数据结构与算法

Data Structures and Algorithms

第四部分 图

回顾：图--3

1. 最小生成树（连通带权无向图）

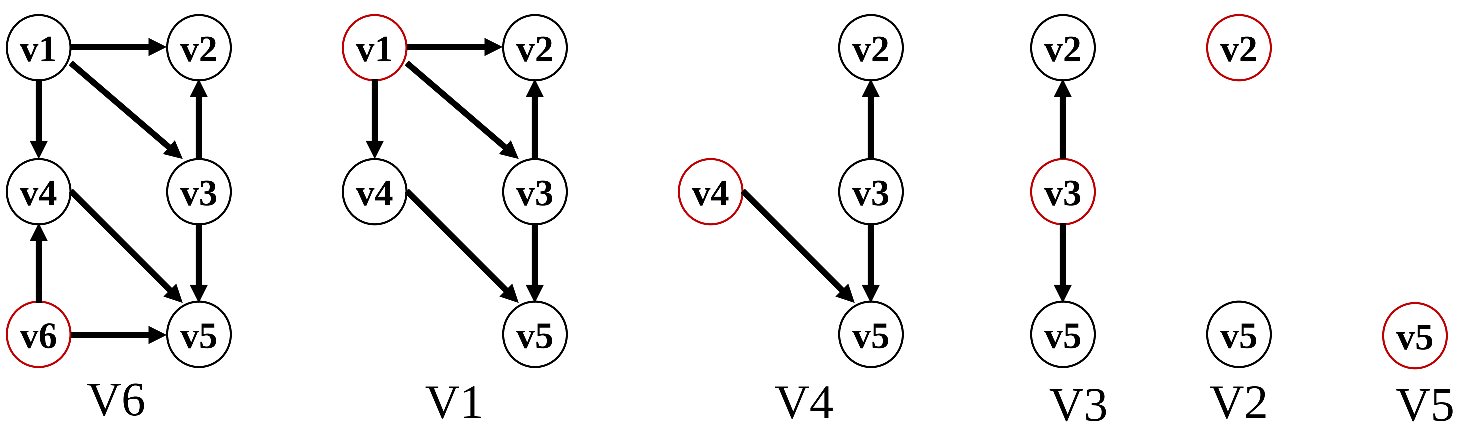
算法一： **Prim** （普里姆算法）

（1）**CloseST[*i*]**为**U**中的一个顶点

（2）边**(*i* , CloseST[*i*])**具有最小的权**LowCost[*i*]**

算法二： **Kruskal** （克鲁斯卡尔算法）

2. 拓扑排序（有向无环图，**AOV**网）



考纲内容

（一）图的基本概念

（二）图的存储结构及基本操作

邻接矩阵法；邻接表法；邻接多重表；十字链表

（三）图的遍历

深度优先搜索；广度优先搜索

（四）图的应用

最小（代价）生成树；拓扑排序；最短路径；关键路径

图

- 图的基本概念
- 图的存储结构：邻接矩阵法；邻接表法；邻接多重表；十字链表
- 图的两种遍历方式：深度优先搜索；广度优先搜索
- 图的应用
  - 最小生成树：**Prim**算法；**Kruskal**算法；
  - 拓扑排序：**AOV**网；
  - 关键路径：**AOE**网；
  - 最短路径：**Dijkstra**算法；**Floyd**算法。

4.8 拓扑分类 有向无环图及其应用

拓扑排序算法概要

增加一个存放各顶点入度的数组，并设置一个线性表，如栈。

(1) 扫描入度数组, 将入度为零的顶点入栈;

(2) while(栈非空) {

- 弹出栈顶元素vi并输出;
- 检查vi的出边表, 将每条出边<vi,vj>的终点vj的入度减1;
- 若vj的入度减至0, 则vj入栈; }

(3) 若输出的顶点数小n, 则“有回路”；否则正常结束。

拓扑分类算法—应用栈

```
Status   Topologicalsort( ALGRAPH  G )
{   FindInDegree( G, InDegree ) ;//计算每个顶点的入度
     MakeNull( S ) ;
     for( v=0; v<n ; ++v )
         if ( !InDegree[v] )  push( v, S ) ;// 入度为零，入栈
     count = 0 ;
     while ( !Empty( S ) )//栈非空
         {  v = Pop ( S ) ; printf( v );  ++count ; //出栈，顶点个数+1
            for( 邻接于 v 的每个顶点 w )
                {
                   if( !(--InDegree[w])) push(w, S) ;// 邻接顶点度-1；若为0，入栈
                }
         }
      if ( count < n )  count<<"图中有环路" ;
      else return OK;
}
```

拓扑分类算法—应用队列

```
Status   Topologicalsort( L )
{   QUEUE  Q ;
    MakeNull( Q ) ;
    for( v=1; v<=n ; ++v )
        if ( InDegree[v] = 0 )  EnQueue( v, Q ) ;
    nodes = 0 ;
    while ( !Empty( Q ) )
        {  v = Front(Q) ;
           DeQueue( Q ) ;
           cout << v ;   nodes ++ ;
           for( 邻接于 v 的每个顶点 w )
             if( !(--InDegree[w])) EnQueue(w,Q) ;
         }
     if ( nodes < n )  cout<<"图中有环路"  ;
}
```

拓扑分类算法— DFS遍历生成拓扑序列

```
Void topodfs  ( v )//深度优先遍历的修改
{   Push( v, S ) ;
    mark[v]=TRUE;
    for ( L[v] 中的每一个顶点w)  do //邻接表中顶点
        if ( mark[w] = FALSE )
              topodfs ( w ) ;
    printf ( top( S ) ) ;
    POP ( S ) ;
}
```

```
Void  dfs-topo ( GRAPH  L )
{  MakeNull( S );
   for( u=1;u<=n;u++)
       mark[u]=FALSE;
   for( u=1;u<=n;u++)
       if ( !mark[u] )
              topodfs( u ) ;
}
```

思想：借助栈，在DFS中，把第一次遇到的顶点入栈，到达某一顶点递归返回，从栈中弹出顶点并输出。

拓扑序列的逆序列

有向无环图的应用（重点）

有向无环图在工程计划和管理方面应用广泛。几乎所有的工程都可分为若干个称作“活动”的子工程， 并且这些子工程之间通常受着一定条件的约束。

例如：某些子工程必须在另外的一些子工程完成之后才能开始。对整个工程和系统，人们主要关心的是两方面的问题：

（1）工程能否顺利进行； ⇐ 拓扑排序

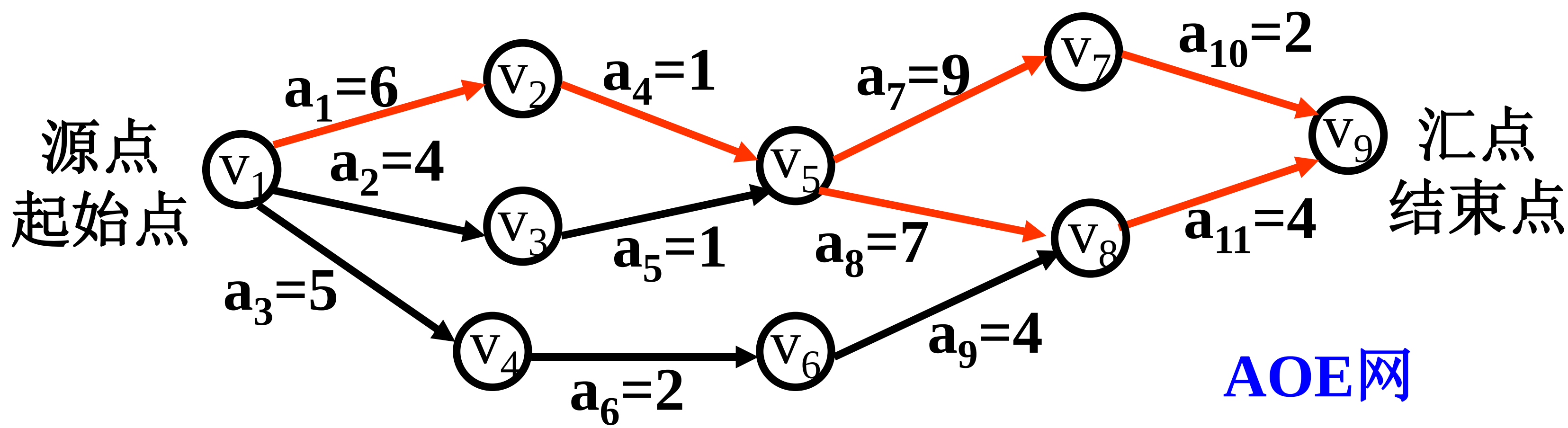
（2） 完成整个工程所必须的最短时间。 ⇐ 关键路径

哪些子工程项将影响整个工程完成的关键工程？

4.9 关键路径

AOE网（Activity On Edge Network）

在带权的有向无环图中，用顶点表示事件，用弧表示活动，弧上的权表示活动的开销（如持续时间），则称此有向图为边表示活动的网络，简称AOE网。

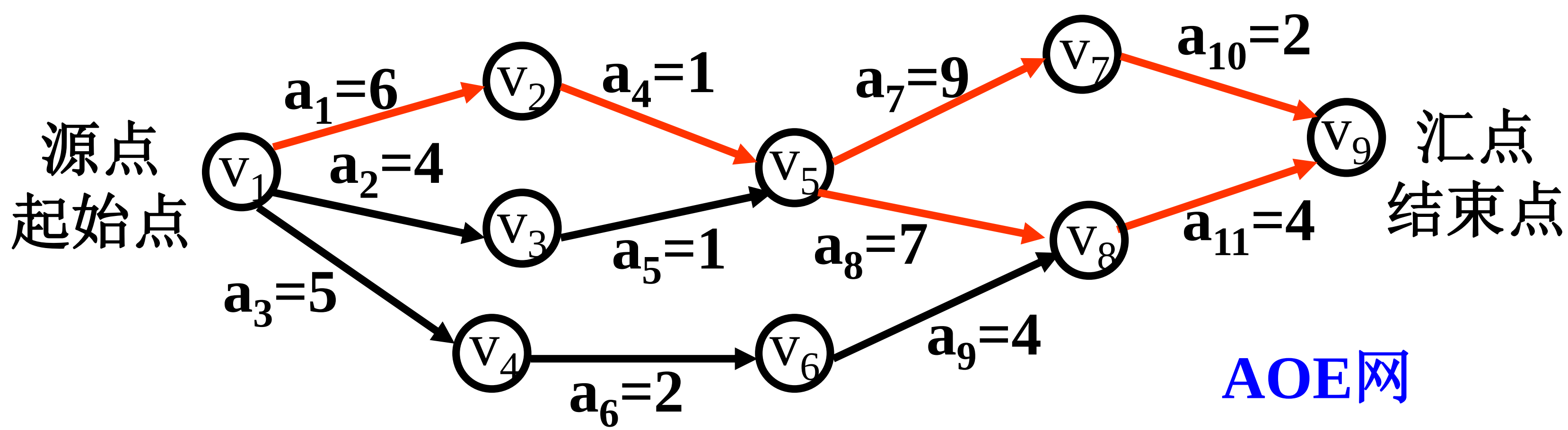


AOE网的性质：

◆只有在某个顶点所代表的事件发生后，从该顶点出发的各有向边代表的活动才能开始；

◆只有在进入某一顶点的各有向边代表的活动已经结束，该顶点所代表的事件才能发生；

◆表示实际工程计划的**AOE**网应该是无环的，并且存在唯一的入度为**0**的开始顶点（源点）和唯一的出度为**0**的结束点（汇点）。

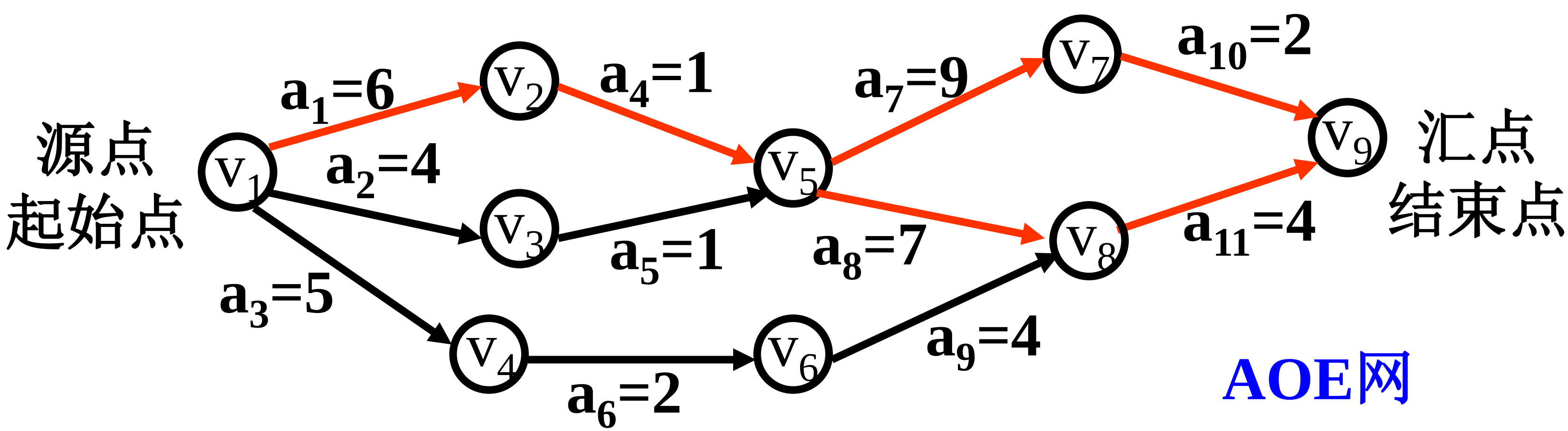


AOE网研究的主要问题:

如果用AOE 网表示一项工程，那么仅仅考虑各个子工程之间的优先关系还不够，更多地是关心整个工程完成的最短时间是多少，哪些活动的延迟将影响整个工程进度,而加速这些活动能否提高整个工程的效率,因此AOE网有待研究的问题是:

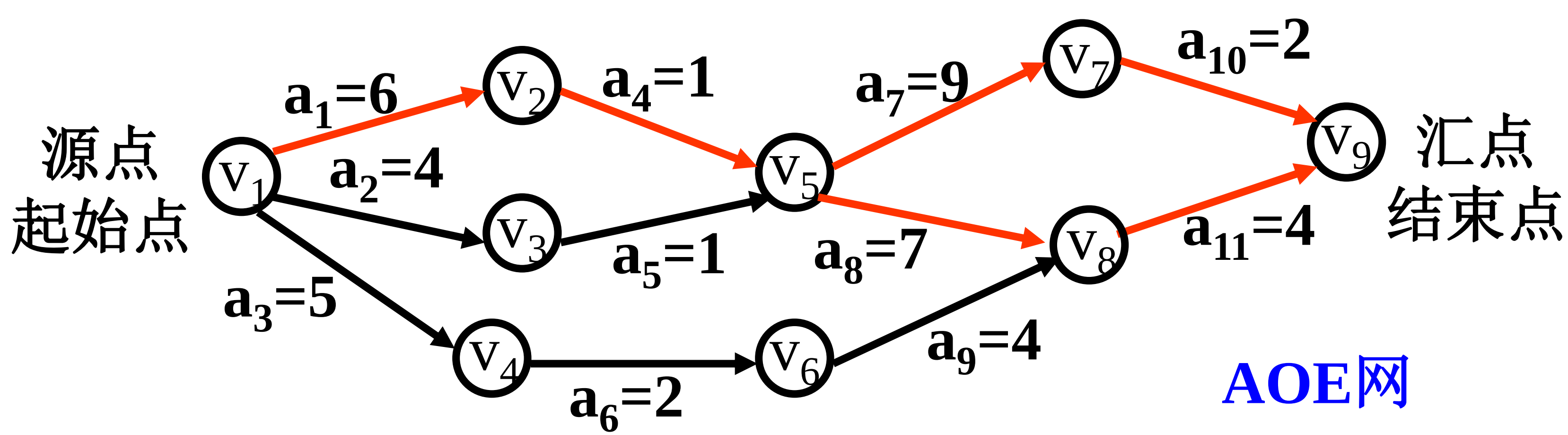
(1) 完成整个工程至少需要多少时间?

(2) 哪些活动是影响工程进度的关键活动?



路径长度、关键路径、关键活动：

- 路径长度：是指从源点到汇点路径上所有活动的持续时间之和。
- 关键路径：在AOE网中，由于有些活动可以并行，所以完成工程的最短时间是从源点到汇点的最大路径长度。因此，把从源点到汇点具有最大长度的路径称为关键路径。
- 一个AOE中，关键路径可能不只一条。
- 关键活动：关键路径上的活动称为关键活动。



关键路径和关键活动性质分析：（与计算关键活动有关的量）

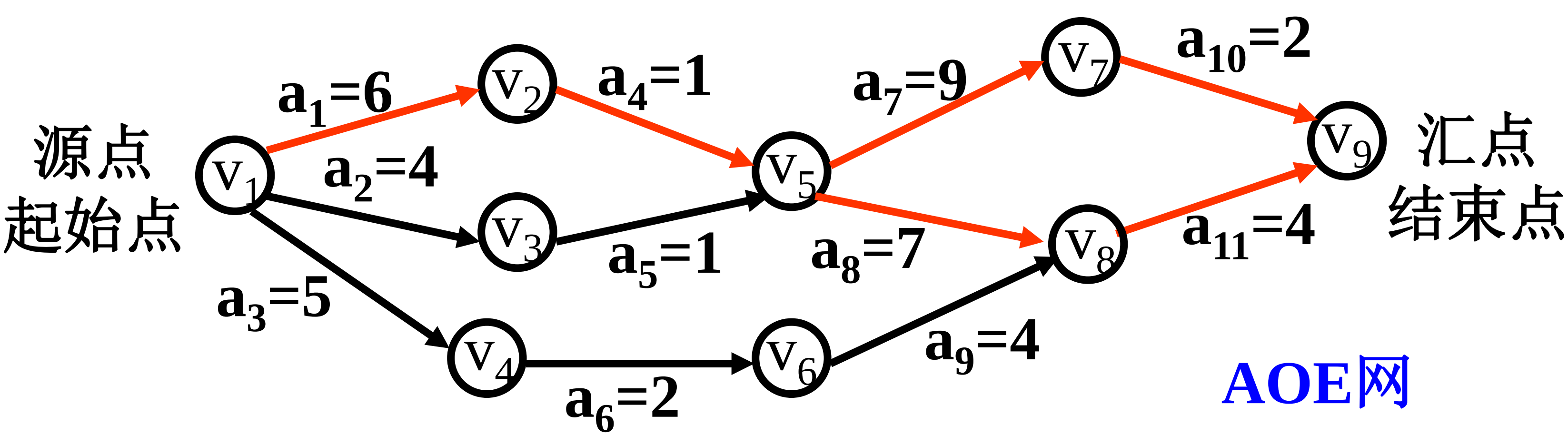
① 事件V_j 的最早可能发生时间*VE(j)*

是从源点V_1 到顶点V_j 的最长路径长度。

② 事件V_j的最迟发生时间*VL(j)*

是在保证汇点V_n在*VE(n)*时刻完成的前提下，事件V_j的允许的最迟开始时间。

在不推迟工期的情况下，一个事件最迟发生时间*VL(j)*应该等于汇点的最早发生时间*VE(n)*减去从V_j到V_n的最大路径长度。



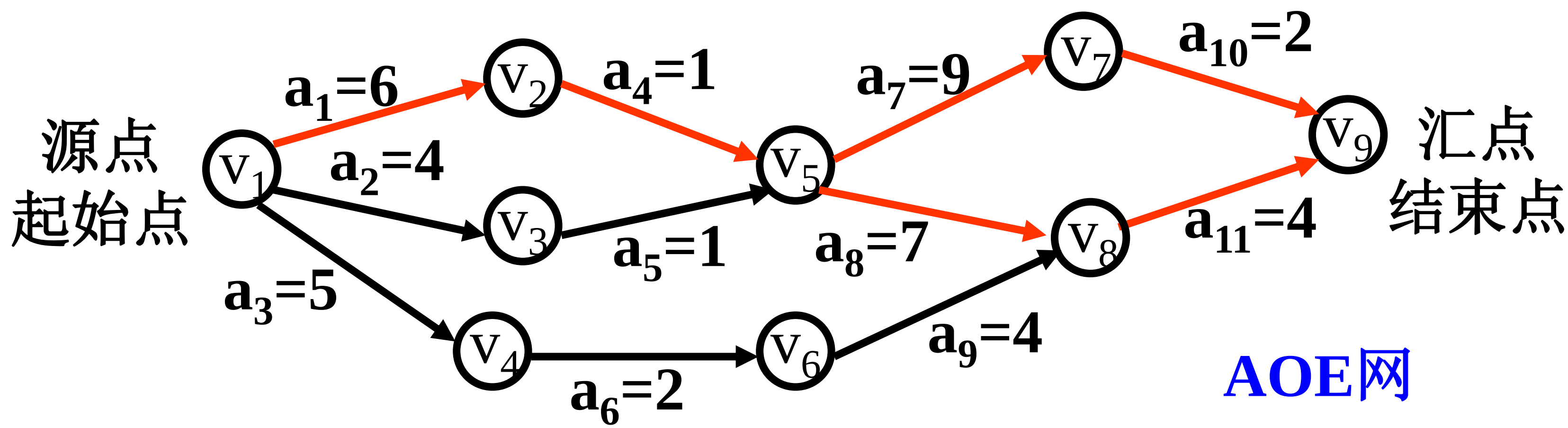
AOE网

③ 活动a_i 的最早可能开始时间 $E(i)$

事件V_j 最早可能发生时间$VE(j)$是从源点V_1 到顶点V_j 的最长路径长度。

设活动a_i在边< V_j, V_k>上, 则$E(i)$也是从源点V_1到顶点V_j 的最长路径长度。这是因为事件V_j发生表明以V_j为起点的所有活动a_i可以立即开始。因此,

$$E(i) = VE(j) \quad \ldots\ldots\ldots\ldots\ldots(1)$$



④ 活动a_i 的最迟允许开始时间 $L(i)$

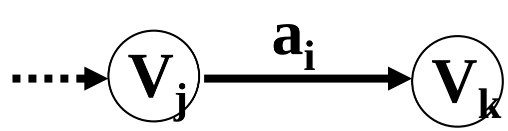
是指在不会引起工期延误的前提下，活动a_i允许的最迟开始时间。

因为事件V_k发生表明以V_k为终点的入边所表示的所有活动均已完成，所以事件V_k的最迟发生时间$VL(k)$也是所有以V_k为终点的入边$<V_j, V_k>$所表示的活动a_i能够最迟结束的时间。

显然，为不推迟工期，活动a_i的最迟开始时间$L(i)$应该是V_k的最迟完成时间$VL(k)$减去a_i的持续时间，即：

$$L(i) = VL(k) - ACT[j][k] \quad \ldots\ldots\ldots\ldots\ldots (2)$$

其中, ACT[j][k]是活动a_i 的持续时间（$< V_j, V_k>$ 上的权）。



⑤时间余量 *L(i) - E(i)*

*L(i) - E(i)*表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。

关键路径上的活动都满足：*L(i) = E(i)* ……………(3)

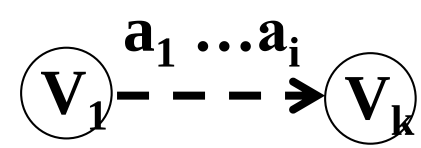
*L(i) = E(i)*表示活动是没有时间余量的关键活动。

由上述分析知，为找出关键活动，需要求各个活动的*E(i)*与*L(i)*，以判别一个活动a_i是否满足*L(i) = E(i)*。 *E(i)*和*L(i)*可有公式 (1)和(2)。而*VE(k)* 和*VL(k)*可由拓扑分类算法得到。

利用拓扑分类算法求关键路径和关键活动。

利用拓扑排序算法求关键路径和关键活动：

◆ **Step 1**(前进阶段)：从源点V_1出发，令$VE(1) = 0$，按拓扑序列次序求出其余各顶点**事件的最早发生时间**：



$$VE(k) = \max_{j \in T}\{ VE(j)+ACT[j][k] \}$$

其中T是以顶点V_k为尾的所有边的头顶点的集合$(2 \le k \le n)$

如果网中有回路，不能求出关键路径则算法中止；否则转Step2。

◆ **Step 2**(回退阶段)：从汇点V_n出发，令$VL(n) = VE(n)$，按逆拓扑有序求其余各顶点**事件的最晚发生时间**：

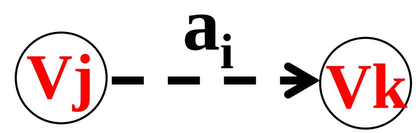
$$VL(j) = \min_{k \in S}\{ VL(k)-ACT[j][k] \}$$

其中S是以顶点V_j为头的所有边的尾顶点的集合$(2 \le j \le n-1)$。

◆ **Step 3**:

求每一项活动a_i的最早开始时间:

$$E(i) = VE(j)$$



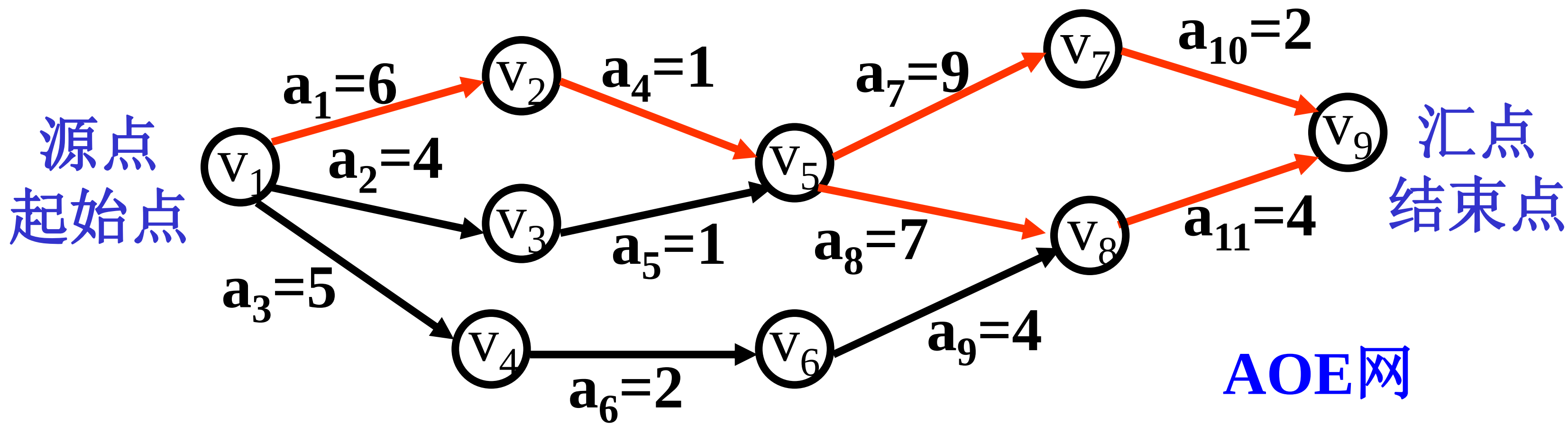
最晚开始时间:

$$L(i) = VL(k) - ACT[j][k]$$

若某条边满足$E(i) = L(i)$，则它是关键活动。

- ✓ 为了简化算法, 可以在求关键路径之前已经对各顶点实现拓扑排序, 并按拓扑有序的顺序对各顶点重新进行编号。
- ✓ 不是任意一个关键活动的加速一定能使整个工程提前。
- ✓ 想使整个工程提前，要考虑各个关键路径上所有关键活动。

【例4-9】关键路径1



事件	*VE*	*VL*	*活动*	*E(i)*	*L(i)*	*L(i)-E(i)*
v_1	0	0	a1	0	0	0
v_2	6	6	a2	0	2	2
v_3	4	6	a3	0	3	3
v_4	5	8	a4	6	6	0
v_5	7	7	a5	4	6	2
v_6	7	10	a6	5	8	3
v_7	16	16	a7	7	7	0
v_8	14	14	a8	7	7	0
v_9	18	18	a9	7	10	3
			a10	16	16	0
			a11	14	14	0

注：

VE(i)：事件最早可能发生时间

源点到达该事件的最长路径

VL(i)：事件最迟发生时间

VE(n) - maxL_{ik}

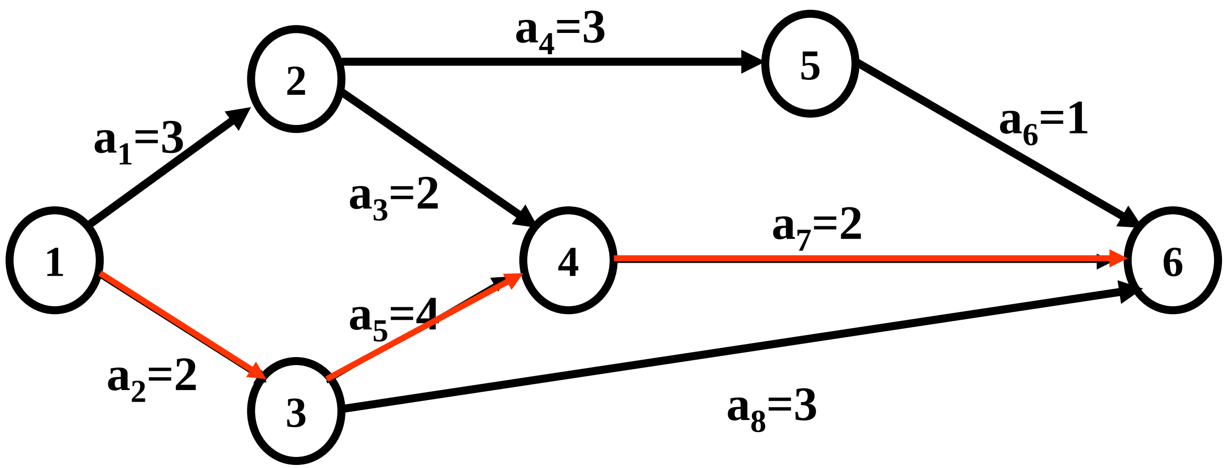
E(i)：活动最早可能开始时间

VE(活动起点)

L(i)：活动最迟允许开始时间

VL(活动终点) - a_i

【例4-10】关键路径2



顶点	*VE(i)*	*VL(i)*	*活动*	*E(i)*	*L(i)*	*L(i)-E(i)*
1	0	0	a1	0	1	1
2	3	4	a2	0	0	0
3	2	2	a3	3	4	1
4	6	6	a4	3	4	1
5	6	7	a5	2	2	0
6	8	8	a6	6	7	1
			a7	6	6	0
			a8	2	5	3

注：

事件最早可能发生时间***VE(i)***

源点到达该事件的最长路径

事件最迟发生时间***VL(i)***

***VE(n)-maxL**$_{ik}$*

活动最早可能开始时间 ***E(i)***

VE(活动起点)

活动最迟允许开始时间 ***L(i)***

***VL(活动终点)-a**$_i$*

算法实例

```
Status TopologicalOrder(ALGraph G, Stack &T)
{
    // 有向网G采用邻接表存储结构，求各顶点事件的最早发生时间ve(全局变量)。
    // T为拓扑序列定点栈，S为零入度顶点栈。
    // 若G无回路，则用栈T返回G的一个拓扑序列，且函数值为OK，否则为ERROR。
    Stack S;
    int count=0,k;
    char indegree[40];
    ArcNode *p;
    InitStack(S);
    FindInDegree(G, indegree);  // 对各顶点求入度indegree[0..vernum-1]
    for (int j=0; j<G.vexnum; ++j)     // 建零入度顶点栈S
        if (indegree[j]==0)
            Push(S, j);  // 入度为0者进栈
    InitStack(T);//建拓扑序列顶点栈T
    count = 0;
    for(int i=0; i<G.vexnum; i++)
        ve[i] = 0;  // 初始化
    while (!StackEmpty(S))
    {
        Pop(S, j);  Push(T, j);  ++count;        // j号顶点入T栈并计数
        for (p=G.vertices[j].firstarc;  p;  p=p->nextarc)
        {
            k = p->adjvex;             // 对j号顶点的每个邻接点的入度减1
            if (--indegree[k] == 0) Push(S, k);   // 若入度减为0，则入栈
            if (ve[j]+p->info > ve[k])  ve[k] = ve[j]+p->info;
        }//for  *(p->info)=dut(<j,k>)
    }
    if(count<G.vexnum)
        return ERROR;  // 该有向网有回路
    else
        return OK;
}
```

```
Status CriticalPath(ALGraph G)
{
    // G为有向网，输出G的各项关键活动。
    Stack T;
    int a,j,k,el,ee,dut;
    char tag;
    ArcNode *p;
    if (!TopologicalOrder(G, T))
        return ERROR;
    for(a=0; a<G.vexnum; a++)
        vl[a] = ve[G.vexnum-1];     // 初始化顶点事件的最迟发生时间
    while (!StackEmpty(T))        // 按拓扑逆序求各顶点的vl值
        for (Pop(T, j), p=G.vertices[j].firstarc;  p;  p=p->nextarc)
        {
            k=p->adjvex;  dut=p->info;     //dut<j,k>
            if (vl[k]-dut < vl[j])
                vl[j] = vl[k]-dut;
        }
    for (j=0; j<G.vexnum; ++j)           // 求ee,el和关键活动
        for (p=G.vertices[j].firstarc;  p;  p=p->nextarc)
        {
            k=p->adjvex;dut=p->info;
            ee = ve[j];  el = vl[k]-dut;
            tag = (ee==el) ? '*' : ' ';
            printf(j, k, dut, ee, el, tag);   // 输出关键活动
        }
    return OK;
}
```

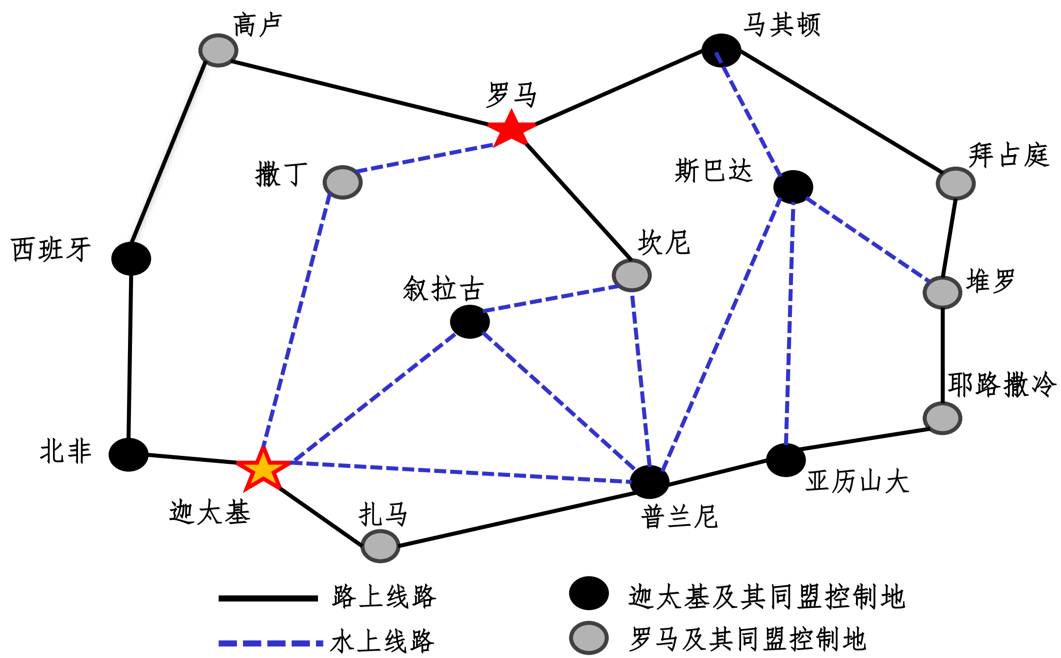
修改后的拓扑排序　　　　　　关键活动的求解

时间复杂度：O(n+e)

4.10 单源最短路径

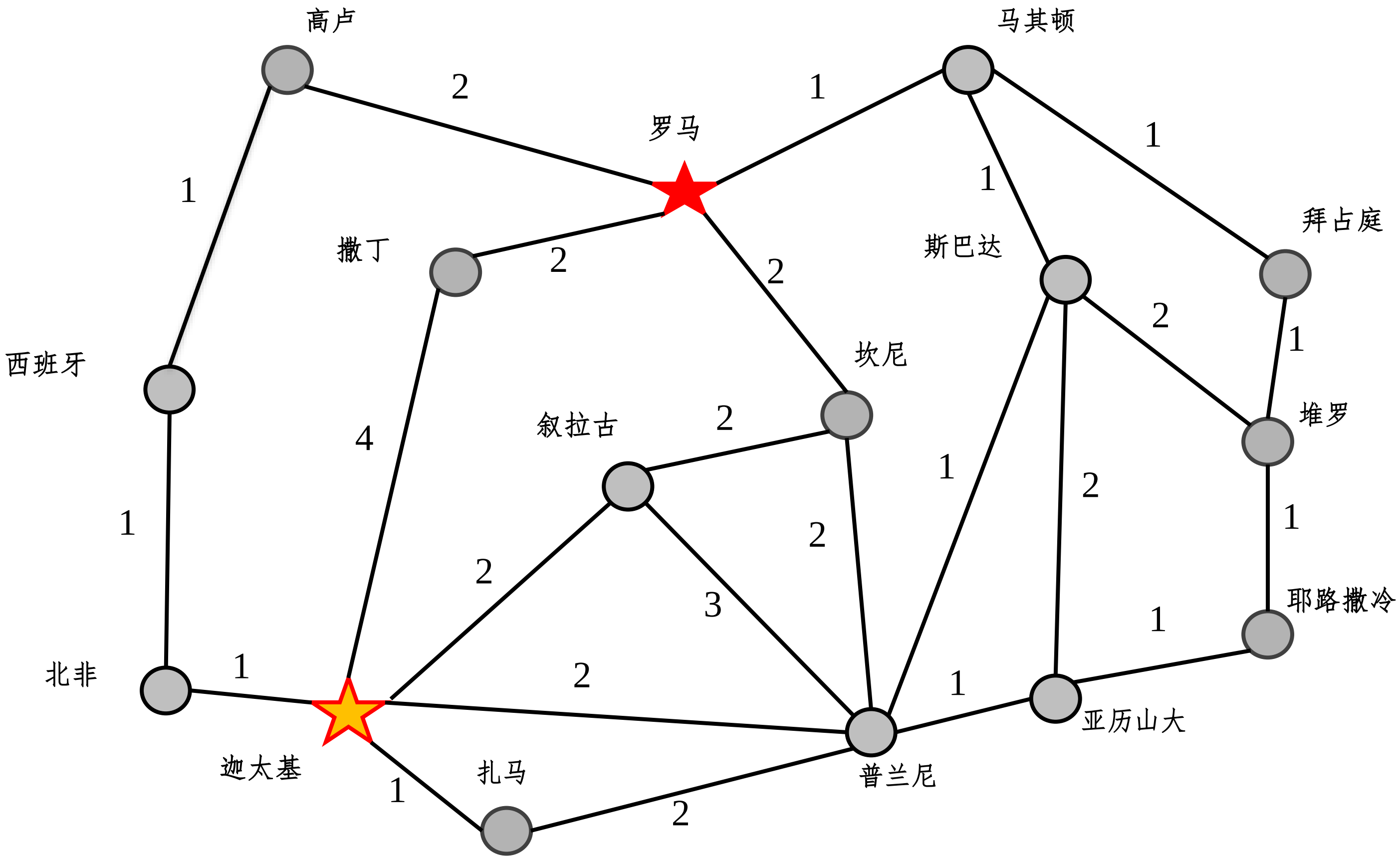
汉尼拔・巴卡（Hannibal Barca）；
公元前247年—前183年；
北非古国迦太基名将，军事家；
是欧洲历史最伟大四大军事统帅之一；
誓言终身与古罗马为敌；
被誉为战略之父。

公元前218年，
迦太基卓越的军事统帅汉尼拔决定进军罗马，准备最后解决罗马问题。选择行军路线成为首要问题。



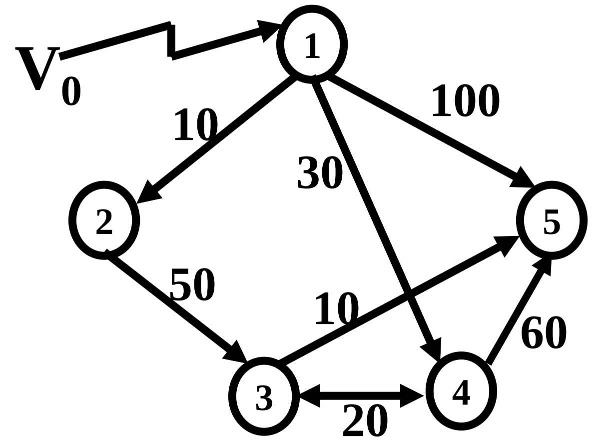
罗马帝国的海陆交通示意图

卢浮宫汉尼拔雕像



罗马帝国的海陆交通示意图—时间抽象版

4.10 单源最短路径



$$C\begin{pmatrix} \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & 20 & 10 \\ \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

约束：不能出现负的权重。

集合 S = { 1 }　　{1, 2, 3, 4, 5 }

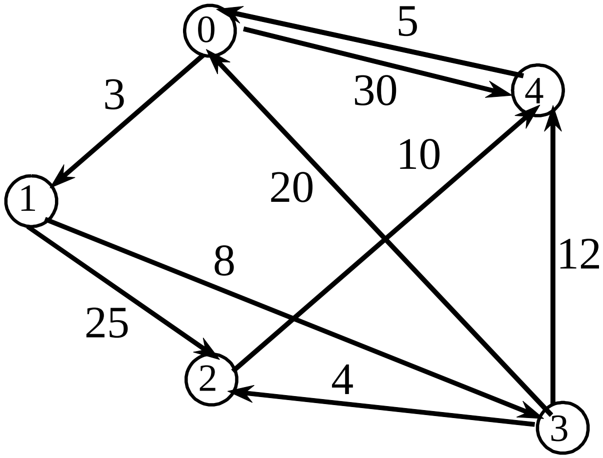
D

D[1]	D[2]	D[3]	D[4]	D[5]
∞	10	∞	30	100

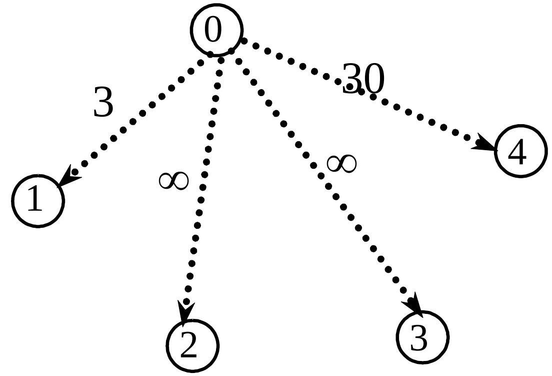
核心想法：以最短路径长度递增，逐次生成最短路径的算法

Dijkstra算法基本思想：

- 集合S的初值为S={1}；
- D为各顶点当前最小路径
- 从V-S中选择顶点w，使D[w]的值最小并将 w加入集合 S，则w的最短路径已求出；
- 调整其他各结点的当前最小路径
  D[k]=min{D[k], D[w]+C[w][k]}
- 直到S中包含所有顶点。

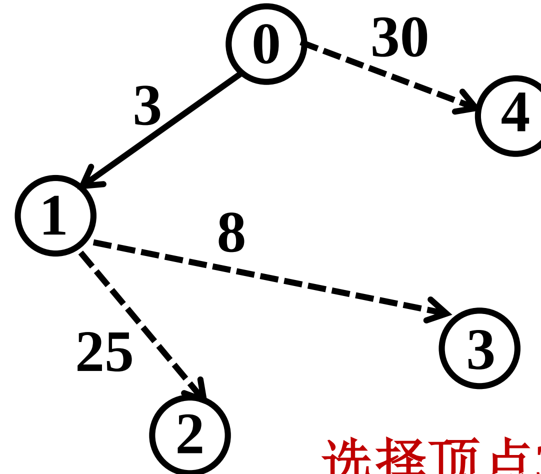


(a) 一个有向网点



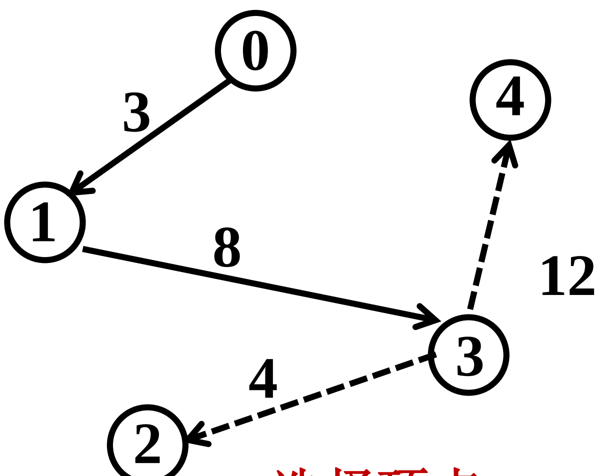
选择顶点1

(b) 源点 0 到其它顶点的初始距离



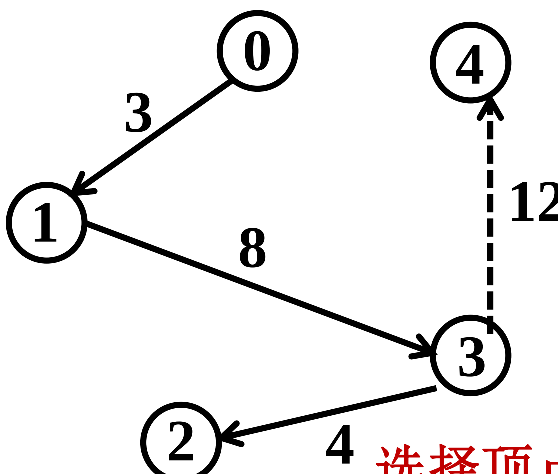
选择顶点3

(c) 第一次求得的结果



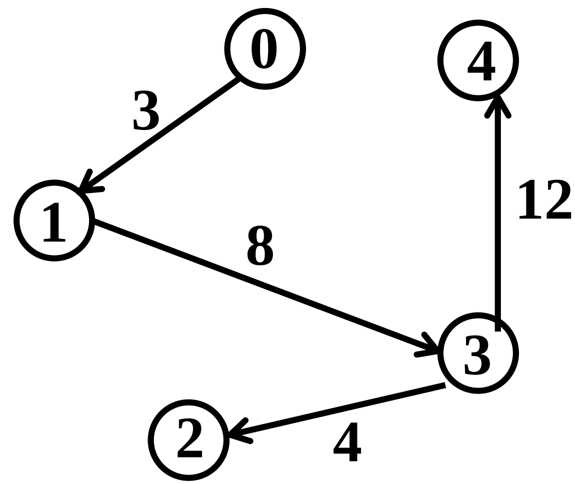
选择顶点2

(d) 第二次求得的结果

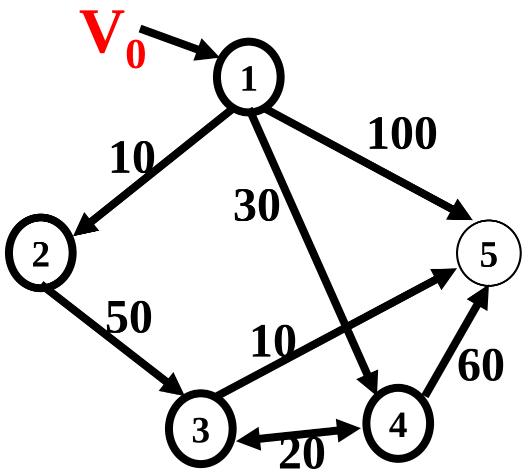


选择顶点4

(e) 第三次求得的结果



(f) 第四次求得的结果



循环	S	w	D[2]	D[3]	D[4]	D[5]
初态	{1}	-	10	∞	30	100
1	{1,2}	2	10	60	30	100
2	{1,2,4}	4	10	50	30	90
3	{1,2,4,3}	3	10	50	30	60
4	{1,2,4,3,5}	5	10	50	30	60

Dijkstra算法框架：

```
Void  Dijkstra( G )
{ S = { 1 } ;
  for( i=2; i<=n; i++ )
           D[i] = C[1][i];
  for( i=2; i<=n; i++ )
  {  从V-S中选出一个顶点w,使D[w]最小；
     S = S + {w} ;
     for ( V-S中的每一个顶点v )
     D[v]=min( D[v],  D[w]+C[w][v] );
  }
}
```

Dijkstra算法：时间复杂度O(n^2)

最小路径经过了哪些点？

顶点的集合

$$S[i]=\begin{cases}1 & i \in S\\ 0 & i \notin S\end{cases}$$

```
Void  Dijkstra(GRAPH  G, costtype D[MAXVEX+1] )
{  int  S[MAXVEX+1] ;
   for ( i=1 ; i<=n; i++ )
       {  D[i]=G[1][i] ;  S[i]=FALSE ;  }
   S [1]= TRUE ;
   for( i=2; i<=n; i++)
   {  w=mincost ( D, S ) ;
      S[w]=TRUE ;
      for ( k=2 ; k<= n ; k++ )//更新最短路径
          if ( S[k]!=TRUE )
             {   sum=D[w] + G[w][k] ;
                 if (sum < D[k] )   D[k] = sum ; }
   }
}
```

```
int  mincost ( D, S )
{
    temp = INFINITY ;
    w = 2 ;
    for ( i=2 ; i<=n ; i++ )
        if ( !S[i] && D[i]<temp)
           {  temp = D[i] ;
              w = i ;
           }
     return  w ;//返回顶点
}
```

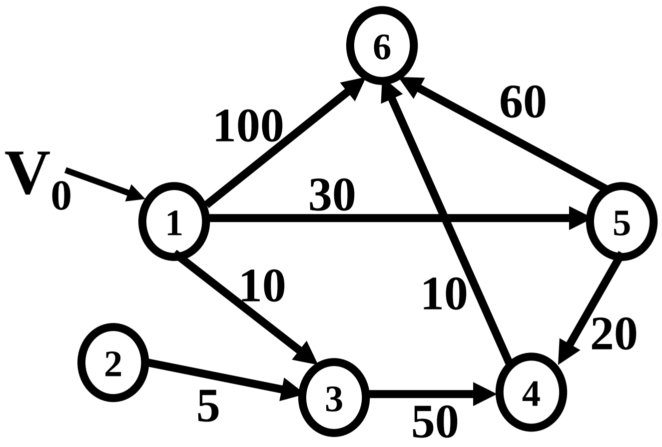
Dijkstra算法（带路径）：

```
Void  Dijkstra(GRAPH  G, costtype D[MAXVEX+1] )
{  int  S[MAXVEX+1],  P[MAXVEX+1];
   for ( i=1 ; i<=n; i++ )
       {  D[i]=G[1][i] ;  S[i]=FALSE ; P[i]=1 ; }
   S [1]= TRUE ;
   for( i=2; i<=n; i++)
   {  w=mincost ( D, S ) ;
      S[w]=TRUE ;
      for ( k=2 ; k<= n ; k++ )
          if ( S[k]!=TRUE )
            {   sum=D[w] + G[w][k] ;
                if (sum < D[k] ) {  D[k] = sum ;   P[k]=w; }
            }
   }
}
```

	P[1]	P[2]	P[3]	P[4]	P[5]
P	1	1	4	1	3

```
void DisplayPath(int *P, int v)
{   //源点到v的最短路径
    if(P[v]!=1)
    {
        DisplayPath(P,P[v]);
        printf("%d--",P[v]);
    }
}
```

【例4-11】



$$C = \begin{pmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

D

D[1]	D[2]	D[3]	D[4]	D[5]	D[6]
∞	∞	10	∞	30	100

循环	S	w	D[2]	D[3]	D[4]	D[5]	D[6]
初态	{1}	-	∞	10	∞	30	100
1	{1,3}	3	∞	10	60	30	100
2	{1,3,5}	5	∞	10	50	30	90
3	{1,3,5,4}	4	∞	10	50	30	60
4	{1,3,5,4,6}	6	∞	10	50	30	60
5	{1,3,5,4,6,2}	2	∞	10	50	30	60

P[1]	1
P[2]	1
P[3]	1
P[4]	5
P[5]	1
P[6]	4

Prim 与 Dijkstra 算法对比：完全不同的两个内容，不要弄混

区别	Prim算法构造最小生成树	Dijkstra算法构造单源最短路径
图类型	无向连通网	有向连通网
起始点	任选一个结点	有一个确定的起点（源点），其余为终点
连通顶点	连通所有顶点，且总造价最低	源点到各终点的两顶点间的最短路径
加入集合外顶点的修改方式	if (C[k][j] < LowCost[j] && LowCost[j] != INFINITY) { LowCost[j]=C[k][j]; CloseST[j]=k; }	sum=D[w] + G[w][v] ; if (sum < D[v]) { D[v] = sum ; p[v]=w; }
记录路径数组	无	有一个P数组记录从源点到各终点的路线
构成结果图	所构造的连通网的权值之和最小	一定是源点到终点的两路径权值最短
重复次数	重复n-1次	重复n-2次

4.11 每一对顶点间的最短路径

Floyd算法的基本思想：

- 假设求顶点v_i到顶点v_j的最短路径。如果从 v_i 到 v_j 存在一条长度为C[*i*][*j*]的路径，该路径不一定是最短路径，尚需进行 *n* 次试探。
- 首先考虑路径 (v_i, 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_j) 的路径长度取长度较短者为从 v_i 到 v_j 的中间顶点的序号不大于0的最短路径。
- 假设在路径上再增加一个顶点 v_1，也就是说，如果 $(v_i ,\ldots, v_1)$和 $(v_1,\ldots, v_j)$ 分别是当前找到的中间顶点的序号不大于0的最短路径,那么 $(v_i,\ldots, v_1 ,\ldots, v_j)$就是有可能是从$v_i$到$v_j$的中间顶点的序号不大于1的最短路径。将它与已经得到的从v_i到v_j中间顶点序号不大于0的最短路径相比较，从中选出中间顶点的序号不大于1的最短路径，再增加一个顶点v_2, 继续进行试探。
- 一般情况下，若 $(v_i ,\ldots, v_k)$ 和 $(v_k, \ldots, v_j)$ 分别是从 v_i到v_k和从v_k到v_j的中间顶点序号不大于 k-1 的最短路径，则将 $(v_i ,\ldots, v_k ,\ldots, v_j)$ 和已经得到的从 v_i到 v_j且中间顶点序号不大于k-1的最短路径相比较，其长度较短者便是从v_i到 v_j 的中间顶点的序号不大于 *k* 的最短路径。

算法步骤概述

若$<V_i,V_j>$存在，则存在路径$\{V_i,V_j\}$；

// 路径中不含其它顶点

若$<V_i,V_1>$, $<V_1,V_j>$存在，则存在路径$\{V_i, V_1, V_j\}$；

// 路径中所含顶点序号不大于1

若$<V_i, \ldots , V_2>$, $<V_2, \ldots , V_j>$存在，

则存在一条路径$\{V_i, \ldots , V_2, \ldots , V_j\}$；

// 路径中所含顶点序号不大于2

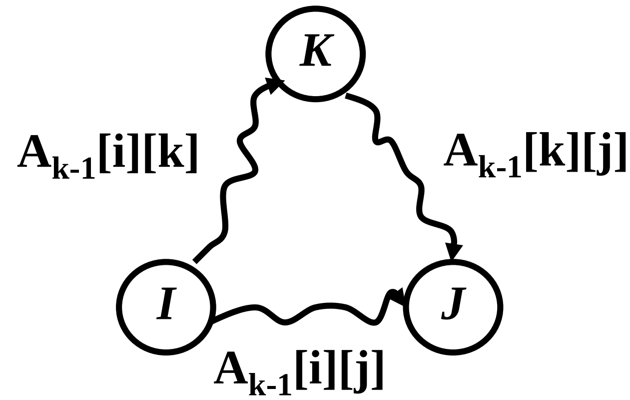
若$<V_i, \ldots , V_k>$, $<V_k, \ldots , V_j>$存在，

则存在一条路径$\{V_i, \ldots , V_k, \ldots , V_j\}$；

// 路径中所含顶点序号不大于k

依次类推，则V_i 至 V_j 的最短路径应使上述这些路径中，路径长度最小者。

$$C[i][j]=\begin{cases}0 & ,\ i=j \\ \infty & ,(i,j)\notin E \text{且 } i\neq j \\ \text{边（i，j)的长度} & ,(i,j)\in E\end{cases}$$

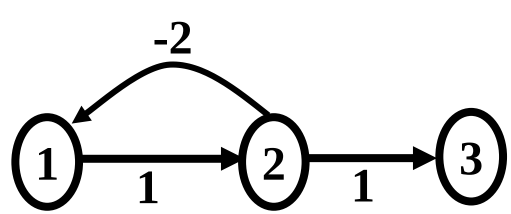


$$A_k[i][j] = \min(\ A_{k-1}[i][j]\ ,\ A_{k-1}[i][k] + A_{k-1}[k][j]\ )$$

A [i][j] 经过 *k* 的路径

不要求所有的C[i][j]都大于0，允许有负的权，**但不允许有负长度的环路**，否则此环路的任意两点之间的最短路径将是负无穷。

有负长度环路的有向图上式不成立！



$$A_0=\begin{bmatrix}0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0\end{bmatrix}$$

路径（1，2，1，2，1，……，3）趋于$-\infty$

算法实现步骤

（1）定义一个方阵序列{A_1，…，A_k}，A_1 初始化为图的邻接矩阵；

（2）依次在任意< i , j >顶点对中间插入中间顶点v_k（ $1\le k \le n$）；

（3）求从 *i* 到 *k* 和从 *k* 到 *j* 的两条路径，更新带有k结点的当前最短路径$A_k[i][j]$;

（4）用$A_k[i][j]$表示从 *i* 到 *j* 所经过顶点的标号不大于 *k* 的最短路径之长，则

$$A_k[i][j] = \min_{1\le i\le n} \{ A_{k-1}[i][k] + A_{k-1}[k][j] , A_{k-1}[i][j] \}$$ 。

其中$A_0[i][j] = C[i][j]$ $1\le i$，$j\le n$

从 *i* 到 *j* 且穿过中间点标号不大于 *k* 的最短路径若存在，则必经过顶点*k*，则:

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

否则:

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

所以有： $A_k[i][j] = \min(A_{k-1}[i][j] , A_{k-1}[i][k] + A_{k-1}[k][j]), k>=j$

另一种解释（建议看懂这个）

该算法依次递推得到n个方阵{$A_1, \ldots, A_n$}来记录每一步的当前最短路径长度。

假设：$A_k[i][j]$ 表示顶点v_i和v_j之间通过绕行了v_k以后，确定的当前最短路径，相当于任意顶点对之间强制绕行v_k，然后修改了当前最短路径长度存入A_k。

（1）定义一个n阶方阵序列{$A_0, A_1, \ldots, A_n$}，其中，$A_0 = C$（邻接矩阵）。

（2）在A_0上实现如下操作，在任意顶点对v_i和v_j之间插入顶点v_1，计算

$A_1[i][j] = \min\{ A_0[i][1] + A_0[1][j], A_0[i][j] \}$ 其中$A_0[i][j] = C[i][j]$ $1 \le i$，$j \le n$。

（3）继续在A_1上操作，在任意顶点对v_i和v_j之间插入顶点v_2，计算

$$A_2[i][j] = \min\{ A_1[i][2] + A_1[2][j], A_1[i][j] \}$$

推广到插入v_k的情况：

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

此时，我们可以说：$A_k[i][j]$ 所表达的最短路径长度是从v_i到v_j、中间顶点序号不大于k的最短路径的长度。

（4）上述操作直至推广到插入v_n的情况，结束。

这个时候，看懂下面的算法就OK了！

Floyd算法:

输入：表示有向图G = (V, E)的邻接矩阵C

输出：表示任意两点之间最短路长的矩阵A

算法要点:

```
Void  Floyd( A , C , n )
{  for ( i = 1; i <= n; i++ )
       for ( j = 1; j<=n; j++ )
           A[i][j] = C[i][j] ;
   for ( k = 1; k <= n; k++ )
       for ( i = 1; i <= n; i++ )
           for ( j = 1; j <=n; j++ )
               if ( A[i][k] + A[k][j] < A[i][j] )
                   A[i][j] = A[i][k] + A[k][j] ;
}
```

时间复杂度：$O(n^3)$

Floyd算法:

输入：表示有向图G = (V, E)的令

输出：表示任意两点之间最短路t

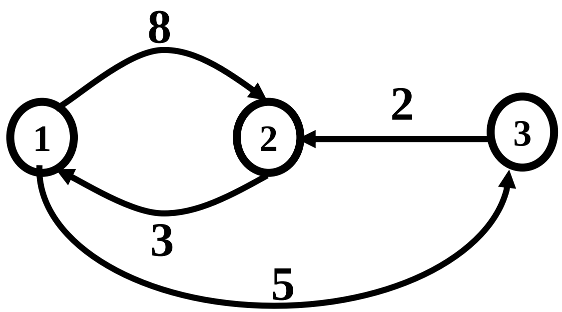
算法要点:

```
Void  Floyd( A , C , n )
{   for ( i = 1; i <= n; i++ )
         for ( j = 1; j<=n; j++ )
              { A[i][j] = C[i][j] ; P[i][j]=0; }
    for ( k = 1; k <= n; k++ )
         for ( i = 1; i <= n; i++ )
              for ( j = 1; j <=n; j++ )
                   if ( A[i][k] + A[k][j] < A[i][j] )
                        { A[i][j] = A[i][k] + A[k][j] ; P[i][j]=k; }
}
```

输出*i*到*j* 最短路径除*i,j*之外的顶点

```
Void Path(i,j)
{   k=P[i][j];
    if(k!=0)
     {  Path(i,k);
        count<<k<<endl;
        Path(k,j);          }
}
```

时间复杂度：$O(n^3)$



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

路径矩阵P

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

A[2][3]= ∞,k=1
A[2][1]+A[1][3]=8

A[3][1]= ∞,k=2
A[3][2]+A[2][1]=5

A[1][2]= 8,k=3
A[1][3]+A[3][2]=7

	1	2	3
1	0	8	5
2	3	0	∞
3	∞	2	0

$A_0[i][j]$

	1	2	3
1	0	8	5
2	3	0	8
3	∞	2	0

$A_1[i][j]$

	1	2	3
1	0	8	5
2	3	0	8
3	5	2	0

$A_2[i][j]$

	1	2	3
1	0	7	5
2	3	0	8
3	5	2	0

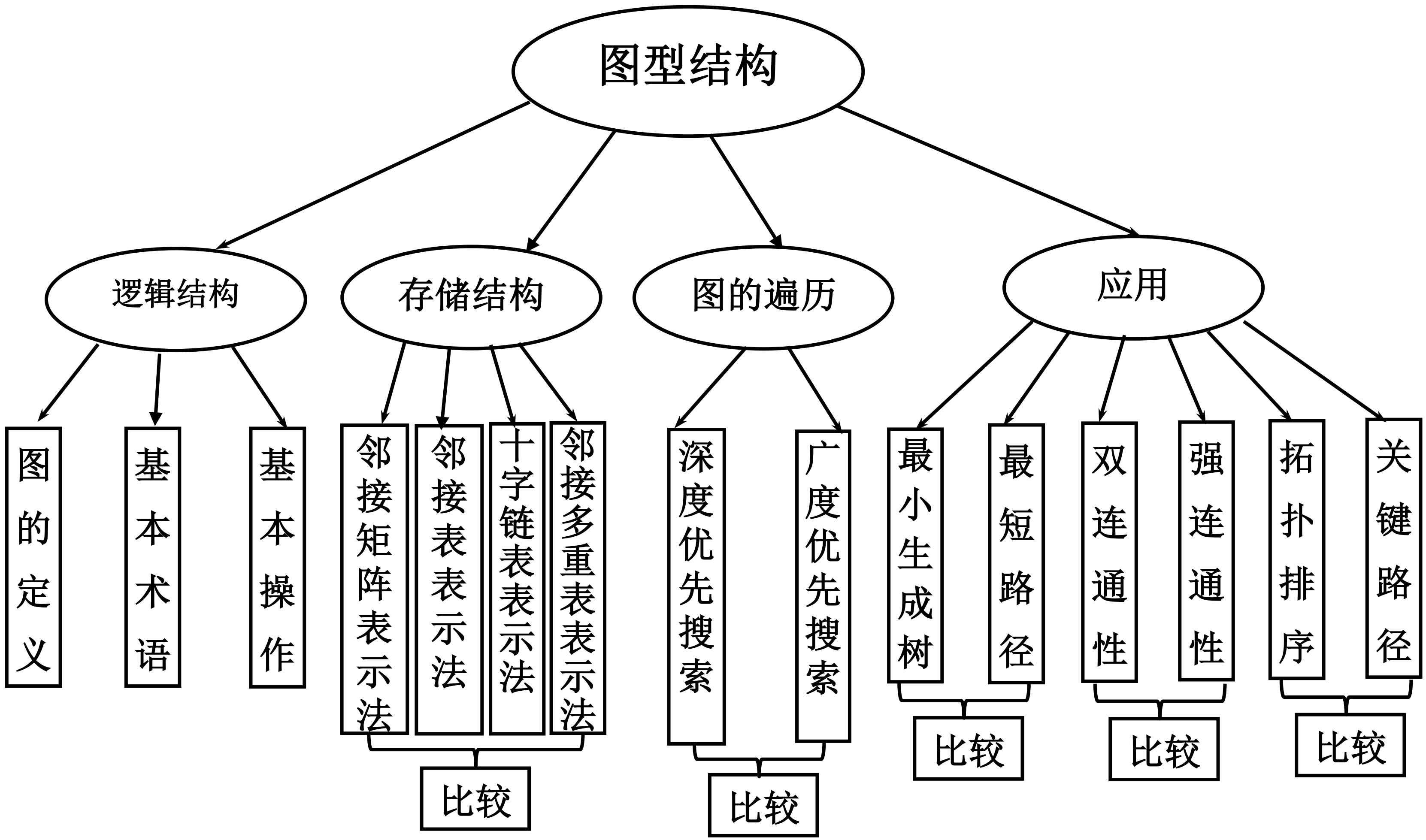
$A_3[i][j]$

- 图（有向图、无向图）与树的关系
  - 深度优先（先深DFS）/ 广度优先（先广BFS）遍历（算法）
  - 先深生成树（森林）和先广生成树（森林）
  - 树边、非树边
- 无向图、无向网
  - 开放树、最小生成树（Prim和Kruskal算法）
  - 由边的等价，引出双连通分量
  - 关节点、双连通图
- 有向图、有向网
  - 树边、向前边、回退边、横边
  - 由顶点的等价，引出强连通分量
  - 强连通图
  - 归约图
- 有向无环图
  - 拓扑排序（算法）、关键路径（算法）
- 最短路径
  - 单源最短路径（算法）、每对顶点间最短路径（算法）

图

- 图的ADT
  - 数学模型
  - 操作集
- 有向图　无向图　有向网　无向网
- 图的存储结构
  - 邻接矩阵
  - 邻接表
- 图的遍历
  - 深度优先(DFS)
  - 广度优先(BFS)
- 最小生成树　有向图环路问题
- 拓扑序列　关键路径
- 最短路径
  - 单源最短路径Dijkstra
  - 每一对顶点间最短路径Floyd

知识点总结


图型结构
逻辑结构
存储结构
图的遍历
应用
图的定义
基本术语
基本操作
邻接矩阵表示法
邻接表表示法
十字链表表示法
邻接多重表表示法
深度优先搜索
广度优先搜索
最小生成树
最短路径
双连通性
强连通性
拓扑排序
关键路径
比较
比较
比较
比较
比较

归纳与总结

三大数据结构		线性表					树		图	
类型		线性结构					层次结构		网状结构	
		线性表	栈	队列	串	数组	二叉树	树	无向图	有向图
逻辑结构	定义									
	ADT									
物理结构	顺序									
	链式									
遍历	常规									
	其它									
典型算法										
其它										
…… ……										