第九章　图

图是一类复杂数据结构，可用于表示具有各种复杂关系的数据集合，在实际中应用广泛（例如第一章的例子）。

由于图的结构较复杂，因此有许多可能实现方式，有许多典型的算法。本课程介绍有关图的最基本知识，图的基本实现方法，以及图中的若干最基本计算问题和重要算法。

重点算法:

- 图的深度优先搜索与广度优先搜索
- 最小生成树的Prim算法
- 最小生成树的Kruskal算法
- 单源最短路径Dijkstra算法
- 所有顶点对之间最短路径的Floyd算法
- 拓扑排序
- 关键路径

这些算法是本章最重要的内容

9.1 图的基本概念:

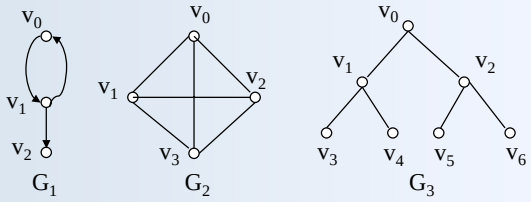
图是二元组 G = (V, E)，其中V是顶点的有穷非空集合（也可以有空图的概念），E是顶点偶对 (边) 的集合，$E \subseteq V \times V$。V 中的顶点也称为图G的顶点，E中的边也称为图G的边。

有向图：有向图中的边有方向，边是顶点的有序对。有序对用尖括号表示。$<v_i,v_j>$ 表示从 v_i 到 v_j 的有向边，v_i 是边的始点，v_j是边的终点。$<v_i,v_j>$ 和 $<v_j,v_i>$ 是两条不同的有向边。

无向图：无向图中的边没有方向，是顶点的无序对，无序对用圆括号表示，(v_i ,v_j) 和 (v_j ,v_i) 表示同一条无向边。

如果图 G 里有边 $<v_i,v_j> \in E$ 或者 $(v_i ,v_j) \in E$，顶点 v_j 称为 v_i 的邻接顶点。这种边称为与顶点 v_i 相关联的边或 v_i 的邻接边

图可用图形自然表示（圆圈表示顶点，线段或箭头表示边）：



两个限制：1）不考虑顶点到自身的边，若 (v_i ,v_j) 或$<v_i ,v_j>$ 是 G 的边，则$v_i \neq v_j$；2）顶点间没有重复出现的边，若(v_i ,v_j) 或$<v_i ,v_j>$ 是 G 的边，则它是这两个顶点间的唯一边。

去掉这些限制，可以得到稍微有些不同的研究对象。“图论”中也研究它们。

完全图：任意两个顶点之间都有边的图（有向图或无向图）。

显然:

- n 个顶点的无向完全图有 n*(n-1)/2 条边
- n 个顶点的有向完全图有n*(n-1) 条边。

度（顶点的度）：与一个顶点关联的边的个数。

对于有向图，还分为入度和出度，分别表示以该顶点为始点或者终点的边的个数。

无论是有向图还是无向图，顶点数 n，边数 e 和度数间满足关系：$D(v_i)$ 表示 v_i 的度数

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

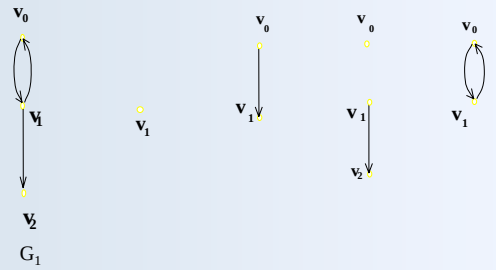
路径

- 路径：对G = (V, E)，若存在顶点序列 $v_{i0},v_{i1},\ldots,v_{i(m)}$，使 (v_{i0},v_{i1})，$(v_{i1},v_{i2}),\ldots,(v_{i(m-1)}, v_{i(m)})$ 都在E中(对有向图是$<v_{i0},v_{i1}>$，$<v_{i1},v_{i2}>,\ldots,<v_{i(m-1)}, v_{i(m)}>$ 都在E中)，则称从顶点v_{i0}到$v_{i(m)}$存在一条路径 $<v_{i0},v_{i1},\ldots,v_{i(m)}>$
- 路径长度：路径上的边数
- 回路（环）：起点和终点相同的路径。如果其中的其他顶点均不相同，则称为简单回路
- 简单路径：内部不包含环路的路径，即，路径上的顶点除起点和终点可能相同外，其它顶点均不相同
- 简单回路也是简单路径

子图

对于图G = (V，E) 和 G' = (V'，E')，如果 V' 是 V 的子集，而且 E' 是 E 的子集，那么就称 G' 是 G 的子图。

下面是有向图G_1的几个子图。



G_1

根

- 有向图 G 中若存在一顶点v，从 v 有路径可以到达图G中其它所有顶点，则称 G 为有根图，v 称为图 G 的根。
- 有根图中的根可能不唯一。

树是有唯一根的有根图（且所有顶点的入度至多为1）。

连通、连通图（连通无向图）

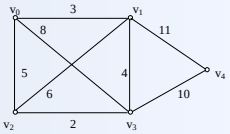
- 连通：若无向图 G 中存在从 v_i 到 v_j 的路径（显然它也是从 v_j 到 v_i 的路径），则称 v_i 与 v_j 连通（默认 v 到 v 连通）
- 连通图：如果无向图 G 中的任意两个不同顶点 v_i 和 v_j 都连通（即存在路径），则称 G 为连通图
- 连通分量：无向图G中的一个极大连通子图（不存在包含它的更大的连通子图）称为G的一个连通分量。若G不连通，它的连通分量将多于一个，它们形成G的一个划分

有向图的连通性

- 强连通图：如果有向图 G 中任意两个不同顶点 v_i 和 v_j 之间都存在从v_i到v_j以及从v_j和v_i的路径，则称 G 是强连通图
- 强连通分量：有向图G的极大强连通子图称为其强连通分量

带权图　网络

- 若图的每条边都被赋以一个权值，这种图称为带权图
- 带权的连通无向图称为网络。权值可用于表示实际应用中的某种与顶点之间的关联有关的量



右图为一个网络。

9.2 图的基本运算与周游

9.2.1 基本运算

1. 创建空图
   Graph createGraph ( )
2. 判断图g是否空图，是则返回真，否则返回假
   int isNullGraph (g)
3. 把图g置为空图
   void makeNullGraph (g)
4. 销毁一个图g，即释放图g占用的空间
   void destroyGraph (g)
   与有序树类似，图中的顶点和边也可以规定顺序
5. 找图中的第一个顶点，如果图是空图则返回NULL
   Vertex firstVertex (g)

6. 找图中顶点vi的下一个顶点
   Vertex nextVertex (g , vi)
7. 查找图中值为value的顶点
   Vertex searchVertex (g , value)
8. 在图g中增加一个值为value的顶点（不增加边）
   Graph addVertex (g , value)
9. 在图g中删除一个顶点和与该顶点相关联的所有边
   Graph deleteVertex (g , vertex)
10. 在图g中增加一条边<vi,vj>或者(vi,vj)
    Graph addEdge (g , vi , vj)
11. 在图g中删除一条边e（ <vi,vj>或者(vi,vj) ）
    Graph deleteEdge (g , vi , vj)

12. 判断图g中是否存在一条指定边<vi,vj>或者(vi,vj)

    int findEdge (g , vi , vj)

13. 找图g中与顶点v相邻的第一个顶点

    Vertex firstAdjacent (g , v)

    v与返回顶点构成的边也称为与v相关联的第一条边。

14. 找图g中与vi相邻的，相对相邻顶点vj的下一个相邻顶点

    Vertex nextAdjacent (g , vi , vj)

    vi与返回值构成的边也称关联vi与vj的边的下一条边

9.2.2 图的周游

图的周游：从图中某个顶点出发，按照某种方式系统访问图中所有顶点，且每个顶点仅访问一次。也称图的遍历。

这种周游的基本部分是访问一个顶点所在的连通分支里的全部结点。如果不是连通图，还需要去访问其他连通分支。

常见的图周游方法有:

1、深度优先周游（通过深度优先搜索的方式周游）

2、广度优先周游（通过广度优先搜索的方式周游）

两种方式对有向图和无向图都适用。

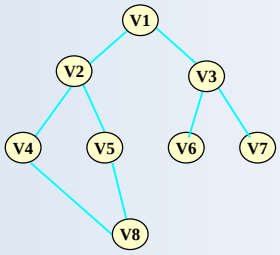
深度优先周游

深度优先周游 (Depth-First Traversal) 的策略是深度优先搜索（Depth-First Search），具体思想是:

1. 从指定顶点 v 出发，先访问 v 并将其标记为已访问过
2. 依次从 v 的未被访问过的各邻接顶点 w 出发进行深度优先搜索（算法需要为 v 的邻接结点假定一种顺序），直到图中与 v 连通的所有顶点都访问过
3. 如果图中还有未访问顶点，则选一个未访问顶点，由它出发重复上述过程，直到图中所有顶点都被访问为止

对图进行深度优先周游时，按访问顶点的先后次序所得到的顶点序列，称为该图的深度优先周游序列，简称 DFS 序列。

例子:



如果假定各个顶点的邻接顶点从左到右排序，得到的 DFS 序列:

v1→v2→ v4 → v8→v5→v3→v6→v7

示例算法:

```
void dft ( Graph g ) {
    Vertex v ;
    for ( v =firstVertex ( g ) ; v != NULL ;
          v = nextVertex ( g , v ) )
        if ( unvisited(v) ) dfs ( g , v ) ; /* 调用 dfs */
}

void dfs ( Graph g , Vertex v ) {
    Vertex v1;
    mark_visited(v);
    for ( v1 = firstAdjacent ( g , v ); v1 != NULL ;
          v1=nextAdjacent ( g ,v, v1 ) )
        if ( unvisited(v1) ) dfs ( g ,v1 ); /* 递归调用 dfs */
}
```

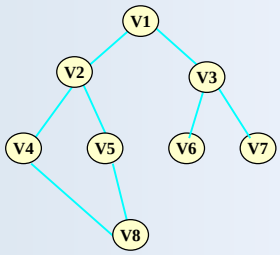

广度优先周游

广度优先周游(Breadth-First Traversal)又称为广度优先搜索（Breadth-First Search），具体过程是:

1. 从指定顶点v出发，先访问顶点v 并将其标记为已访问;
2. 依次访问 v 的所有相邻结点 $w_1, w_2, \ldots, w_x$（算法需要为 v 的邻接结点假定一种顺序），然后依次访问与 $w_1, w_2, \ldots, w_x$ 邻接的所有未访问顶点;
3. 直到所有已访问顶点的相邻结点都已访问为止。
4. 如果图中还有未访问顶点，则选择一个未访问顶点，由它出发进行广度优先搜索，直到所有顶点都已访问为止。

通过广度优先周游得到的顶点序列称为广度优先周游序列，或BFS序列

例子:



V1→v2→ v3 → v4→v5→v6→v7→v8

19

```
void bft ( Graph g ) {
    Vertex v ;
    for ( v =firstVertex (g) ; v != NULL; v = nextVertex (g , v) )
        if ( unvisited(v) ) bfs ( g , v ) ;
}
void bfs ( Graph g , Vertex v ) {
    Vertex v1 , v2;
    Queue q = createEmptyQueue ( ) ; /* 元素类型为Vertex* */
    for (mark_visited(v), enQueue (q ,v); !isEmptyQueue(q); ) {
        v1 = frontQueue(q) ;  deQueue(q);
        v2 = firstAdjacent(g ,v1);
        while (v2!=NULL) {
            if ( unvisited(v2) ){
                mark_visited(v2) = TRUE; enQueue ( q,  v2 );
            }
            v2 = nextAdjacent ( g , v1, v2 ) ;
        }
    }
}
```

20

9.3 图的存储

图的结构比较复杂，图中任意两个顶点间都可能存在联系

需要表示顶点及顶点间的联系，存储方法很多。应根据具体应用和需要做的操作，选择图的存储表示法

邻接矩阵表示法

邻接表表示法

其他方法：邻接多重表表示法*、图的十字链表*等

21

9.3.1 邻接矩阵表示法

邻接矩阵是表示顶点间相邻关系的矩阵

设G = (V，E) 为n个顶点的图，其邻接矩阵为如下n阶方阵:

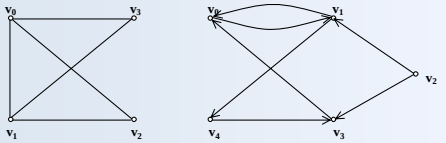
$$A[i,j]=\begin{cases}1 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是图G的边}\\ 0 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是图G的边}\end{cases}$$

邻接矩阵只表示了图中顶点个数和顶点间的联系（边）。

如果顶点本身还有其他信息，就需要用另外的方式表示顶点。例如另外用一个顶点表。

22

无向图G_5和有向图G_6的邻接矩阵分别为A_1和A_2



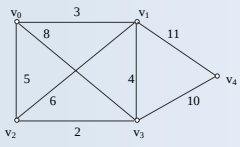
$$A_1=\begin{bmatrix}0&1&1&1\\1&0&1&1\\1&1&0&0\\1&1&0&0\end{bmatrix}\qquad A_2=\begin{bmatrix}0&1&0&0&0\\1&0&0&0&1\\0&1&0&1&0\\1&0&0&0&0\\0&0&0&1&0\end{bmatrix}$$

23

如果G是带权图，w_{ij}是边(v_i,v_j)或$<v_i,v_j>$的权，则其邻接矩阵定义为（用邻接矩阵元素记录边的权）：

$$A[i,j]=\begin{cases}w_{ij} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是图G的边}\\ 0\text{或}\infty & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是图G的边}\end{cases}$$

下图带权图的两种邻接矩阵分别为A_3和A_4。可以根据实际需要选择用 0 或者∞表示没有边。



$$A_3=\begin{bmatrix}0&3&5&8&0\\3&0&6&4&11\\5&6&0&2&0\\8&4&2&0&10\\0&11&0&10&0\end{bmatrix}\qquad A_4=\begin{bmatrix}\infty&3&5&8&\infty\\3&\infty&6&4&11\\5&6&\infty&2&\infty\\8&4&2&\infty&10\\\infty&11&\infty&10&\infty\end{bmatrix}$$

24

邻接矩阵表示法的存储结构定义

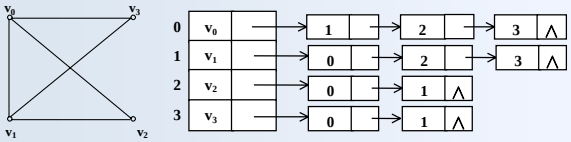
```
#define MAXVEX    常数
typedef char VexType;
typedef double AdjType; /* 无权图可以用 int 表示 0/1 */

typedef struct {
    VexType vexs[MAXVEX];  /* 顶点信息 */
    AdjType arcs[MAXVEX][MAXVEX];/* 边信息 */
    int n;                       /* 图的顶点个数 */
} GraphMatrix;
```

如果需要，可以定义 VexType 为表示顶点的结构。

9.3.2 邻接表表示法

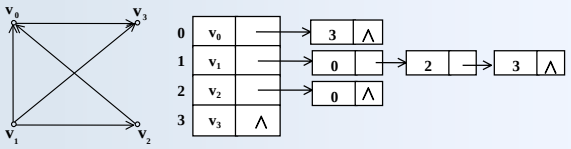
无向图:

- 一个顶点表，其中每个顶点有一个关联边的表;
- 每条边 (v_i,v_j) 在两个顶点 v_i，v_j 的边表中各有一个结点，因此每条边在边表中存储两次。
- 顶点 v_i 的边表中结点个数为顶点 v_i 的度

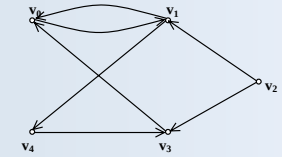


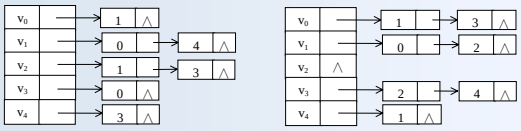
有向图

- 一个顶点表，每个顶点关联一个边表
- 顶点 v_i 的边表中每个结点对应的是以 v_i 为始点的一条边，因此，将有向图邻接表的边表称为出边表。
- 顶点 v_i 的边表中结点个数为顶点 v_i 的出度。
- 也可采用入边的表，表中结点个数是顶点的入度



出边表





(a) 是保存出边表的情况　　(b)是保存入边表的情况。

邻接表表示法的存储结构定义:

```
typedef struct EdgeNode, * PEdgeNode;
typedef struct EdgeNode {   /* 边结构 */
    int endvex;             /* 相邻顶点字段 */
    AdjType weight;         /* 边的权 */
    PEdgeNode nextedge;     /* 链字段 */
} EdgeNode, * EdgeList;     /* 边表 */

typedef struct {            /* 顶点结构 */
    VexType vertex;         /* 顶点信息 */
    EdgeList edgelist;      /* 边表头指针 */
} VexNode;
typedef struct {            /* 顶点表 */
    VexNode vexs[MAXVEX];
    int n;                  /* 图的顶点个数 */
} GraphList;
```


9.4 最小生成树

基本概念:

- 生成树
  - DFS生成树
  - BFS生成树
- 最小生成树
- Prim 算法
- Kruskal算法

生成树:

从连通无向图或强连通有向图 G 中的任一顶点 v_0 出发，或从有根有向图的根 v_0 出发，存在到其他各结点的路径。

若 G 有 n 个顶点，必可找到 n-1 条边，其中包含从 v_0 到其他所有结点的路径（通过归纳易证）。这 n-1 条边形成 G 的一个子图 T（包含 G 所有顶点和 n-1条边）。

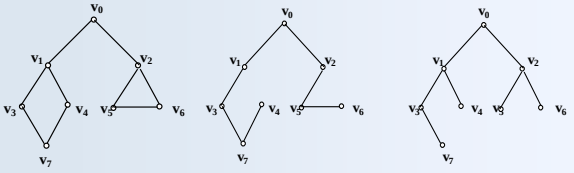
对无向图 G，T 是最小连通子图（去掉任意一条边后图不再连通）。n 个顶点 n-1 条边的图形成树形。因此称子图 T 为 G 的生成树。对有向图的定义类似。

无向“树”就是连通无环图。有向“树”的边都位于从根到其他结点的（有方向的）路径上。

生成树可能不唯一。非连通的无向图可划分为一组连通分量，因此可以讨论它们的生成树林。

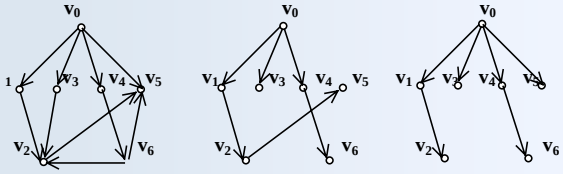
从连通无向图或强连通有向图中任一顶点出发周游，或从有根有向图的根顶点出发周游，可以访问所有顶点。周游经过的边加上所有顶点构成该图的一棵生成树。

构造生成树的过程可以按深度优先周游或广度优先周游，周游中记录访问的所有顶点和经过的边，得到的是深度优先生成树或广度优先生成树，简称为DFS生成树或BFS生成树。



例：无向图　　DFS生成树　　BFS生成树

有向图



DFS生成树　　BFS生成树

也可把连通无向图的生成树定义为图的最小连通子图，把一般无向图的生成树林定义为各连通分量的最小连通子图的集合。

生成树描述了图的一种框架结构，也称为支撑树。

最小生成树

网络 G（带权连通无向图）的边带有权值，其生成树中各条边的权值之和称为该生成树的权。

图 G 可能有多棵不同生成树，其权值可能不同。其中权值最小的生成树称为 G 的最小生成树 (Minimum Spanning Tree, MST)。（最小生成树可能不唯一）

最小生成树有许多实际应用:

- 把网络顶点看作城市，边的权看作连接城市的通讯线路成本。根据最小生成树建立的是城市间成本最低的通讯网。可类似考虑成本最低的公路网等
- 农村村庄之间的输电网络、有线电视网
- 输水管线、暖气管线、配送中心与线路的规划

MST性质:

设G = (V，E) 是网络，U是顶点集V的任一真子集。

假设有边 e = (u,v)，其顶点 u ∈ U，v ∈ V - U，而且 e 是图 G 中所有一个端点在 U 另一端点在 V - U 里的边中权值最小的边，则一定存在 G 的一棵包括边 e 的最小生成树。

MST性质的证明:

对 G 的任一最小生成树 T，其中必有一条一端在U另一端在V-U的边e'。加上e得到一个包含环路的图，去掉 e' 得到另一生成树，其权不大于T 的权。

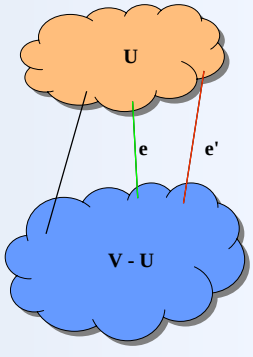


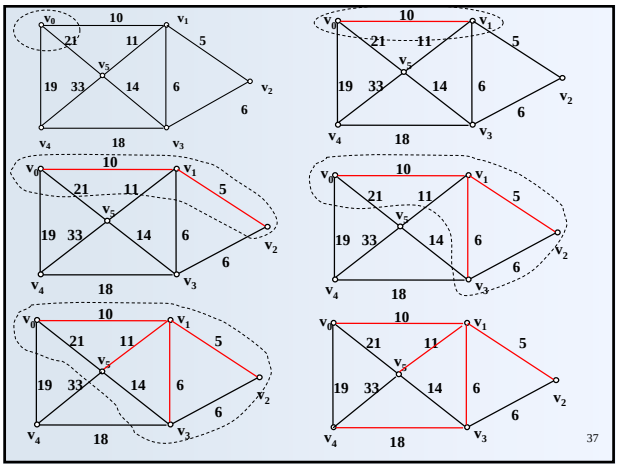
图 G

Prim算法

基本思想:

1. 从图 G 的顶点集 V 中任取一顶点 (例如取顶点v_0) 放入集合 U 中，这时U = {v_0}，TE = { }，(U, TE) 是树
2. 在所有一个顶点在集合 U 里另一顶点在集合 V – U 的边中，找出权值最小的边 e = (u, v)（u ∈ U，v ∈ V-U），将顶点 v 加入集合 U，并将 e 加入 TE。(U, TE) 仍是一棵树
3. 重复2，直至 U = V（树中包含所有顶点）。这时 TE 有n-1 条边，T = (U, TE) 就是 G 的一棵最小生成树

算法直接利用 MST 性质构造最小生成树



37

Prim算法的实现:

1. 图采用邻接矩阵表示。用顶点对的下标 (在顶点表中的下标) 表示边，定义:

```
typedef struct{
    int start_vex, stop_vex;     /* 边的起点和终点 */
    AdjType weight;              /* 边的权 */
} Edge;
```

2. 构造中用一个类型为Edge的数组mst（n为顶点数）

   Edge mst[n-1];

   开始时把 v0 放入顶点集U，在 mst里记录从v0到其余顶点的最短边和权（没有边时权值取无穷）

   反复选择最小权顶点加入 mst，同时更新到其他顶点的边和权，使 mst 始终记录从 U 到其他顶点的最短边和权

   算法结束时 mst 中存放着最小生成树的n-1条边

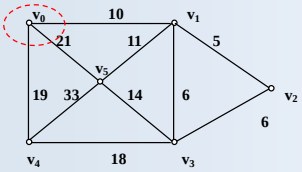
38

算法实现梗概:

- 开始v0 属于最小生成树 T 的结点集 U，mst 初始化为从v0 到其他各顶点的边和权值，无边时权值取无穷大
- 循环 n-1 次，向T加入 n-1 个顶点和相关边（在检查循环条件时，mst[0] .. mst[i-1] 给出了属于U的边和结点）
  - 找出关联到 U 之外的顶点的最短边
  - 把该边和关联点加入T（把相关边与smt[i] 交换位置）
  - 检查新顶点 v_i 与 U 之外的顶点之间边的权值，如果它比 mst 中记录的到那个顶点的边的权值小，就用 v_i 到那个顶点的边代替 mst 里原记录的相关边
- 将 n-1 个顶点与边加入后，算法结束。如果mst里还有权为无穷大的边，那么图不连通，没有最小生成树。

39

例: 已知图G_9及其邻接矩阵，构造该图的最小生成树



$$\begin{bmatrix} 0 & 10 & \infty & \infty & 19 & 21 \\ 10 & 0 & 5 & 6 & \infty & 11 \\ \infty & 5 & 0 & 6 & \infty & \infty \\ \infty & 6 & 6 & 0 & 18 & 14 \\ 19 & \infty & \infty & 18 & 0 & 33 \\ 21 & 11 & \infty & 14 & 33 & 0 \end{bmatrix}$$

1、n = 6，只有顶点v_0在最小生成树中。

mst = {{0,1,10}, {0,2,∞}, {0,3,∞}, {0,4,19}, {0,5,21} }

2、在mst[0]到mst[4]中找出权值最小的边mst[0]，即(v_0, v_1)，将顶点v_1及边(v_0, v_1)加入最小生成树。

而后考虑是否要调整一些点的“已知最近联系途径”

40

3、考虑mst[1]到mst[4]

$$\begin{bmatrix} 0 & 10 & \infty & \infty & 19 & 21 \\ 10 & 0 & 5 & 6 & \infty & 11 \\ \infty & 5 & 0 & 6 & \infty & \infty \\ \infty & 6 & 6 & 0 & 18 & 14 \\ 19 & \infty & \infty & 18 & 0 & 33 \\ 21 & 11 & \infty & 14 & 33 & 0 \end{bmatrix}$$

(v_1, v_2)=5 小于(v_0, v_2)　调整

(v_1, v_3)=6 小于(v_0, v_3)　调整

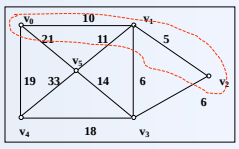
(v_1, v_4)=∞ 大于(v_0, v_4)　不变

(v_1, v_5)=11 小于(v_0, v_5)　调整

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {0,4,19}, {1,5,11} }

红字表示已经在最小生成树中的边

4、在mst[1]到mst[4]找出权值最小的边mst[1]，(v_1, v_2)，将顶点v_2及边(v_1, v_2)加入最小生成树



5、考虑 mst[2] 到 mst[4]

(v_2, v_3)=6 大于 (v_1, v_3) 不变

(v_2, v_4)=∞ 大于(v_0, v_4) 不变

(v_2, v_5)=∞ 大于(v_1, v_5) 不变

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {0,4,19}, {1,5,11} }

41

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {0,4,19}, {1,5,11} }

6、在mst[2]到mst[4]中找出权值最小的边mst[2]，即(v_1, v_3)，将顶点v_3及边(v_1, v_3)加入最小生成树

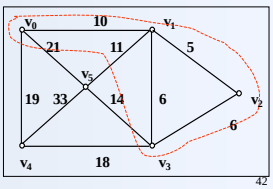
$$\begin{bmatrix} 0 & 10 & \infty & \infty & 19 & 21 \\ 10 & 0 & 5 & 6 & \infty & 11 \\ \infty & 5 & 0 & 6 & \infty & \infty \\ \infty & 6 & 6 & 0 & 18 & 14 \\ 19 & \infty & \infty & 18 & 0 & 33 \\ 21 & 11 & \infty & 14 & 33 & 0 \end{bmatrix}$$

7、考虑 mst[3]到mst[4]

(v_3, v_4)=18 小于(v_0, v_4) 调整

(v_3, v_5)=14 大于(v_1, v_5) 不变

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {3,4,18}, {1,5,11} }



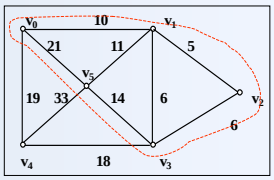
42

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {3,4,18}, {1,5,11} }

8、在mst[3]到mst[4]中找出权值最小的边mst[4]，即(v_1, v_5)，将顶点v_5及边(v_1, v_5)加入最小生成树。互换mst[3]和mst[4]

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {1,5,11} , {3,4,18}}

$$\begin{bmatrix} 0 & 10 & \infty & \infty & 19 & 21 \\ 10 & 0 & 5 & 6 & \infty & 11 \\ \infty & 5 & 0 & 6 & \infty & \infty \\ \infty & 6 & 6 & 0 & 18 & 14 \\ 19 & \infty & \infty & 18 & 0 & 33 \\ 21 & 11 & \infty & 14 & 33 & 0 \end{bmatrix}$$



mst = {{0,1,10}, {1,2,5}, {1,3,6}, {1,5,11} , {3,4,18}}

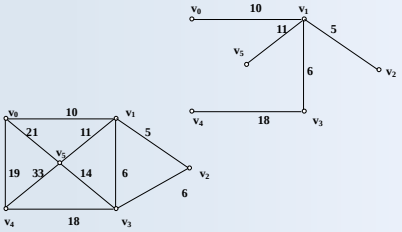
9、 调整mst[4]

(v_5, v_4)=33 大于(v_3, v_4) 不变

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {1,5,11} , {3,4,18}}

10、将mst[4]加入最小生成树。

mst = {{0,1,10}, {1,2,5}, {1,3,6}, {1,5,11} , {3,4,18}}



$$\begin{bmatrix} 0 & 10 & \infty & \infty & 19 & 21 \\ 10 & 0 & 5 & 6 & \infty & 11 \\ \infty & 5 & 0 & 6 & \infty & \infty \\ \infty & 6 & 6 & 0 & 18 & 14 \\ 19 & \infty & \infty & 18 & 0 & 33 \\ 21 & 11 & \infty & 14 & 33 & 0 \end{bmatrix}$$


```
void prim(GraphMatrix * pgraph, Edge mst[]) {
  int i, j, min, vx, vy, n = pgraph->n;
  double weight;   Edge edge;
  for (i = 0; i < n-1; ++ i) { /* 边组初始化为(v0, vi) */
    mst[i].start_vex = 0;
    mst[i].stop_vex = i+1; /* mst[i] 保存v0到vi+1的边信息 */
    mst[i].weight = pgraph->arcs[0][i+1];
  }
  /* mst[i]..mst[n-2]记录从 U 到 V-U 顶点的最短边 */
  for (i = 0; i < n-1; ++ i) { /* 循环加入各边，共n-1条 */
    weight = MAX;  min = i;
    for (j = i; j < n-1; ++ j)
      /* 从边mst[i] .. mst[n-2]中选最短的 (vx,vy) */
      if(mst[j].weight < weight) {
        weight = mst[j].weight;
        min = j;
      }
```



```
    /* mst[min]是 U 到 V-U 的边 (vx,vy) 中最短的，
       将mst[min]加入MST（换到 mst[i]） */
    edge = mst[min];  mst[min] = mst[i];  mst[i] = edge;
    vx = mst[i].stop_vex; /* vx为刚加入mst的顶点的下标 */
    /* 考察mst[i+1] .. mst[n-2]，是否该用来自vx的边取代 */
    for(j = i+1; j < n-1; ++ j) {
      vy = mst[j].stop_vex;
      weight = pgraph->arcs[vx][vy];
      if (weight < mst[j].weight) {
        mst[j].weight = weight;
        mst[j].start_vex = vx;
      }
    }
  }
}
```


时间复杂度:

Prim算法的时间主要用在选择最小生成树的n-1条边上。外循环执行n-1次，两个内循环，时间耗费为：

$$\sum_{i=0}^{n-2}\left(\sum_{j=i}^{n-2}O(1)+\sum_{j=i+1}^{n-2}O(1)\right)\approx 2\sum_{i=0}^{n-2}\sum_{j=i}^{n-2}O(1)$$

因此算法的时间复杂度为$O(|V|^2)$。

采用不同辅助数据结构实现，可能使 Prim算法的复杂性变为 O(|E| log |V|) 或 O(|E| + |V| log |V|)。这里 |E| 和 |V| 是边集合和结点集合的大小。

在边稀疏的图上速度快得多。

参考书：T. H. Corman等，Introduction to Algorithms，MIT Press（高等教育出版社影印），2001

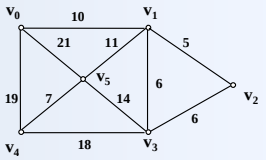
Kruskal算法

基本思想:

设G = (V，E) 是网络，构造最小生成树的过程:

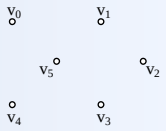
- 初始状态是只有n个顶点而无边的非连通图T = (V， φ)，T中每个顶点自成一个连通分量。
- 将 E 中的边按递增顺序排列。构造中从小到大顺序选择两端点在T的两个不同连通分量的边 e，把 e 加入 T 使这两个连通分量由于该边连接变成一个连通分量（每次操作，T 减少一个连通分量）。
- 不断加入新边，直到 T 所有顶点都在一个连通分量上为止，这个连通分量就是 G 的一棵最小生成树。
- 若不能得到一个连通分量，则图不连通，无最小生成树。

例子：用Kruskal方法构造右图的最小生成树

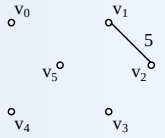


E中的边按权递增顺序排列为：
(v_1, v_2):5, (v_1, v_3):6, (v_2, v_3):6, (v_4, v_5):7, (v_0, v_1):10, (v_1, v_5):11, (v_3, v_5):14, (v_3, v_4):18, (v_0, v_4):19, (v_0, v_5):21,

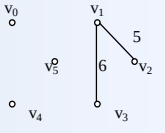
①．初始，T为只包含6个顶点的非连通图



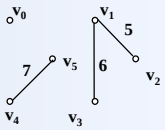
② 边(v_1, v_2)的两个顶点v_1，v_2分别属于两个连通分量，将边(v_1, v_2)加入T



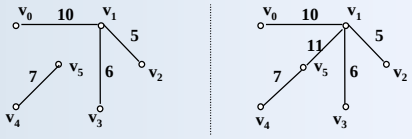
③．同理，将边 (v_1, v_3) 加入T



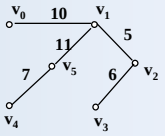
④．由于边(v_2, v_3)的两个顶点v_2，v_3属于同一个连通分量，因此舍去这条边。下一最短边是(v_4, v_5)，将它加入 T



⑤．将边(v_0, v_1)、(v_1, v_5)加入T，边(v_3, v_4)加入T。这时T中含的边数为5条，成为一个连通分量，T就是G的一棵最小生成树



说明：图的最小生成树不一定唯一。上例中边(v_1, v_3)和(v_2, v_3)长度相同，该图另一棵最小生成树如下



算法描述：

```
T = (V, ϕ)
while(T中所含边数<n-1) {
    从E中选取当前最短边(u,v);
    从E中删去边(u,v);
    if ( (u,v) 加入T中后不产生回路) 将边(u,v)加入T中;
}
```

采用的数据结构：

- 边数组mst记录构造中的最小生成树，num是已有边数
- 整数数组 status[0 .. n-1] 记录各个结点所属的连通分支，每个连通分支用一个顶点（代表顶点）的下标代表
- 初始时各顶点属于自己的连通分支，构造过程中修改为新确定的代表顶点的下标


```
int kruskal(GraphMatrix  *graph, Edge mst[]) {
   int i, j, num = 0, start, stop, n = graph->n;
   double minweight;
   int* status = (int *)malloc( sizeof(int)*n );
   for (i = 0; i < n; ++ i)
      status[i] = i; /* 每个顶点属于自己代表的连通分支 */
   while (num < n - 1){
      minweight = MAX;
      for (i = 0; i < n-1; ++ i)
         for (j = i+1; j < n; ++ j)
            if ( graph->arcs[i][j] < minweight ){
               start = i; stop = j;
               minweight = graph->arcs[i][j];
            }
      /* start和stop是找到的最短边的起点和终点 */
      if (minweight == MAX) return FALSE; /* 无 MST */
```



```
      /* 加入start和stop组成的边不产生回路*/
      if (status[start] != status[stop]){ /* 不属于同一连通分支 */
         mst[num].start_vex = start;
         mst[num].stop_vex = stop;
         mst[num].weight = graph->arcs[start][stop];
         ++ num;
         /* 原 status[stop] 代表的连通分支 的结点，现都属于
            status[start] 代表的连通分支，该代表顶点 */
         for ( j = status[stop], i = 0; i < n; ++ i)
            if (status[i] == j) status[i] = status[start];
      }
      /* 删除start和stop组成的边*/
      graph->arcs[start][stop] = MAX;
   }
   return TRUE; /* 得到了最小生成树*/
}
```

算法分析:

算法的时间复杂性由三重循环确定。外层循环做n-1次，两个内层循环做 n*(n+1)/2次（n = |V|）。因此复杂性为 $O(|V|^3)$

采用其他辅助结构和实现方式，可能使 Kruskal 算法的复杂性变为 O(|E| log |V|)。

在边稀疏的情况下，O(|E| log |V|) 优于$O(|V|^3)$。

参考书：T. H. Corman等，Introduction to Algorithms，MIT Press（高等教育出版社影印），2001

9.5 最短路径

最短路径问题:

在网络或带权有向图里，从顶点 v_i 到 v_j 的一条路径上各边的长度之和称为该路径的长度。从 v_i 到 v_j 的所有路径中长度最短的路径就是最短路径。

最短路径在实际中很有价值，许多调度问题与此有关：运输（最短里程，最短运费，最低成本），加工流程等等。

9.5.1 从一个顶点到其它各顶点的最短路径（Dijkstra算法）

9.5.2 各对顶点之间的最短路径（Floyd算法）

9.5.1 一个顶点到其它各个顶点的最短路径

Dijkstra算法给出图中一个顶点 v_0 到所有其他结点的最短路径，该算法要求图中所有边的权大于等于 0。

Dijkstra 算法的工作过程与 Prim 算法类似。假设要找从 v_0 到其他顶点的最短路径，算法执行过程中把顶点分为两个集合:

当时已知最短路径的顶点集合 U

尚不知最短路径的顶点集合 V - U

算法逐步扩充已知最短路径的顶点集合，每次从 V – U 中找到一个（当时已经知道最短路径的）新顶点，加入 U。

反复这样做，直到找出从 v_0 到所有顶点的最短路径，以及这些最短路径的长度（距离）。

如何为 U 扩充顶点，在 V-U 里找到下一能确定最短路径的顶点?

$v_i \in$V-U 的已知最短距离定义为:

若存在 $v_j \in U$ 有边 (v_j, v_i)，则

$cdis(v_i) = min\{dis(v_j) + w(v_j,v_i) \mid v_j \in U\}$

$dis(v_j)$ 表示从 v_0 到 v_j 的距离，$w(v_j,v_i)$ 是 (v_j, v_i) 的权;

若没有这种 v_j，定义 $cdis(v_i) = \infty$

性质：若 v_i 是 V-U 中 *cdis* 值最小的结点，那么 $dis(v_i) = cdis(v_i)$。

即：从 v_0 到 v_i 的最短路径已知，可以把 v_i 加入集合 U。

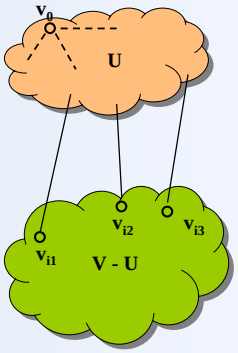
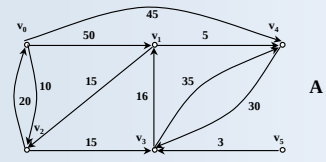


图 G

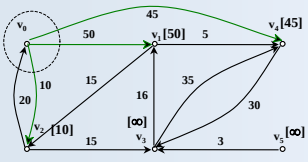
算法梗概:

- 初始时集合 U 中放入顶点 v_0，v_0 的距离为 0，V – U 中顶点 v_i 与 v_0 的已知最短路长为 $w(v_0, v_i)$（i=1, 2, …, n-1），若v_0和v_i间无边直接相连，则 v_i 的已知最短路长为 ∞
- 从 V – U 中选出当时已知最短路长最小的顶点 v_{min} 加入集合 U（这时到 v_{min} 的已知最短路长就是 v_0 到 v_{min} 距离）
- 更新 V – U 中各顶点的已知最短路长：如果从 v_0 经过 v_{min} 到 v_i 的路径长度比原来的已知最短路长更小，那么就更新到 v_i 的已知最短路长，并记录经过 v_{min} 的这条路径
- 反复上述两步操作，直到从v_0出发可达的所有顶点都已经在集合U中为止
- 如果有顶点不能加入 U，说明图不连通，到这些顶点的最短路径长度为无穷

例：已知带权图G_{10}如图8.18所示及其邻接矩阵A，求从顶点v_0到其它各顶点的最短路径

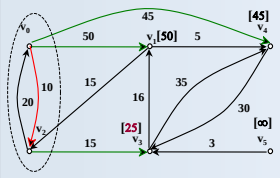


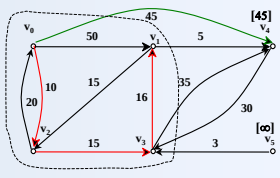
$$A = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 16 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

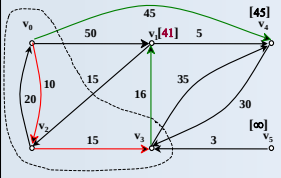


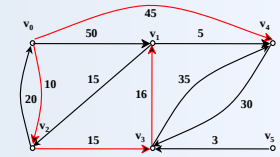
绿色为从U到V-U的边，

选 (v_0, v_2) 把v_2加入U









61

Dijkstra 算法的实现：

图用邻接矩阵表示。

数组 dist[n] 用于存放各顶点，顶点 v_0 到其它顶点的已知最短路径和最短路径长度，其元素结构为：

```
typedef struct {
    VexType vertex;  /* 顶点信息 */
    AdjType len;     /*已知最短路长/最短路径长度 */
    int  prevex;     /* v0 到 vi 最短路径上 vi 的前一顶点 */
} Path;
Path dist[n];        /* n为图中顶点个数*/
```

注意：dist[i] 在算法里只用于表示 v_0 到 v_i 的路径

Path里的vertex（用于表示结点信息）在算法中没有用

62

算法细节：

1. 初始时U中只有顶点v_0（在图的对角线里记录），v_0到v_i (i = 1, …,n-1) 的已知最短路长为 $w(v_0,v_i)$，记在 dist[i]，路径上的前驱顶点都是v_0。无边时为无穷大，前驱记 -1
2. 从V - U中找出距离最小的顶点v_{min}加入U，从 v_0 到 v_{min}的已知最短路长就是v_{min}的距离。算法把邻接矩阵的对角线元素置 1 表示顶点放入U（也可以用其他方法）
3. 调整 V-U 剩余顶点的距离。若通过新加入U的v_{min}可使v_0到 v_i (v_i ∈ V-U) 的距离更小，就修改v_i的已知最短路长：

   当 dist[i].len > dist[min].len + graph.arcs[min][i] 时

   将顶点v_i的距离改为dist[min].len+graph.arcs[min][i];

   把路径上v_i的前趋改为 v_{min}（设dist[i].prevex = min）
4. 重复(2)，(3) 步骤n-1次，可得到所有可达结点的最短路径（根据前驱追溯）及其长度。

63

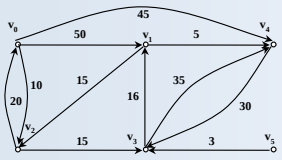
```
void init(GraphMatrix* pgraph, Path dist[]) {
  int i;
  dist[0].length = 0;
  dist[0].prevex = 0;
  dist[0].vertex = pgraph->vexs[0]; /* 下面没用到 */
  pgraph->arcs[0][0] = 1;
  /* 用矩阵对角线元素记录集合 U，为 1 表示在U中。
     也可修改 Path 类型的定义，在数组 dist 里记录 */
  for(i = 1; i < pgraph->n; ++ i) { /* 初始化V-U中顶点的距离 */
    dist[i].len = pgraph->arcs[0][i];
    if (dist[i].len != MAX)
      dist[i].prevex = 0;
    else  dist[i].prevex = -1;
  }
}
```

64

```
void dijkstra(GraphMatrix *graph, Path dist[]) {
  int i, j, mv, n = graph->n;   AdjType min;
  init(graph, dist); /* 初始化，集合U中只有顶点v0 */
  for(i = 1; i < n; ++ i) {
    min = MAX; mv = 0;
    for (j = 1; j < n; ++ j) /*在V-U中选出距 v0 最近的顶点*/
      if( graph->arcs[j][j] == 0 && dist[j].len < min )
      {  mv = j;   min = dist[j].len;  }
    if (mv == 0) break;  /* v0 与 V-U 的顶点不连通，结束 */
    graph->arcs[mv][mv] = 1;/* 顶点mv加入U */
    for (j = 1; j < n; ++ j) { /*调整V-U 顶点的已知最短路径*/
      if (graph->arcs[j][j] == 0 &&  /*为V-U的顶点*/
          dist[j].len > dist[mv].len + graph->arcs[mv][j]) {
        dist[j].prevex = mv; /* 调整已知最短路径信息 */
        dist[j].len = dist[mv].len + graph->arcs[mv][j];
  }  } }
}
```

65

例：已知带权图G_{10}如图8.18所示及其邻接矩阵A，求从顶点v_0到其它各顶点的最短路径



$$A = \begin{bmatrix} 0 & 50 & 10 & \infty & 45 & \infty \\ \infty & 0 & 15 & \infty & 5 & \infty \\ 20 & \infty & 0 & 15 & \infty & \infty \\ \infty & 16 & \infty & 0 & 35 & \infty \\ \infty & \infty & \infty & 30 & 0 & \infty \\ \infty & \infty & \infty & 3 & \infty & 0 \end{bmatrix}$$

数组 dist 的变化过程：(下划线为U顶点，红为新加入顶点)

{{0, 0}, {50, 0}, {10, 0}, {MAX, -1}, {45, 0}, {MAX, -1} }

{{0, 0}, {50, 0}, {10, 0}, {25, 2}, {45, 0}, {MAX, -1} }

{{0, 0}, {41, 3}, {10, 0}, {25, 2}, {45, 0}, {MAX, -1} }

{{0, 0}, {41, 3}, {10, 0}, {25, 2}, {45, 0}, {MAX, -1} }

{{0, 0}, {41, 3}, {10, 0}, {25, 2}, {45, 0}, {MAX, -1} }

66

{{0, 0}, {41, 3}, {10, 0}, {25, 2}, {45, 0}, {MAX, -1} }

由dist的prevex可得到最短路径。如 v_0 到 v_1 的最短路径:

dist[1].prevex = 3 可知路径上顶点 v_1 的前一个顶点是 v_3
dist[3].prevex = 2可知路径上顶点v_3的前一个顶点是v_2
dist[2].prevex = 0可知路径上前一个顶点是v_0

即最短路径为(v_0 , v_2 , v_3 , v_1)，路径长度为 dist[1].length = 41

Dijkstra算法的时间复杂度:

算法中的初始化部分的时间复杂度为O(n)，

求最短路径部分由一个大循环组成，外循环 n-1 次，两个内循环各 n-1 次。算法时间复杂度为$O(n^2)$。

采用其他数据结构可能得到算法复杂性 O(|V| log |V| + |E|)。如果图较稀疏，可能快得多。

9.5.2 各对顶点间的最短路径

- 可以用Dijkstra算法:

  依次把图中每个顶点作为起始点，用Dijkstra方法求出从该顶点到其它顶点的最短路径

- Floyd-Warshall算法，直接算出各对顶点间的最短路径

Floyd算法的基本思想:

设图G = (V，E) 有n个顶点，用邻接矩阵作为存储结构

如果边 (v_i, v_j)∈E，那么从 v_i 到顶点 v_j 有长度为arcs[i][j]的路径，但该路径未必是从 v_i 到 v_j 的最短路径，可能存在从 v_i 到 v_j，途中经过其它顶点的路径

需要在 v_i 到 v_j 的可能经过任何顶点的路径中找最短路径

开始，k = 0: 已知从 v_i 到 v_j 途中不经任何其他结点的路径

k = 1: 对每对v_i 和 v_j，(除已知直接路径外) 从v_i 到 v_j 途经顶点的下标小于 k（不大于0） 的路径可分为两段:

$<v_i, v_0>$ $<v_0, v_j>$ (没有就认为有长∞的路径)

其长度是两段路径的长度之和。比较这一路径和直接路径，可确定 v_i 到 v_j 的途径顶点的下标不大于 0 的最短路径。

k = 2: 对每对 v_i 和 v_j，(除至此已知的路径外) 从 v_i 到 v_j 途经顶点的下标小于k（不大于 1）的路径可分为两段:

$<v_i, \ldots, v_1>$ $<v_1, \ldots, v_j>$

两段所经顶点的下标不大于0，长度是两段路径的长度之和

用这一新路径与从已知 v_i 到 v_j 的最短路径比较，就可确定v_i 到 v_j 的途经顶点的下标不大于 1 的最短路径

一般而言，如果已考察过从 v_i 到 v_j 的途经顶点的下标小于 k 的所有路径，而且已知这样路径中的最短路径。

对每一对v_i 和 v_j，（除至此已知的路径外）v_i 到 v_j 途经顶点的下标小于 k + 1 的路径必可分为两段:

$<v_i, \ldots, v_k>$ $<v_k, \ldots, v_j>$

两段途径顶点的下标都不大于 k-1，路径长度是两段路径的长度之和。用该路径与已知的从 v_i 到 v_j的最短路径比较，就可确定从v_i 到 v_j 的途经顶点的下标小于 k+1 的最短路径

如此继续，直到 做完k = n（途径结点的下标不大于 n - 1），就确定了从v_i 到 v_j 的所有路径中最短的路径

这里假定结点下标为 0 到 n - 1

Floyd算法的实现

定义一系列 n×n 方阵: 其中$A_k[i][j]$ 表示从 v_i 到 v_j，途径顶点可为 $v_0, v_1, \ldots, v_{k-1}$ 的最短路径的长度 ($0 \leqslant k \leqslant n$)。

显然 A_0 就是图的邻接矩阵，$A_0[i][j]$ 表示从 v_i 到 v_j 不经过任何顶点的最短路径长度。$A_n[i][j]$ 是从 v_i 到 v_j 的最短路径长度

矩阵序列 A_0，A_1，A_2，…，A_n 可以递推计算:

$A_0[i][j] = arcs[i][j]$ $\quad 0 \leqslant i \leqslant n-1, 0 \leqslant j \leqslant n-1$

$A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+A_k[k][j]\}$，$0 \leqslant k \leqslant n-1$，这时考虑的是有顶点 v_k 为中间顶点的路径

$A_k[i][j]$ 为 v_i 到 v_j 的经由顶点下标小于 k 的最短路径的长度

（注意，顶点编号为 0 .. n-1）

为记录找到的路径，另外安排一系列矩阵$nextvex_k$ [n][n]，$nextvex_k[i][j]$ 的值为在从 v_i 到 v_j 中间允许有顶点$v_0, v_1, \ldots, v_{k-1}$的最短路径上，顶点 v_i 的后继顶点

初始时，若 $A_0[i][j] = \infty$（没有边），则令$nextvex_0$ [i][j]= －1，否则令 $nextvex_0$ [i][j] = j，表示 v_j 是 v_i 的后继顶点。

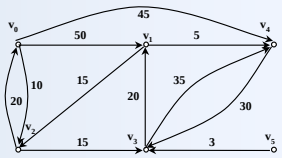
在由A_k计算A_{k+1}时，若 $A_{k+1}[i][j]$ 被更新为 $A_k[i][k]+ A_k[k][j]$，那么就设 $nextvex_{k+1}$ [i][j] = $nextvex_k$ [i][k]（在从 v_i 到 v_j 的路径上 v_i 的后继顶点就是原 v_i 到 v_k 路径上 v_i 的后继顶点）

计算完成时，$nextvex_{k+1}[i][j]$ 就是在从 v_i 到 v_j 的可以途径顶点 $v_0, v_1, \ldots, v_k$ 的最短路径上，顶点v_i的后继顶点。

最后 $nextvex_n[i][j]$ 就是从v_i到v_j的最短路径上 v_i 的后继结点。追溯这个矩阵，可得到任何一对结点之间的最短路径。

例：用Floyd方法求下图各顶点间的最短路径长度

$$arcs=\begin{bmatrix}0&50&10&\infty&45&\infty\\\infty&0&15&\infty&5&\infty\\20&\infty&0&15&\infty&\infty\\\infty&20&\infty&0&35&\infty\\\infty&\infty&\infty&30&0&\infty\\\infty&\infty&\infty&3&\infty&0\end{bmatrix}$$



$$A_0=\begin{bmatrix}0&50&10&\infty&45&\infty\\\infty&0&15&\infty&5&\infty\\20&\infty&0&15&\infty&\infty\\\infty&20&\infty&0&35&\infty\\\infty&\infty&\infty&30&0&\infty\\\infty&\infty&\infty&3&\infty&0\end{bmatrix}\quad nextvex_0=\begin{bmatrix}0&1&2&-1&4&-1\\-1&1&2&-1&4&-1\\0&-1&2&3&-1&-1\\-1&1&-1&3&4&-1\\-1&-1&-1&3&4&-1\\-1&-1&-1&3&-1&5\end{bmatrix}$$

$$A_0=\begin{bmatrix}0&50&10&\infty&45&\infty\\\infty&0&15&\infty&5&\infty\\20&\infty&0&15&\infty&\infty\\\infty&20&\infty&0&35&\infty\\\infty&\infty&\infty&30&0&\infty\\\infty&\infty&\infty&3&\infty&0\end{bmatrix}$$

考虑经过顶点v_0的路径，求：

$A_1[i][j]=\min\{A_0[i][j], A_0[i][0]+A_0[0][j]\}\quad 0\leqslant i\leqslant n-1,\ 0\leqslant j\leqslant n-1$

$$A_1=\begin{bmatrix}0&50&10&\infty&45&\infty\\\infty&0&15&\infty&5&\infty\\20&70&0&15&65&\infty\\\infty&20&\infty&0&35&\infty\\\infty&\infty&\infty&30&0&\infty\\\infty&\infty&\infty&3&\infty&0\end{bmatrix}\quad nextvex_1=\begin{bmatrix}0&1&2&-1&4&-1\\-1&1&2&-1&4&-1\\0&0&2&3&0&-1\\-1&1&-1&3&4&-1\\-1&-1&-1&3&4&-1\\-1&-1&-1&3&-1&5\end{bmatrix}$$

$$A_1=\begin{bmatrix}0&50&10&\infty&45&\infty\\\infty&0&15&\infty&5&\infty\\20&70&0&15&65&\infty\\\infty&20&\infty&0&35&\infty\\\infty&\infty&\infty&30&0&\infty\\\infty&\infty&\infty&3&\infty&0\end{bmatrix}$$

考虑经过顶点v_1的路径，求：

$A_2[i][j]=\min\{A_1[i][j],A_1[i][1]+A_1[1][j]\}\quad 0\leqslant i\leqslant n-1,\ 0\leqslant j\leqslant n-1$

$$A_2=\begin{bmatrix}0&50&10&\infty&45&\infty\\\infty&0&15&\infty&5&\infty\\20&70&0&15&65&\infty\\\infty&20&35&0&25&\infty\\\infty&\infty&\infty&30&0&\infty\\\infty&\infty&\infty&3&\infty&0\end{bmatrix}\quad nextvex_2=\begin{bmatrix}0&1&2&-1&4&-1\\-1&1&2&-1&4&-1\\0&0&2&3&0&-1\\-1&1&1&3&1&-1\\-1&-1&-1&3&4&-1\\-1&-1&-1&3&-1&5\end{bmatrix}$$

考虑经过顶点v_2的路径，

$A_3[i][j]=\min\{A_2[i][j], A_2[i][2]+A_2[2][j]\}\quad 0\leqslant i\leqslant n-1,\ 0\leqslant j\leqslant n-1$

$$A_3=\begin{bmatrix}0&50&10&25&45&\infty\\35&0&15&30&5&\infty\\20&70&0&15&65&\infty\\55&20&35&0&25&\infty\\\infty&\infty&\infty&30&0&\infty\\\infty&\infty&\infty&3&\infty&0\end{bmatrix}\quad nextvex_3=\begin{bmatrix}0&1&2&2&4&-1\\2&1&2&2&4&-1\\0&0&2&3&0&-1\\1&1&1&3&1&-1\\-1&-1&-1&3&4&-1\\-1&-1&-1&3&-1&5\end{bmatrix}$$

考虑经过顶点v_3的路径，

$A_4[i][j]=\min\{A_3[i][j], A_3[i][3]+A_3[3][j]\}\quad 0\leqslant i\leqslant n-1,\ 0\leqslant j\leqslant n-1$

$$A_4=\begin{bmatrix}0&45&10&25&45&\infty\\35&0&15&30&5&\infty\\20&35&0&15&40&\infty\\55&20&35&0&25&\infty\\85&50&65&30&0&\infty\\58&23&38&3&28&0\end{bmatrix}\quad nextvex_4=\begin{bmatrix}0&2&2&2&4&-1\\2&1&2&2&4&-1\\0&3&2&3&3&-1\\1&1&1&3&1&-1\\3&3&3&3&4&-1\\3&3&3&3&3&5\end{bmatrix}$$

$$A_4=\begin{bmatrix}0&45&10&25&45&\infty\\35&0&15&30&5&\infty\\20&35&0&15&40&\infty\\55&20&35&0&25&\infty\\85&50&65&30&0&\infty\\58&23&38&3&28&0\end{bmatrix}\quad nextvex_4=\begin{bmatrix}0&2&2&2&4&-1\\2&1&2&2&4&-1\\0&3&2&3&3&-1\\1&1&1&3&1&-1\\3&3&3&3&4&-1\\3&3&3&3&3&5\end{bmatrix}$$

A_5、A_6与A_4相同；$nextvex_5$、$nextvex_6$与$nextvex_4$相同

找出最短路径

例如想知道v_0到v_1的最短路径：

由 A[0][1] 可知 v_0 到 v_1 的最短路径长度为 45

要追溯具体路径，由 nextvex[0][1] = 2 可知顶点 v_0 的下一顶点为v_2，由 nextvex[2][1] = 3 可知 v_2 的下一顶点为 v_3，由 nextvex[3][1] = 1 可知 v_3 的下一顶点为 v_1。因此从 v_0 到 v_1 的最短路径为 $v_0\rightarrow v_2\rightarrow v_3\rightarrow v_1$

Floyd算法从 A_0 = arcs（图的邻接矩阵）开始。递推生成一系列的矩阵A_1，A_2，…，A_n。后一矩阵可能与前一矩阵不同。

问题：是否需要用一个新的两维数组存放下一矩阵？

假设已经算出了 A_k 存在矩阵 A 里，现考虑 A_{k+1} 的计算。公式：$A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+A_k[k][j]\}$。

注意：所有 $A_{k+1}[i][j]$ 或者就是 $A_k[i][j]$（如果它小），或者是由下面的一列和一行元素算出来的：

$A_k[0][k]$，$A_k[1][k]$，...，$A_k[n-1][k]$

$A_k[k][0]$，$A_k[k][1]$，...，$A_k[k][n-1]$

如果计算 A_{k+1} 时可能修改矩阵第 k 行或第 k 列的元素，后来取元素 [i][k] 或 [k][j] 时得到的不是 A_k 的元素，那就必须保留原来的 A_k，需要另外用一个二维数组。

实际上 A_{k+1} 计算中不会修改矩阵第 k 行或者第 k 列的元素：

$A_{k+1}[i][k] = \min\{A_k[i][k], A_k[i][k]+A_k[k][k]\}$

$A_{k+1}[k][j] = \min\{A_k[k][j], A_k[k][k]+A_k[k][j]\}$

已知 A_k 对角线元素值都为0，$A_k[m][m] = 0$ 对所有m，所以

$A_{k+1}[i][k] = A_k[i][k]$　　$A_{k+1}[k][j] = A_k[k][j]$

因此可以用一个 A 实现所有 A_k，矩阵的递推计算可以通过直接在 A 里更新元素的方式实现。

计算 $A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+A_k[k][j]\}$ 中赋值可以用 A[i][j] = A[i][k] + A[k][j] 实现。计算所有顶点对之间最短路径长度，只需要一个矩阵。

算法中的 ShortPath 包含两个n×n矩阵成员：a 记录已知最短路径长度，nextvex 记录已知最短路径上的下一顶点。


```
void Floyd(GraphMatrix * pgraph, ShortPath * path) {
   int i, j, k, n = pgraph->n;
   for (i = 0; i < n; ++ i)
      for (j = 0; j < n; ++ j) {
         if (pgraph->arcs[i][j] != MAX) path->nextvex[i][j] = j;
         else path->nextvex[i][j] = -1;
         path->a[i][j] = pgraph->arcs[i][j]; /* 复制邻接矩阵 */
      }
   for (k = 0; k < n; ++ k)
      for (i = 0; i < n; ++ i)
         for (j = 0; j < n; ++ j) {
            if ( path->a[i][k] == MAX || path->a[k][j] == MAX )
               continue;
            if ( path->a[i][j] > path->a[i][k]+ path->a[k][j] ) {
               path->a[i][j] = path->a[i][k] + path->a[k][j];
               path->nextvex[i][j] = path->nextvex[i][k];
}       } }
```


时间复杂度：

Floyd算法初始化部分是一个循环，其外层循环共执行 n 次，内层循环也执行 n 次，初始化部分的时间复杂度为 $O(n^2)$。

迭代生成矩阵 A 和路径 nextvex 的部分为三重循环， Floyd 算法的时间复杂度为 $O(|V|^3)$。

令 n = |V|，时间复杂度为 $O(n^3)$。

9.6 拓扑排序

AOV网　　拓扑排序

9.6.1 AOV网

AOV网：有向图中的顶点表示“工程”中的活动，边表示有关活动之间的先后关系，这样的有向图称为顶点活动网(Activity On Vertex network，简称 AOV 网)

AOV网中的弧常用于表示活动之间存在的制约关系。问题是需要根据这种制约关系做出有关活动的安排。

AOV网络被用于各种工程计划等。

在一些问题里，AOV网上的顶点或者边还可能带有权值。

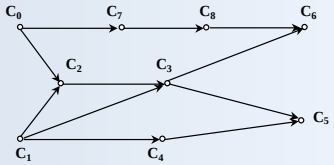
AOV网的一个典型实例是大学课程的先修关系。当学生想选某一门课程时，需要考察是否已经修过先修课程。

例：计算机专业学生须完成规定的基础课和专业课才能毕业，这时的工程就是完成给定的学习计划，而活动就是学习课程，这些课程的名称与代号如表所示

课程代号	课程名称	先修课程
C_0	高等数学	
C_1	程序设计语言	
C_2	离散数学	C_0,C_1
C_3	数据结构	C_1,C_2
C_4	算法语言	C_1
C_5	编译技术	C_3,C_4
C_6	操作系统	C_2,C_8
C_7	普通物理	C_0
C_8	计算机原理	C_7

可以用AOV网中的顶点表示课程，用有向边表示课程之间的先修关系，如果课程 C_i 是课程 C_j 的先修课，则在相应的 AOV网中必定存在一条有向边$<C_i, C_j>$。

前面表中各课程的AOV网如下图所示



9.6.2 拓扑排序

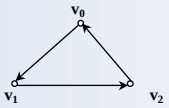
定义：对一个 AOV 网，考虑其中所有顶点能否排成一个线性序列 $v_{i1}, v_{i2}, \ldots, v_{in}$，满足：如果在该 AOV 网中从顶点 v_i到顶点 v_j 存在一条路径，那么在序列$v_{i1}, v_{i2}, \ldots, v_{in}$中，顶点 v_i 一定排在顶点 v_j 之前。具有这种性质的线性序列称为该 AOV网的一个拓扑序列，构造拓扑序列的操作称为拓扑排序。

1、一个AOV网的拓扑序列不一定唯一。

2、假设某个AOV网代表一个工程，如果有条件限制各个动作只能串行进行，则那么该 AOV网 的一个拓扑序列就是整个工程得以顺利完成的一种可行方案。

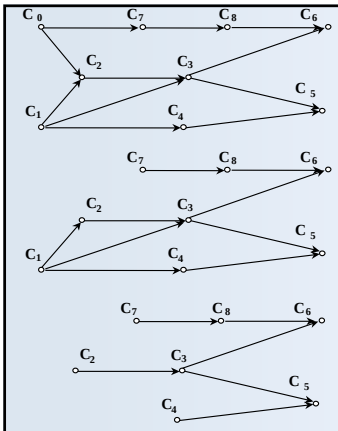
3、有回路的 AOV 网不能完成拓扑排序。（回路意味着某些活动的开始以其完成为先决条件，这种现象称为死锁)

下图的AOV网就没有拓扑序列：



4、任何无回路 AOV 网都可以做出拓扑序列，方法：

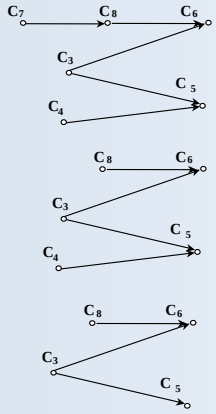
- 从 AOV 网中选择一个入度为 0 的顶点作为序列的下一顶点
- 在 AOV 网中删除此顶点及其所有的出边
- 反复执行以上两步，直到所有顶点都已经输出为止，此时拓扑排序完成
- 如果还剩下入度不为0的顶点，就说明该 AOV 网存在回路，拓扑排序无法完成



C_0

$C_0, C_1,$

$C_0, C_1, C_2,$



$C_0, C_1, C_2, C_7,$

$C_0, C_1, C_2, C_7, C_4,$

出现多个入度为0的顶点时，就出现了一次任意选择

$C_0, C_1, C_2, C_7, C_4, C_3,$

$C_0, C_1, C_2, C_7, C_4, C_3, C_5, C_8, C_6$

图用邻接表表示：

```
typedef struct EdgeNode {   /* 边表中的结点结构 */
   int endvex;              /* 相邻顶点字段 */
   EdgeNode* next;  /* 链字段 */
} *EdgeList;
typedef struct {            /* 顶点表中的结点 */
   /*VexType vertex;*/      /* 顶点信息 */
   EdgeList elist;          /* 边表头指针 */
} VexNode;
typedef struct{             /* 图顶点表 */
   int n;                   /* 图的顶点个数 */
   VexNode vexs[MAXVEX];
} GraphList;
typedef struct {            /* 记录拓扑排序序列 */
   int n;
   int vexsno[MAXVEX];      /* 顶点在顶点表中的下标值 */
} Topo;
```


整数数组 inDegree 记录各顶点入度

还在其中实现一个栈，保存入度为 0 的顶点，简化后续操作。栈底元素存 –1，栈顶位置用变量 top 保存。

findInDegree 统计顶点入度：

```
void findInDegree(GraphList* g, int *inDegree){
   int i;
   EdgeList p;
   for (i = 0; i < g->n; ++ i) inDegree[i] = 0;
   for (i = 0; i < g->n; ++ i)
      for (p = g->vexs[i].elist; p != NULL; p = p->next)
         ++inDegree[p->endvex]; /* 邻接终点的入度加一 */
}
```

```
int topoSort(GraphList * paov, Topo * ptopo) {
   int i, nd = 0, top = -1;
   int indegree[MAXVEX];
   findInDegree(paov, indegree);  /* 求图所有顶点的入度 */
   for (i = 0; i < paov->n; ++ i)  /* 入度0的顶点入栈 */
      if (indegree[i] == 0) {
         indegree[i] = top; top = i;
      }
   /* 拓扑排序，将顶点序列记入ptopo，返回记入的顶点数 */
   nd = topoList (paov, ptopo, indegree, top);
   ptopo->n = nd;
   if (nd < paov->n) { /* AOV网中存在回路 */
      printf ("The AOV has no topo-sort sequence.\n");
      return FALSE;
   }
   return TRUE;
}
```



```
int topoList(GraphList *paov, Topo *ptopo,
          int indegree[], int top) {
   EdgeList p;
   int i, k, nd = 0;
   while ( top != -1) {   /* 有入度0的顶点 */
      i = top;
      top = indegree[top];  /* top 顶点退栈 */
      ptopo->vexsno[nd] = i; ++ nd;  /* 将 i 记入拓扑序列 */
      for (p = paov->vexs[i].elist; p != NULL; p = p->next) {
         k = p->endvex;
         if (--indegree[k] == 0) {
            indegree[k] = top; top = k; /* 入度0的顶点入栈 */
         }
      }
   }
   return nd;
}
```


算法复杂度:

设AOV网有n个顶点，e条边。

算法初置每个顶点的入度，其中需要检查每条边一次，而后检查入度为零的顶点。总花费的时间为 O(e+n)。

在拓扑排序的主要工作部分，每个顶点至多连接到 top 链里一次，而且每个边结点都被检查一次（且至多检查一次），时间代价也为O(n+e)。

因此，拓扑排序算法的时间复杂度为O(n+e)

也可以用邻接矩阵实现这一算法，这时:

设置入度需要遍历矩阵，初始准备的时间为 $O(n^2)$。拓扑排序中的工作时间也是 $O(n^2)$ 。

9.7 关键路径

AOE网

关键路径

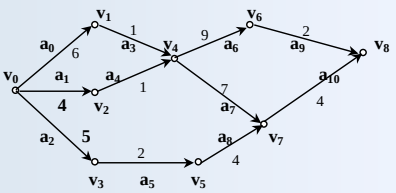
9.7.1 AOE网

AOE网：如果带权有向图的顶点表示事件，有向边表示活动，边上的权值表示活动的持续时间，这样的带权有向图称为AOE网(Activity On Edge network)。

顶点表示的事件，就是它的入边所表示的活动都已完成，它的出边所表示的活动可以开始的状态。

例：下面 AOE 网包括11项活动，9个事件，事件 v_0 表示整个工程可以开始的状态；事件 v_4 表示活动 a_3、a_4 已经完成，活动 a_6、a_7 可以开始的状态，事件 v_8 表示整个工程结束。

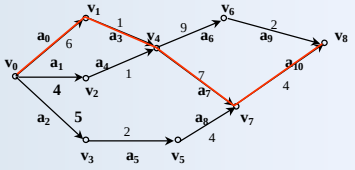
若权表示的时间单位是“天”，活动a_0需要6天完成，活动a_1需要4天完成等等。整个工程开始，活动 a_0、a_1、a_2 就可以并行进行，而活动a_3、a_4、a_5分别在事件 v_1、v_2、v_3发生之后才能进行，当活动a_9、a_{10}完成时，整个工程完成。



9.7.2 关键路径

AOE网中有些活动可以并行进行，所以完成整个工程的最短时间就是从开始顶点到完成顶点的最长路径长度（路径长度为路径上各边的权值之和）。

从开始顶点到完成顶点的最长路径称为关键路径。



v_0, v_1, v_4, v_7, v_8是一条关键路径，长度为18，也就是说，完成整个工程至少需要 18 天

问题：如何确定AOE网的关键路径

总假定 v_0 是开始事件，v_{n-1} 是结束事件。

首先定义几组变量（$w(<v_i,v_j>)$为边$<v_i,v_j>$的权）：

1、事件 v_j 的最早可能发生时间ee(j):

$ee(0) = 0$

$ee(j) = \max\{ ee(i) + w(<v_i,v_j>) \mid <v_i,v_j> \in T,\ 1 \leqslant j \leqslant n-1 \}$

T 是所有以 v_j 为终点的入边的集合

2、事件 v_i 的最迟允许发生时间le(i):

$le(n-1) = ee(n-1)$

$le(i) = \min\{ le(j) - w(<v_i, v_j>) \mid <v_i,v_j> \in S,\ 0 \leqslant i \leqslant n-2 \}$

S 是所有以v_i为开始顶点的出边的集合

3、活动$a_k=<v_i,v_j>$的最早可能开始时间e(k):

$e(k) = ee(i)$

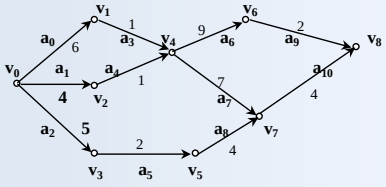
4、活动 $a_k=<v_i,v_j>$的最迟允许开始时间l(k)（在保证整个工程不拖延工期的情况下）：

$l(k) = le(j) - w(<v_i,v_j>)$

所有 $e(k) = l(k)$ 的活动 a_k 称为关键活动。因为它们的推迟开始就会延误整个工程的工期。

$l(k) - e(k)$ 表示完成活动 a_k 的时间余量，这是在不延误工期的前提下，活动a_k可以推迟的时间量。

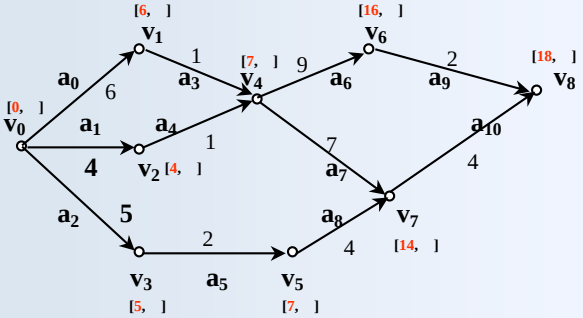
例题：求图中的AOE网的关键路径



按前面公式分别求出事件的最早发生时间及最迟发生时间和活动的最早开始时间及最晚开始时间。

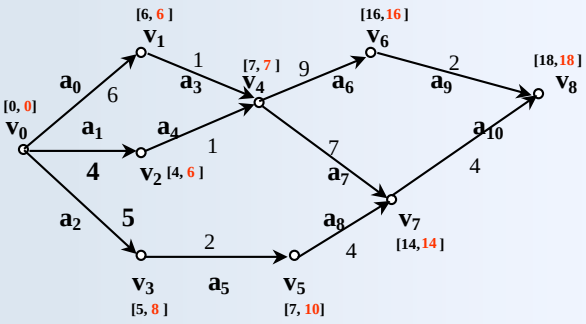
为求事件的最早发生时间，必须已知其所有前驱事件的最早发生时间；求最迟发生时间，必须已知其后继事件的最迟发生时间。因此要先有图中结点的一个拓扑排序，而后按拓扑序列里的事件顺序工作。取拓扑序列：$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

计算事件的最早可能发生时间



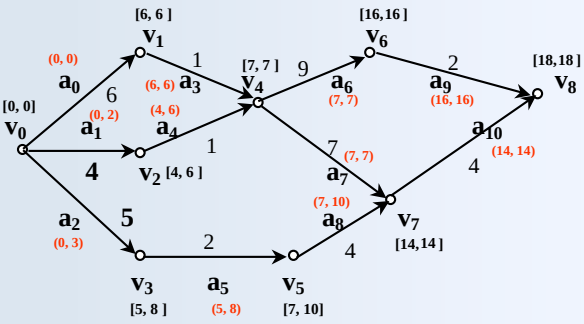
$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

计算事件的最迟允许发生时间



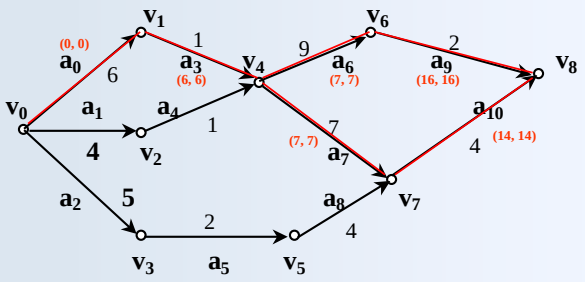
$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

计算活动的最早开始时间和最晚开始时间



$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

关键活动和关键路径



图用邻接表（同拓扑排序），边结点增加 weight 域记录权值

```
typedef struct EdgeNode {  /* 边表中的结点结构 */
   int endvex; AdjType weight;      /* 相邻顶点和权 */
   EdgeNode* next;
} *EdgeList;
typedef struct {                    /* 顶点表中的结点 */
   EdgeList elist;                  /* 边表头指针 */
} VexNode;
typedef struct{                     /* 图顶点表 */
   int n;                           /* 图的顶点个数 */
   VexNode vexs[MAXVEX];
} GraphList;
typedef struct {                    /* 记录拓扑排序序列 */
   int n;
   int vexsno[MAXVEX];    /* 顶点在顶点表中的下标值 */
} Topo;
```


关键路径算法

先做出一个拓扑序列，而后按照这个序列计算出各个数组。

```
int CriticalPath(GraphList * paoe) {
   AdjType ee[MAXVEX], le[MAXVEX], l[MAXEDGE],
           e[MAXEDGE];
   Topo topo;
   if (topoSort(paoe, &topo) == FALSE)  /* 求一个拓扑序列 */
      return FALSE;
   comp_ee(paoe, &topo, ee);          /*计算数组ee*/
   comp_le(paoe, &topo, ee, le);      /*计算数组le*/
   comp_el(paoe, &topo, ee, le, e, l);/*计算数组e,l并输出结果*/
   printf("\n");
   return TRUE;
}
```


计算数组ee

把所有事件的最早发生时间初始化为0，而后逐个顶点去更新其后继顶点（事件）的最早发生时间

```
void comp_ee(GraphList* paoe, Topo* ptopo, AdjType ee[]) {
   int i, j, k;  EdgeList p;
   for (i = 0; i < paoe->n; ++ i) ee[i] = 0;
   for (k = 0; k < paoe->n; ++ k) {
      /* 求事件v_j可能的最早发生时间ee(j) */
      i = ptopo->vexsno[k];
      for ( p = paoe->vexs[i].elist; p; p = p->next) {
         j = p->endvex;
         if (ee[i] + p->weight > ee[j]) ee[j] = ee[i] + p->weight;
      }
   }
}
```


计算数组le

把所有事件的最迟发生事件初始化为最后顶点的最迟发生时间，根据后继事件的最迟时间更新自己的最迟时间。

```
void comp_le(GraphList * paoe, Topo* ptopo, AdjType ee[],
             AdjType le[]) {
   int i, j, k;  EdgeList p;
   /* 求事件v_i允许的最迟发生时间le(i) */
   for (i = 0; i < paoe->n; ++ i) le[i] = ee[paoe->n - 1];
   for (k = paoe->n - 2; k >= 0; -- k) {
      i = ptopo->vexsno[k];
      for (p = paoe->vexs[i].elist; p; p = p->next) {
         j = p->endvex;
         if( le[j] - p->weight < le[i] ) le[i] = le[j] - p->weight;
      }
   }
}
```


计算数组e, l并输出结果

根据始点事件和终点事件的事件，计算出各个活动的最早和最迟开始事件。如果两个时间相等就输出（也可以记录）。

```
void comp_el(GraphList * paoe, Topo* ptopo, AdjType ee[],
             AdjType le[], AdjType e[], AdjType l[]) {
   int i, j, k = 0; EdgeList p;
   /* 求活动a_k的最早开始时间e(k) 及最晚开始时间l(k) */
   for (i = 0; i < paoe->n; ++ i) {
      for (p = paoe->vexs[i].elist; p; ++k, p = p->next) {
         j = p->endvex;
         e[k] = ee[i];  l[k] = le[j] - p->weight;
         if (e[k] == l[k]) printf("<v%2d, v%2d>\n", i, j);
      }
   }
}
```

算法分析

算法复杂度：设AOE网有n个顶点，e条边，在求事件可能的最早发生时间及允许的最迟发生时间，以及活动的最早开始时间和最晚开始时间时，都要对图中所有顶点及每个顶点边表中所有的边结点进行检查，时间花费为O(n+e)。

因此，求关键路径算法的时间复杂度为O(n+e)

空间复杂度：需要保存拓扑序列的数组和4个保存事件的时间和活动的时间的数组，空间复杂度也为 O(n+e)。

小结

图是一种复杂的非线性结构。

本章介绍图的基本概念

两种常用的表示方法（相邻矩阵和邻接表）

图的计算问题：宽度优先和深度优先周游、最小生成树、最短路径、拓扑排序及关键路径等，给出了相应算法

本章重点是:

- 掌握图的存储表示
- 掌握（除 Floyd 算法之外）各种算法的工作过程