第八章 图

线性结构：唯一前驱，唯一后继，线性关系
树形结构：唯一前驱，多个后继，层状关系
图形结构：多对多、任意，网状关系

图应用广泛：网络布设，航线管理，交通管理，语言学等。

图是更复杂的非线性结构。

基本内容：

- 基本概念（逻辑）
- 存储表示
  - 邻接矩阵
  - 邻接表
- 图的运算
  - 图的周游（遍历）

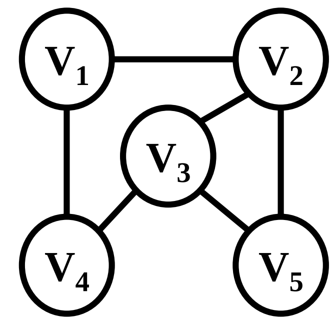
    深度优先、广度优先
  - 最小生成树
  - 最短路径
- 拓扑排序
- 关键路径

8.1 基本概念

1. 图的定义

$G=(V, E)$, V – 顶点集合，E – 边(v_i, v_j)或弧$<v_i, v_j>$集合

2. 弧与边



无向图

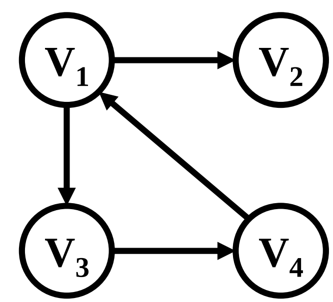
$<v_i, v_j>$为有序对（弧），$<v_i, v_j>$与$<v_j, v_i>$不同

(v_i, v_j)为无序对（边），(v_i, v_j)与(v_j, v_i)相同

3. 有向图与无向图

有向图：顶点+弧； 无向图：顶点+边。

4. 完全图



有向图

任意一对顶点间都有边（弧）相连

无向完全图：n(n-1)/2条边，C_n^2

有向完全图：n(n-1)条弧，$2\times C_n^2$

5. 顶点的度：与顶点相关联的边（弧）数， D(v)

有向图中：入度ID(v)+出度OD(v) = D(v)

顶点数n， 边（弧）数e和顶点度有如下关系：

$$e = \frac{1}{2}\sum_{i=1}^{n} D(v_i)$$ 每条边（弧）涉及两个顶点。

6. 子图

设有图G=(V，E)和G'=(V'，E')，如果V'是V的子集，E'是E的子集，则称G'是G的子图。

7. 路径与简单路径

路径：在图中从顶点v到顶点v'所经过的所有顶点的序列

路径长度：路径上边（弧）数目

简单路径：序列中顶点不重复出现的路径

8. 环（回路）与简单环（回路）

回路或环：第一个顶点和最后一个顶点相同的路径。

简单回路或环：除第一个和最后一个顶点，其余顶点不重复出现的路径。

9. 有根图

有向图中，若存在一顶点v，从该顶点到图中其它顶点都有路径，则称此有向图为有根图，v称为图的根。

10. 连通、连通图、连通分量（无向图）

在无向图中，如果从v到v'存在路径，则称v和v'是连通的。

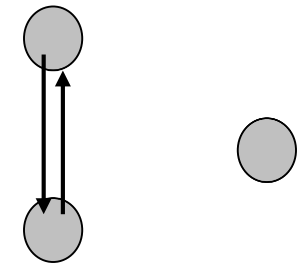
无向图G中如果任意两个顶点v_i、v_j之间都是连通的，则称图G是连通图。

无向图中的极大连通子图称为连通分量（可有多个）。

11. 强连通图、强连通分量[有向图]

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

有向图中的极大强连通子图称为强连通分量。

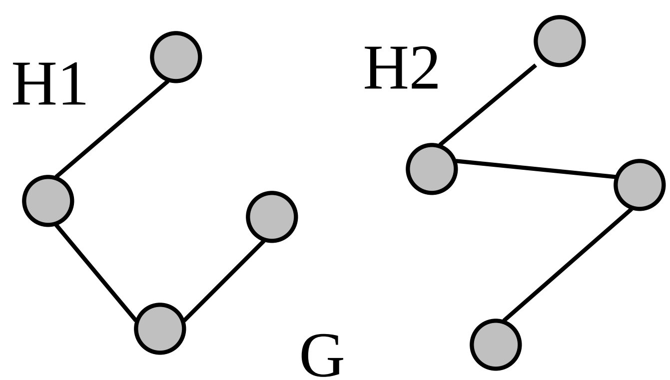


G的两个强连通分量

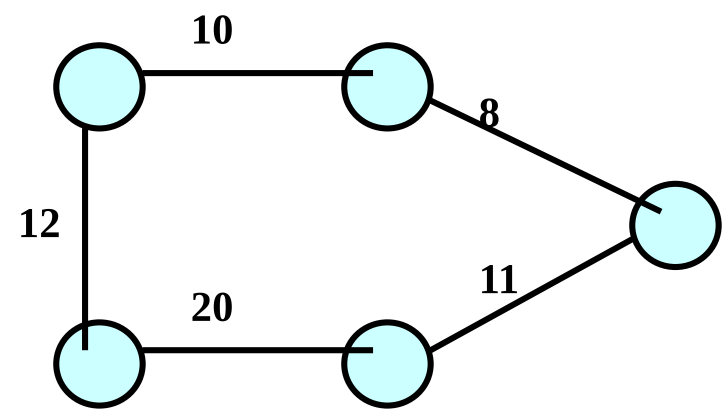
12. 带权图与网络

带权图：图的每条边（弧）都赋上一个权值；

网络：带权的连通图（强连通图）。



G有两个连通分量



网络示例

13. 连通图的生成树

是连通图的一个极小连通子图，它含有图中的全部n个顶点，但只有足以构成一棵树的n-1条边。

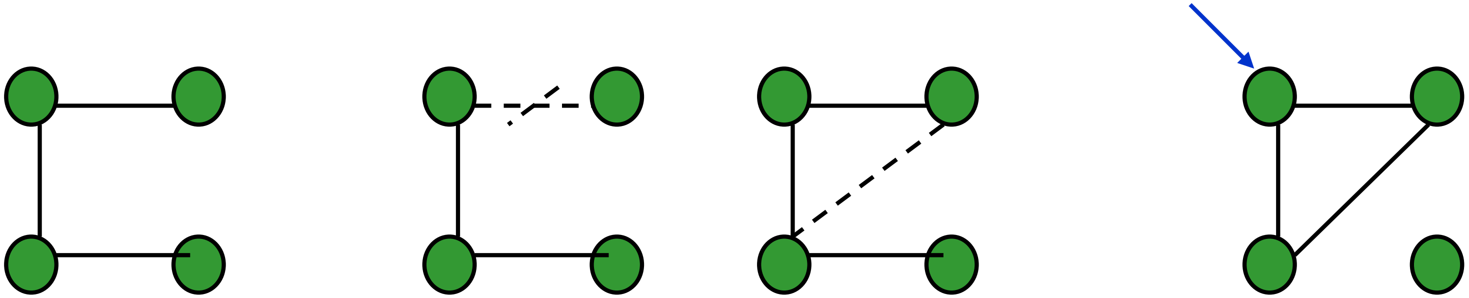
如在一棵生成树中增加一条边，则必定构成环；

如在一棵生成树中去掉一条边，则连通图变为不连通的；

顶点数为n，边数小于n-1的无向图必定是不连通的；

顶点数为n，边数大于n-1的无向图必定存在环；

顶点数为n，边数为n-1的无向图不一定是生成树。



图的基本运算：

- 创建空图；
- 判空；
- 销毁，释放空间；
- 找第一个顶点；
- 找下一个顶点；
- 查找；
- 增加一个新的顶点；
- 删除一个顶点；
- 增加一条边（或弧）；
- 删除一条边（或弧）；
- 判断某两个顶点间是否存在边（或弧）；
- 找与某个顶点相邻的下一个顶点；
- 找与某个顶点相邻，相对于另一个顶点的下一个相邻顶点[下一条边（或弧）]。

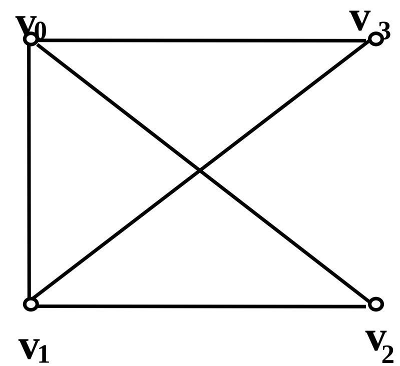
8.2 存储表示

1. 邻接矩阵表示法：

一个顶点信息表 + 一个顶点关系矩阵

顶点信息表： 顺序表vexs[n]存储；

顶点关系矩阵：n阶方阵



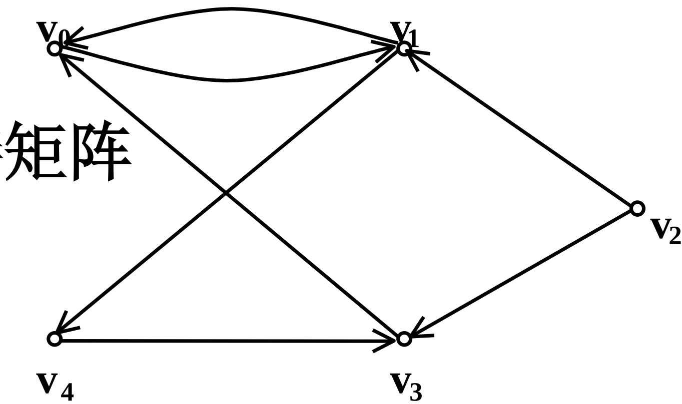
无向图G_1

A[i,j] = 1 若(V_i,V_j)或$<V_i, V_j>$是图的边或弧

0 若(V_i, V_j)或$< V_i, V_j >$不是图的边或弧

无向图G_1的邻接矩阵A_1和有向图G_2的邻接矩阵A_2

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

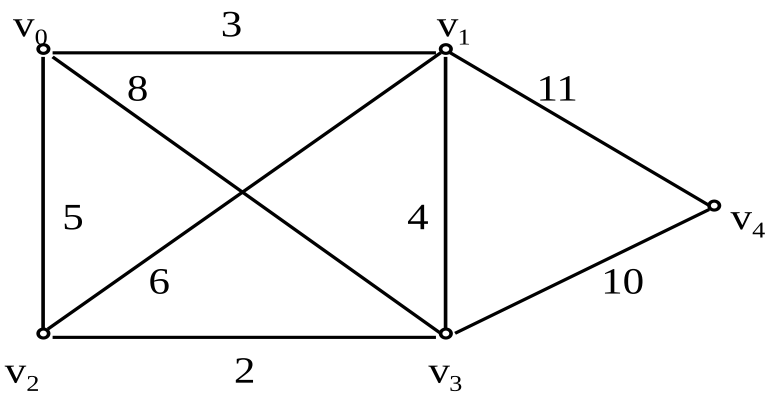


有向图G_2

如果G是网络，w_{ij}是边(v_i,v_j)或$< v_i,v_j>$的权，则其邻接矩阵定义为:

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

下面网络的邻接矩阵如下：



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

图的邻接矩阵存储结构：

```
typedef struct
{
    VexType vexs[MAXVEX];                /* 顶点信息 */
    AdjType arcs[MAXVEX][MAXVEX];  /* 邻接矩阵信息 */
    int   arcCount, vexCount;            /* 图的顶点个数 */
} Graph, *PGraph;
```

邻接矩阵的特点：

- 无向图的邻接矩阵一定是一个对称方阵。
- 无向图的邻接矩阵的第i行(或第i列)非零元素(或非∞元素)个数为第i个顶点的度$D(v_i)$。

- 有向图的邻接矩阵的第i行非零元素(或非∞元素)个数为第i个顶点的出度$OD(v_i)$，第i列非零元素(或非∞元素)个数就是第i个顶点的入度$ID(v_i)$。
- 邻接矩阵表示图，很容易确定图中任意两个顶点之间是否有边相连。

邻接矩阵表示法的优缺点：

优点：各种基本操作都易于实现。

缺点：$e \ll n^2$时，空间浪费严重。某些算法时间效率低。

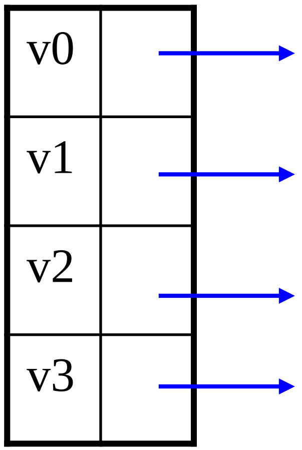
空间代价：$O(n^2)$

2. 邻接表表示法:

邻接表表示法包括两大部分:

- 顺序存储的顶点表(顶点信息+指向边表中第一个结点指针)
- n个链式存储的边表

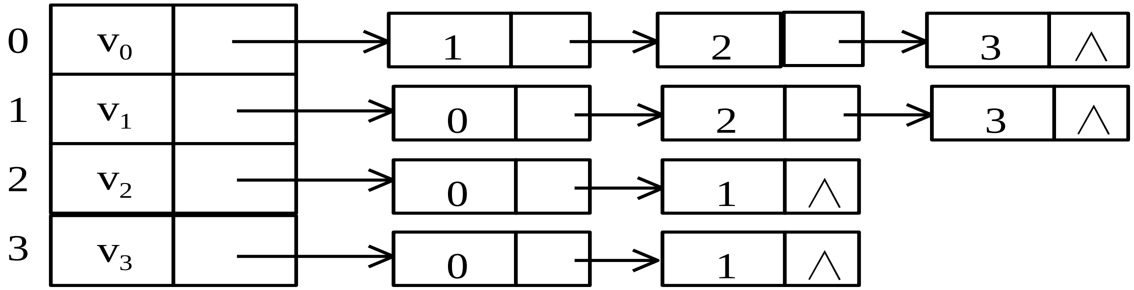
(存放对应顶点相关联的边或弧)

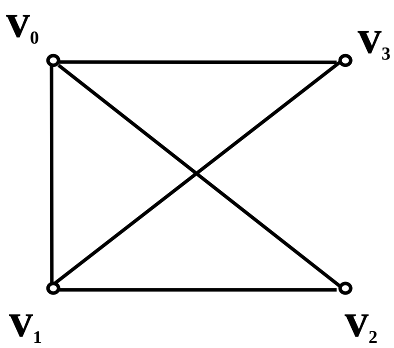


无向图中，边表的结点代表一条边;

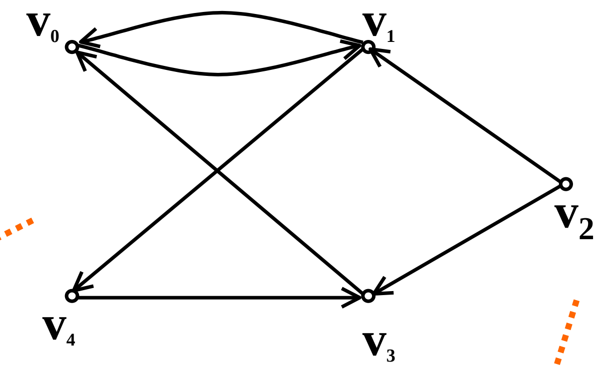
有向图中，边表的结点代表一条由顶点出去的弧

【出边】;

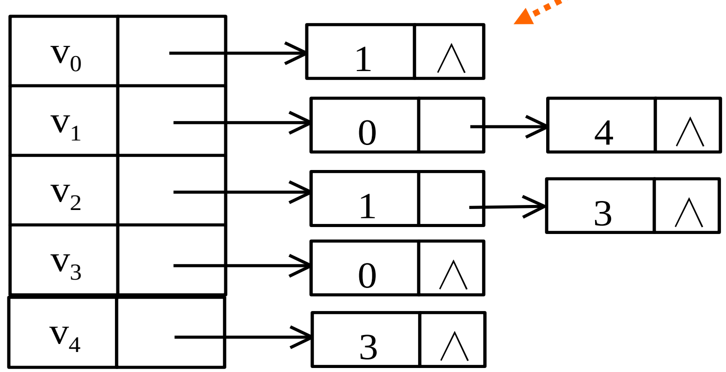


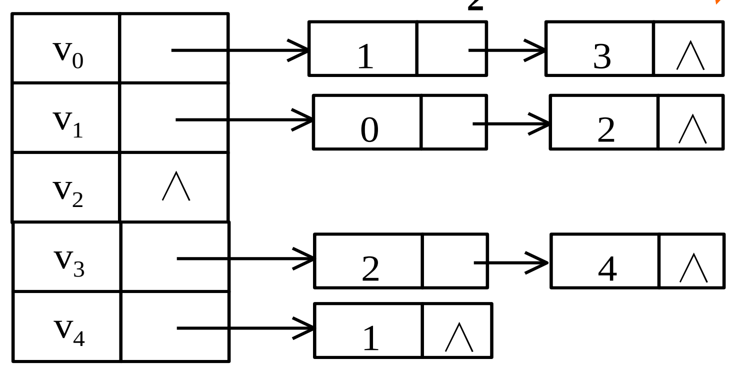


无向图G_1的邻接表



有向图G_2





有向图G_2的邻接表(出边表）

有向图G_2的逆邻接表（入边表）

一些基本操作的实现：

- 求无向图中某个顶点的度：
  该结点所指向的链表中的结点总数。

- 求有向图的出度：
  该结点所指向的链表中的结点总数。
- 求有向图的入度：
  必须搜索整个邻接表才能得到，统计所有顶点的边表中，包含该顶点的结点个数。（改进：增加逆邻接表）

逆邻接表：所有顶点的边表中的边都是以该顶点为终点的边.

邻接表表示法的存储结构：

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

typedef struct
{       VexType vertex;                   /* 顶点信息 */
        PEdgeList edgelist;               /* 边表头指针 */
} VexNode;                                /* 顶点表 */

typedef struct
{       VexNode vexs[MAXVEX];
        int vexNum, edgeNum;              /* 图的顶点个数 */
}GraphList;
```

邻接表的优缺点：

优点：容易找任一结点的第一邻接点和下一个邻接点；

$e<<n^2$时，存储量小。

缺点：判定任意两个结点之间是否有边或弧不方便。

(需要扫描某个顶点的边表）

空间代价：无向图：O(n+2e)，有向图：O(n+e)

3. 其它表示法:

有向图的十字链表

无向图的邻接多重表

有向图的十字链表存储结构：

弧尾 → 弧头

弧结点

tailvex	headvex	hlink	tlink	info

弧尾顶点

弧头顶点

弧头相同的下一条弧

弧信息[加权值？]

弧尾相同的下一条弧

弧头相同的弧在同一链表上
弧尾相同的弧在同一链表上

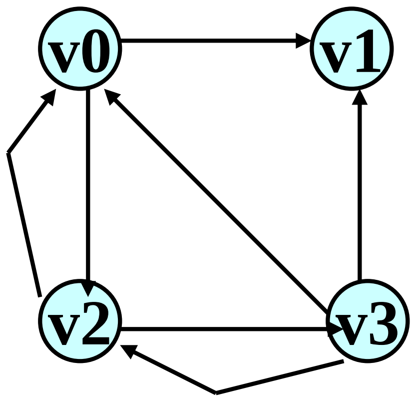
顶点结点

data	firstin	firstout

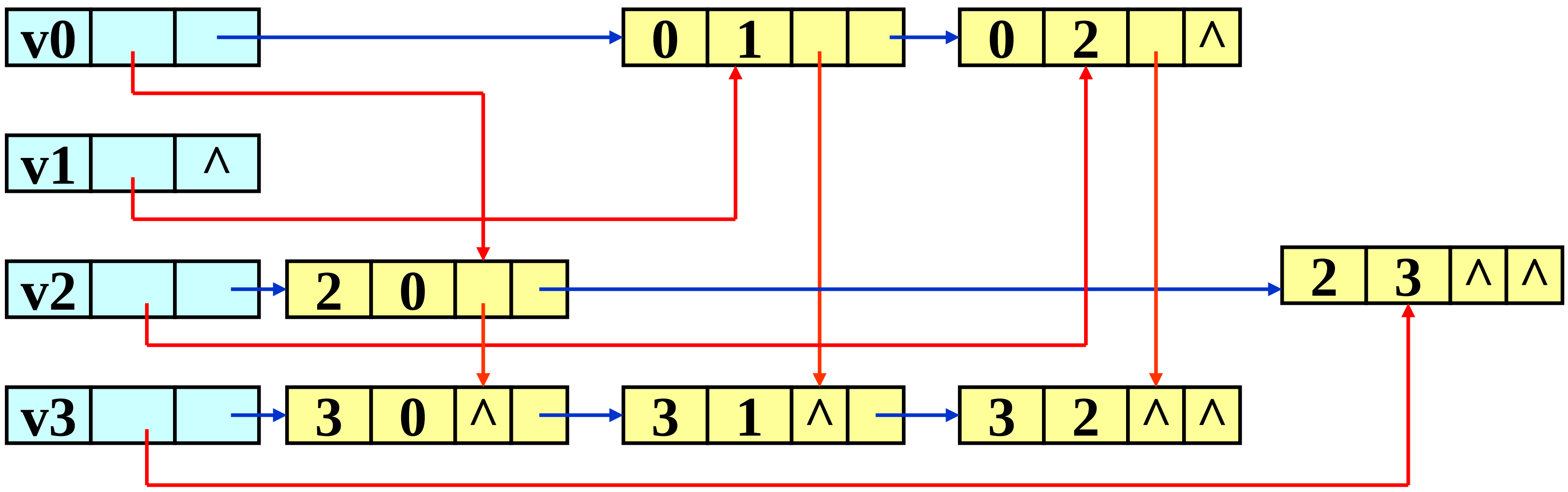
该顶点为弧头的第一个弧结点

顶点数据

该顶点为弧尾的第一个弧结点



有向图及其十字链表存储



优点：方便求得以v为头的弧和以v为尾的弧，
因而方便求入度、出度

无向图的邻接多重表存储结构：

边结点

mark	ivex	ilink	jvex	jlink	info

标志域

边的两个顶点

下一条依附ivex的边

下一条依附jvex的边

边信息[加权值？]

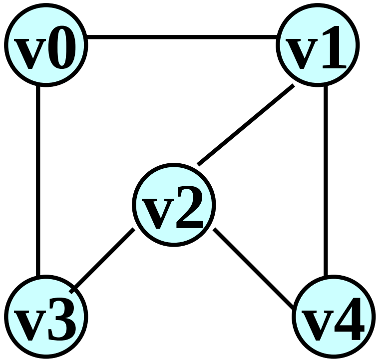
所有依附于同一顶点的边串联在同一链表上
每个边结点同时在两个链表上

顶点结点

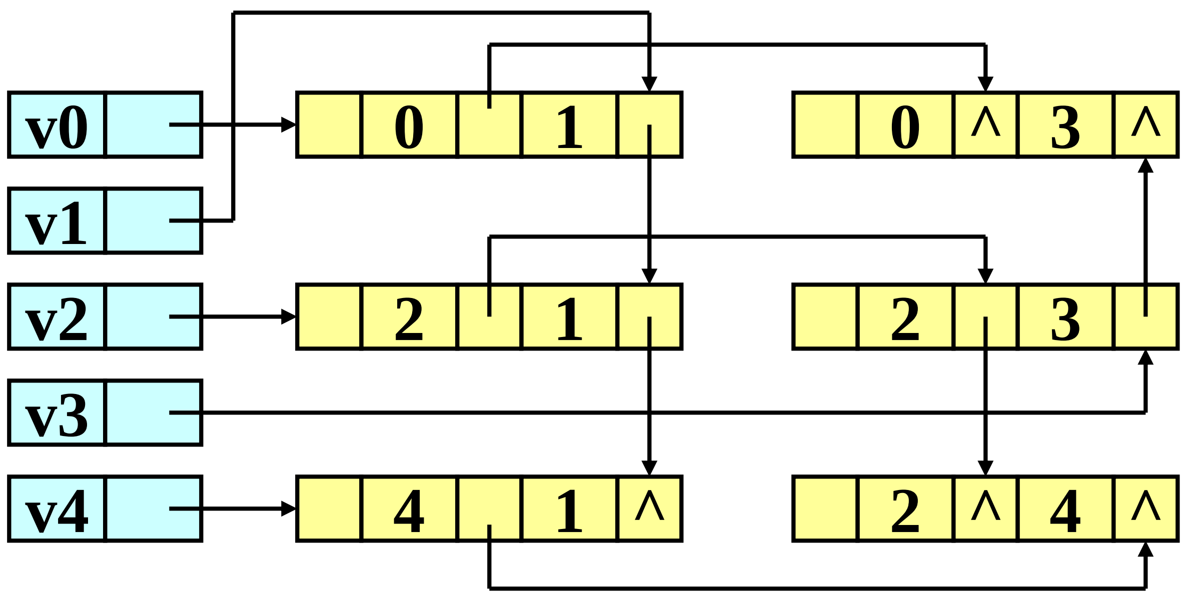
data	firstedge

顶点数据

指向第一条依附于该顶点的边



无向图及其邻接多重表存储



思考：如何计算顶点度?

8.3 图的周游（遍历）

定义：从图中某一顶点出发访遍图中其余结点，且使每一个结点被访问且仅被访问一次。

图的周游算法是求解图的连通性问题、拓扑问题和求关键路径等算法的基础。

图的周游通常有两种方法：

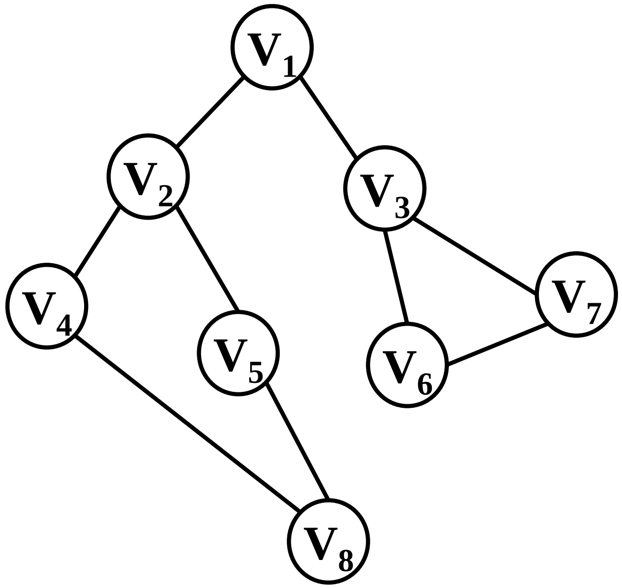
深度优先搜索（**DFS)**，广度优先搜索**(BFS**）

1. 深度优先遍历（DFS):

周游规则:从图的指定顶点v出发，先访问顶点v，并将其标记为已访问过，然后依次从v的未被访问过的邻接顶点w出发进行深度优先搜索，直到图中与v相连的所有顶点都被访问过。如果图中还有未被访问的顶点，则从另一未被访问过的顶点出发重复上述过程，直到图中所有顶点都被访问过为止。

V_1出发深度优先：V_1->V_2->V_4->V_8->V_5->V_3->V_6->V_7

访问过的结点需要特殊标志，避免回路。



邻接矩阵递归算法实现：

```
void dFSInMatrix(Graph * pGraph, int visited[], int i)
{   int  j;
    printf("node: %c\n", pGraph->vexs[i]); /* 访问出发点vi */
    visited[i]=TRUE;
    for(j=0; j<pGraph->n; j++)
    {
        //如果有未被访问的邻接点vj，从vj开始继续深度优先遍历
        if((pGraph->arcs[i][j]==1) && (visited[j]==FALSE) )
            dFSInMatrix(pGraph,visited,j);
    }
}
```

```
void traverDFS(Graph * pGraph)
{   int visited[MAXVEX];                    /* 初始化数组visited */
    for(i=0;i<pGraph->n;i++)  visited[i]=FALSE;
    for(i=0; i<n; i++)
    {
        if(visited[i]==FALSE) dFSInMatrix(pGraph,visited,i);
        /* 对于邻接表表示 */
        // if(visited[i]==FALSE) dFSInList(pGraph,visited,i);
        /* 调用多少次，表示图中有多少个连通分量 */
     }
}
```

效率分析：

空间复杂度：标志数组和栈，O（n）

时间复杂度：为O（n^2）

邻接表递归算法实现：

```
void dFSInList(GraphList * pgraphlist, int visited[], int i)
{  int      j;
   PEdgeNode p;
   printf("node: %c\n", pgraphlist->vexs[i].vertex);
   visited[i]=TRUE;
   p=pgraphlist->vexs[i].edgelist;  /* 取边表中的第一个边结点 */
   while(p!=NULL)
   {
     //如果有未被访问的邻接点，从邻接点继续深度优先遍历
     if(visited[p->endvex]==FALSE)
          dFSInList(pgraphlist,visited,p->endvex);
     p=p->nextedge;                /* 取边表中的下一个边结点 */
   }
}
```

效率分析：空间复杂度：标志数组和栈，O（n）

时间复杂度：O（n+e）

2. 广度优先遍历(BFS):

周游规则:从图的指定顶点v出发，先访问顶点v，接着依次访问v的所有邻接点w_1，w_2，…，w_x，然后，再依次访问与w_1，w_2，…，w_x邻接的所有未被访问过的顶点，以此类推，直到所有已访问顶点的邻接点都被访问过为止。如果图中还有未被访问过的顶点，则从另一未被访问过的顶点出发进行广度优先搜索，直到所有顶点都被访问过为止。

对于广度优先周游，关键在于怎么保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问(队列控制）。

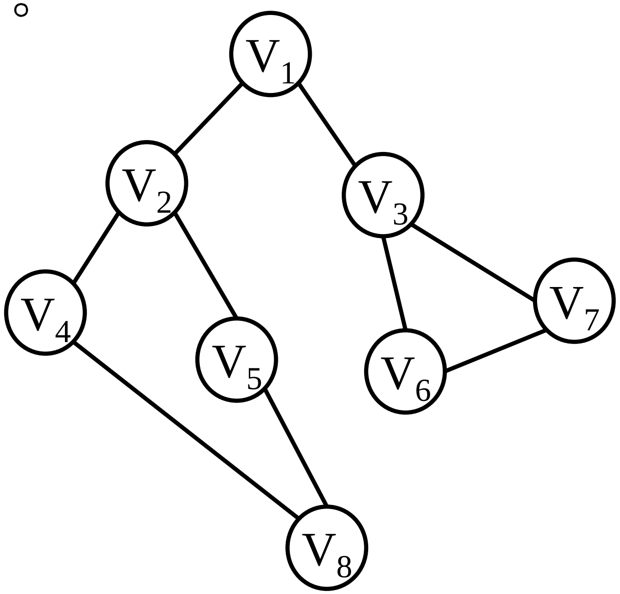
V_1出发广度优先：V_1->V_2->V_3->V_4->V_5->V_6->V_7->V_8

访问过的结点需要特殊标志，避免回路。

邻接矩阵广度优先搜索算法实现：

设立一个队列pq，将访问过的顶点依次进对，

按顶点进对先后顺序访问它们的邻接点。



```
void bFSInMatrix(Graph * pGraph, int visited[], int i)
{   PLinkQueue pq;
    int          j, k;
    pq=creatEmptyQueue_link();             /* 置队列为空 */
    printf("node:%c\n", pGraph->vexs[i]);
    visited[i]=TRUE;
    enQueue_link(pq,i);                    /* 将顶点序号进队 */
    while( !isEmptyQueue_link(pq) )        /* 队列非空时执行 */
    {   k=deQueue_link(pq);                /* 队头顶点出队 */
        for(j=0; j<pGraph->n; j++)
          if( (pGraph->arcs[k][j]==1) && (!visited[j]) )
          /*访问相邻接的未被访问过的顶点 */
          {  printf("node:%c\n", pGraph->vexs[j]);
             visited[j]=TRUE;
             enQueue_link(pq,j);           /* 新访问的顶点入队 */
          }
    }
}
```

队列定义：

```
typedef int DataType;
struct Node        /* 队列结点结构 */
{   DataType  vexs;
    struct Node *next;
};
struct LinkQueue/* 链队列结构 */
{   struct Node *f, *r;
};
typedef struct LinkQueue *PlinkQueue;
void traverBFS(Graph *pGraph)
{   int    visited[MAXVEX];
    int    i,n;
    n=pGraph->n;
    for(i=0;i<n;i++)     visited[i]=FALSE;
    for(i=0; i<n; i++)
        if(visited[i]==FALSE)  bFSInMatrix(pGraph,visited,i);
                                 /* bFSInList(pGraph,visited,i); */
}
```

效率分析：

空间复杂度：标志数组和队列，O(n)

时间复杂度：对于邻接矩阵为$O(n^2)$

对于邻接表为O(n+e)

```
void bFSInList(GraphList *pgraphlist, int visited[], int i)
{   PLinkQueue pq;    PEdgeNode p;    int  j;
    pq=creatEmptyQueue_link();                  /* 置队列为空 */
    printf(“node:%c\n”,pgraphlist->vexs[i].vertex);
    visited[i]=TRUE;    enQueue_link(pq,i); /* 将顶点序号进队 */
    while  (!isEmptyQueue_link(pq) )          /* 队列非空时执行 */
    {   j=deQueue_link(pq);                   /* 队头顶点出队 */
        p=pgraphlist->vexs[j].edgelist;
        while( p!=NULL)
        {   if (!visited[p->endvex]) /*访问相邻接未被访问过的顶点 */
            {  printf(“node:%c\n”,pgraphlist->vexs[p->endvex].vertex);
               visited[p->endvex]=TRUE;
               enQueue_link(pq,p->endvex);    /* 新访问的顶点入队 */
            }
            p=p->nextedge;
        }
    }
}
```

邻接表广度优先搜索算法实现

8.4 最小生成树

1. 概述:

对于连通的无向图和强连通的有向图可以从任何一点出发遍历，访问图中所有结点；遍历得到的边（弧）加上顶点构成了图的一个连通子图，该连通子图成为一棵**生成树**。

对于n个顶点的连通图，生成树中必定包含n个顶点和n-1条边；如果加入任何一条边，必定构成回路；如果删除一条边，必定成为非连通图。

由于遍历方法不同（DFS、BFS)，起始点也可能不同，因此相同的连通图有不同的生成树。

DFS生成树：从连通图的任一顶点出发，进行深度优先周游，记录周游中访问的所有顶点及经过的边，便得到深度优先生成树。

BFS生成树：从连通图的任一顶点出发，进行广度优先周游，记录周游中访问的所有顶点及经过的边，便得到广度优先生成树。

对于网络，生成树的边带有加权值。

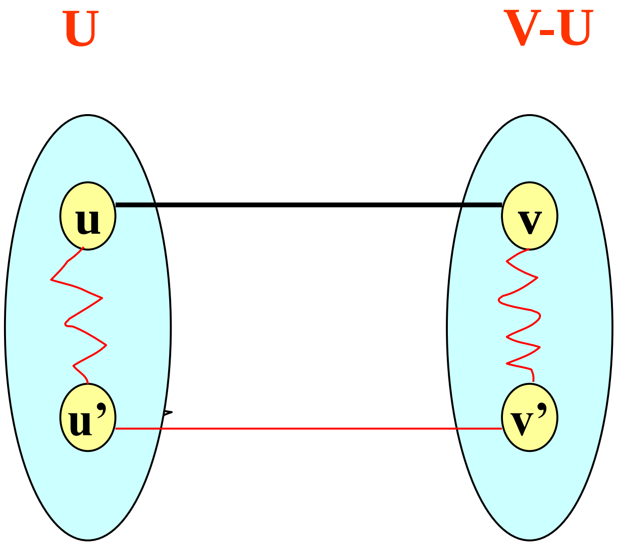
生成树的权：生成树中各边权值和。

最小生成树：在网络中，具有最小权值的生成树。

最小生成树的应用非常广泛，例如：城市间通讯线路的布设，n个城市最多有n(n-1)/2条线路连接，但根据架设通讯线路的代价需要，可以在n(n-1)/2条可以选择的线路中选择连接n个城市并且代价最小的n-1条线路，组成这n座城市之间的通讯连接。

最小生成树的生成方法：基于MST性质选择n-1条边。

MST性质：设G=(V，E)是一个网络，U是顶点集合V的一个真子集。如果边(u,v)的顶点u∈U，v∈V-U，且边(u,v)是图G中所有的一个端点在U里，另一端点在V-U里的边中权值最小的边，则一定存在G的一棵最小生成树包括边(u,v)。



教材p295页的反证。

（会得到一棵“权值和”更小的最小生成树）

最小生成树的构造利用了MST性质，一条边一条边地加入。

主要有两种算法：

- **Prim算法**
- **Kruskal算法**

如果(u,v)不属于任何MST，u和v是连通的：（u…u’, v’…v)的值必定大于(u, v),
与最小生成树的定义矛盾

2. Prim算法

G=(V, E)具有n个顶点的网络，T=(U, TE)为G的最小生成树。
初始状态：U=NULL, TE = NULL.

最小生成树T构造的基本过程：

（1）从集合V中任取一顶点v_0放入集合U中，这时U={v_0}, TE=NULL；

（2）在所有一个顶点在集合U里，另一个顶点在集合V-U里的边中，找出权值最小的边(u,v)(u∈U，v∈V-U)，将边加入TE，并将顶点v加入集合U；

（3）重复过程（2）操作，直到U=V为止。

顶点x到集合的边：顶点x到集合中所有顶点的最小边。

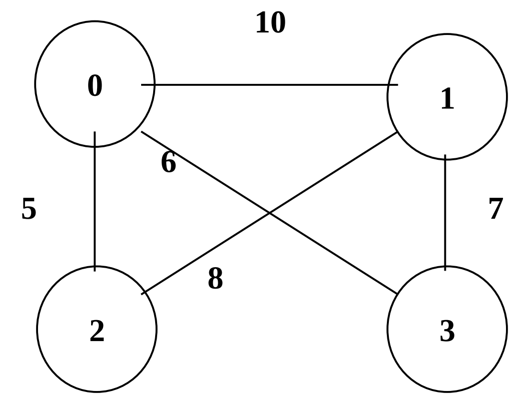
初始时， $U=\{v_0\}$

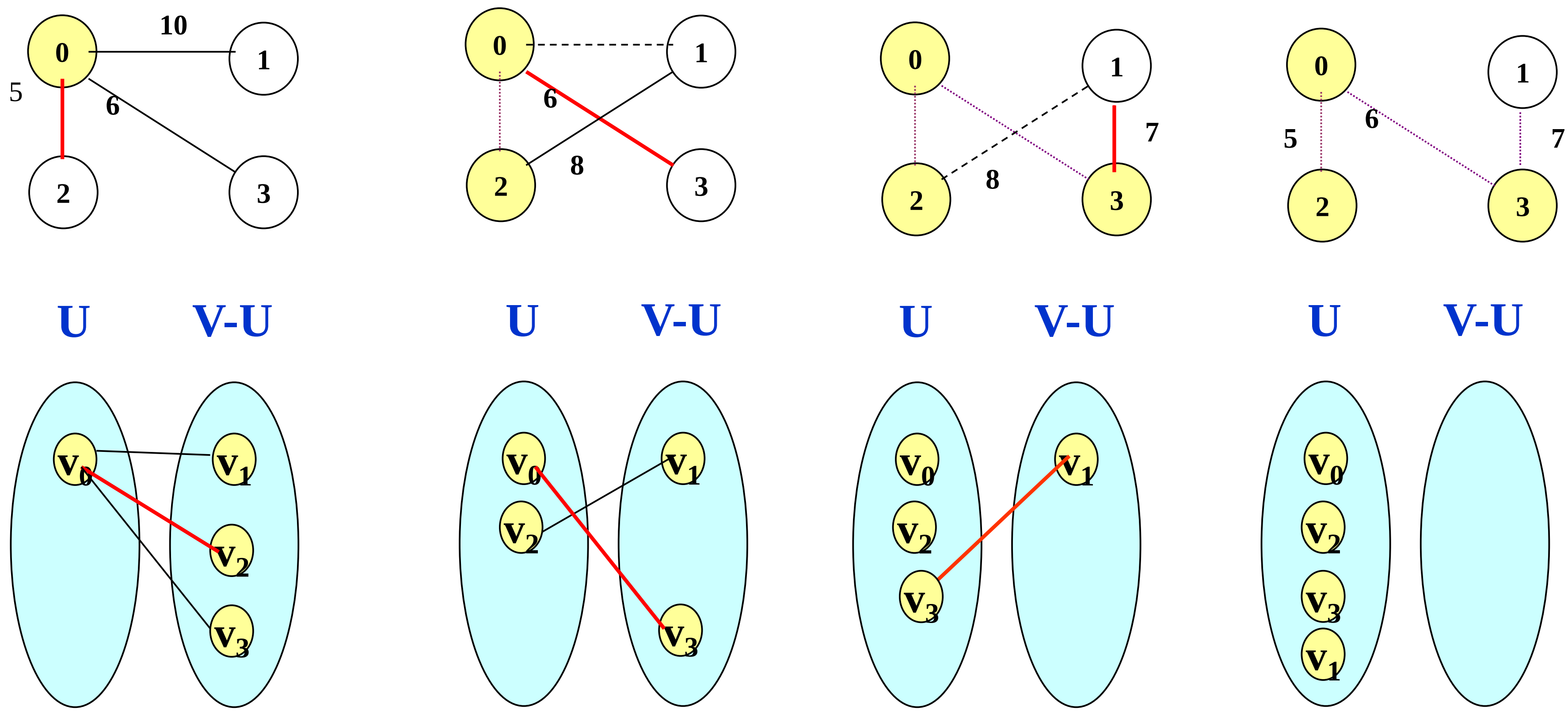
（1）统计“V-U”集合中每个顶点到“U”集合的边。

找出权值最小的边(u，v)(u∈U， v∈V-U)，

将边加入TE，并将顶点v加入集合U;

（2）重复上述操作直到U=V为止(即：V-U为空)。





构造方法（假定采用邻接矩阵表示图）

边的表示如下：

```
typedef struct
{    int start_vex, stop_vex;  //边的起点和终点（顶点表中下标）
     AdjType  weight;          //边的加权值
} Edge;
Edge mst[n-1];    //算法结束时，存放n-1条边
```

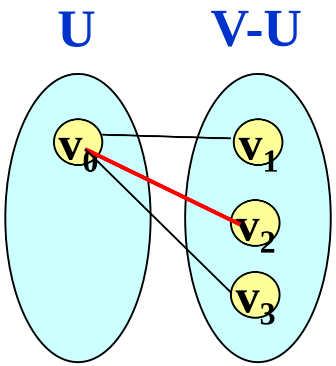
中间第i条边选择前：

mst[0]~mst[i-2]存的是已经选好的i-1条边，

mst[i-1]~mst[n-2]存的是当前V-U集合（包含了n-i个顶点）每个顶点到集合U（包含i个顶点）的边【最小边】。

o	o	o	o	n	n	n	n
0	1	2	i-2	i-1	i	……	n-2

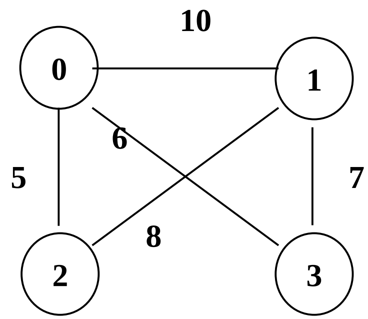
（1）初始时，U中只有顶点v_0，mst中存放v_0到其它n-1个顶点的边。如果(v_0,v_j)不存在，用∞表示。

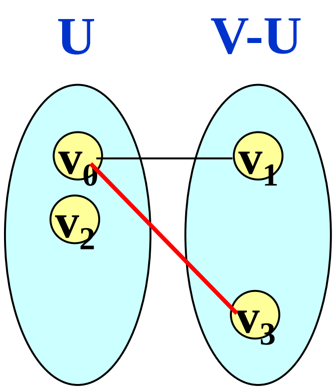


下面依次求n-1条边：

（2）当前mst中存放的都是起点在U中，终点在V-U中的边。因此在mst[0]到mst[n-2]中选择权值最小的边mst[min]加入最小生成树。

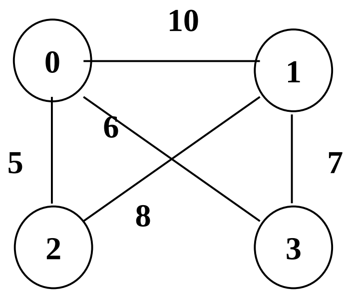
对于mst[min]有mst[min].start_vex = v_0，mst[min].stop_vex = v_x，将v_x加入最小生成树的新顶点，并将mst[min]与mst[0]互换。此时，mst[0]中存放一条最小生成树的边。

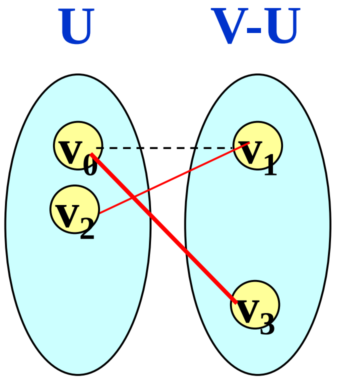




（3）调整mst[1]到mst[n-2]

如果集合V-U中顶点v_y到顶点v_x的边长度（权）比原来v_y到集合U中顶点的长度小，则将原来的边调整为（v_x, v_y）；否则，不需要调整。

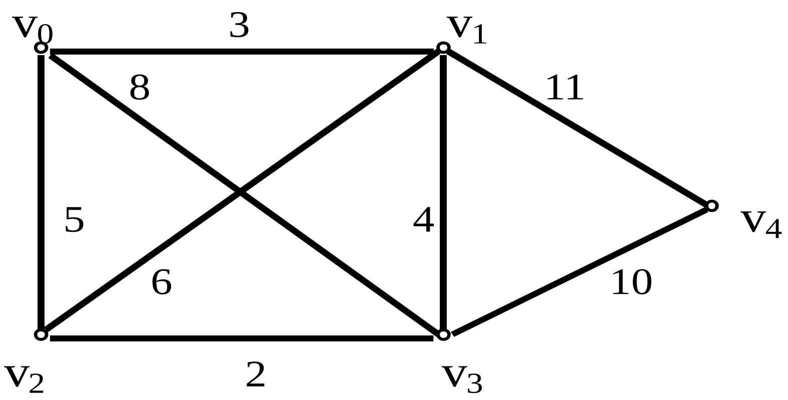




（4）从mst[1]到mst[n-2]重复（2）、（3）操作。

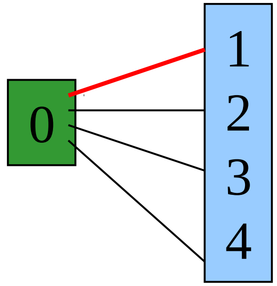
每次在一个顶点在U中，另一个顶点在V-U中的所有边中，选择一条权值最小的边，并将这条边对应的顶点加入到最小生成树中，并调整未加入最小生成树中那些边，直到n个顶点都在生成树中为止。

构造示例：



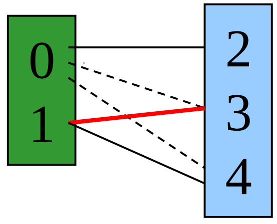
$$A_3 = \begin{bmatrix} \infty & 3 & 5 & 8 & \infty \\ 3 & \infty & 6 & 4 & 11 \\ 5 & 6 & \infty & 2 & \infty \\ 8 & 4 & 2 & \infty & 10 \\ \infty & 11 & \infty & 10 & \infty \end{bmatrix}$$

1) n=5,初始时只有v_0在最小生成树中。
   mst[4] = {(0,1,3),(0,2,5),(0,3,8),(0,4,∞)}



2) 在mst[0]到mst[3]中找权值最小的边mst[0]，将顶点v_1及边(v_0,v_1)加入到最小生成树中。

3) 按照新加入顶点v_1调整mst[1]到mst[3]
   (v_1, v_2) = 6　大于(v_0, v_2)，不需要调整
   (v_1, v_3) = 4　小于(v_0, v_3)，需要调整
   (v_1, v_4) = 11　小于(v_0, v_4)，需要调整
   mst[4] = {(0,1,3),(0,2,5),(1,3,4),(1,4,11)}



4) 在mst[1]到mst[3]中找权值最小的边mst[2]，将顶点v_3及边 (v_1,v_3)加入到最小生成树中。交换mst[1]和mst[2]，得到：

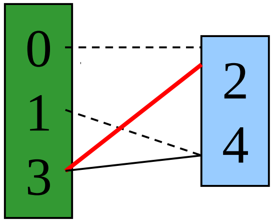
mst[4] = {(0,1,3),(1,3,4),(0,2,5),(1,4,11)}

5) 按照新加入顶点v_3调整mst[2]到mst[3]

$(v_3, v_2) = 2$ 小于(v_0, v_2)，需要调整

$(v_3, v_4) = 10$ 小于(v_1, v_4)，需要调整

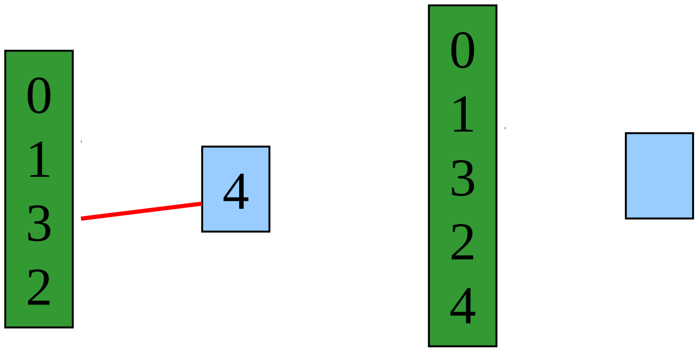
mst[4] = {(0,1,3),(1,3,4),(3,2,2),(3,4,10)}

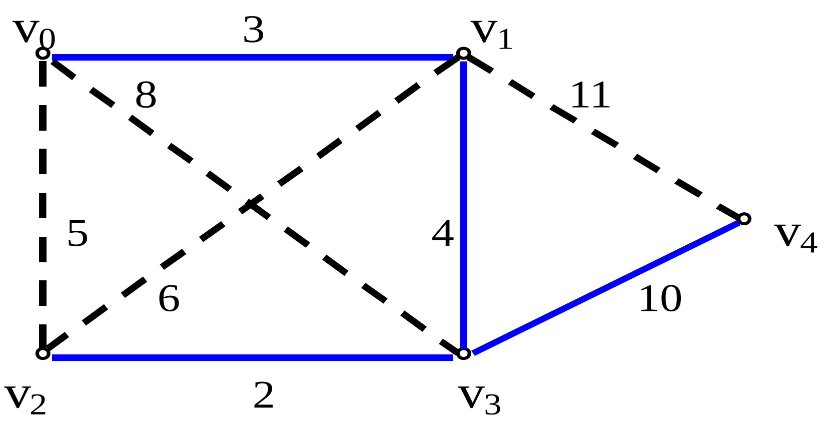


6) 在mst[2]到mst[3]中找权值最小的边mst[2]，将顶点v_2及边 (v_3,v_2)加入到最小生成树中。调整判断后，得到：

mst[4] = {(0,1,3),(1,3,4),(3,2,2),(3,4,10)}

7) 将v_4和mst[4]加入到最小生成树中，得到下面的最小生成树：





```
void prim(Graph * pGraph, Edge mst[])
{
    int        i, j, min, vx, vy;
    float     weight, minweight;
    Edge      edge;

    //初始状态下，U只包含v0。计算MST:
    for(i=0; i<pGraph->n-1; i++)
    {  mst[i].start_vex=0;
       mst[i].stop_vex=i+1;
       mst[i].weight=pGraph->arcs[0][i+1];
    }
```

接下页……

```
for(i=0; i<pGraph->n-1; i++)                /* 共n-1条边 */
{   /* 从所有边(vx,vy)(vx∈U,vy∈V-U)中选出最短的边 */
    minweight=MAX;    min=i;
    for(j=i; j<pGraph->n-1; j++)
    {   if(mst[j].weight<minweight)
          {  minweight=mst[j].weight;   min = j; }
     }     /* mst[min]是最短的边(vx,vy)(vx∈U, vy∈V-U)，  */

     /* 将mst[min]加入最小生成树【交换到i位置】  */
    edge=mst[min];    mst[min]=mst[i];     mst[i]=edge;
    vx=mst[i].stop_vex;      /* vx为刚加入最小生成树的顶点下标 */
    for(j=i+1; j<pGraph->n-1; j++)  /* 调整mst[i+1]到mst[n-1] */
    {   vy=mst[j].stop_vex;       weight=pGraph->arcs[vx][vy];
        if(weight<mst[j].weight)
        {   mst[j].weight=weight;    mst[j].start_vex=vx;  }
    }
}
}
```

效率分析：空间复杂度：O（n）： 存放挑选出的边

时间复杂度：O（n^2）：与边数无关，适用于稠密图

3. Kruskal算法

设连通网N＝(V,E)。令最小生成树的初始状态为只有n个顶点而无边的非连通图T＝(V,Φ)，图中每个顶点自成一个连通分量。在E中选择代价最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入T，否则舍去此边而选择下一条代价最小的边，依次类推，直到T中所有顶点都在同一连通分量为止。

E中的边按照权值递增顺序排列；

$T = (V, \phi)$

While (T中所含边数 < n-1)

{ 从E中选取当前最短边(u, v);

从E中删除边(u,v);

if ((u,v)加入T后不产生回路) 将边(u,v)加入T中；

}

S_1、S_2…S_m共m个集合包含当前已选择的顶点(m个联通分量)，

u和v是否同属于一个集合S_k中？

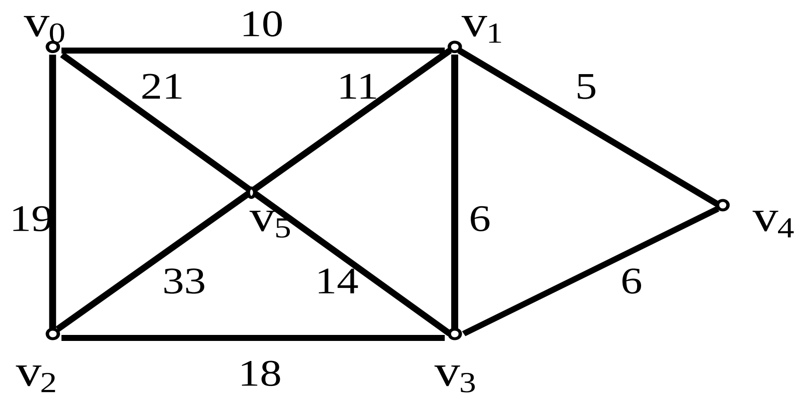
如果不是：加入边(u,v)，u和v所在的集合合并。

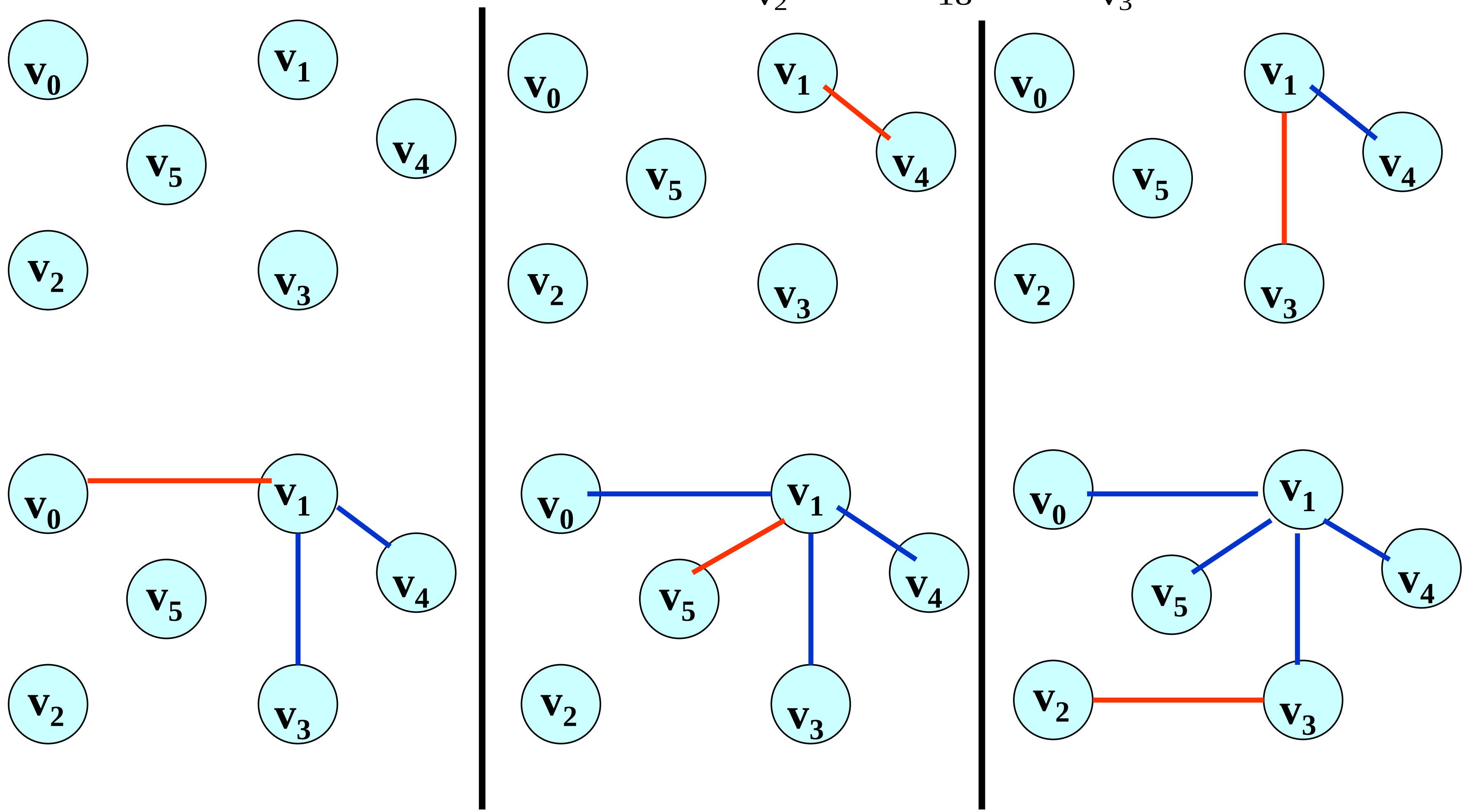
最后所有顶点在一个集合中。

最小生成树的Kruskal生成过程

5(1,4),6(1,3),6(3,4),10(0,1),11(1,5),

14(3,5),18(2,3),19(0,2),21(0,5),33(2,5)



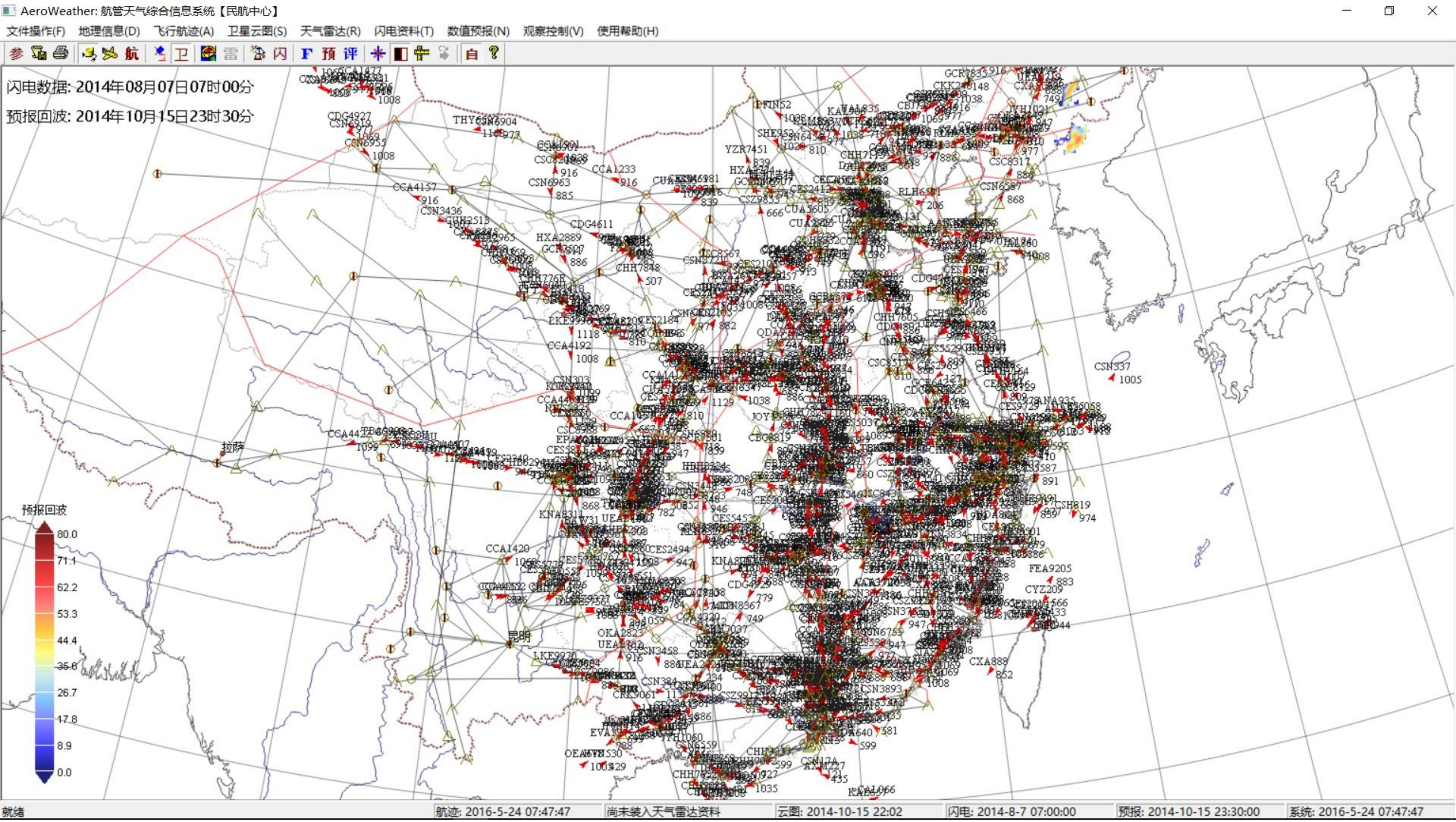


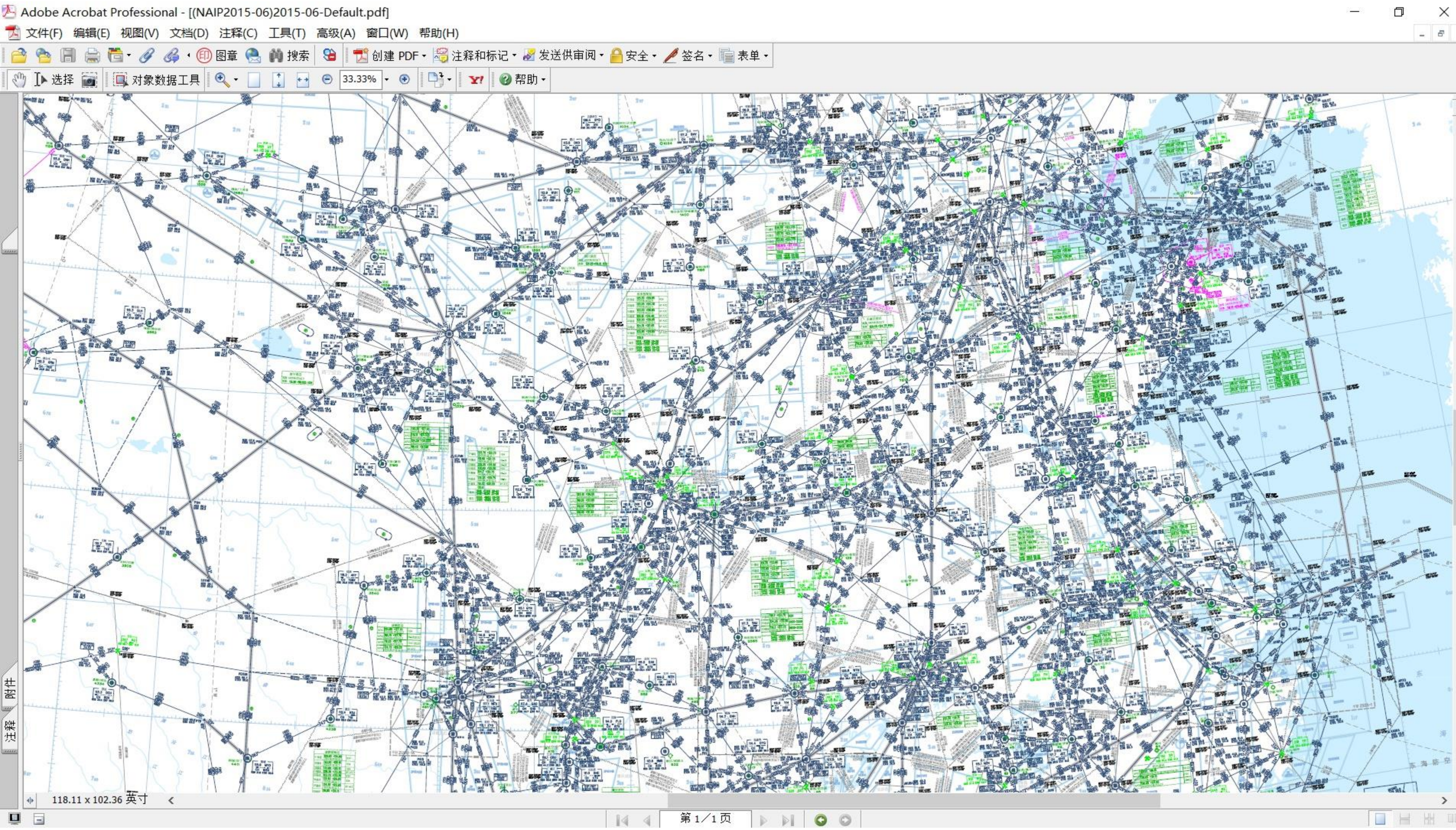
```
Kruskal （G， mst[])
{
    EDGE *edges = sort_edges(G);           //按照从小到大排序所有边
    int sets[NMAXV];                       //集合：初始化n个，每个包含1个顶点
    int i, j, k = 0, m= 0,  n = G->n;
    for (i = 0; i < n; i++) sets[i] = i;   //集合初始化
    while (m < n-1)                        //n-1条边的选取
    {   s = edges[k].s,  e = edges[k].e;   //选择当前最小的
        if (sets[s] != sets[e])            //是否同属一个连通分量（集合）
        {   add2mst(edges[k], mst, m);     //选择一条边加入到MST
            m++;
            j = sets[e];                   //集合合并：两个连通分量合并
            for (i = 0; i < n; i++)
            {   if (sets[i] == j)   sets[i] = sets[s];  }
        }
        k++;
    }
}
```

```
typedef struct
{
    int s, e;
    float w;
} EDGE;
```

时间复杂度：$O(n^2)$

空间复杂度：$O(n)$


AeroWeather: 航管天气综合信息系统【民航中心】
文件操作(F) 地理信息(D) 飞行航迹(A) 卫星云图(S) 天气雷达(R) 闪电资料(T) 数值预报(N) 观察控制(V) 使用帮助(H)
闪电数据: 2014年08月07日07时00分
预报回波: 2014年10月15日23时30分
拉萨
昆明
西宁
预报回波
80.0
71.1
62.2
53.3
44.4
35.6
26.7
17.8
8.9
0.0
就绪
航迹: 2016-5-24 07:47:47
尚未装入天气雷达资料
云图: 2014-10-15 22:02
闪电: 2014-8-7 07:00:00
预报: 2014-10-15 23:30:00
系统: 2016-5-24 07:47:47


Adobe Acrobat Professional - [(NAIP2015-06)2015-06-Default.pdf]
文件(F) 编辑(E) 视图(V) 文档(D) 注释(C) 工具(T) 高级(A) 窗口(W) 帮助(H)
图章 搜索 创建 PDF 注释和标记 发送供审阅 安全 签名 表单
选择 对象数据工具 33.33% 帮助
附件
注释
118.11 x 102.36 英寸
第 1／1 页

8.5 最短路径

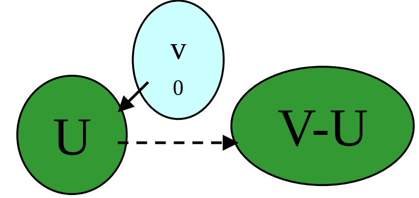
最短路径：从一个顶点到另一个顶点，可能存在多条路径。其中长度最短（路径上边或弧的加权值和最小）的路径。

应用广泛: 两个城市(地点)之间有多条道路,求其中距离最短的。

从某个源点到其余各顶点的最短路径

每一对顶点之间的最短路径

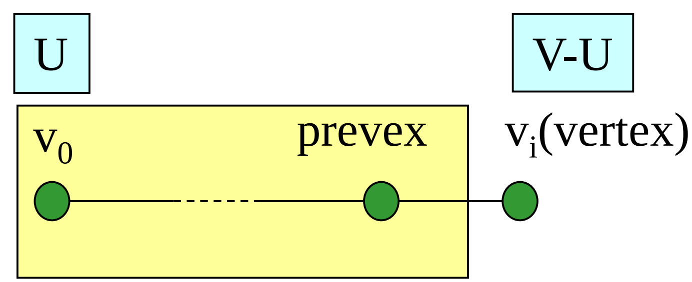
1. 从某个源点到其余各顶点的最短路径



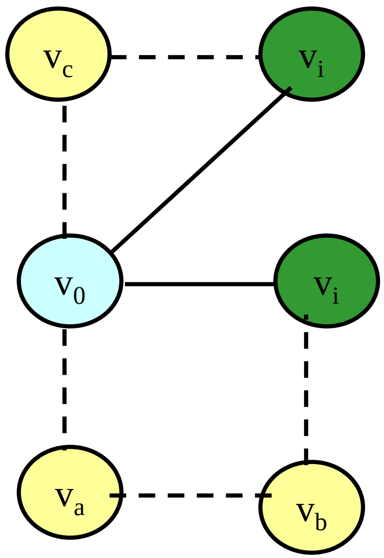
基本思想(Dijkstra方法，按照长度递增的次序产生)：设U存放已求出最短路径的顶点，V-U是尚未确定最短路径的顶点集合；U中顶点的距离值是从v_0到该顶点的最短路径长度，V-U中顶点的距离值是从v_0到该顶点的只包括U中顶点为中间顶点的最短路径长度。

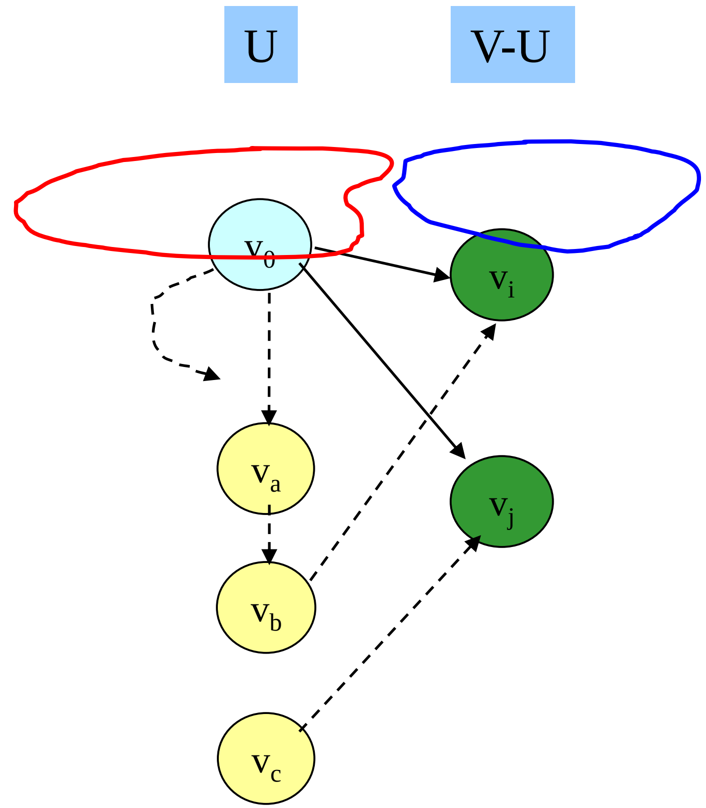
设置一个数组dist[n]，用于存放v_0到其它各个顶点的最短路径：

```
typedef struct
{
    AdjType  length;
     int      prevex;        //从v0到达vi(i=1,2,…,n-1)的最短路径上vi的前驱顶点
} Path;
Path dist[n];    //n个顶点
```



dist[i]的意义



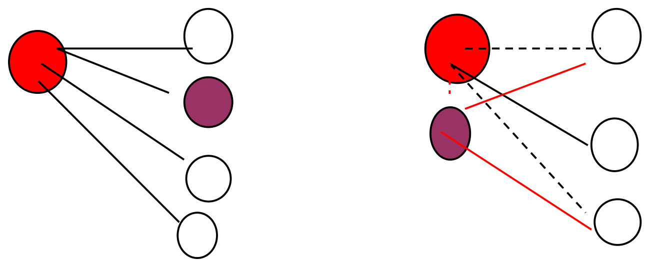


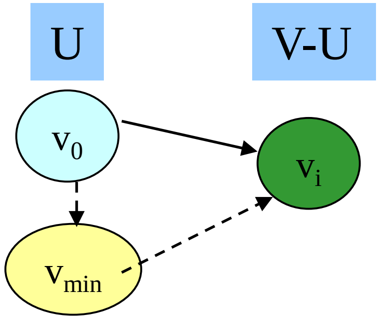
已经求出v_0到v_a、v_b、v_c的最短距离，
现在求v_0到v_i和v_j的最短距离。

可能：从v_0经过U里的中间点更短。

图采用邻接矩阵存储

(1) 初始时U中只有v_0，集合V-U中顶点v_i的距离值为边(v_0, v_i)的权值；如果(v_0,v_i)不存在，用∞表示。





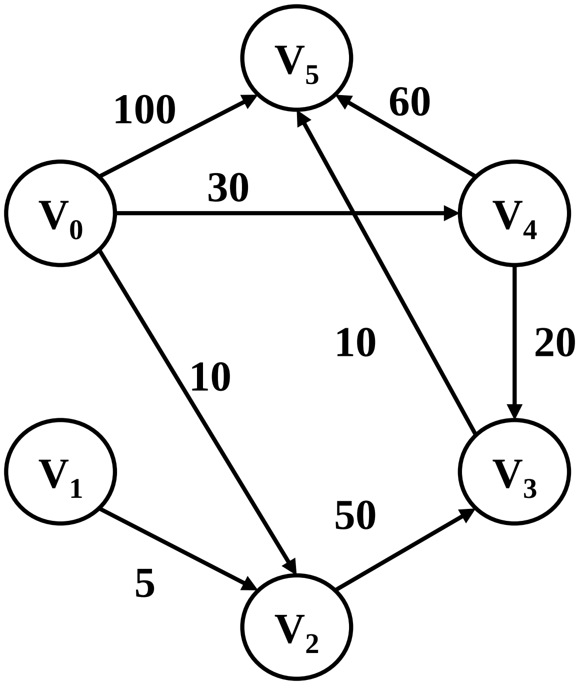
在V-U中，选择距离值最小的顶点v_{min}加入U，然后对V-U中各顶点的距离值修正。如果加入v_{min}为中间顶点后，使v_0到$v_i \in$ V-U的距离值比原距离值小，则修改v_i的距离值。

如果dist[min].length+graph.arcs[min][i] < dist[i].length 则将顶点v_i的距离修正为dist[min].length+graph.arcs[min][i]，并将路径上v_i的前驱顶点修改为v_{min}，即：dist[i].prevex = min。

(3) 反复操作，直到从v_0出发可以到达的所有顶点都在U中为止。

例： 对下图求顶点v_0和其余顶点之间的最短距离为：

始点	终点	最短路径	路径长度
v_0	v_1	-----	
	v_2	(v_0,v_2)	10
	v_3	(v_0,v_4,v_3)	50
	v_4	(v_0,v_4)	30
	v_5	(v_0,v_4,v_3,v_5)	60



（1）初始：只有v_0

dist[n] = {(0,0), (MAX,-1), (10,0), (MAX,-1), (30, 0), (100,0)}

（2）在dist[n]中找距离最小顶点v_2，将顶点v_2加入集合U中。

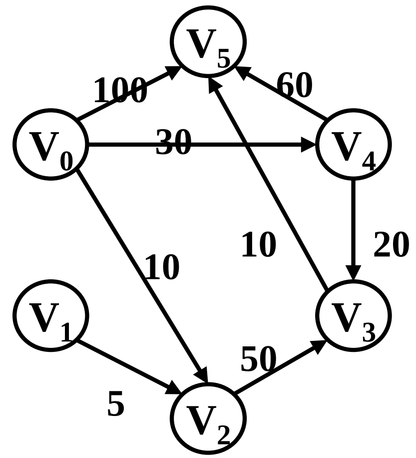
dist[n] = {(0,0), (MAX,-1), (10,0), (MAX,-1), (30, 0), (100,0)}

调整集合V-U中顶点距离值，此时min=2

因为， dist[1].length=MAX,

dist[2].length+graph.arcs[2][1] = 10+MAX

顶点v_1的距离值不需要调整;



因为， dist[3].length=MAX

dist[2].length+graph.arcs[2][3] = 10+50=60

顶点v_3的距离值需要调整为60，并且其前驱顶点为v_2;

同理判断v_4, v_5不需要调整。

dist[n] = {(0,0), (MAX,-1), (10,0), (60,2), (30, 0), (100,0)}

（3）在dist[n]中找距离最小顶点v_4，将顶点v_4加入集合U中。

dist[n] = {(0,0), (MAX,-1), (10,0), (50,4), (30, 0), (90,4)}

（4）在dist[n]中找距离最小顶点v_3，将顶点v_3加入集合U中。

dist[n] = {(0,0), (MAX,-1), (10,0), (50,4), (30, 0), (60,3)}

（5）在dist[n]中找距离最小顶点v_5，将顶点v_5加入集合U中。

dist[n] = {(0,0), (MAX,-1), (10,0), (50,4), (30, 0), (60,3)}

（6）没有可以加入到U的顶点了，说明从v_0到v_1无路径。

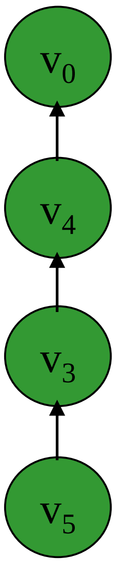
从dist的prevex得到v_0到各个顶点的最短路径。

如：v_0到v_5的最短路径

dist[5].prevex=3 可知路径上v_5的前一个顶点为v_3，

dist[3].prevex=4 可知路径上v_3的前一个顶点为v_4，

dist[4].prevex=0 可知路径上v_4的前一个顶点为v_0，



因此：v_0到v_5的最短路径为： v_0-v_4-v_3-v_5

同理可以求出v_0到其它顶点的最短路径。

```
void dijkstra(Graph graph, Path dist[])
{   int  i, j, minvex;
    AdjType min, new_length;

    dist[0].length=0;    dist[0].prevex=0;
    graph.arcs[0][0]=1;            /* 表示顶点v0在集合U中 */

    for(i=1; i<graph.n; i++)      /* 初始化集合V-U中顶点的距离值 */
    {
        dist[i].length=graph.arcs[0][i];
        if(dist[i].length != MAX)
                dist[i].prevex=0;
        else
                dist[i].prevex= -1;
    }
    接下页……
```

```
for(i=1; i<graph.n; i++)
{   min=MAX;     minvex=0;
    for(j=1; j<graph.n; j++)      /*在V-U中选出距离值最小顶点*/
        if ( (graph.arcs[j][j]==0) && (dist[j].length<min) )
        {   min=dist[j].length;   minvex=j;   }
    if(minvex==0)   break;     /*结束，从v0没有路径可通往集合V-U中的顶点 */

    graph.arcs[minvex][minvex]=1;  /* V-U路径最小顶点minvex, 加入U集合 */
    for(j=1; j<graph.n; j++)           /* 调整集合V-U中顶点的最短路径 */
    {  if (graph.arcs[j][j]==1)  continue;  /* U集合中的顶点 */
       new_length = dist[minvex].length+graph.arcs[minvex][j];
       if (new_length < dist[j].length)
       {  dist[j].length = new_length;
          dist[j].prevex = minvex;
       }
    }
  }
}
```

效率分析：

空间复杂度：$O(n)$

时间复杂度：$O(n^2)$

2. 每一对顶点之间的最短路径

两种方法：

a）依次变换图中的每个顶点为起点，用**Dijkstra**方法求该顶点到其余所有顶点的最短距离。时间复杂度为$O(n^3)$

b）动态规划法：**Floyd**算法，时间复杂度也为$O(n^3)$，但算法简单，容易理解。

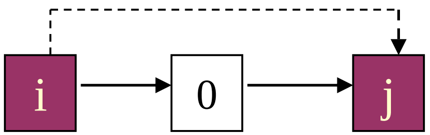
对于图G=(V,E)，有n个顶点，采用邻接矩阵存储。

如果边$(v_i,v_j)\in E$，则从v_i到v_j存在一条长度为arcs[i][j]的路径，但该路径不一定为最短路径，因为可能存在包含其它顶点为中间顶点的更短路径。因此，应该考虑v_i到v_j的所有路径，求长度最短的。

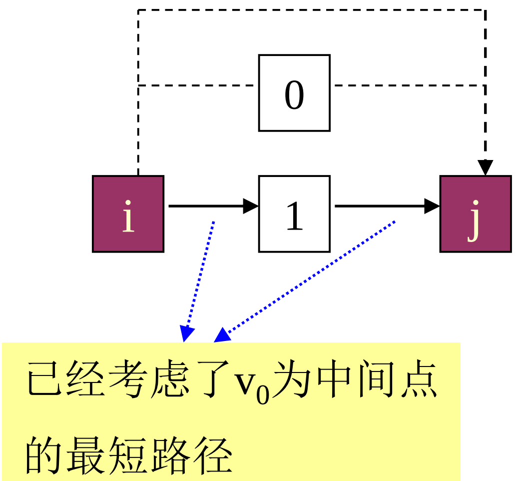
如何确定所有的路径，求最短路径？

思想： 对于<u>所有的</u>V_i到V_j，依次考虑加入 V_0、V_1、…、V_k、…、V_{n-1}为中间点后路径的变化。

首先考虑路径(v_i, v_0)和(v_0,v_j)是否存在，如果存在，则比较(v_i,v_j)和$(v_i,v_0)+(v_0,v_j)$的路径长度，取较短者为当前的最短路径。该路径是从v_i到v_j允许一个顶点v_0为中间点的最短路径。

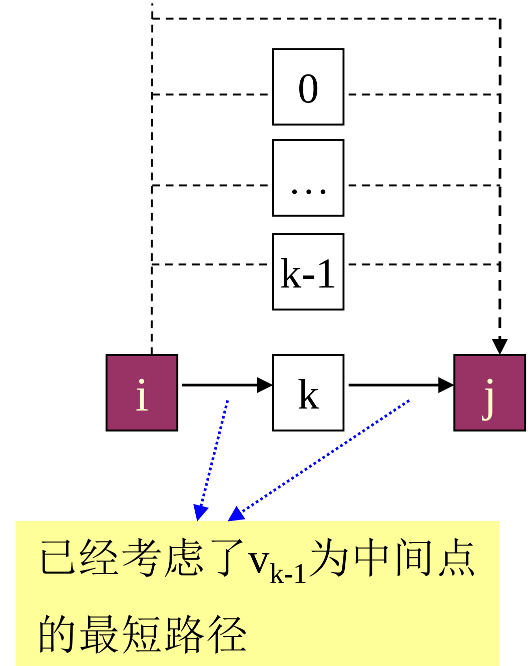


其次，考虑从v_i到v_j是否存在包含v_1为中间点的路径: $(v_i,\dots,v_1,\dots,v_j)$，其中$(v_i,\dots,v_1)$和$(v_1,\dots,v_j)$分别是<u>前一次找到的</u>允许顶点$v_0$为中间点的最短路径。如果存在这样的路径，则$(v_i,\dots,v_1)+(v_1,\dots,v_j)$为路径$(v_i,\dots,v_1,\dots,v_j)$的长度，将其与前一次求得的允许$v_0$为中间点的最短路径长度比较，取较短者为当前的最短路径。



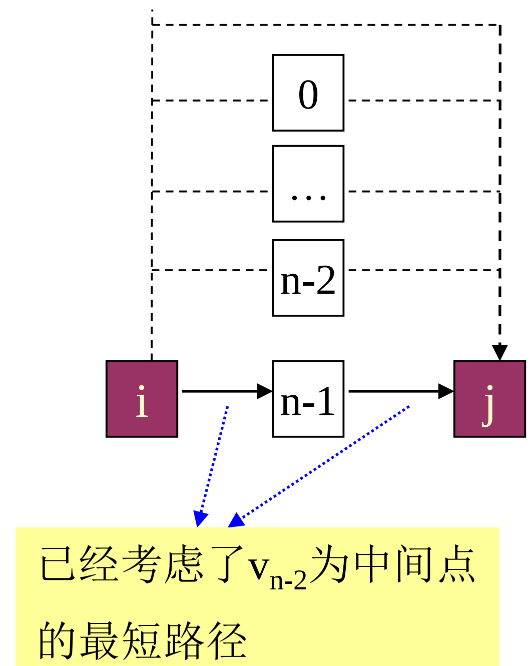
………………

如果$(v_i,\ldots,v_k)$和$(v_k,\ldots,v_j)$分别是从v_i到v_k和从v_k到v_j允许k个顶点$v_0,v_1,\ldots,v_{k-1}$为中间点的最短路径，则将 $(v_i,\ldots,v_k)+(v_k,\ldots,v_j)$ 与已经得到的从v_i到v_j允许k个顶点$v_0,v_1,\ldots,v_{k-1}$为中间点的最短路径进行比较，取较短者为从v_i到v_j允许k+1个顶点$v_0,v_1,\ldots,v_k$为中间点的最短路径。



…………………

依次类推，直到加入顶点v_{n-1}为止，则得到的是v_i到v_j允许n个顶点$v_0,v_1,\ldots,v_{n-1}$为中间点的最短路径。此时，已经考虑了所有顶点为中间点的可能性，因此，得到结果。



实现：

定义一个n×n的方阵序列：$A_0, A_1, \ldots, A_n$，其中$A_k[i][j]$表示从v_i到v_j允许k个顶点$v_0, v_1, \ldots, v_{k-1}$为中间点的最短路径$(0 \le k \le n)$，并且$A_0$等于图的邻接矩阵。$A_0[i][j]$表示从$v_i$到$v_j$不经过任何中间点的最短路径，$A_n[i][j]$就是从$v_i$到$v_j$的最短路径。

$$A_0[i][j] = arcs[i][j]$$

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

其中, $0 \le i \le n-1$, $0 \le j \le n-1$, $1 \le k \le n$

主要计算过程：关系矩阵上的迭代和修改。

另外，设置一个n×n的矩阵nextvex存放v_i到v_j最短路径上v_i的后继顶点。初始时，如果$A_0[i][j] = \infty$，则$nextvex[i][j] = -1$，否则$nextvex[i][j]=j$，表示v_j是v_i的后继顶点。

在由A_{k-1}计算A_k时，如果$A_k[i][j] = A_{k-1}[i][k-1]+ A_{k-1}[k-1][j]$，则 nextvex[i][j]=nextvex[i][k-1], nextvex[i][j]表示从v_i到v_j允许k个顶点$v_0,v_1,\ldots,v_{k-1}$为中间点的最短路径上顶点v_i的后继顶点。

路径：长度 $A_n[i][j]$

nextvex[i][j] = m_1，

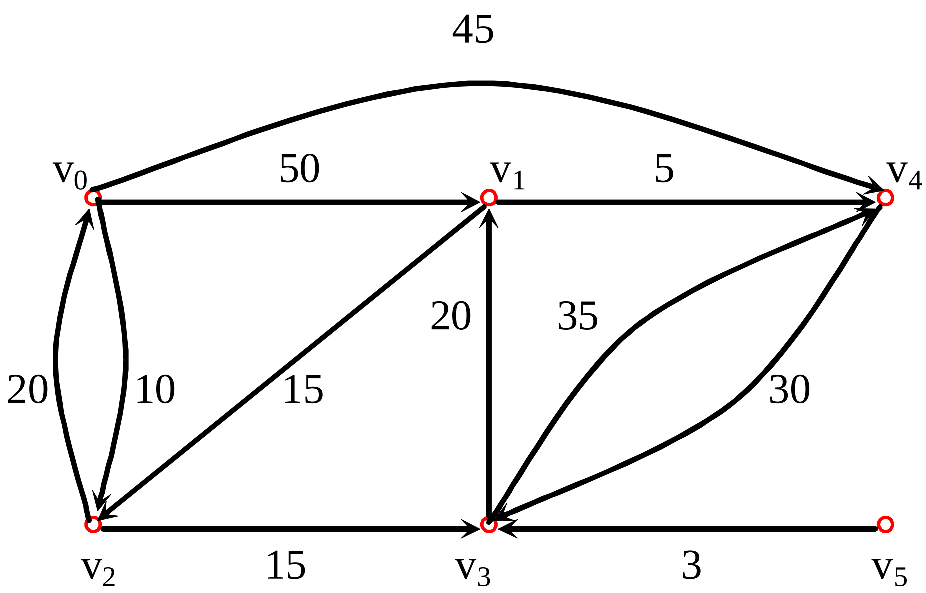
nextvex[m_1][j] = m_2,

……,

nextvex[m_k][j]=j

$v_i \to v_{m1} \to v_{m2} \to \ldots \to v_{mk} \to v_j$

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



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

$$\mathbf{nextvex}_0 = \begin{bmatrix} 0 & 1 & 2 & -1 & 4 & -1 \\ -1 & 1 & 2 & -1 & 4 & -1 \\ 0 & -1 & 2 & 3 & -1 & -1 \\ -1 & 1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & -1 & 5 \end{bmatrix}$$

加入第一个顶点v_0,

$$A_1[i][j]=\min\{A_0[i][j],\ A_0[i][0]+A_0[0][j]\}$$

$$0\le i\le n-1,\ 0\le j\le n-1$$

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

$$\mathbf{nextvex}_1=\begin{bmatrix} 0 & 1 & 2 & -1 & 4 & -1 \\ -1 & 1 & 2 & -1 & 4 & -1 \\ 0 & 0 & 2 & 3 & 0 & -1 \\ -1 & 1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & -1 & 5 \end{bmatrix}$$

加入顶点v_1，

$$A_2[i][j]=\min\{A_1[i][j],A_1[i][1]+A_1[1][j]\}$$

$$0\le i\le n-1,\ 0\le j\le n-1$$

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

$$\text{nextvex}_2=\begin{bmatrix} 0 & 1 & 2 & -1 & 4 & -1 \\ -1 & 1 & 2 & -1 & 4 & -1 \\ 0 & 0 & 2 & 3 & 0 & -1 \\ -1 & 1 & 1 & 3 & 1 & -1 \\ -1 & -1 & -1 & 3 & 4 & -1 \\ -1 & -1 & -1 & 3 & -1 & 5 \end{bmatrix}$$

加入顶点v_2，

$$A_3[i][j]=\min\{A_2[i][j], A_2[i][2]+A_2[2][j]\}$$

$$0\le i\le n-1,\ 0\le j\le n-1$$

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

加入顶点v_3

$$A_4[i][j]=\min\{A_3[i][j],\ A_3[i][3]+A_3[3][j]\}$$

$$0\le i\le n-1,\ 0\le j\le n-1$$

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

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

例如，想知道v_0到v_1的最短路径

路径长度：A[0][1]=45;

最短路径：

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$v_0 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1$

算法:

最短路径及长度存储结构

```
typedef struct
{    /* 存放每对顶点间最短路径长度 */
    AdjType a[MAXVEX][MAXVEX];
    /* nextvex[i][j]存放vi 到vj最短路径上vi的后继顶点的下标值 */
     int nextvex[MAXVEX][MAXVEX];
}ShortPath;
```

void floyd(Graph * pGraph, ShortPath * ppath)

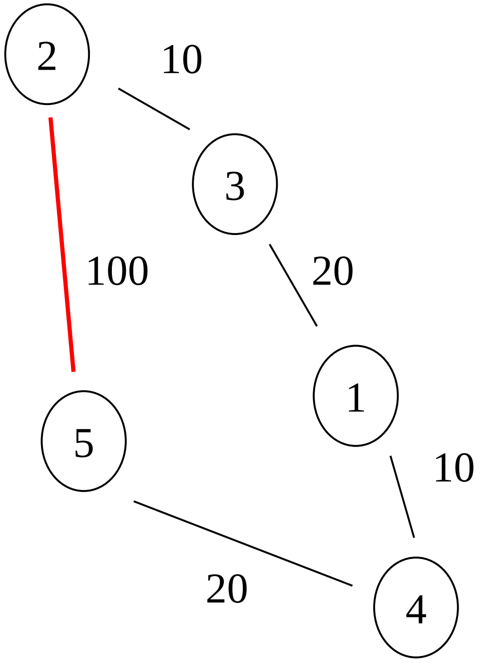
```
{
    int  i, j, k;
    for (i=0; i<pGraph->n; i++)          //初始化
    for (j=0; j<pGraph->n; j++)
    {
      if (pGraph->arcs[i][j]!=MAX)   ppath->nextvex[i][j]=j;
      else                           ppath->nextvex[i][j]= -1;
      ppath->a[i][j]=pGraph->arcs[i][j];
    }  //接下页→
```

```
//计算A0～An， nextvex
for (k=0; k<pGraph->n; k++)
{
    for (i=0; i<pGraph->n; i++)
    for (j=0; j<pGraph->n; j++)
    {
        if ( (ppath->a[i][k]==MAX)||(ppath->a[k][j]==MAX) )
            continue;
        distance =ppath->a[i][k]+ ppath->a[k][j];
        if (distance < ppath->a[i][j])
        {
            ppath->a[i][j]= distance;
            ppath->nextvex[i][j]=ppath->nextvex[i][k];
        }
    }
}
}
```

复杂度分析：

时间复杂度：$O(n^3)$ 三层循环

空间复杂度：$O(n^2)$ 两个n×n矩阵



对于2-5，考虑1、2、3中间点时，都不变化（100）。

考虑v4中间点时：

2-4：已经考虑了1~3中间点的所有路径中最少的=40

4-5：20

40+20 < 100

因此，2-5：修改为60

8.6 拓扑排序

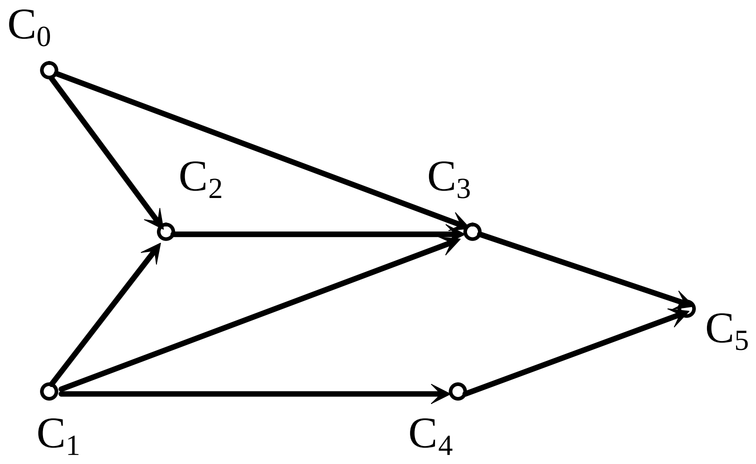
- 8.6.1 AOV网
- 8.6.2 拓扑排序

8.6.1 AOV网

基本概念：

AOV网：如果用图中的顶点表示活动，边表示活动间的先后关系，则这样的有向图称为顶点活动网(<u>A</u>ctivity <u>O</u>n <u>V</u>ertex network，简称AOV网)

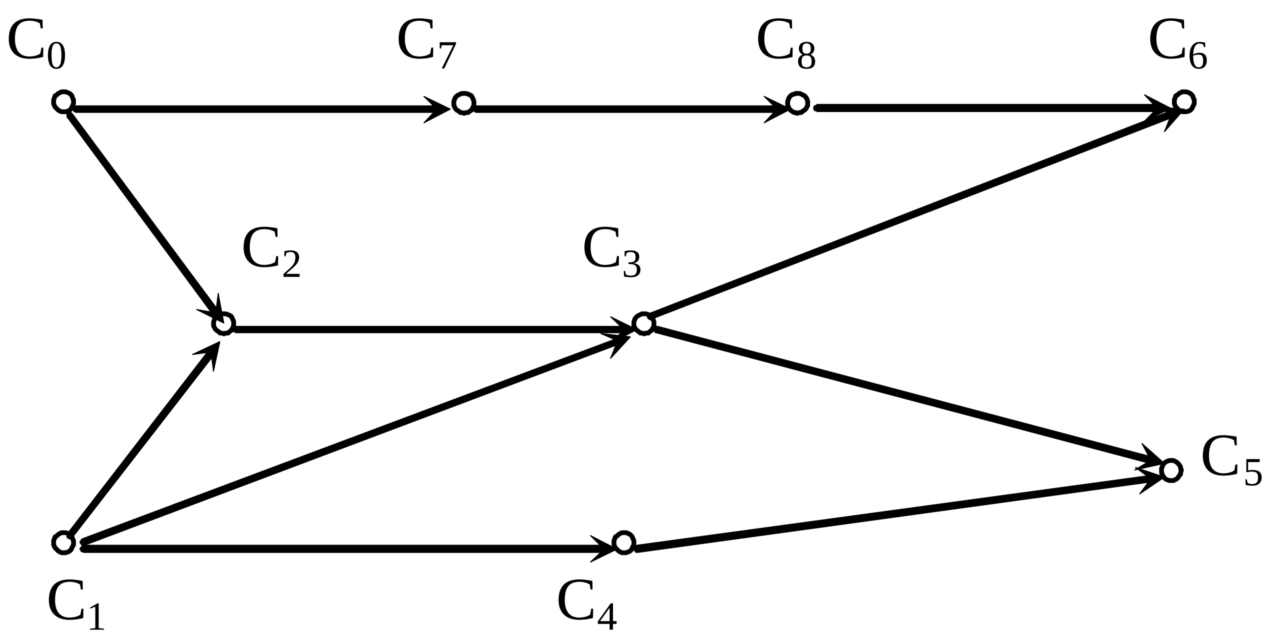
AOV网中的弧表示活动之间存在的先后制约关系



例：计算机专业的学生必须完成一系列规定的基础课和专业课才能毕业，这时工程就是完成给定的学习计划，而活动就是学习课程，这些课程的名称和代号如下表所示：

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_3,C_8
C_7	普通物理	C_0
C_8	计算机原理	C_7

在AOV网中，用顶点表示课程，有向边表示课程之间的优先关系，如果课程C_i是课程C_j的先修课，则在AOV网中必定存在一条有向边$<C_i, C_j>$。表中各课程的AOV网如下图所示



8.6.2 拓扑排序

对于一个**AOV网**，其所有顶点可以排成一个线性序列v_1，v_2，…，v_n，该线性序列具有以下性质：如果在AOV网中，从顶点v_i到顶点v_j存在一条路径，则在线性序列中，顶点v_i一定排在顶点v_j之前。

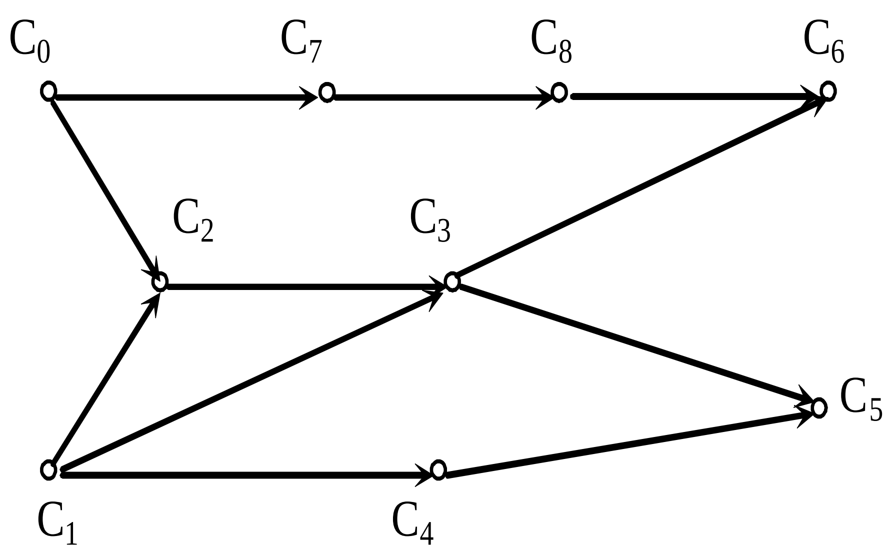
具有这种性质的线性序列称为拓扑序列，构造拓扑序列的操作称为拓扑排序

例：对下图中的AOV网进行拓扑排序

得到的一个拓扑序列：$C_0, C_1, C_2, C_4, C_3, C_5, C_7, C_8, C_6$，

另外一个拓扑序列：$C_0, C_7, C_8, C_1, C_4, C_2, C_3, C_6, C_5$

如果一个学生一学期只能选修一门课，则他必须按照某一个拓扑序列的次序学习，才能保证学习任何一门课时，其先修课程已学过。

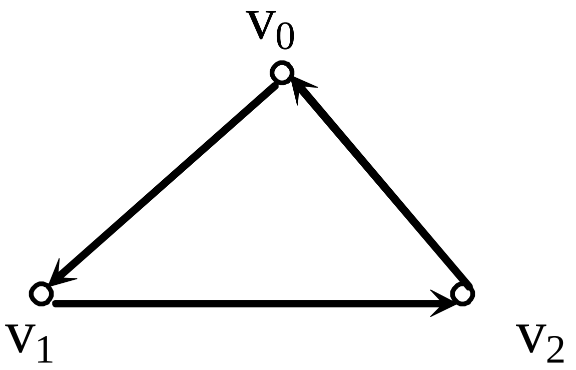


注意：

1）一个AOV网的拓扑序列不一定是唯一的。

2）假设AOV网代表一个工程，如果条件限制只能串行工作，则AOV网某一拓扑序列就是整个工程得以顺利完成的一种可行方案。**AOV网中一定不能出现回路**（因为出现回路意味着，某些活动的开工是以自己工作的完成作为先决条件，这种现象称为死锁）。

下图所示的AOV网，就无法把顶点排成满足拓扑序列条件的线性序列。

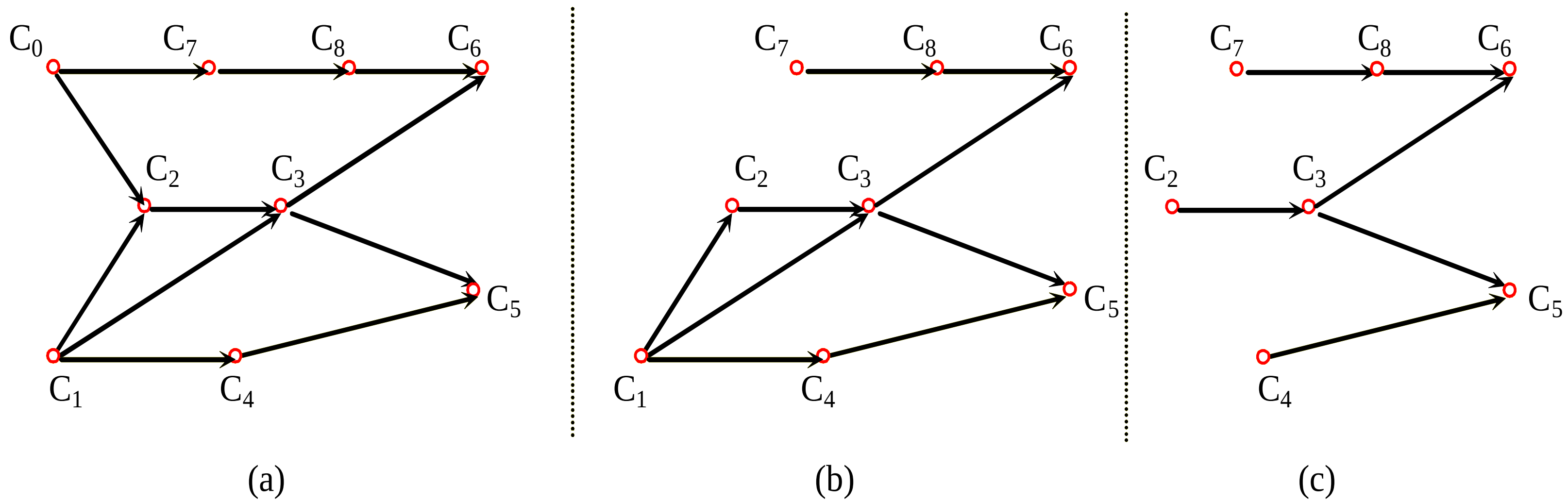


任何无回路的AOV网，其顶点都可以排成一个拓扑序列，方法如下：

1) 从AOV网中选择一个入度为0的顶点将其输出。
2) 在AOV网中删除此顶点及其所有的出边，修改出边关联的顶点的入度（减）。
3) 反复执行以上两步，直到所有顶点都已经输出为止，此时整个拓扑排序完成；
   或者直到剩下的顶点的入度都不为0为止，此时说明AOV网中存在回路，拓扑排序无法再进行。

①. 顶点C_0和C_1的入度为0，可以任选一个输出，选择C_0，将C_0及其所有的出边删除，得到图(b)。

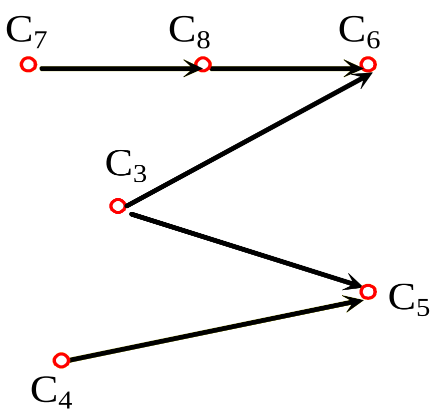
②. 这时，图中入度为0的顶点是C_1和C_7，选择C_1输出，并删除C_1和它的所有出边，得到图(c)。



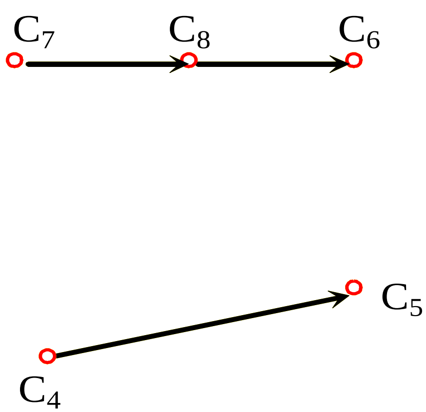
(a) 初态 (b)输出C_0后 (c) 输出C_1后

③. 依此类推，依次选择C_2, C_3, C_4, C_5, C_7, C_8, C_6输出，每次输出并删除所有出边后的图为(d)，(e)，(f)，(g)，(h)，(i)。

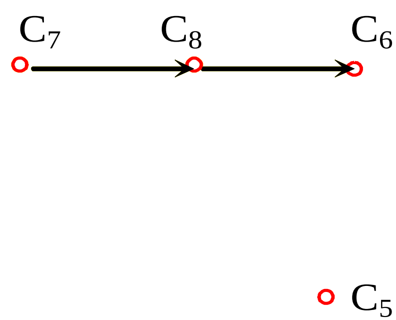
最后得到的拓扑序列为C_0, C_1, C_2, C_3, C_4, C_5, C_7, C_8, C_6。



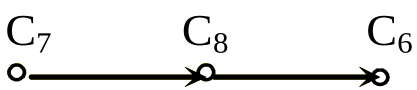
(d) 输出C2后



(e) 输出C3后

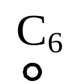


(f) 输出C4后



(g) 输出C5后

(h) 输出C7后



(i) 输出C8后

拓扑排序算法：

设AOV网采用邻接表表示，边表为出边表。

算法中定义一个indegree数组，存放各顶点的入度。

由于可能存在多个入度为0的顶点，为了控制输出次序，设置一个链栈存储入度为0的顶点。

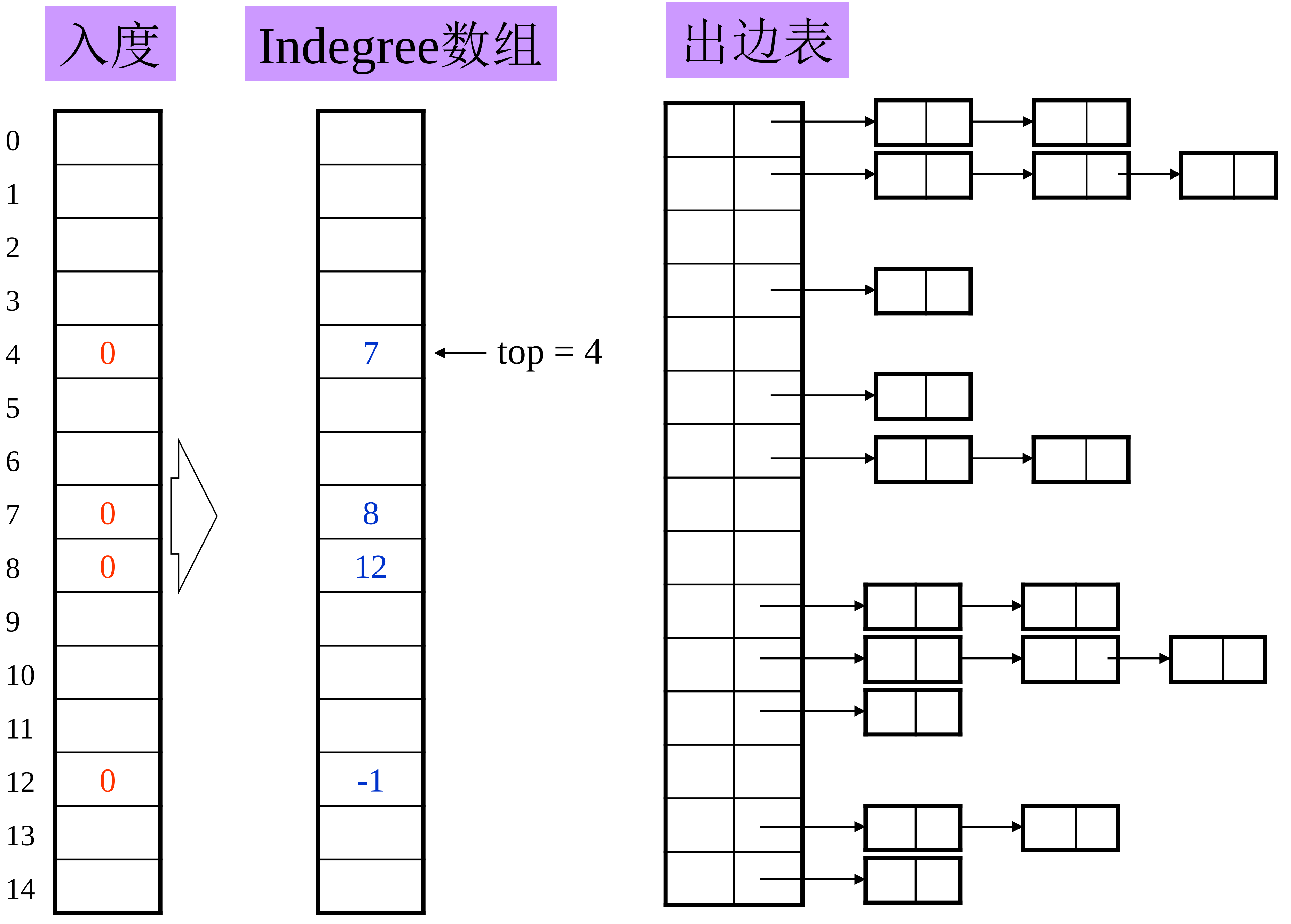
拓扑排序前，先计算所有顶点的入度，然后将所有入度为0的顶点压栈（其入度已经没有意义，利用“入度”构成静态链栈。一旦弹栈出去，该结点再没有用途）。

从栈顶取出一个顶点将其输出，由它的出边表可以得到以该顶点为起点的出边，将这些边终点的入度减1，即删除这些边。如果某条边终点的入度为0，则将该顶点入栈。

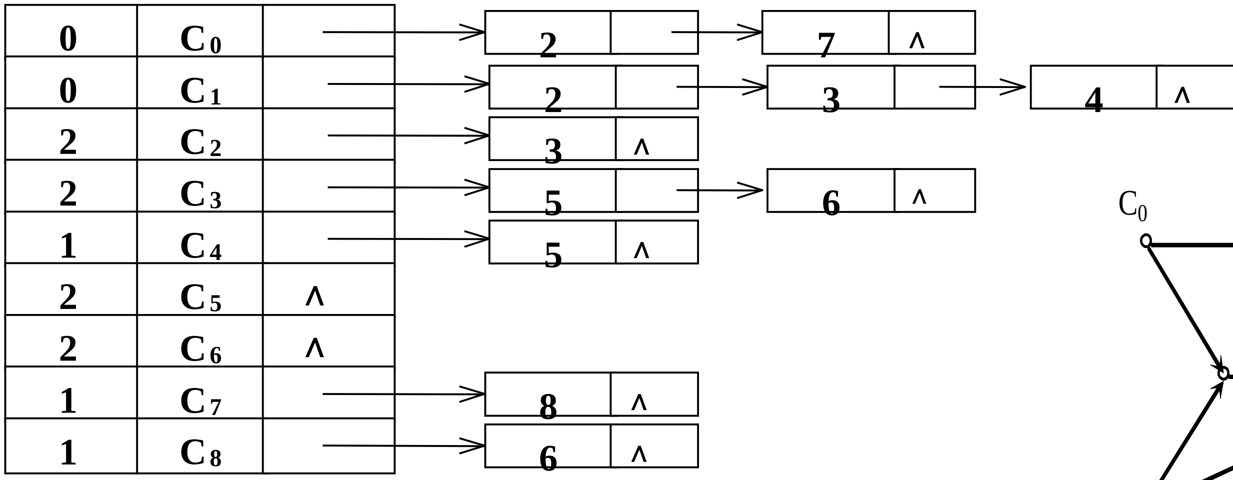
反复进行上述操作，直到栈为空。此时，如果输出的顶点个数小于n，则说明该AOV网中存在回路；否则，拓扑排序正常结束。

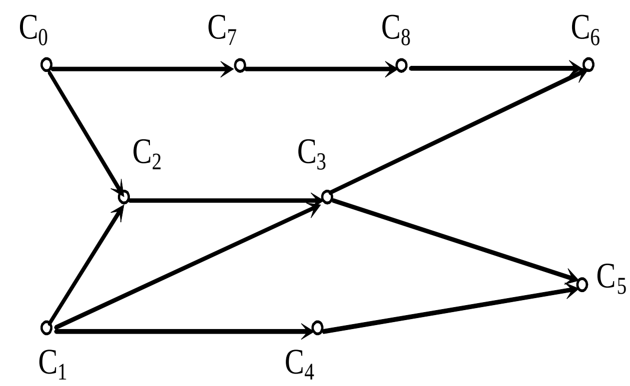
算法结束后，拓扑序列存放在变量ptopo中。

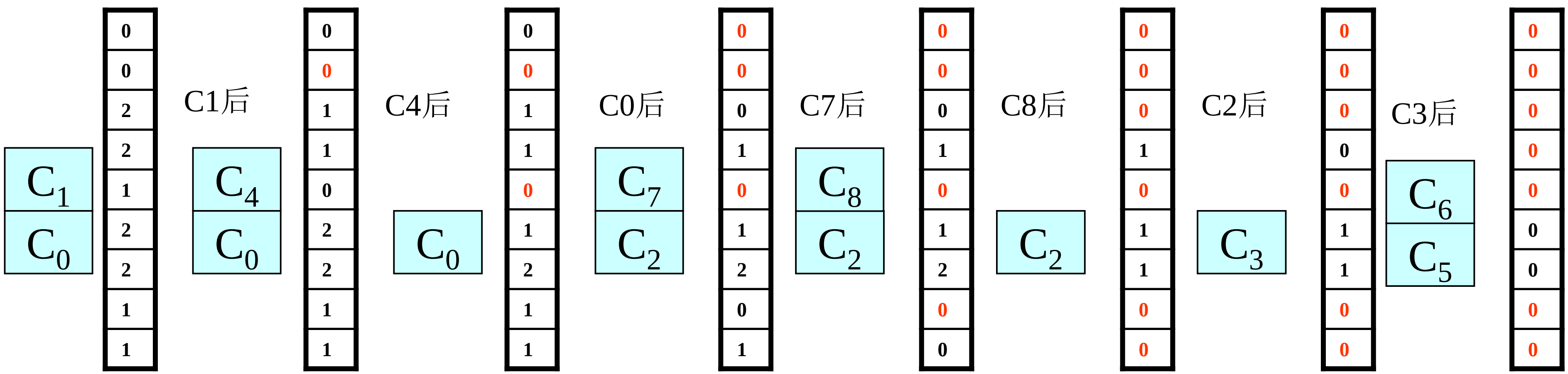
具体实现时，链栈可以利用顶点表中，值为0的indegree元素实现。


入度
Indegree数组
出边表
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
0
0
0
0
7
8
12
-1
top = 4

例：AOV网的邻接表表示如图所示。按上述算法进行拓扑排序







拓扑序列为:$C_1, C_4, C_0, C_7, C_8, C_2, C_3, C_6, C_5$

算法复杂度：

设AOV网有n个顶点，e条边，算法最初首先统计各个顶点的入度，并检查入度为零的顶点，并将这些顶点压栈，花费的时间为O(n+e)。

下面进行拓扑排序时，每个顶点都入栈一次，且每个顶点边表中的边结点都被检查一遍，运行时间为O(n+e)。

因此，拓扑排序算法的时间复杂度为O(n+e)

8.7 关键路径

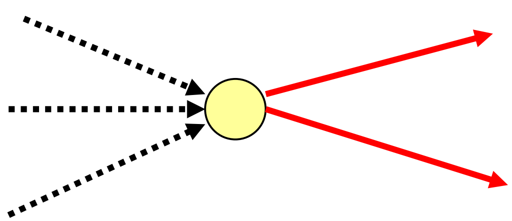
- 8.7.1 AOE网
- 8.7.2 关键路径

8.7.1 AOE网

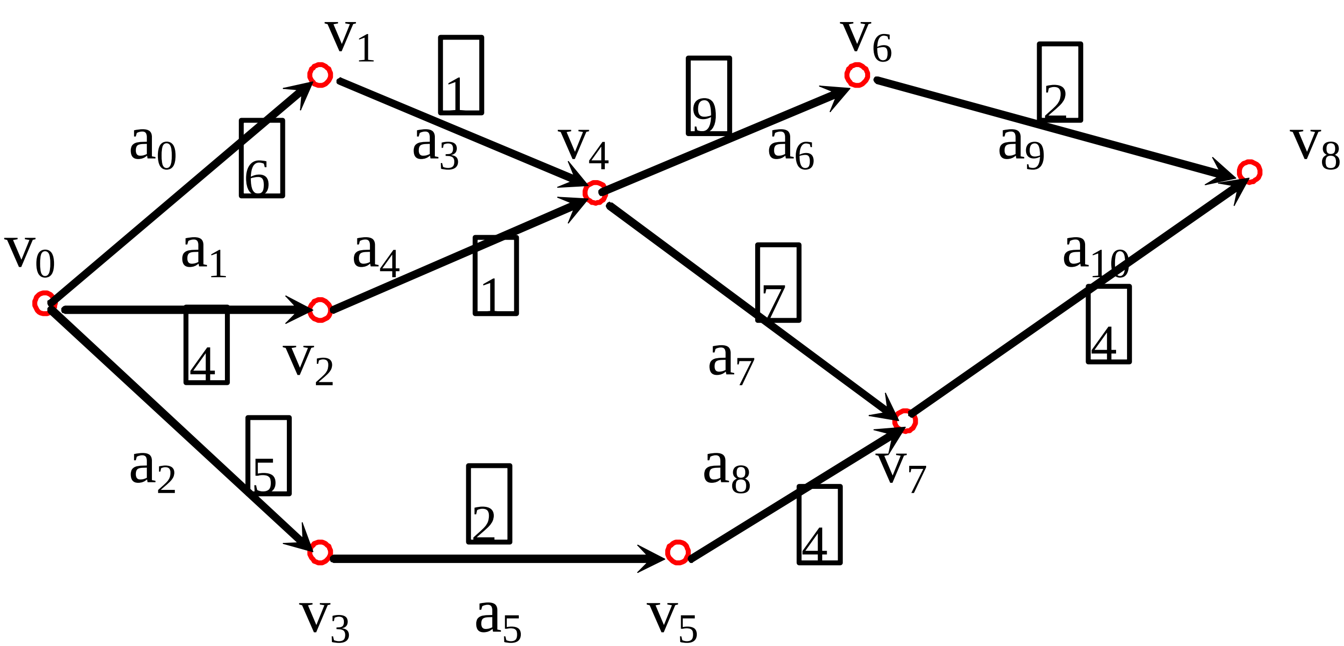
AOE网：如果在带权的有向图中，用顶点表示事件，用有向边表示活动，边上的权值表示活动持续的时间，则此带权的有向图称为**边活动网** (Activity On Edge network，简称AOE网)。

AOE网通常用于估计工程的完成时间。

顶点所表示的事件实际上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这样一种状态。



例：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这样一个状态；事件v_4表示活动a_3、a_4已经完成，活动a_6、a_7可以开始这个状态，事件v_8表示整个工程结束。如果权所表示的时间单位是天，则活动a_0需要6天完成，活动a_1需要4天完成，等等。整个工程一开始，活动a_0、a_1、a_2就可以并行进行，而活动a_3、a_4、a_5只有当事件v_1、v_2、v_3分别发生后才能进行，当活动a_9、a_{10}完成时，整个工程也就完成。



表示实际工程的AOE网应无回路，只有一个入度为0的起始顶点和一个出度为0的终止顶点。

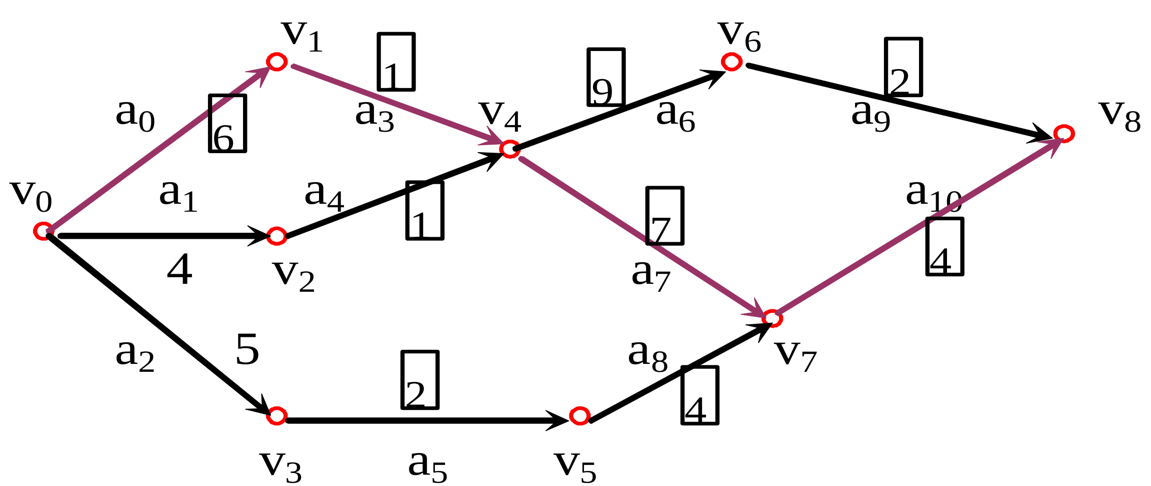
利用AOE网进行工程管理时，需要讨论以下两个问题：

1）完成整个工程至少需要多少时间？

2）哪些活动是影响工程进度的关键活动？

8.7.2 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。把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，也就是整个工程至少18天才能完成。减少关键活动的完成时间，则整个工程就有可能提前完成。

如何确定AOE网的关键路径?

首先定义几个变量

（1）<u>事件v_j可能的最早发生时间ee(j)</u>

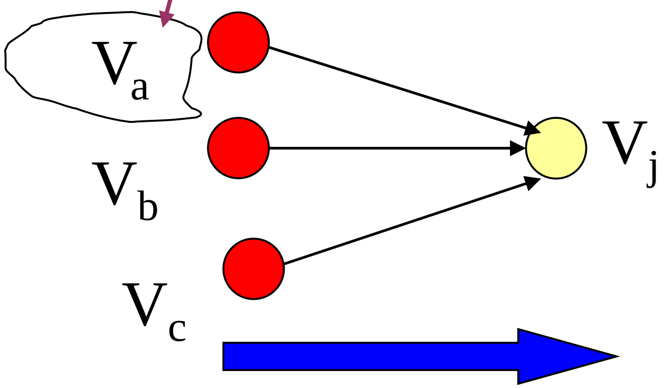
从开始点到v_j的最长路径长度。也是从v_j开始的活动能够开工的最早时间。只有进入v_j的活动$<v_i\ ,v_j>$都结束，v_j代表的事件才能发生。

$ee(0)=0$

$ee(j)=\max\{\ ee(i)+weight(<v_i\ ,v_j>)\ \}\quad <v_i\ ,v_j>\in T,\ 1\le j\le n-1$

T是所有以v_j为终点的入边的集合，$weight(<v_i\ ,v_j>)$为边$<v_i\ ,v_j>$的权。

<u>ee由前往后推算。</u>



（2）事件v_i允许的最迟发生时间le(i)

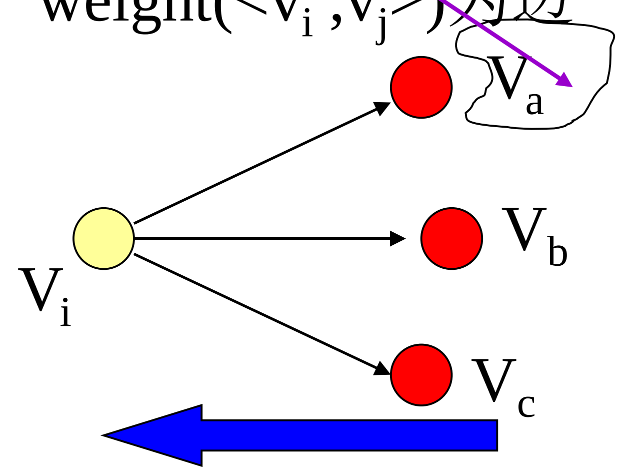
不推迟整个工期的前提下， v_i允许的最晚发生时间。

为了不拖延整个工期， v_i发生的最晚时间不得迟于其后继事件v_j的最晚发生时间减去活动$<v_i, v_j>$的持续时间。

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

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

S是所有以v_i为开始顶点的出边的集合，weight($<v_i, v_j>$)为边$<v_i, v_j>$的权。

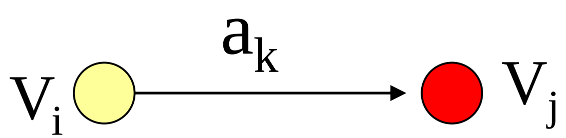


le由后往前推算。

（3）活动a_k=< v_i ,v_j>的最早开始时间ea(k)

只有事件v_i发生了，活动a_k才能开始。

ea(k)=ee(i)



（4）活动a_k=< v_i ,v_j>的最晚开始时间la(k)

事件v_j的最迟发生时间减去活动a_k的持续时间。

la(k)=le(j)－weight(<v_i,v_j>)

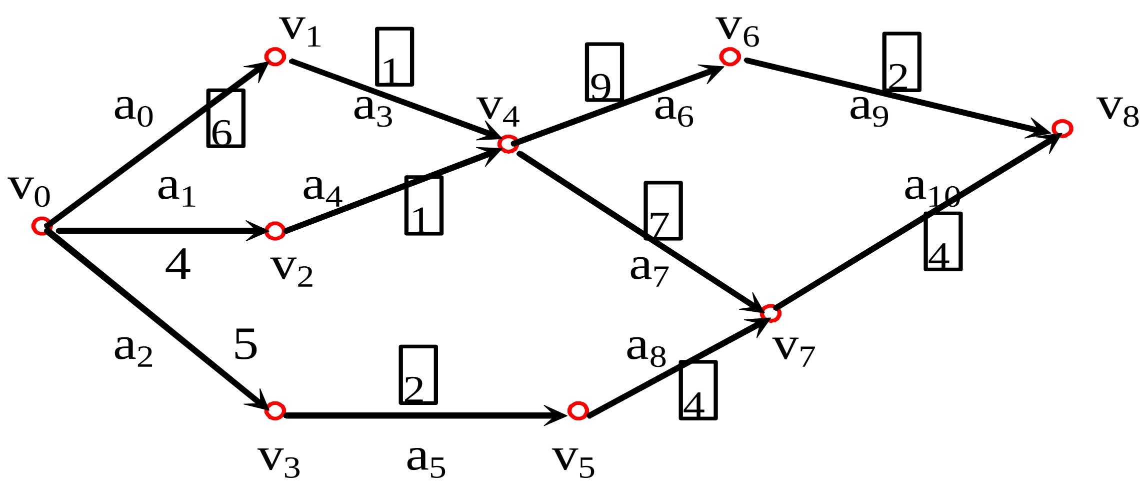
weight(<v_i,v_j>)为边<v_i,v_j>的权值。

la(k)－ea(k)：表示完成活动a_k的时间余量，是在不延误工期的前提下，活动a_k可以延迟的时间。

把ea(k)=la(k)的活动a_k称为关键活动。

关键路径由关键活动构成。

例题:求图中的AOE网的关键路径



按上述公式分别求出:

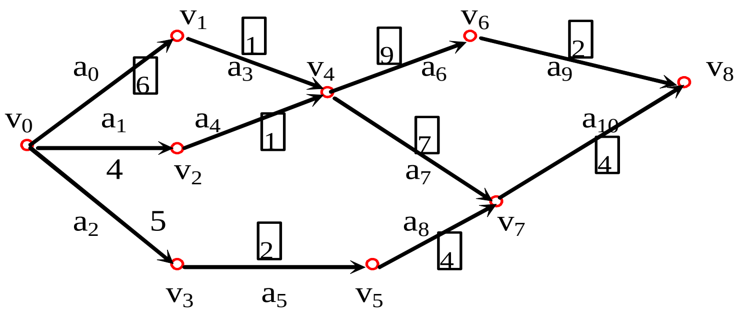
事件的最早发生时间： ee(i) (0≤i≤n-1）

事件的最迟发生时间： le(i) (0≤i≤n-1）

活动的最早开始时间： ea(k) (0≤k≤e-1)

活动的最晚开始时间： la(k) (0≤k≤e-1)

ee(0)=0



$ee(1)=ee(0)+weight(<v_0,v_1>)=0+6=6$

$ee(2)=ee(0)+ weight(<v_0,v_2>)=0+4=4$

$ee(3)=ee(0)+ weight(<v_0,v_3>)=0+5=5$

$ee(4)=max\{ee(1)+ weight(<v_1,v_4>), ee(2)+ weight(<v_2,v_4>)$

$=max\{6+1, 4+1\}=7$

$ee(5)=ee(3)+ weight(<v_3,v_5>)=5+2=7$

$ee(6)=ee(4)+ weight(<v_4,v_6>)=7+9=16$

$ee(7)=max\{ee(4)+ weight(<v_4,v_7>), ee(5)+ weight(<v_5,v_7>)$

$=max\{7+7, 7+4\}=14$

$ee(8)=max\{ee(6)+ weight(<v_6,v_8>), ee(7)+ weight(<v_7,v_8>)$

$=max\{16+2, 14+4\}=18$

$$le(8)=ee(8)=18$$

$$le(7)=ee(8)-weight(<v_7,v_8>)=18-4=14$$

$$le(6)=ee(8)-weight(<v_6,v_8>)=18-2=16$$

$$le(5)=ee(7)-weight(<v_5,v_7>)=14-4=10$$

$$le(4)=min\{le(7)—weight(<v_4,v_7>),\ le(6)—weight(<v_4,v_6>)\}$$

$$=min\{14-7,\ 16-9\}=7$$

$$le(3)=le(5)-weight(<v_3,v_5>)=10-2=8$$

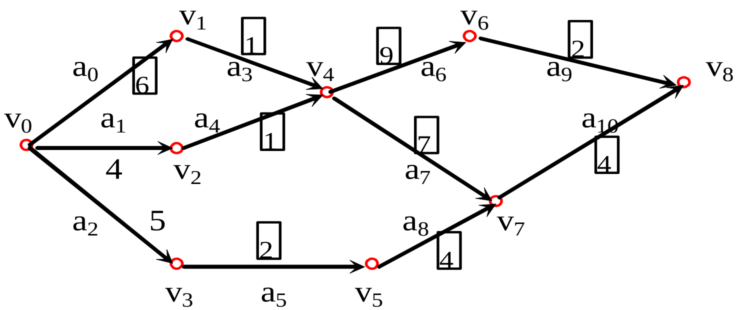
$$le(2)=le(4)-weight(<v_2,v_4>)=7-1=6$$

$$le(1)=le(4)-weight(<v_1,v_4>)=7-1=6$$

$$le(0)=min\{le(1)—weight(<v_0,v_1>),\ le(2)—weight(<v_0,v_2>),$$

$$le(3)—weight(<v_0,v_3>)\}$$

$$=min\{6-6,\ 6-4,\ 8-5\}=0$$



ea(0)=ee(0)=0

ea(1)=ee(0)=0

ea(2)=ee(0)=0

ea(3)=ee(1)=6

ea(4)=ee(2)=4

ea(5)=ee(3)=5

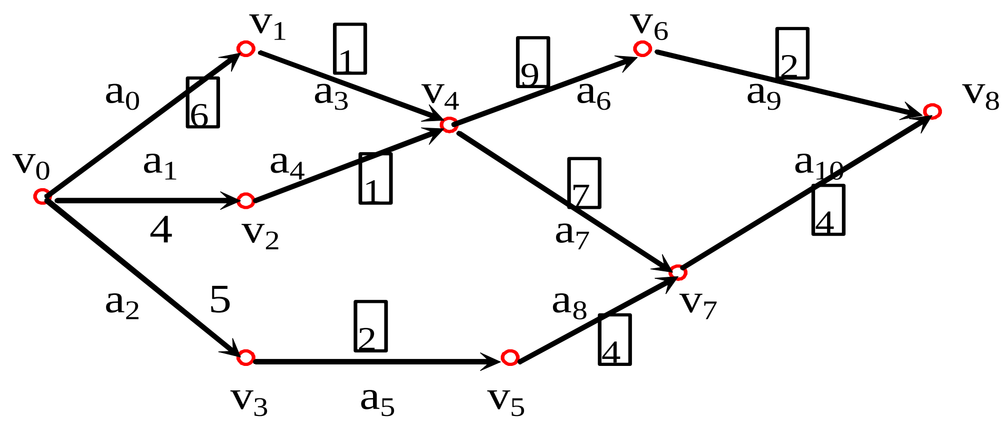
ea(6)=ee(4)=7

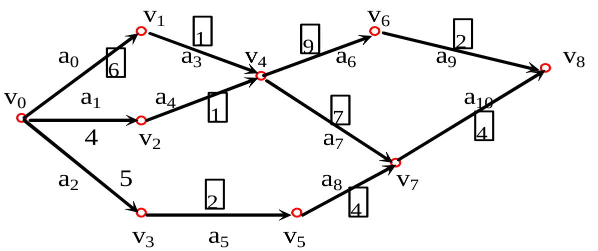
ea(7)=ee(4)=7

ea(8)=ee(5)=7

ea(9)=ee(6)=16

ea(10)=ee(7)=14





$la(0)=le(1)-weight(<v_0,v_1>)=6-6=0$

$la(1)=le(2)-weight(<v_0,v_2>)=6-4=2$

$la(2)=le(3)-weight(<v_0,v_3>)=8-5=3$

$la(3)=le(4)-weight(<v_1,v_4>)=7-1=6$

$la(4)=le(4)-weight(<v_2,v_4>)=7-1=6$

$la(5)=le(5)-weight(<v_3,v_5>)=10-2=8$

$la(6)=le(6)-weight(<v_4,v_6>)=16-9=7$

$la(7)=le(7)-weight(<v_4,v_7>)=14-7=7$

$la(8)=le(7)-weight(<v_5,v_7>)=14-4=10$

$la(9)=le(8)-weight(<v_6,v_8>)=18-2=16$

$la(10)=le(8)-weight(<v_7,v_8>)=18-4=14$

$$la(0) - ea(0) = 0 - 0 = 0$$

$$la(1) - ea(1) = 2 - 0 = 2$$

$$la(2) - ea(2) = 3 - 0 = 3$$

$$la(3) - ea(3) = 6 - 6 = 0$$

$$la(4) - ea(4) = 6 - 4 = 2$$

$$la(5) - ea(5) = 8 - 5 = 3$$

$$la(6) - ea(6) = 7 - 7 = 0$$

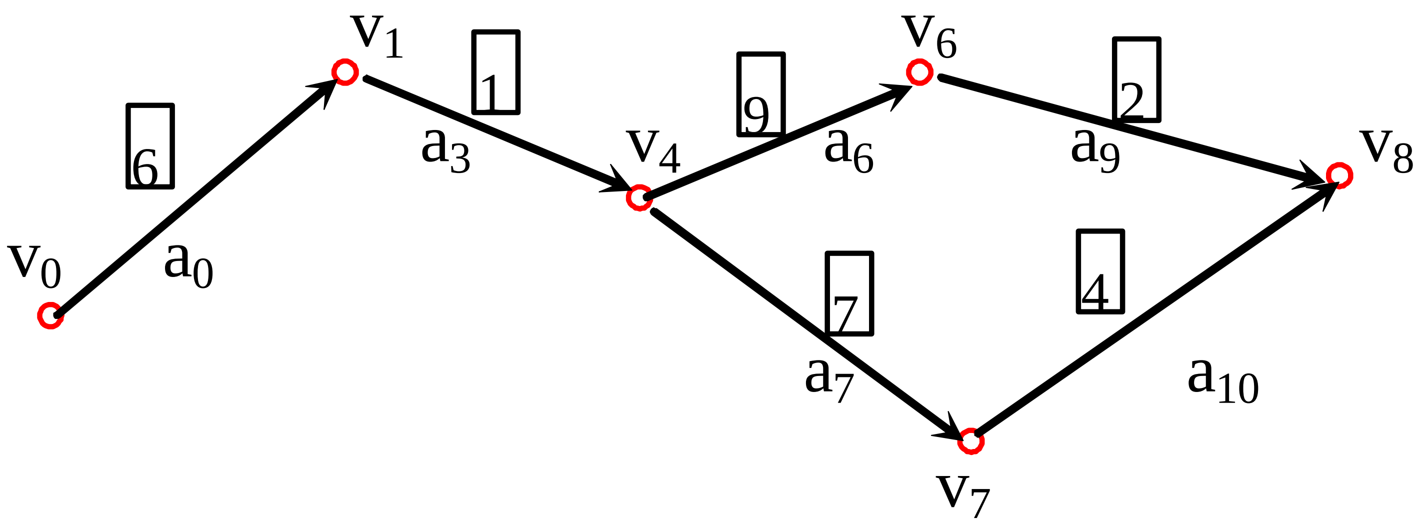
$$la(7) - ea(7) = 7 - 7 = 0$$

$$la(8) - ea(8) = 10 - 7 = 3$$

$$la(9) - ea(9) = 16 - 16 = 0$$

$$la(10) - ea(10) = 14 - 14 = 0$$

结果：活动a_0, a_3, a_6, a_7, a_9, a_{10}为关键活动



讨论：

1）延误关键活动的持续时间，推迟整个工程时间；
2）缩短非关键活动的持续时间，对整个工程时间无影响；
3）只有缩短关键活动的持续时间，才能缩短整个工程时间，但可能引起关键活动的变化；
4）缩短某些关键活动的持续时间，并不一定提前工程完成时间；

关键路径算法

存储结构：AOE网采用邻接表（出边表）表达，四个数组ee、 le、 ea和la分别表示：

- ee - 事件的可能最早发生时间
- le - 事件的允许最迟发生时间
- ea – 活动的最早开始时间
- la – 活动的最晚开始时间

关键活动用一对相关顶点输出。

计算ee(j)必须在顶点v_j所有前驱顶点的最早发生时间都已经求出的前提下进行，而计算le(i)必须在顶点v_i所有后继顶点的最迟发生时间都已经求出的前提下进行，因此，顶点序列必须是一个拓扑序列。首先检查是否有环，如无则找出ea[k] = la[k]的关键活动。

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)

本章小结：

图是一种复杂的非线性结构。

本章介绍了图的基本概念，图的相邻矩阵和邻接表两种常用的存储表示方法，讨论了图的周游、最小生成树、最短路径、拓扑排序及关键路径等问题，并给出了相应的算法。

重点是掌握图的存储表示和各种算法的基本思想。

1）图：无向图，有向图， 边（弧）带权=>网络

2）图的存储：邻接矩阵，邻接表（逆邻接表）

通过存储表示，顶点入度、出度计算

无向图的邻接多重表，有向图的十字链表

3）图的遍历：DFS(深度优先)、BFS(广度优先)

不同存储表示（邻接矩阵、邻接表）下的实现算法。

4）最小生成树：邻接矩阵存储表示下的Prim算法、kruskal算法

5）最短路径： 邻接矩阵存储表示下的问题求解。

Dijkstra算法用于求一个顶点到其它顶点的最短路径

Floyd算法用于求每一对顶点间的最短路径

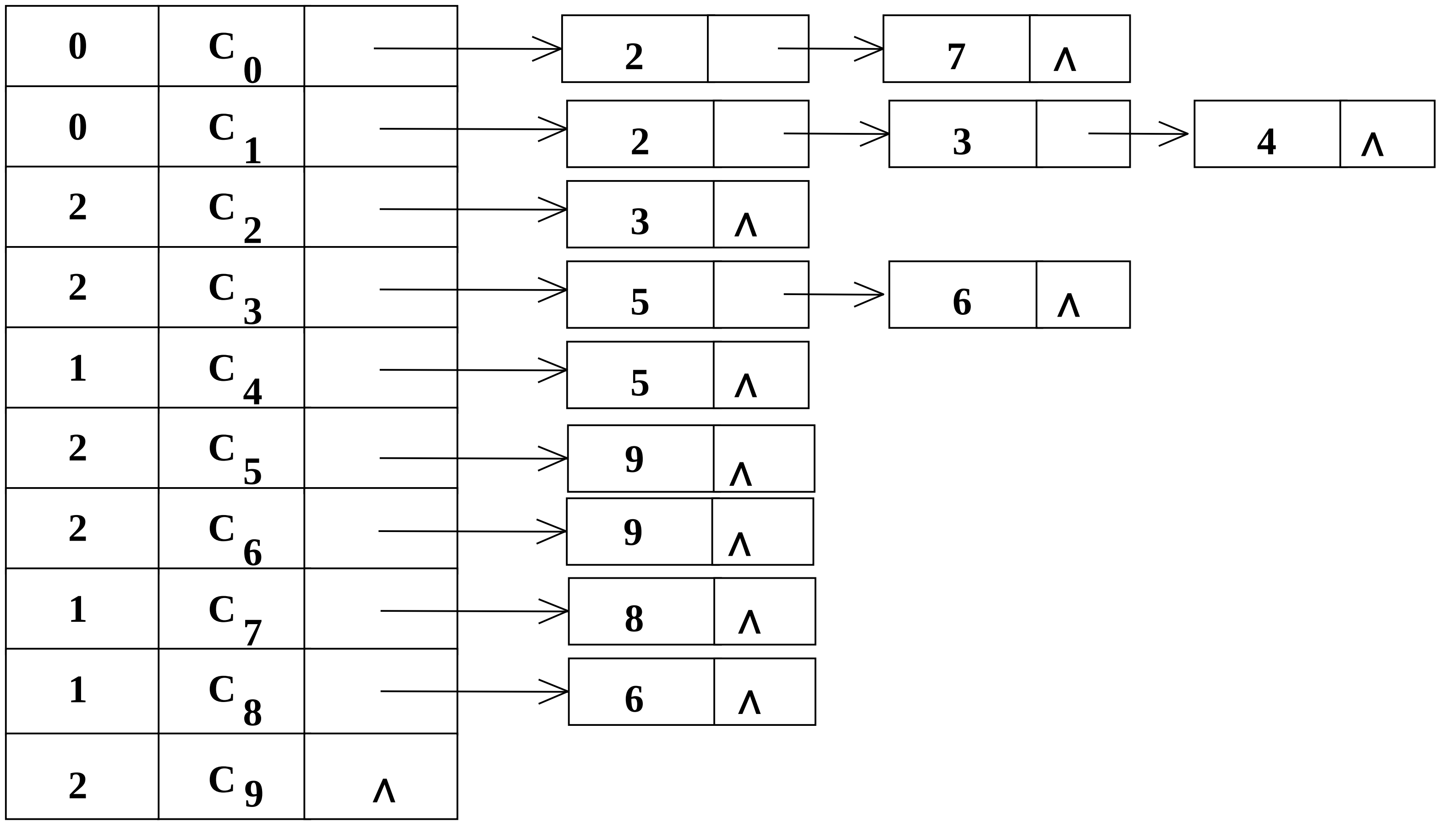
6）拓扑排序

AOV网，拓扑排序

7）关键路径

AOE网，关键路径

习题：书p317~319(无书面作业，但需要思考、复习)


0 C0 2 7 ∧
0 C1 2 3 4 ∧
2 C2 3 ∧
2 C3 5 6 ∧
1 C4 5 ∧
2 C5 9 ∧
2 C6 9 ∧
1 C7 8 ∧
1 C8 6 ∧
2 C9 ∧