第12章　图（二）

计算机学院

主要内容

- 最短路径
- 拓扑排序
- 关键路径

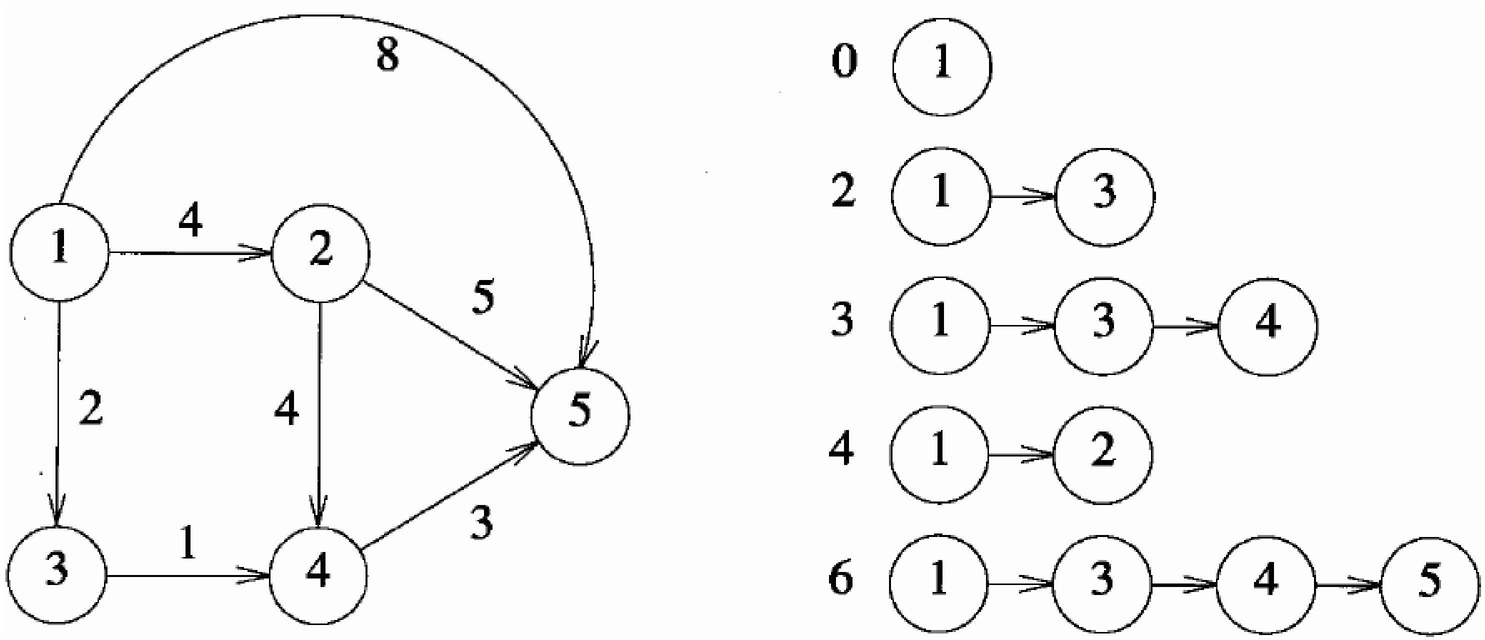
最短路径问题

- 无权图的最短路径问题
  - 比较简单，即两点之间边数最少的路径
- **有向带权图**的最短路径问题
  - 可理解为两地间交通费用最少问题
  - 分为单源最短路径、每对点最短路径两类

单源最短路径

- 有向图G，每条边都有非负权重（耗费）
- 路径长度——路径中边的权重之和
- 单源最短路径：给定源顶点s，求它到其他任意顶点（目的顶点）的最短路径

单源最短路径例



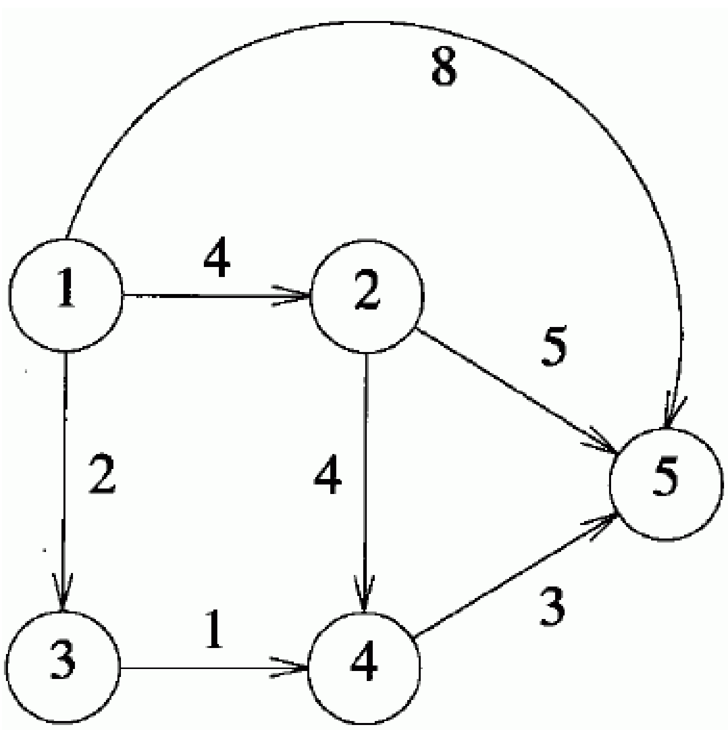
Dijkstra算法

- S：“已求出最短路径顶点集合”，初始为{s}
- L=V - S
- 每个步骤从L选取一个顶点v加入S
- 贪心准则：v是L中距s距离最短者
- 新最短路径=已有最短路径+一条边
  每个顶点无需保存其完整路径，保存路径中它的前一顶点即可

实现

- 数组p：保存路径（上例：[0, 1, 1, 3, 4]）
  - p[i]：最短路径中顶点i的前驱顶点
  - 从终点开始，逆向即可获取路径
- 数组d：当前最短路径长度
  - i在S中，d[i]：s→i的真正最短路径长度（最终结果）
  - i在L中，d[i]：当前最短路径长度 s→j(∈S)→i的路径长度（最短者）：d[j]+a[j][i]
  - 从L中选择v加入S后，S发生变化，L中的d[i]可能变得更小，应进行更新

运算实例：解题形式1



S={1}

1 2 3 4 5
d={0, 4, 2, ∞, 8}
p={0, 1, 1, 0, 1}

S={1, 3}, 1→3

1 2 3 4 5
d={0, 4, 2, 3, 8}
p={0, 1, 1, 3, 1}

S={1, 3, 4}, 1→3→4

1 2 3 4 5
d={0, 4, 2, 3, 6}
p={0, 1, 1, 3, 4}

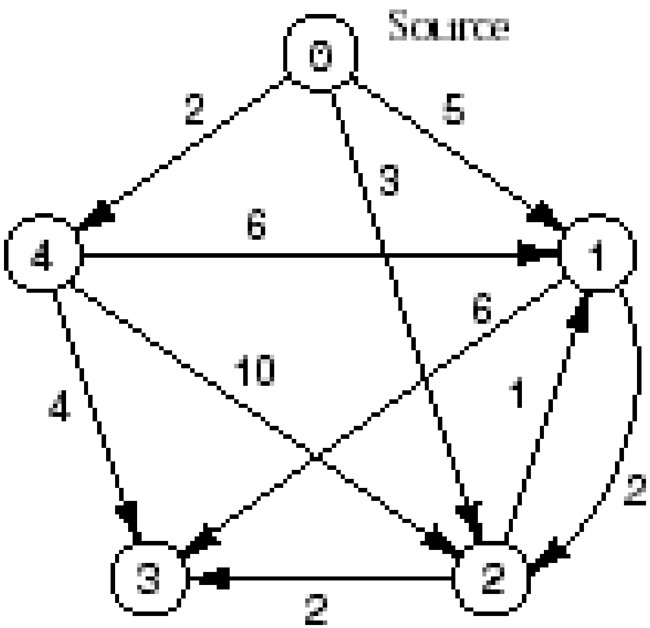
S={1, 3, 4, 2}, 1→2

1 2 3 4 5
d={0, 4, 2, 3, 6}
p={0, 1, 1, 3, 4}

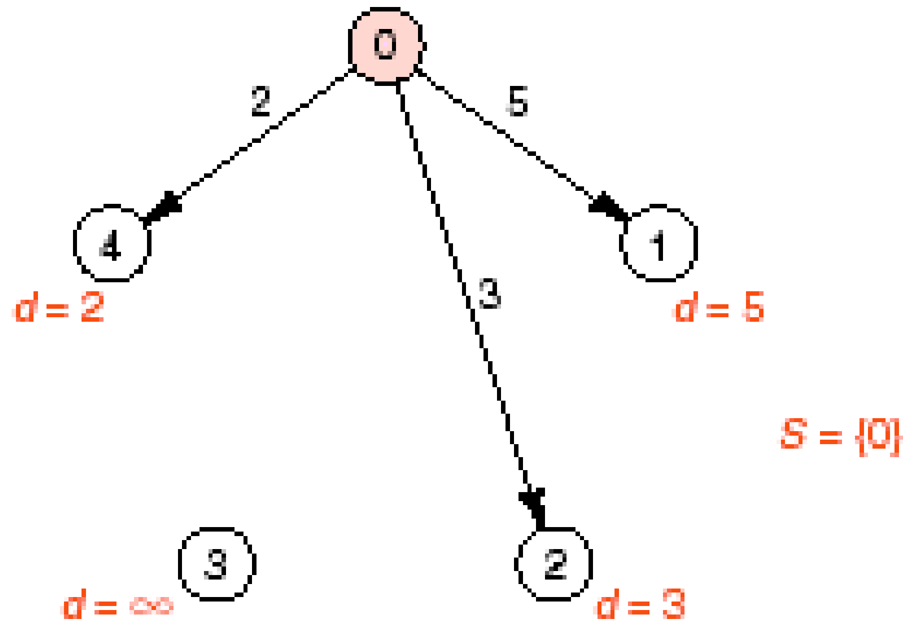
S={1, 3, 4, 2, 5}
1→3→4→5

1 2 3 4 5
d={0, 4, 2, 3, 6}
p={0, 1, 1, 3, 4}

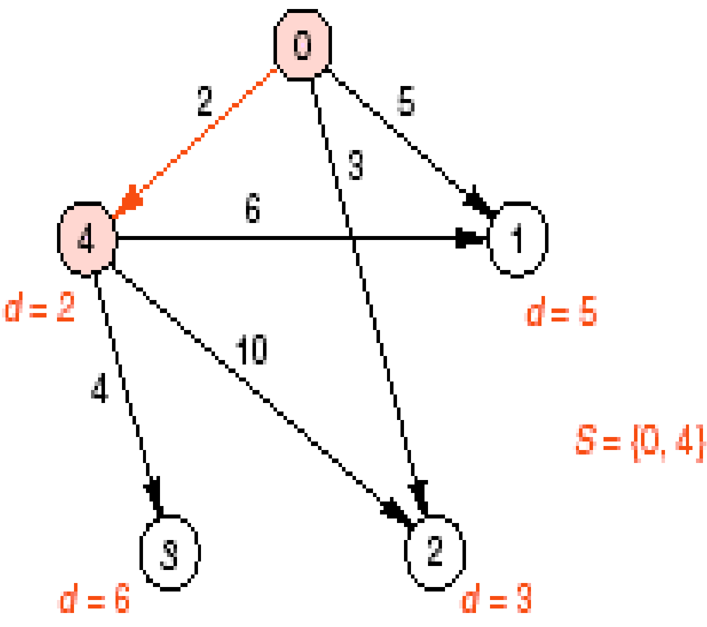
运算实例



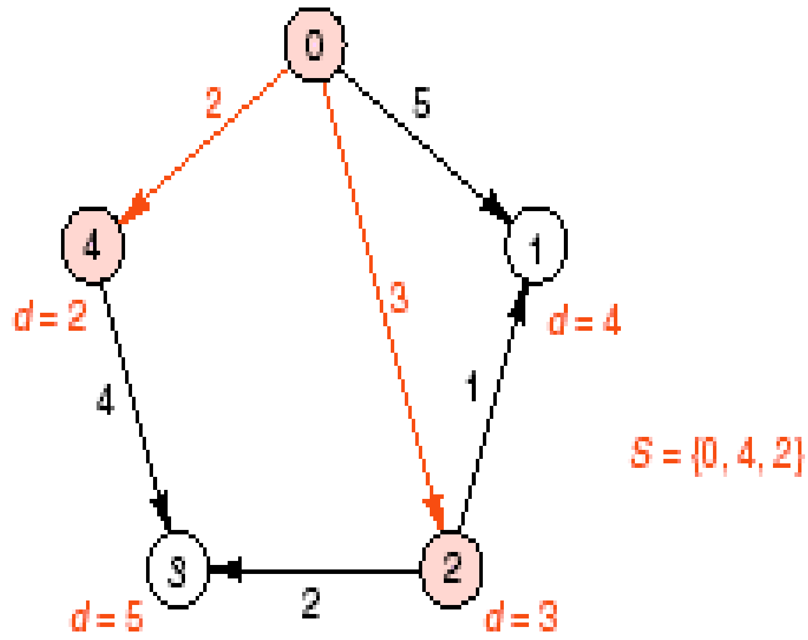
(a)



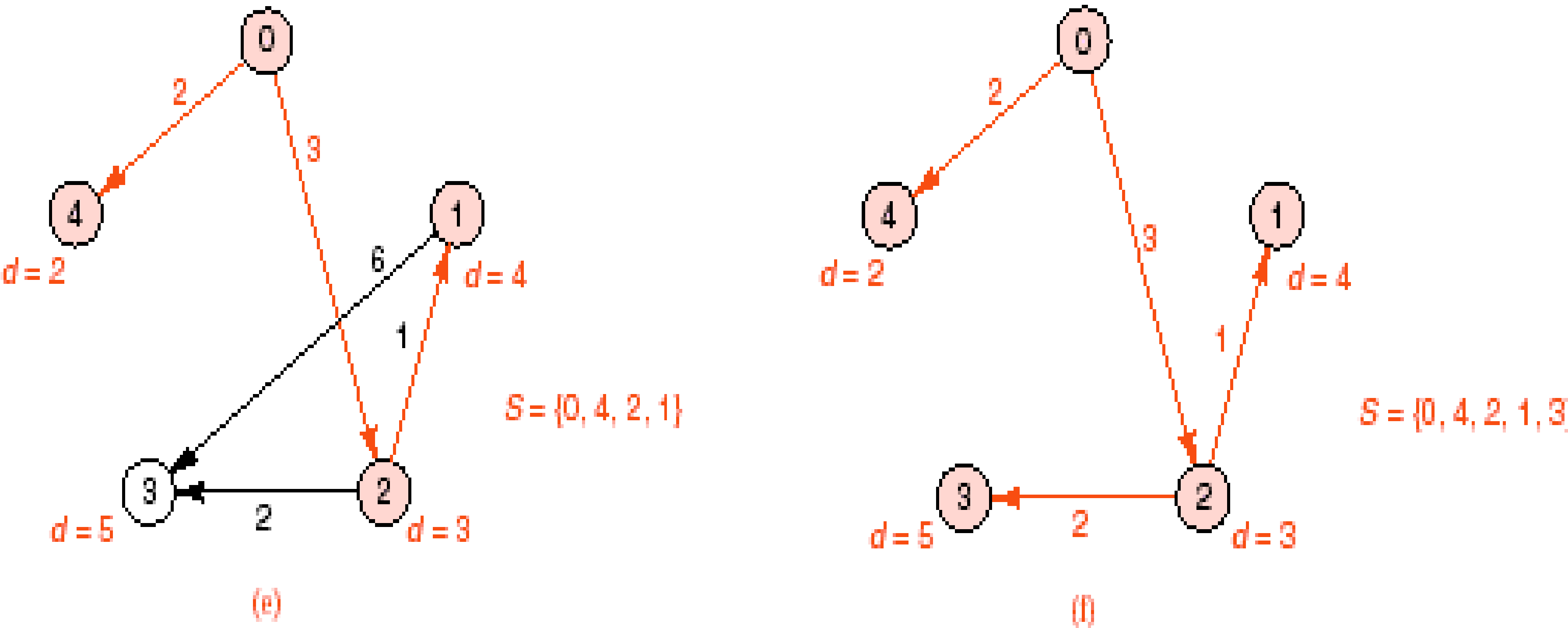
(b)



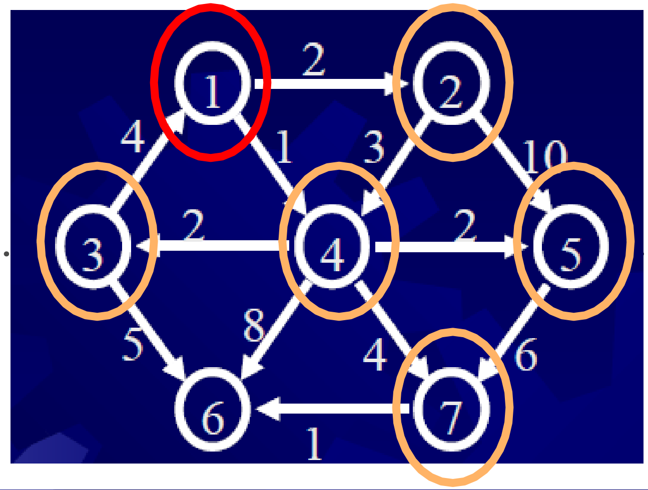
(c)



(d)



0
4
1
3
2
2
3
6
1
2
d = 2
d = 4
d = 5
d = 3
S = {0, 4, 2, 1}
(e)
0
4
1
3
2
2
3
1
2
d = 2
d = 4
d = 5
d = 3
S = {0, 4, 2, 1, 3}
(f)



v_2	2 (v_1, v_2)	2 (v_1, v_2)				
v_3	∞	3 (v_1,v_4,v_3)	3 (v_1,v_4,v_3)			
v_4	1 (v_1,v_4)					
v_5	∞	3 (v_1,v_4,v_5)	3 (v_1,v_4,v_5)	3 (v_1,v_4, v_5)		
v_6	∞	9 (v_1,v_4,v_6)	9 (v_1,v_4,v_6)	8 (v_1,v_4,v_3,v_6)	8 (v_1,v_4,v_3,v_6)	6 (v_1,v_4,v_7,v_6)
v_7	∞	5 (v_1,v_4,v_7)	5 (v_1,v_4,v_7)	5 (v_1,v_4, v_7)	5 (v_1,v_4,v_7)	
v_j	v_4	v_2	v_3	v_5	v_7	v_6

Dijkstra算法伪代码

1) 初始化$d[i]=a[s][i]$（$1\leqslant i\leqslant n$）
对于邻接于s的所有顶点i，置$p[i]=s$，对于其余的顶点置$p[i]=0$
对于$p[i]\neq 0$的所有顶点建立L表

2) 若L为空，终止，否则转至3)

3) 从L中删除d 值最小的顶点

4) 对于与i邻接的所有还未到达的顶点j，更新$d[j]$值为$min\{d[j], d[i]+a[i][j]\}$
若$d[j]$发生了变化且j还未在L中，则置$p[j]=i$，并将j加入L，转至2

无序链表实现

```
template<class T>
void AdjacencyWDigraph<T>::ShortestPaths(int s,
                         T d[], int p[])
{// Shortest paths from vertex s, return shortest
 // distances in d and predecessor info in p.
   if (s < 1 || s > n) throw OutOfBounds();
   Chain<int> L; // list of reachable vertices for
                 // which paths have yet to be found
   ChainIterator<int> I;
▯
```

无序链表实现（续）

```
// initialize d, p, and L
for (int i = 1; i <= n; i++){
  d[i] = a[s][i];
  if (d[i] == NoEdge) p[i] = 0;
  else {p[i] = s;
        L.Insert(0,i);}
  }
```

无序链表实现（续）

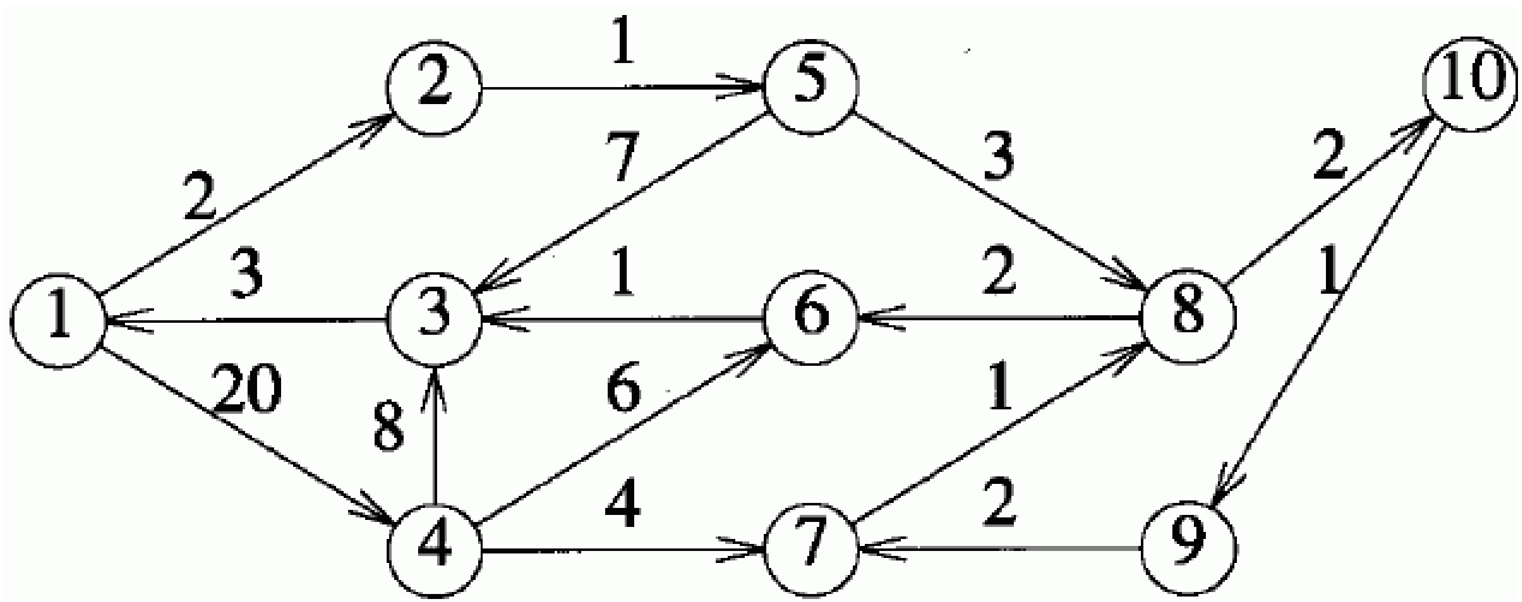
```
// update d and p
while (!L.IsEmpty()) {// more paths exist
  // find vertex *v in L with least d
  int *v = I.Initialize(L);
  int *w = I.Next();
  while (w) {
    if (d[*w] < d[*v]) v = w;
    w = I.Next();}
```

无序链表实现（续）

```
int i = *v;
    L.Delete(*v);
    for (int j = 1; j <= n; j++) {
      if (a[i][j] != NoEdge && (!p[j] ||
            d[j] > d[i] + a[i][j])) {
            d[j] = d[i] + a[i][j];
       if (!p[j]) L.Insert(0,j);
         p[j] = i;}
      }
    }
}
```

每一对点的最短路径

- 所有点对间的最短路径，all-pairs shortest-paths problem，n(n-1)条
- 简单算法：每个顶点执行单源最短路径算法

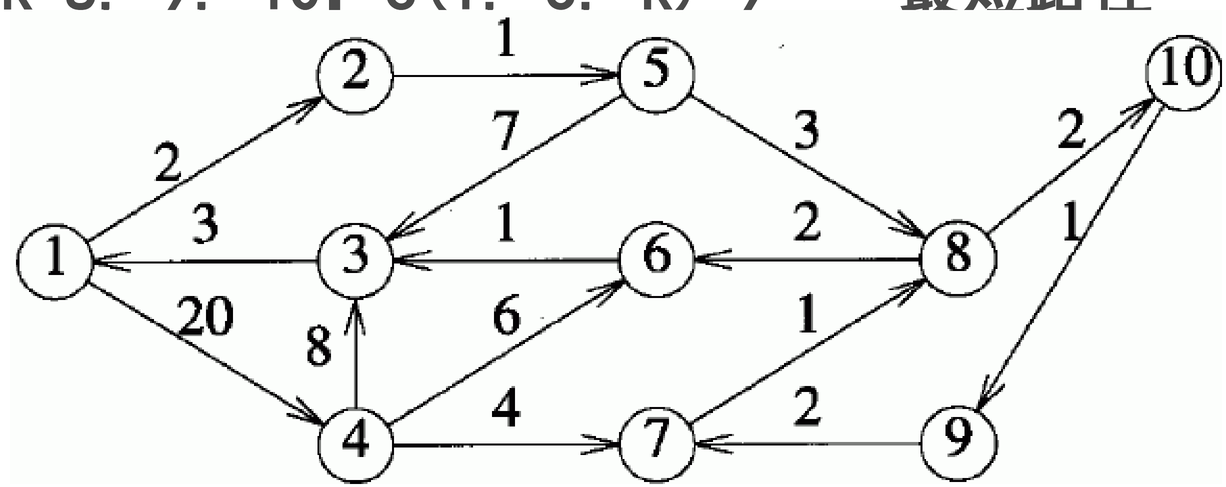


Floyd算法

- 顶点编号为1～n
- c(i, j, k)：i→j的“最短路径”长度——加了限制条件，路径中顶点的最大编号为k
  - 存在边<i，j>→c(i，j，0)=<i，j>的长度
  - 不存在边<i，j>→c(i，j，0)=+∞
  - c(i，i，0)=0
  - c(i，j，n)——最短路径长度

Floyd算法

- 例15.6：考虑上图
  - k=0, 1, 2, 3, c(1, 3, k)= +∞ ；c(1, 3, 4)=28
  - k=5, 6, 7, c(1, 3, k)=10;
  - k=8. 9. 10. c(1. 3. k)=9——最短路径



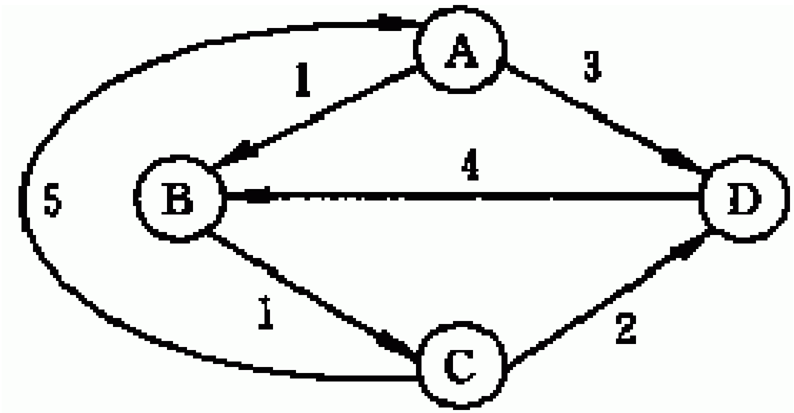
如何计算c(i, j, k)

- 顶点最大编号不超过k，两种情况
  - 路径不包含k，c(i, j, k)=c(i, j, k−1)
  - 包含k，c(i, j, k)=c(i, k, k−1) + c(k, j, k−1)
  - ➔c(i, j, k)=min{c(i, j, k−1), c(i, k, k−1)+c(k, j, k−1)}
- 递归算法，$\Theta(n2^n)$
- 迭代计算→$\Theta(n^3)$

迭代计算伪代码

```
//寻找最短路径的长度
//初始化c(i, j, 1)
for (int i=1; i <= n ; i ++)
  for (int j =1; j <=n; j ++)
     c( i , j, 0) = a (i , j); // a 是长度邻接矩阵
//计算c ( i , j, k ) ( 0 < k <= n )
for(int k=1;k<=n;k++)
  for (int i=1;i<=n;i++)
     for (int j= 1 ;j< = n ;j+ + )
        if (c(i,k,k-1)+c(k, j, k - 1) < c (i, j, k - 1))
           c(i, j, k) = c(i, k, k - 1) + c(k, j, k - 1) ;
        else c(i, j, k) = c (i, j, k - 1);
```

Floyd算法例



C0

	A	B	C	D
A	0	1	∞	3
B	∞	0	1	∞
C	5	∞	0	2
D	∞	4	∞	0

Floyd算法例（续）

$c(i,j,1)=\min(c(i,j,0),\ c(i,1,0)+c(1,j,0))$

C0

	1	2	3	4
1	0	1	∞	3
2	∞	0	1	∞
3	5	∞	0	2
4	∞	4	∞	0

C1

	1	2	3	4
1	0	1	∞	3
2	∞	0	1	∞
3	5	**6**	0	2
4	∞	4	∞	0

Floyd算法例（续）

c(i,j,2)=min(c(i,j,1), c(i,2,1)+c(2,j,1))

C1

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

C2

	1	2	3	4
1	0	1	**2**	3
2	∞	0	1	∞
3	5	6	0	2
4	∞	4	**5**	0

Floyd算法例（续）

C2

$c(i,j,3)=\min(c(i,j,2),\ c(i,3,2)+c(3,j,2))$

C3

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

	1	2	3	4
1	0	1	2	3
2	**6**	0	1	**3**
3	5	6	0	2
4	**10**	4	5	0

Floyd算法例（续）

$c(i,j,4)=\min(c(i,j,3),\ c(i,4,3)+c(4,j,3))$

C3

	1	2	3	4
1	0	1	2	3
2	6	0	1	3
3	5	6	0	2
4	10	4	5	0

C4

	1	2	3	4
1	0	1	2	3
2	6	0	1	3
3	5	6	0	2
4	10	4	5	0

最终代码

```
template<class T>
void AdjacencyWDigraph<T>::AllPairs(T **c, int **kay)
{// All pairs shortest paths.
 // Compute c[i][j] and kay[i][j] for all i and j.
   // initialize c[i][j] = c(i,j,0)
   for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) {
         c[i][j] = a[i][j];
         kay[i][j] = 0;
         }
   for (i = 1; i <= n; i++)
      c[i][i] = 0;
```

最终代码（续）

```
// compute c[i][j] = c(i,j,k)
  for (int k = 1; k <= n; k++)
    for (int i = 1; i <= n; i++)
      for (int j = 1; j <= n; j++) {
        T t1 = c[i][k];
        T t2 = c[k][j];
        T t3 = c[i][j];
        if (t1 != NoEdge && t2 != NoEdge &&
          (t3 == NoEdge || t1 + t2 < t3)) {
            c[i][j] = t1 + t2;
            kay[i][j] = k;}
        }
}
```

最终代码（续）

```
void outputPath(int **kay, int i, int j)
{// Actual code to output i to j path.
   if (i == j) return;
   if (kay[i][j] == 0) cout << j << ' ';
   else {outputPath(kay, i, kay[i][j]);
         outputPath(kay, kay[i][j], j);}
}
```

最终代码（续）

```
template<class T>
void OutputPath(T **c, int **kay, T NoEdge,
                    int i, int j)
{// Output shortest path from i to j.
  if (c[i][j] == NoEdge) {
    cout << "There is no path from " << i << " to "
         << j << endl;
    return;}
  cout << "The path is" << endl;
  cout << i << ' ';
  outputPath(kay,i,j);
  cout << endl;
}
```

主要内容

- 最短路径
- 拓扑排序
- 关键路径

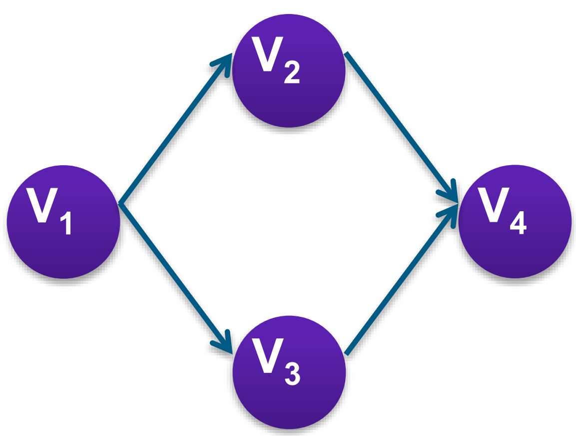
工程和有向无环图

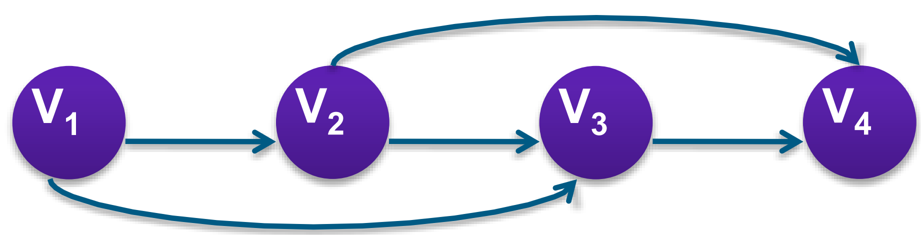
- 有向无环图是描述复杂工程的有效工具
- 工程可以分解为多个活动（Activity）
- 活动之间具有前后约束关系

- 工程问题转化为
  - 工程能否顺利进行?
  - 完成工程的最短时间是多少?

拓扑排序

- 偏序：集合中仅有部分成员之间可比较
- 全序：集合中全体成员之间均可比较
- **拓扑排序**（Topological Sort）：由某个集合上的一个偏序得到该集合上的一个全序

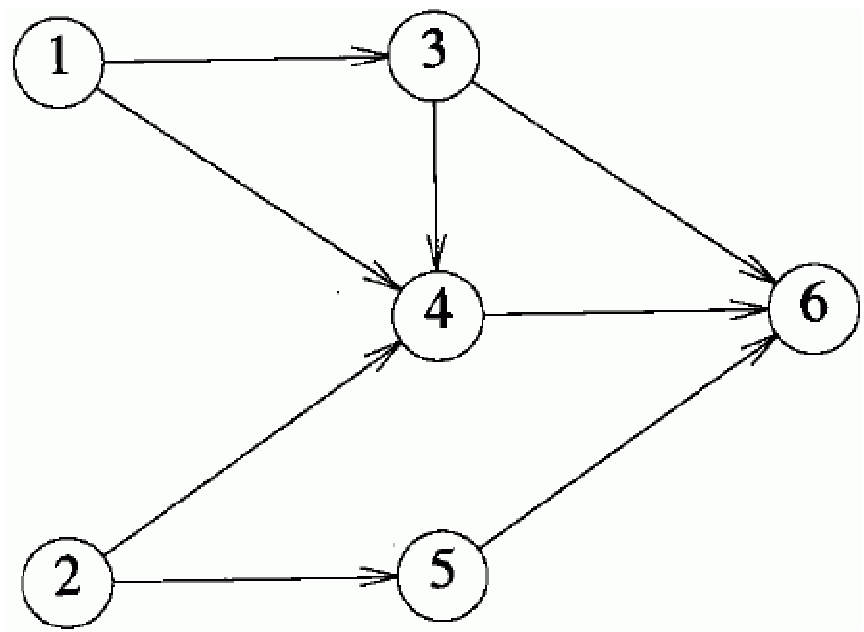




AOV图

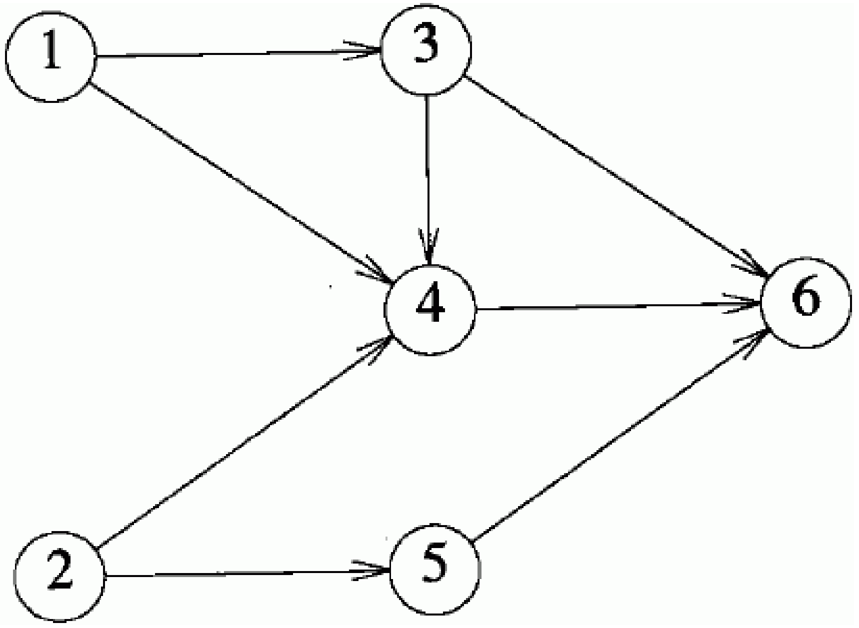
- 用顶点表示活动，用箭头表示活动间优先关系的有向图称为顶点活动网络（**Activity On Vertex，AOV**）
  - 顶点i在顶点j之前，意味着活动i是活动j的先决条件
  - 显然不应该出现有向环，否则“活动k是它自己的先决条件”，不成立

AOV网络示例



拓扑序列例

- 123456、132456、215346
- 142356不是！



利用贪心算法进行拓扑排序

- 在有向图中选一个没有前驱的顶点且输出
- 从图中删除该顶点和所有从其发出的箭头
- 重复上述两步，直至所有顶点均已输出，或者当前图中不存在无前驱的顶点为止

算法描述

设*n*是有向图中的顶点数

设*V*是一个空序列

while (*true*) {

设*w*不存在入边（*v*, *w*），其中顶点*v*不在*V*中

如果没有这样的*w*，*break*

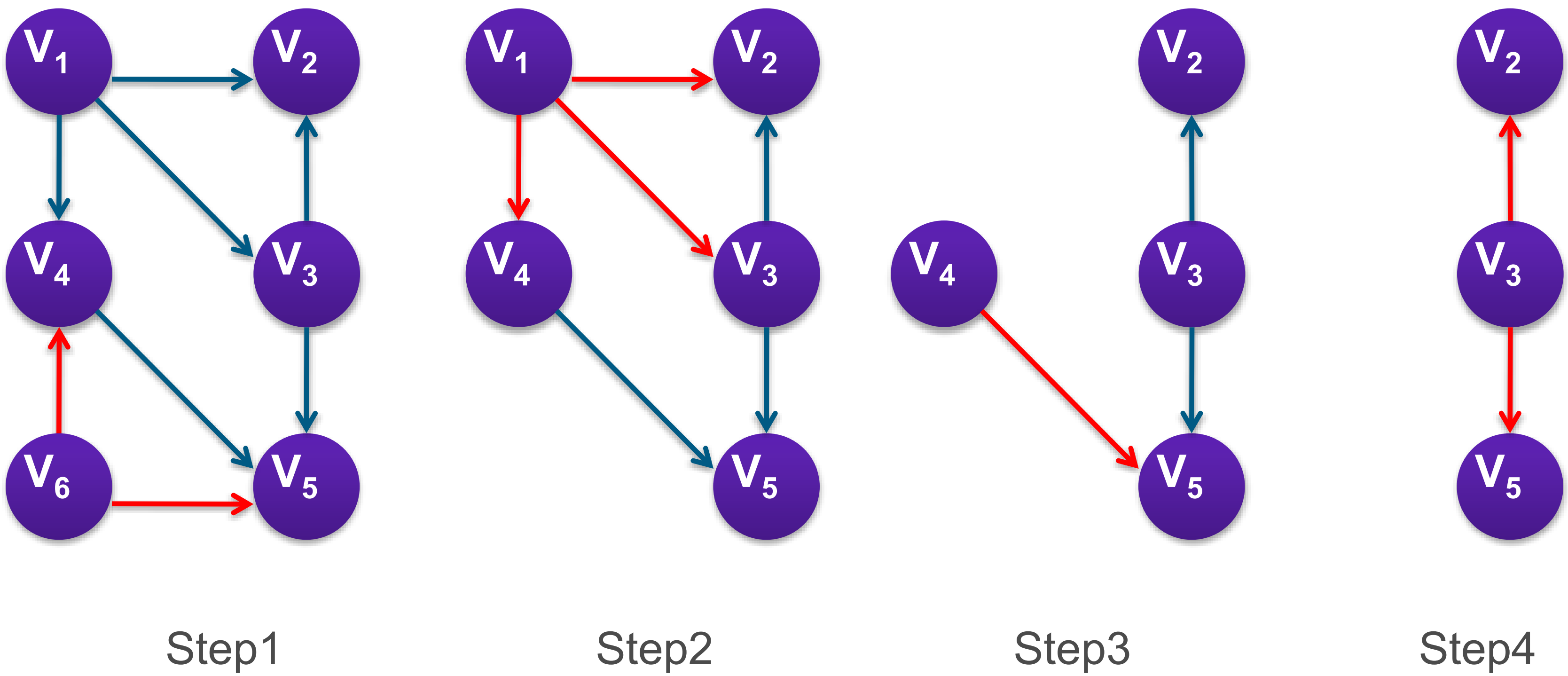
把w添加到V的尾部

}

if(*V*中的顶点数少于*n*)算法失败

else *V*是一个拓扑序列

算法示例



$V_6 \quad V_1 \quad V_4 \quad V_3 \quad V_2 \quad V_5$

算法细化——数据结构的选择

- 拓扑序列V如何描述？
  如何找出候选顶点？
- V———一维数组v
  栈——保存候选顶点
- InDegree——保存顶点当前入度

算法实现

- 初始
  - V为空
  - InDegree保存图中顶点入度
  - 将入度为0的顶点压栈
- 每个步骤
  - 弹出栈顶顶点p，加入V
  - 并将p的每个邻接顶点的InDegree值减1
  - 若某个顶点的InDegree值变为0，将其压栈

实现

```
bool Network::Topological(int v[])
{
  int n = Vertices();

  // Compute in-degrees
  int *InDegree = new int [n+1];
  InitializePos(); // graph iterator array
  for (int i = 1; i <= n; i++) // initialize
    InDegree[i] = 0;
```

实现（续）

```
for (i = 1; i <= n; i++) {// 遍历所有顶点的出边，计算入度
  int u = Begin(i);
  while (u) {
    InDegree[u]++;
    u = NextVertex(i);}
  }

// Stack vertices with zero in-degree
LinkedStack<int> S;
for (i = 1; i <= n; i++)
  if (!InDegree[i]) S.Add(i);
```

实现（续）

```
// Generate topological order
i = 0;  // cursor for array v
while (!S.IsEmpty()) {// select from stack
   int w;            // next vertex
   S.Delete(w);
   v[i++] = w;
   int u = Begin(w);
   while (u) {// update in-degrees
      InDegree[u]--;
      if (!InDegree[u]) S.Add(u);
      u = NextVertex(w);}
   }
DeactivatePos();
delete [] InDegree;
return (i == n);
}
```

时间复杂度是O(n+e)

拓扑排序要点

- 入度为0的顶点即没有前驱活动的，或前驱活动都已经完成的顶点，工程可以从这个顶点所代表的活动开始或继续

- 算法每输出一个顶点之后，要删去从这个顶点发出的边，这意味着这个顶点所代表的活动已经完成，对于后续顶点所代表的活动来说，该前驱活动已经完成

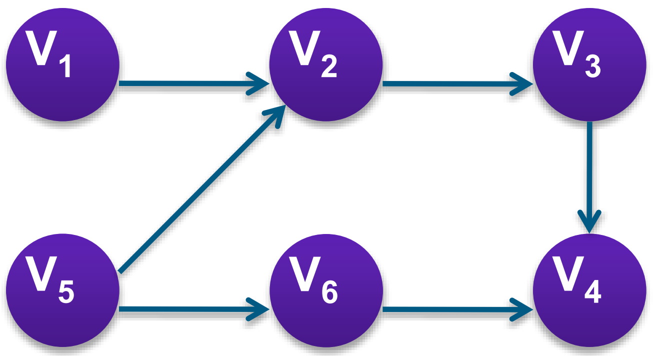
拓扑排序要点

- 如果一个节点有多个直接后继，则拓扑排序的结果通常不唯一

- 由于AOV网络中各顶点的地位是平等的，每个顶点的编号是人为的，因此可以按照拓扑排序的结果重新安排顶点的序号，生成AOV网络的新的邻接矩阵存储表示。其中，对角线以下可以全为零。

小练习

- 写出下图的所有拓扑排序结果
  - 提示：共有7种

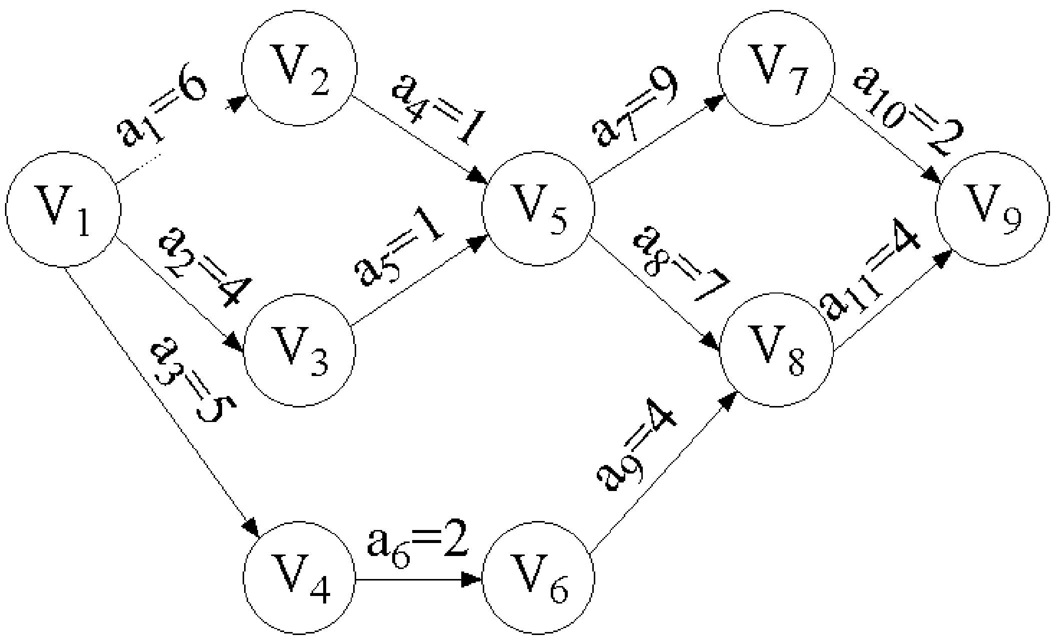


主要内容

- 最短路径
- 拓扑排序
- 关键路径

AOE网

- 一个带权的有向无环图，其中顶点表示事件，边表示活动，权表示活动持续的时间，则该图称为AOE（Activity On Edge）网



左图所示工程：
9个事件（里程碑）
11项活动（必须完成的工作）

V_5含义是：
a4和a5已完成
a7和a8可开始

关键路径

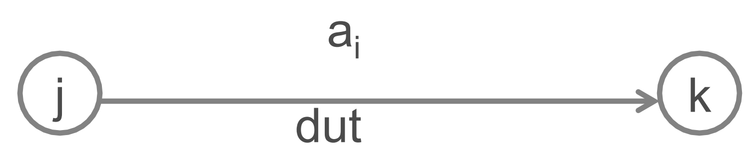
- 将AOE网看作一个工程
  - 只有一个入度为0的点（源点），一个出度为0的点（汇点）
  - 完成整个工程需要多少时间？哪些活动是影响工程进度的关键？
- 关键路径（Critical Path）
  - 从源点到汇点长度（时间和）最长的路径
  - 长度——完成工程的最短时间
  - 上例：（v_1，v_2，v_5，v_8，v_9）

关键活动

- 源点v_1→$_i$的最长路径长度：
  事件v_i的最早发生时间，
  v_i发出的所有边（活动）的最早开始时间
- e(i)：活动a_i的最早开始时间
- 最迟开始时间：l(i)，前提：不影响工程进度
- l(i)=e(i)：关键活动
- 关键路径上的活动都是关键活动
- **通过计算l(i)、e(i)寻找关键活动**

l（i）和e（i）的计算

- 事件的最早发生时间ve（j）
  最迟发生时间vl（j）



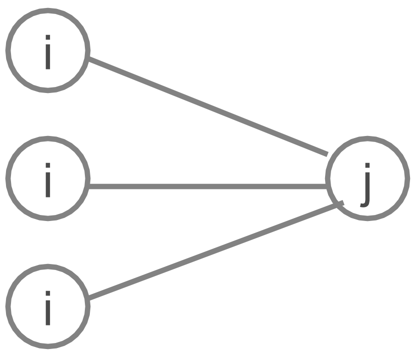
- 活动a_i由边<j，k>表示
  dut（<j，k>）表示其持续时间，则有
  - e（i）=ve（j）
  - 含义是：活动i要想开始至少需要等待的时间
  - l（i）=vl（k）-dut（<j，k>）
  - 含义是：活动i开始之前最多空闲的时间

求e(i)和l(i)可以转换为求ve(j)和vl(k)的问题

ve(i)的计算

- 由ve(0)=0向前递推

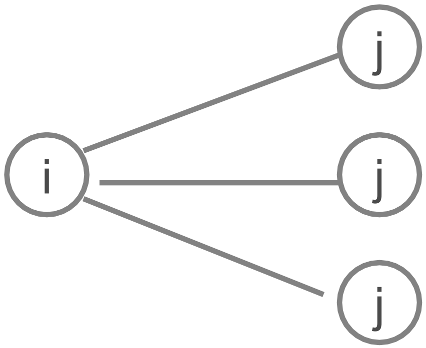
  $ve(j)=\max\{ve(i)+dut(\langle i,j\rangle)\},\ \langle i,j\rangle\in E$



事件j的开始依赖于
所有活动<i,j>的完成
显然应该取其中“最差”者——
j再早也不会比最慢的那个早

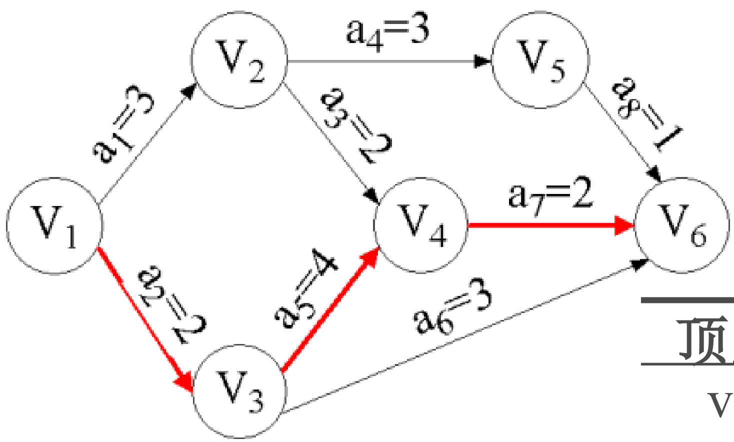
vl(i) 的计算

- 由vl(n-1)=ve(n-1)向后递推
  $vl(i)=\min\{vl(j)-dut(\langle i, j\rangle)\}, \langle i, j\rangle \in E$



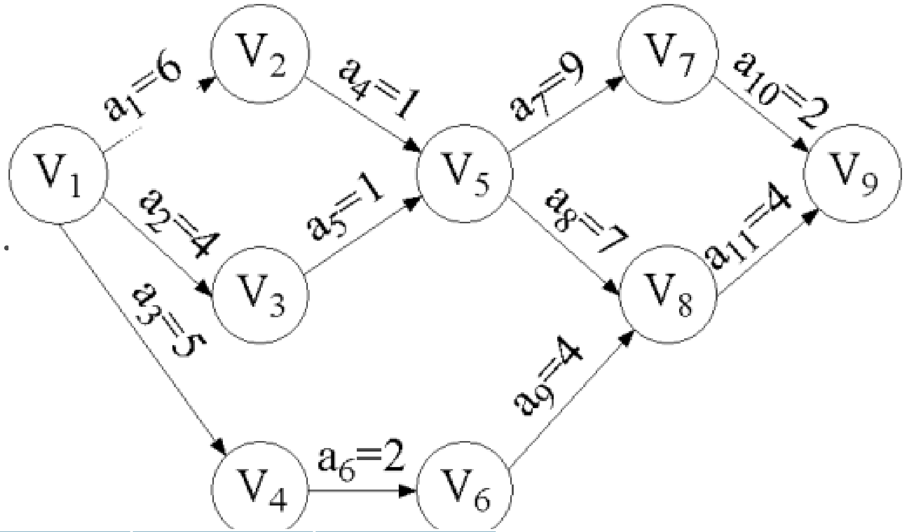
所有的j都依赖于i
i再迟也不应影响j的启动——不能影响工期
也应该取其中“最差”（最早）者

计算实例



顶点	ve	vl	活动	e	l	l-e
v_1	0	0	a_1	0	1	1
v_2	3	4	a_2	0	0	0
v_3	2	2	a_3	3	4	1
v_4	6	6	a_4	3	4	1
v_5	6	7	a_5	2	2	0
v_6	8	8	a_6	2	5	3
			a_7	6	6	0
			a_8	6	7	1

计算实例



顶点	ve	vl	活动	e	l	l-e
v1	0	0	a1	0	0	0
v2	6	6	a2	0	2	2
v3	4	6	a3	0	3	3
v4	5	8	a4	6	6	0
v5	7	7	a5	4	6	2
v6	7	10	a6	5	8	3
v7	16	16	a7	7	7	0
v8	14	14	a8	7	7	0
v9	18	18	a9	7	10	3
			a10	16	16	0
			a11	14	14	0

关键路径要点

- 关键路径长度是完成工程的最短时间，即至少消耗时间
- 研究意义是找到关键路径、设法提高其效率，则**有可能**缩短工期
- 算法
  - Step1：从源点开始计算ve(i)，考察指向顶点i的所有边，寻找**最大值**ve(j)=max{ve(i)+dut(<i， j>)}，<i， j>∈E
  - Step2：从汇点开始计算vl(i)，考察顶点i发出的所有边，寻找**最小值**vl(i)=min{vl(j)-dut(<i， j>)}，<i， j>∈E
  - Step3：求每个活动的e(i)，等于活动发出顶点的ve值
  - Step4：求每个活动的l(i)，等于活动指向顶点的vl值减去活动本身的持续时间
  - Step5：找到那些l(i)=e(i)的活动，即为关键活动；构成的路径即为关键路径。关键路径可能不止一条。

本章结束