]=4，path[2][4]=2，表示顶点1 ← 顶点3 ← 顶点4 ← 顶点2；从顶点2到顶点1最短路径为

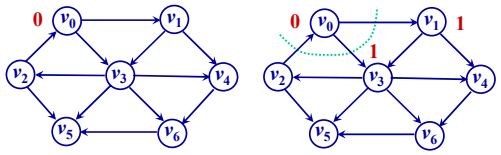
  <2, 4>, <4, 3>, <3, 1>。

- 本章给出的求解最短路径的算法不仅适用于带权有向图，对带权无向图也可以适用。因为带权无向图可以看作是有往返二重边的有向图。

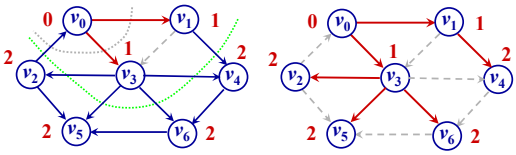
149-119

非带权图的最短路径问题

- 对于一个不考虑边上权重的有向图，若想以某个顶点 v 作源点，求出它到其他各个顶点的最短路径，可以使用广度优先搜索BFS算法来实现。
- 源点到源点的路径长度为0，各顶点旁的数字标识源顶点到该顶点的路径长度。



149-120



- 算法中用到一个队列存放已求得最短路径的顶点，以实现广度优先搜索。

```
#include "MGraph.cpp"
#define queSize 30                //队列长度
void unweighted ( MGraph& G, int v,
   int dist[ ], int path[ ] ) {
```



```
int Q[queSize]; int front = 0, rear = 0;   //队列Q置空
int i, w, n = numberOfVertices(G);         //顶点数
for ( i = 0; i < n; i++ ) dist[i] = maxVertices;
dist[v] = 0; path[v] = -1;        //源点到源点距离为0
Q[rear] = v; rear = ( rear+1 ) % queSize; //源点进队
while ( rear != front ) {         //队列不空循环
   v = Q[front]; front = ( front+1 ) % queSize;
   for ( w = firstNeighbor(G, v); w != -1;
         w = nextNeighbor (G, v, w) )
      if ( dist[w] == maxVertices ) { //w未求最短路径
         dist[w] = dist[v]+1; path[w] = v;
                  //记下路径上前一顶点及长度
```



```
            Q[rear] = w; rear = ( rear+1 ) % queSize;
         }
   }
}
```

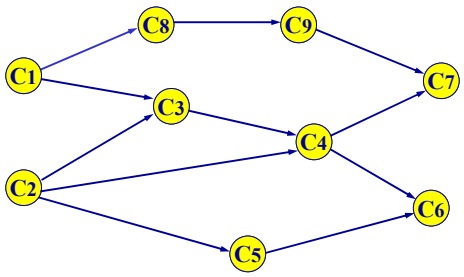
- 如果某些顶点从源点出发是不可达的，则队列可能会过早变空，在这种情况下，相应的 dist 将保持maxValue（相当于∞），这是合理的。
- 使用邻接表存储图，算法的时间代价为$O(n+e)$；使用邻接矩阵存储图，算法的时间代价为$O(n^2)$。
- 算法的附加空间是使用了一个队列，空间复杂度为$O(n)$。

活动网络 (Activity Network)

拓扑排序

- 计划、施工过程、生产流程、程序流程等都是“工程”。除了很小的工程外，一般都把工程分为若干个叫做“活动”的子工程。完成了这些活动，这个工程就可以完成了。
- 例如，计算机专业学生的学习就是一个工程，每一门课程的学习就是整个工程的一些活动。其中有些课程要求先修课程，有些则不要求。这样在有的课程之间有领先关系，有的课程可以并行地学习。

课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1, C_2
C_4	数据结构	C_3, C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5, C_4
C_7	操作系统	C_4, C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8



学生课程学习工程图

- 可以用有向图表示一个工程。在这种有向图中，用顶点表示活动，用有向边<V_i, V_j>表示活动V_i必须先于活动V_j进行。这种有向图叫做顶点表示活动的AOV网络 (Activity On Vertices)。
- 在AOV网络中不能出现有向回路，即有向环。如果出现了有向环，则意味着某项活动应以自己作为先决条件。
- 因此，对给定的AOV网络，必须先判断它是否存在有向环。

149-127

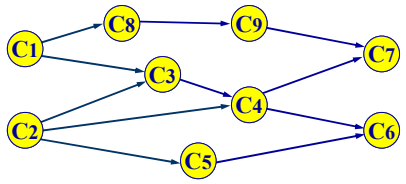
- 检测有向环的一种方法是对AOV网络构造它的拓扑有序序列。即将各个顶点 (代表各个活动)排列成一个线性有序的序列，使得AOV网络中所有应存在的前驱和后继关系都能得到满足。
- 这种构造AOV网络全部顶点的拓扑有序序列的运算就叫做拓扑排序。
- 如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中，则该网络中必定不会出现有向环。
- 如果AOV网络中存在有向环，此AOV网络所代表的工程是不可行的。

149-128

- 例如，对学生选课工程图进行拓扑排序，得到的拓扑有序序列为

$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$

或 $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$



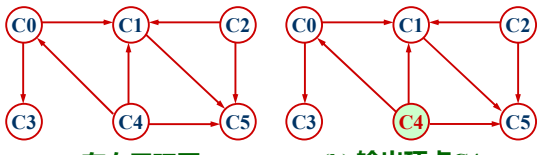
149-129

进行拓扑排序的方法

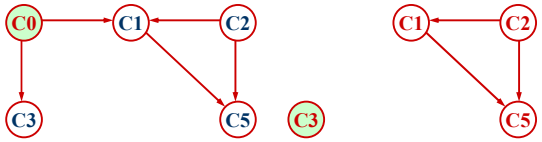
① 输入AOV网络。令 n 为顶点个数。
② 在AOV网络中选一个没有直接前驱的顶点，并输出之;
③ 从图中删去该顶点，同时删去所有它发出的有向边;
④ 重复以上 ②、③步, 直到
- 全部顶点均已输出，拓扑有序序列形成，拓扑排序完成；或
- 图中还有未输出的顶点, 但已跳出处理循环。说明图中还剩下一些顶点, 它们都有直接前驱。这时网络中必存在有向环。

149-130

拓扑排序的过程

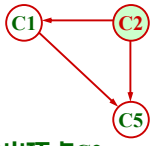


(a) 有向无环图　(b) 输出顶点C4

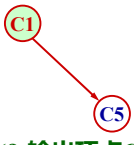


(c) 输出顶点C0　(d) 输出顶点C3

149-131



(e) 输出顶点C2

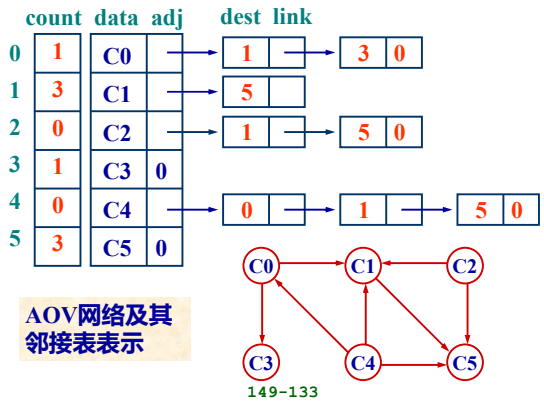


(f) 输出顶点C1

(g) 输出顶点C5　(h) 拓扑排序完成

- 最后得到的拓扑有序序列为 C4, C0, C3, C2, C1, C5。它满足图中给出的所有前驱和后继关系，对于本来没有这种关系的顶点，如C4和C2，也排出了先后次序关系。

149-132



AOV网络及其邻接表表示

- 在邻接表中增设一个数组count[]，记录各顶点入度，入度为零的顶点即无前驱顶点。
- 入度数组count[]初始化后，输入数据。每输入一条边<j, k>, 就使用 count[k]++ 统计入度信息。
- 在算法中, 使用一个存放入度为零的顶点的栈，供选择和输出无前驱的顶点。
- 拓扑排序算法可描述如下：
  a) 建立入度为零的顶点栈;
  b) 当入度为零的顶点栈不空时, 重复执行
    - 从顶点栈中退出一个顶点, 并输出之;
    - 从AOV网络中删去这个顶点和它发出的边, 边的终顶点入度减一;

- 如果边的终顶点入度减至0, 则该顶点进入度为零的顶点栈;

c) 如果输出顶点个数少于AOV网络的顶点个数, 则报告网络中存在有向环。

拓扑排序的算法

```
#include "ALGraph.cpp"
#define maxsize 20          //栈的容量
bool TopologicalSort ( ALGraph& G,
      int topoArray[ ], int& k ) {
//函数通过 topoArray 返回拓扑排序的结果，k 返回
//排入顶点个数
```



```
int i, j, w;  EdgeNode *p;
int  S[maxsize];  int top = -1;    //入度为零顶点的栈
int n = G.numVertices;             //有向图中顶点个数
int *ind = (int*) malloc ( n*sizeof ( int )); //入度数组
for ( i = 0; i < n; i++ ) ind[i] = 0;
for ( i = 0; i < n; i++ ) {        //计算各顶点的入度
   p = G.VerticesList[i].adj;
   while ( p != NULL )
      { ind[p->dest]++;  p = p->link; }
}
for ( i = 0; i < n; i++ )          //检查图的所有顶点
   if ( !ind[i] ) S[++top] = i;    //入度为零顶点进栈
```



```
k = 0;                             //排序元素计数
for ( i = 0; i < n; i++ )          //期望输出n个顶点
   if ( top > -1 ) {               //栈不空，拓扑排序
      j = S[top--];                //退栈顶点为j
      topoArray[k++] = j;          //保存拓扑有序序列
      p = G.VerticesList[j].adj;
      while ( p != NULL ) {        //查 j 所有邻接顶点
         w = p->dest;              //邻接顶点入度减一
         if ( --ind[w] == 0 ) S[++top] = w;
         p = p->link;              //减至零的顶点进栈
      }
   }
```



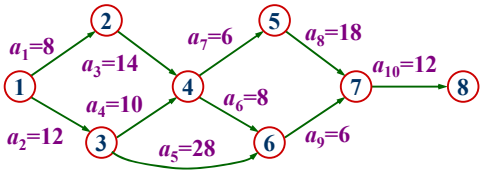
```
   else {
      printf ( "图中有有向环！\n" );
      return false;
   }
return true;                       //正常循环出口，排序成功
}
```

关键路径

- 如果在无有向环的带权有向图中，用有向边表示一个工程中的活动 (Activity)，用边上权值表示活动持续时间 (Duration)，用顶点表示事件 (Event)，则这样的有向图叫做用边表示活动的网络，简称 AOE (Activity On Edges) 网络。
- AOE网络在某些工程估算方面非常有用。例如，可以使人们了解：
  - 完成整个工程至少需要多少时间（假设网络中没有环）？
  - 为缩短完成工程所需的时间，应当加快哪些活动？

- 从开始点到各个顶点，以至从开始点到结束点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但只有各条路径上所有活动都完成了，整个工程才算完成。
- 因此，完成整个工程所需的时间取决于从源点到汇点的最长路径长度，即在这条路径上所有活动的持续时间之和。这条路径长度最长的路径就叫做关键路径(Critical Path)。
- 要找出关键路径，必须找出关键活动，即不按期完成就会影响整个工程完成的活动。

- 关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到关键路径。例如，下图就是一个AOE网。



- 顶点 1 是整个工程的开始点，顶点 8 是整个工程的结束点。a_i = …是各边上的活动及持续时间。

定义几个与计算关键活动有关的量

1. 事件V_i的最早可能开始时间$Ve(i)$
   它是从开始点V_1到顶点V_i的最长路径长度。
2. 事件V_i的最迟允许开始时间$Vl[i]$
   它是在保证结束点V_n在$Ve[n]$时刻完成的前提下，事件V_i的允许的最迟开始时间。
3. 活动a_k的最早可能开始时间 $e[k]$
   设活动a_k在边$<V_i, V_j>$上，则$e[k]$是从开始点V_1到顶点V_i的最长路径长度。因此
   $e[k] = Ve[i]$。

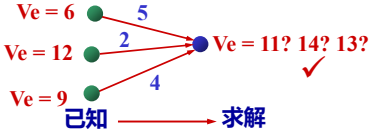
4. 活动a_k的最迟允许开始时间 $l[k]$
   它是在不会引起时间延误的前提下，该活动允许的最迟开始时间。
   $l[k] = Vl[j] - dur(<i, j>)$。
   其中，$dur(<i, j>)$是完成 a_k 所需的时间。
5. 时间余量 $l[k] - e[k]$
   表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。$l[k] == e[k]$ 表示活动 a_k 是没有时间余量的关键活动。

- 为找出关键活动，需要求各个活动的 $e[k]$ 与 $l[k]$，以判别是否 $l[k] == e[k]$。

- 为求得$e[k]$与$l[k]$，需要先求得从开始点V_1到各个顶点V_i的$Ve[i]$和$Vl[i]$。
- 求$Ve[i]$的递推公式
  a) 从$Ve[1] = 0$开始，向前递推

$$Ve[i] = \max_j \{Ve[j] + dur(<V_j, V_i>)\},$$

  $<V_j, V_i> \in S_2$，$i = 2, 3, \ldots, n$
  S_2是所有指向V_i的有向边$<V_j, V_i>$的集合。

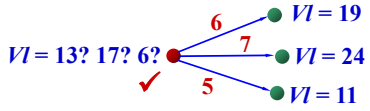


b) 从$Vl[n] = Ve[n]$开始，反向递推

$$Vl[i] = \min_j \{Vl[j] - dur(<V_i, V_j>)\},$$

$$<V_i, V_j> \in S_1, i = n-1, n-2, \ldots, 1$$

S_1是所有源自V_i的有向边$< V_i , V_j >$的集合。



求解 ← 已知

- 这两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。

149-145

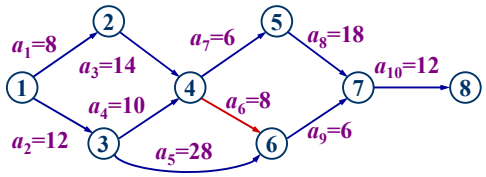
- 设活动a_k ($k = 1, 2, \ldots, e$) 在带权有向边$<V_i, V_j>$上，其持续时间用dur ($<V_i , V_j >$) 表示，则有

  $e[k] = Ve[i]$;

  $l[k] = Vl[j] - dur(<V_i , V_j >)$; $k = 1, 2, \ldots, e$。

  这样就得到计算关键路径的算法。
- 为了简化算法，假定在求关键路径之前已经对各顶点实现了拓扑排序，并按拓扑有序的顺序对各顶点重新进行了编号。

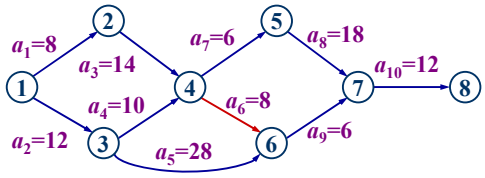
149-146



- $Ve[1] = 0$, $Ve[2] = Ve[1]+a_1 = 0+8 = 8$,
- $Ve[3] = Ve[1]+a_2 = 0+12 = 12$,
- $Ve[4] = \max\{Ve[2]+a_3, Ve[3]+a_4\} = \max\{8+14, 12+10\} = 22$

	1	2	3	4	5	6	7	8
Ve	0	8	12	22				
Vl								

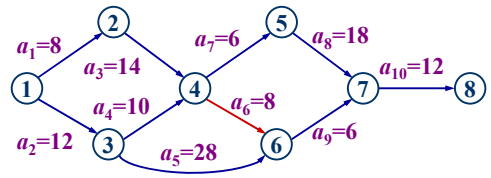
149-147



- $Ve[5] = Ve[4]+a_7 = 22+6 = 28$,
- $Ve[6] = \max\{Ve[4]+a_6, Ve[3]+a_5\} = \max\{22+8, 12+28\} = 40$,

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40		
Vl								

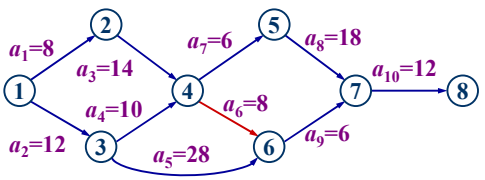
149-148



- $Ve[7] = \max\{Ve[5]+a_8, Ve[6]+a_9\} = \max\{28+18, 40+6\} = 46$,
- $Ve[8] = Ve[7]+a_{10} = 46+12 = 58$,

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl								

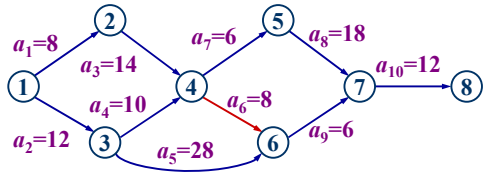
149-149



- $Vl[8] = Ve[8] = 58$, $Vl[7] = Vl[8]-a_{10} = 58-12 = 46$,
- $Vl[6] = Vl[7]-a_9 = 46-6 = 40$,
- $Vl[5] = Vl[7]-a_8 = 46-18 = 28$,

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl					28	40	46	58

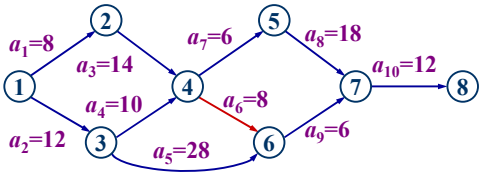
149-150



- $Vl[4] = \min\{Vl[5]-a_7, Vl[6]-a_6\} = \min\{28-6, 40-8\} = 22$,
- $Vl[3] = \min\{Vl[4]-a_4, Vl[6]-a_5\} = \min\{22-10, 40-28\} = 12$,

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl			12	22	28	40	46	58

149-151



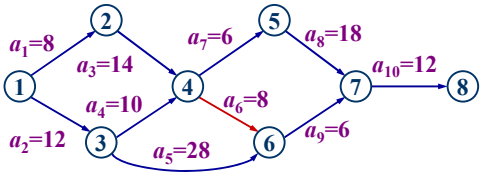
- $Vl[2] = Vl[4]-a_3 = 22-14 = 8$,
- $Vl[1] = \min\{Vl[2]-a_1, Vl[3]-a_2\} = \min\{8-8, 12-12\} = 0$.

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl	0	8	12	22	28	40	46	58

149-152

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl	0	8	12	22	28	40	46	58

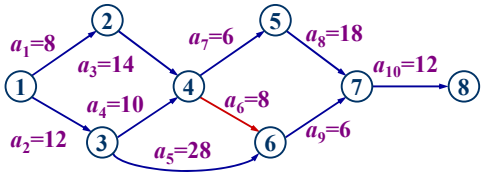
- $e[1] = Ve[1] = 0$, $l[1] = Vl[2]-a_1 = 8-8 = 0$,
- $e[2] = Ve[1] = 0$, $l[2] = Vl[3]-a_2 = 12-12 = 0$,
- $e[3] = Ve[2] = 8$, $l[3] = Vl[4]-a_3 = 22-14 = 8$,



149-153

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl	0	8	12	22	28	40	46	58

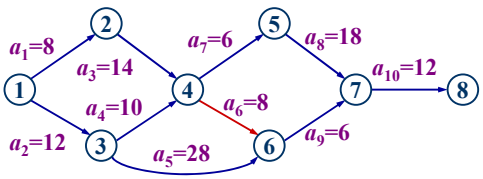
- $e[4] = Ve[3] = 12$, $l[4] = Vl[4]-a_4 = 22-10 = 12$,
- $e[5] = Ve[3] = 12$, $l[5] = Vl[6]-a_5 = 40-28 = 12$,
- $e[6] = Ve[4] = 22$, $l[6] = Vl[6]-a_6 = 40-8 = 32$,



149-154

	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl	0	8	12	22	28	40	46	58

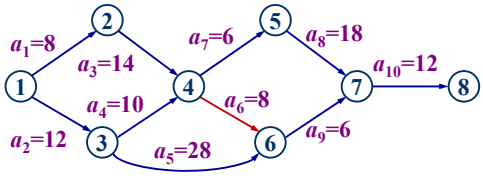
- $e[7] = Ve[4] = 22$, $l[7] = Vl[5]-a_7 = 28-6 = 22$,
- $e[8] = Ve[5] = 28$, $l[8] = Vl[7]-a_8 = 46-18 = 28$,
- $e[9] = Ve[6] = 40$, $l[9] = Vl[7]-a_9 = 46-6 = 40$,



149-155

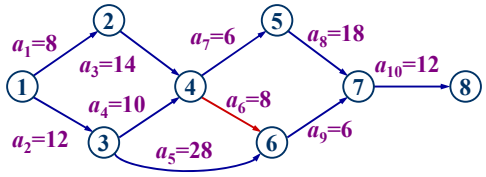
	1	2	3	4	5	6	7	8
Ve	0	8	12	22	28	40	46	58
Vl	0	8	12	22	28	40	46	58

- $e[10] = Ve[7] = 46$, $l[10] = Vl[8]-a_{10} = 58-12 = 46$.
- **从以上计算可得，只有活动 a_6 的$e[6] \neq l[6]$，它不是关键活动，其他活动都是关键活动。**



149-156

	1	2	3	4	5	6	7	8	9	10
e	0	0	8	12	12	22	22	28	40	46
l	0	0	8	12	12	32	22	28	40	46



- 这些关键活动构成 3 条关键路径，即{ V_1, V_2, V_4, V_5, V_7, V_8 }; { V_1, V_3, V_4, V_5, V_7, V_8 }; { V_1, V_3, V_6, V_7, V_8 }。

149-157

- 计算和使用关键路径的原则：
  - (1) 所有顶点按拓扑有序的次序重新编号后，相应的邻接矩阵的下三角部分均为零。
  - (2) 仅计算 *Ve*[*i*] 和 *Vl*[*i*] 是不够的，还须计算 *e*[*k*] 和 *l*[*k*]，才能判断哪些是关键活动。
  - (3) 如果同时存在几条关键路径，那么，不是任一关键活动加速一定能使整个工程提前。
  - (4) 想使整个工程提前，要考虑加速各关键路径上的关键活动。
  - (5) 如果关键路径上的任一关键活动延迟，整个工程将延迟。

149-158