6. 图-3

- 图：基本概念和性质，基本操作
- 图的遍历：宽度优先，深度优先
- 图的表示
- 生成树问题，DFS 生成树，BFS 生成树
- 最小生成树问题，Prim 算法，Kruskal 算法
- 最短路径问题，单源点最短路径和 Dijkstra 算法，所有顶点之间的最短路径和 Floyd 算法
- AOV 网，拓扑排序
- AOE 网，关键路径

求各对顶点之间的最短路径：Floyd 算法

- 前面说过，可以用Dijkstra算法解决这个问题：
  - 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（Floyd-Warshall）算法采用了完全不同的想法，可以一次直接计算出各对顶点间的所有最短路径及其长度

  Floyd 算法的基本想法来自 Warshall 的强连通子图算法（基于邻接矩阵），实际上是求可达性（有边相邻）的传递闭包
- 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
  - 如果有边 (v, v')∈E，那么它就是从顶点 v 到 v' 的路径，其长度可以直接得到，即是 A[v][v']。但这一路径未必是从 v 到 v' 的最短路径，有可能存在从 v 到 v' 途中经过其它顶点的更短路径

  问题就是要在 v 到 v' 的可能经过任何顶点的路径中找出最短路径

Floyd 算法

- 开始：每对 v 到 v' 的途中不经任何结点的路径长度为已知

  有 v 到 v' 的边时就是边的权，无边时认为存在长度为 ∞ 的路径

- k = 0：对每对 v 和 v'，除已知直接路径外，从v 到 v' 途经顶点的下标不大于 k（此时是不大于 0）的路径可分为两段：
  - <v, v_0>，<v_0, v'>　（如果没有，就认为有长度为 ∞ 的路径）
  - 其长度是两段路径的长度和。比较这一路径和直接路径（是已知最短的），可确定 v 到 v' 的途径顶点下标不大于 0 的最短路径
- k = 1：对每对 v 和 v'，除至此已知路径外（途径顶点下标 ≤0），从 v 到 v' 途经顶点下标不大于 k（不大于 1）的新路径可分为两段：
  - <v, …, v_1>，<v_1, …, v'>
  - 这两段内部所经过的顶点下标都不大于 0，路径及其长度都已在前一步确定。这种新路径的长度是两段路径的长度之和
  - 用这样的新路径与从 v 到 v' 的已知最短路径（途径顶点下标 ≤0）比较，就可确定v 到 v' 的途经顶点下标 ≤1 的最短路径

Floyd 算法

- 一般而言，如果已经考察过从 v 到 v' 的途经顶点的下标 ≤k-1 的所有路径，而且已经获知了这样的路径中的最短路径及其长度
- 考虑 k：对每对 v 和 v'，除至此已知的路径（途经顶点的下标 ≤k-1）外，途经顶点的下标 ≤k 的其他路径必定可分为两段：
  - <v, …, v_k>，<v_k, …, v'>
  - 这两段路径中途径顶点的下标都 ≤k-1，两段的长度也在前一步已知，这种新路径的长度是两段路径的长度之和
  - 用该路径与已知的从 v 到 v' 的最短路径比较，就可确定从v 到 v' 的途经顶点的下标 ≤k 的最短路径
- ……
- 如此继续，直到做完 k = n-1（途径结点的下标不大于 n-1），也就对于每对 v 和 v'，确定了从v 到 v' 的所有路径中的最短路径
- 这里假定结点的下标为 0 到 n-1（对下标为 1 到 n 可类似定义）

Floyd 算法的实现

- 实现 Floyd 算法，需要迭代式地算出一系列方阵
- 为此要生成一系列 n×n 方阵 A_k (0≤k≤n)，其中 $A_k[i][j]$ 表示从 v_i 到 v_j 的途径顶点可为 $v_0, v_1,\ldots,v_{k-1}$ 的最短路径的长度：
  - A_0 就是图的邻接矩阵 A（由于是计算路径，对角线元素取 0 值），$A_0[i][j]$ 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 可递推计算（0≤i≤n-1, 0≤j≤n-1）：
  - $A_0[i][j] = A[i][j]$　　直接由邻接矩阵得到
  - $A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+A_k[k][j]\}$，0≤k≤n-1，这时新考虑了途径顶点 v_k 的路径，因此，$A_{k+1}[i][j]$ 为 v_i 到 v_j 的途经顶点的下标不大于 k 的最短路径的长度
  - $A_n[i][j]$ 为 v_i 到 v_j 的最短路径的长度

  （注意，这里采用的顶点编号为 0, ..., n-1）

Floyd 算法

- 实现算法时还需要设法做出所有路径的记录
  - 下面考虑另外安排一系列 n 阶方阵 $nvertex_k$，其中 $nvertex_k[v][v']$ 的值为在从 v 到 v' 的中间允许有顶点 $v_0, v_1,\ldots,v_{k-1}$ 的最短路径上，顶点 v 的后继顶点 v'' 的下标（与前面 A_k 对应）
  - 到最后 v'' 到 v' 的最短路径也应已知，可以根据后继顶点追溯
- 初始时，如果 $A_0[i][j] = \infty$（没有边），则令 $nvertex_0[i][j] = -1$，否则就令 $nvertex_0[i][j] = j$，表示 v_j 是 v_i 的后继顶点。
- 在由 A_k 计算 A_{k+1} 时，若 $A_{k+1}[i][j]$ 被设置为 $A_k[i][k] + A_k[k][j]$，那么就设 $nvextex_{k+1}[i][j] = nvertex_k[i][k]$（在从 v_i 到 v_j 的路径上 v_i 的后继顶点就是原来 v_i 到 v_k 的路径上 v_i 的后继顶点）

  这轮计算完成时，$nvertex_{k+1}[i][j]$ 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。
- 计算完成最后，$nvertex_n[i][j]$ 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

Floyd 算法

- Floyd 算法从 A_0 = A（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：是否真需要用一个新的两维表存放下一个矩阵？

- 假设已经算出了 A_k 保存在矩阵 A 里，现考虑 A_{k+1} 的计算。公式是：

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

- 注意：所有新的 $A_{k+1}[i][j]$ 或者就是 $A_k[i][j]$（如果它比较小），或者是由下面的一列和一行中的元素求和计算出来的：

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

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

- 注意：如果在计算 A_{k+1} 过程中可能修改矩阵第 k 行或第 k 列元素，在后面取元素 [i][k] 或 [k][j] 得到的将不是 A_k 的元素，就必须保留 A_k

  即，如果计算中修改了后面要用的元素，那就需要另外用一个新矩阵，否则就不需要另外用一个新矩阵，可以直接在矩阵里修改

Floyd 算法

- 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，因为：

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

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

  而 A_{k-1} 的对角线元素 $A_{k-1}[m, m]$ 总是 0（对所有m），所以

  $A_{k+1}[i][k] = A_k[i][k]$　　第 k 行不变

  $A_{k+1}[k][j] = A_k[k][j]$　　第 k 列不变

- 因此，计算中可以用同一个 A 实现所有的 A_k，矩阵的递推计算过程可以通过直接更新 A 里元素的方式实现
  - 计算 $A_{k+1}[i][j] = \min\{A_k[i][j], A_k[i][k]+Ak[k][j]\}$，其中新矩阵元素的生成用赋值 A[i][j] := A[i][k] + A[k][j] 实现
  - 计算所有顶点对之间最短路径的长度，只需要一个矩阵
- 算法 all_shortest_paths 用了两个 n×n 矩阵成员：a 记录已知最短路径长度，nvextex 记录已知最短路径上的下一顶点

Floyd 算法

实现 Floyd 算法的 Python 函数：

```
def all_shortest_paths(graph):
    vnum = graph.vertex_num()
    a = [[graph.get_edge(i, j) for j in range(vnum)]
         for i in range(vnum)] # create a copy the adjacent matrix
    nvertex = [[-1 if a[i][j] == infinity else j for j in range(vnum)]
                for i in range(vnum)]
    for k in range(vnum):
        for i in range(vnum):
            for j in range(vnum):
                if a[i][j] > a[i][k] + a[k][j]:
                    a[i][j] = a[i][k] + a[k][j]
                    nvertex[i][j] = nvertex[i][k]
    return (a, nvertex)
```


Floyd 算法的复杂度

- Floyd 算法的复杂度分析非常简单
- 时间复杂度：
  - 算法的初始化部分是一个循环，其外层循环共执行 |V| 次，内层循环也执行 |V| 次，初始化部分的时间复杂度为 $O(|V|^2)$
  - 迭代生成矩阵 a 和路径 nvextex 的部分是一个三重循环，时间复杂度为 $O(|V|^3)$，这也是 Floyd 算法的时间复杂度
- 空间复杂度：
  - 两个结果矩阵，另外用了几个辅助变量
  - 存放计算结果需要 $O(|V|^2)$ 的空间

最短路径

- 两个最短路径算法，其中都蕴涵着有趣的想法
- **Dijkstra** 算法基于类似于最小生成树的想法，或者说是一种“宽度优先搜索”。其中用到了类似 **MST** 的性质
  - 它逐个找出可以确定最短路径的顶点，同时也找到了到新确定最短路径的顶点的路径
  - 做完前一步后更新信息，保证记录的都是至今已知的最短路径
  - 这是典型的动态规划方法（在计算中保留一些信息支持动态决策）
- **Floyd** 算法基于完全不同的考虑，求解所有顶点间的最短路径
  - 其基本方法是为最终问题的解决逐步积累信息，根据已有的信息更新包含部分信息的解的雏形，最终得到问题的解
  - 这一算法也是一个典型的动态规划方法
  - 过程中求子结构（子问题）的最优解，最后得到原问题的最优解
- 两个算法都值得认真学习理解。当然，其中 **Floyd** 算法不适合人做，因为其中的操作更缺乏直观

AOV 网

- 考虑有向图的一类应用“方式”，用图中的顶点表示某种“工程”里的活动，图中的边表示活动之间的先后顺序关系。这样的有向图称为顶点活动网（**Activity On Vertex network**），或称 **AOV** 网
- **AOV**网中的边常用于表示活动之间的制约关系
  - 要解决的一个问题就是根据图中制约关系做出有关活动的安排
  - 这样考虑，可以把 **AOV** 网络用于各种工程计划
- 在一些问题里，**AOV** 网的顶点或边还可能带有权值。可以考虑例如
  - “最优”安排问题
  - 最大或最小流问题
- 简单 **AOV** 网的一个典型实例是大学课程的先修关系
  - 课程知识有前后联系，一门课可能以其他课程的知识为基础
  - 当学生想选某门课程时，要看看是否已修过所有先修课程

拓扑排序

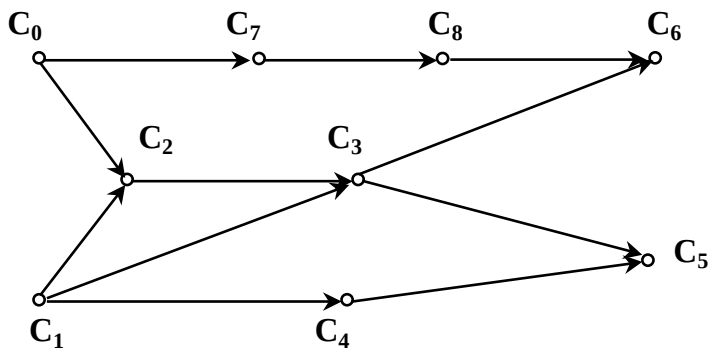
- 例：计算机专业学生必须完成规定的基础课和专业课才能毕业，这时的“工程”就是完成给定课程的学习计划，而活动就是学习课程。假设课程的名称与代号如表所示

  课程之间有先修关系，学习一门课之前必须完成其所有先修课程

课程编号	课程名	先修课程
C0	高等数学	
C1	程序设计	
C2	离散数学	C0, C1
C3	数据结构	C1, C2
C4	程序设计语言	C1
C5	编译技术	C3, C4
C6	操作系统	C3, C8
C7	物理学	C0
C8	计算机原理	C7

拓扑排序

- 可以用 AOV 网表示课程之间的关系
  - 图中顶点表示课程，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，在表示课程关系的 AOV 网中就加入一条有向边 $<C_i, C_j>$。
- 前面表中各课程的AOV网如下图所示

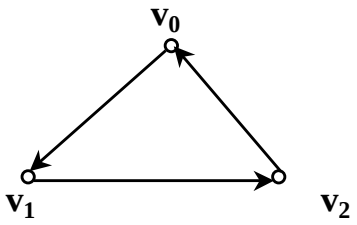


拓扑排序

- 拓扑排序是 AOV 网上的一种操作
- 定义：对给定的 AOV 网 N，如果 N 中的所有顶点能排成一个线性序列 $S = v_{i1}, v_{i2}, \ldots, v_{in}$，满足：

  如果 N 中存在从顶点 v_i 到 v_j 的路径，那么 S 里 v_i 排在 v_j 之前

  则 S 称为 N 的一个拓扑序列，构造拓扑序列的操作称为拓扑排序
- AOV 网未必有拓扑序列。不难证明，一个 AOV 网存在拓扑序列，当且仅当它不包含回路（存在回路，意味着某些活动的开始要以其自己的完成作为先决条件，这种现象称为活动之间的死锁）
- 例如，下图的 AOV 网没有拓扑序列：



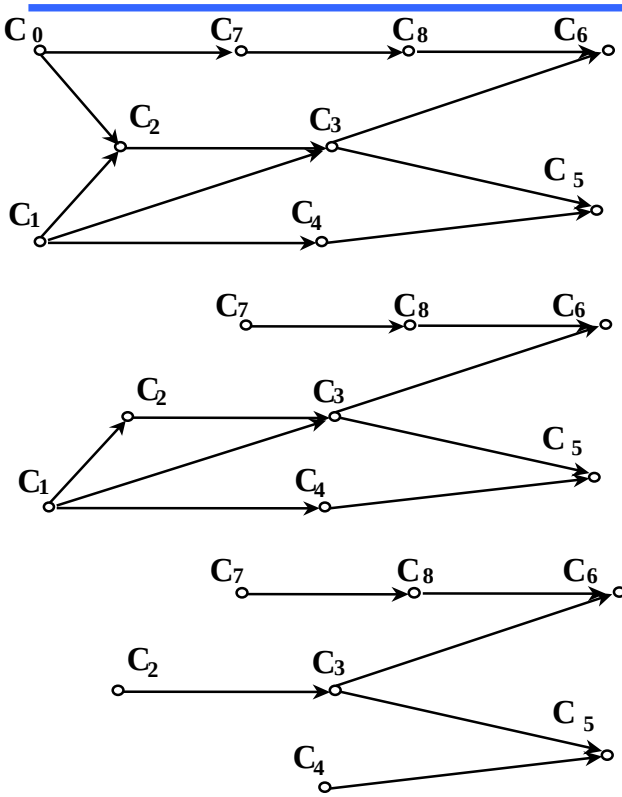
拓扑排序

- 性质：如果一个 AOV 网有拓扑序列，其拓扑序列未必唯一
- 性质：将 AOV 网 N 的拓扑序列反向得到的序列，都是 N 的逆网（即是把原网的所有边反转得到的 AOV 网）的拓扑序列
- 假设用 AOV 网表示一个工程的安排
  - 网中顶点代表工程中的活动（工序），顶点之间的有向边代表活动之间的制约关系（前一活动完成后，后一活动才能进行）
  - 如果在实际条件下各种动作只能串行进行，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
- 任何无回路 AOV 网都可以做出拓扑序列，方法很简单：
  - 从 AOV 网中选出一个入度为 0 的顶点作为序列的下一顶点
  - 从 AOV 网中删除此顶点及其所有的出边

  反复执行上面两步操作，直到输出了所有可输出顶点时拓扑排序结束

  如果剩下入度非 0 顶点就说明该 AOV 网里存在回路，无拓扑序列

拓扑排序：示例



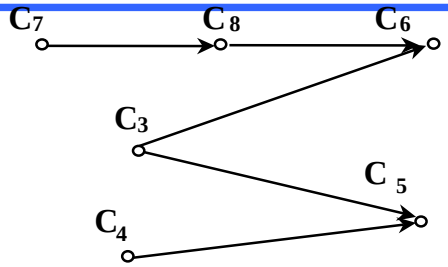
C_0

出现多个入度为0的顶点时，就出现了一次任意选择

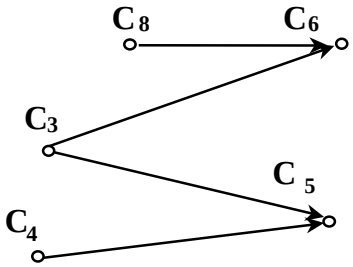
$C_0, C_1,$

$C_0, C_1, C_2,$

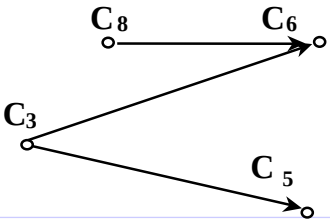
拓扑排序：示例



$C_0, C_1, C_2, C_7,$



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



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

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

拓扑排序：算法

- 现在考虑实现拓扑排序的程序
- 算法中用一个连续表 **indegree** 记录顶点的入度。由于工作中需要反复查找入度 **0** 的顶点，每次扫描数组影响效率，这里用了一种技巧：
  - 在 **indegree** 里维持一个“**0** 度表”记录入度为 **0** 的顶点
  - 用变量 **zerov** 记录“第一个”入度 **0** 的顶点下标，**indegree[zerov]** 里记录下一入度为 **0** 的顶点下标，依此类推；如最后的入度为 **0** 的顶点是 **zv**，在 **ingegree[zv]** 存特殊值 **–1**
  - 这个 **0** 度表就像是在 **indegree** 里保存了一个栈，**zerov** 记录栈中第一个顶点的下标，**–1** 表示栈结束
- **topological_sort** 的基本工作过程是：
  - 找出所有顶点的入度存入 **indegree**，确定其中入度为 **0** 的顶点
  - 反复选择入度为 **0** 的顶点并维护 **0** 度表
  - 最后返回拓扑序列，失败（无拓扑序列）时返回 **False**

拓扑排序：算法

```
def toposort(graph):
    vnum = graph.vertex_num()
    indegree, toposeq, zerov = [0]*vnum, [], -1
    for vi in range(vnum):
        for v, w in graph.out_edges(vi): indegree[v] += 1
    for vi in range(vnum):
        if indegree[vi] == 0:
            indegree[vi] = zerov; zerov = vi
    for n in range(vnum):
        if zerov == -1: return False # Thereis no topo-seq
        toposeq.append(zerov)
        vi = zerov; zerov = indegree[zerov]
        for v, w in graph.out_edges(vi):
            indegree[v] -= 1
            if indegree[v] == 0:
                indegree[v] = zerov; zerov = v
    return toposeq
```

拓扑排序：算法分析

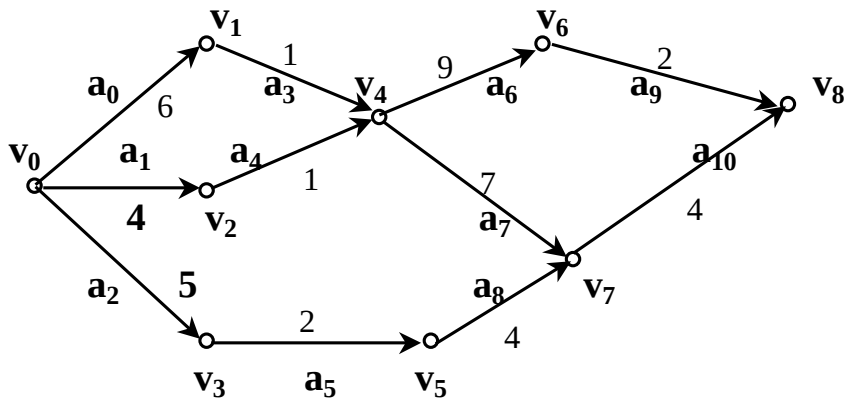
- 时间复杂性：
  - 设置 **indegree** 初值用的两重循环时间复杂度为 **O(max(|E|, |V|))**，检查入度为零的顶点需要 **O(|V|)** 时间
  - 工作的主要部分是一个两重循环时间复杂度也是 **O(max(|E|, |V|))**
  - 所以整个算法的时间复杂度为 **O(|E| + |V|)**，对于连通图就是 **O(|E|)**
- 空间复杂性：
  - **indegree** 和 **toposeq** 都是 **|V|** 大小的数组，**O(|V|)**
- 如果图用邻接矩阵表示，矩阵里可能有许多非边元素，矩阵很稀疏时会浪费很多空间，处理它也花费很多时间：
  - 在设置顶点入度中检查每条边一次，时间 $O(|V|^2)$
  - 主循环的时间复杂度也为 $O(|V|^2)$
  - 因此，采用这种表示的算法时间复杂度为 $O(|V|^2)$

AOE 网

- **AOE网（Activity On Edge network）**是另一类常用带权有向图，这是一种重要 **PERT（Program Evaluation and Review Technique）**模型
  - 最早在美国军方支持下开发出来，用于大型工程的计划和管理
  - 其雏形在 **1940** 年代用于美国原子弹开发的曼哈顿计划
  - 有许多实际的工程应用
- 抽象地看，**AOE网**是一种无环的带权有向图，其中
  - 顶点表示事件，有向边表示活动
  - 边上的权值通常表示活动的持续时间
  - 一个顶点表示的事件，也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（事件）
- 实际工程或事务里的一批相关活动，可以用一个 **AOE** 网描述（抽象），然后基于这种网考虑活动的安排问题

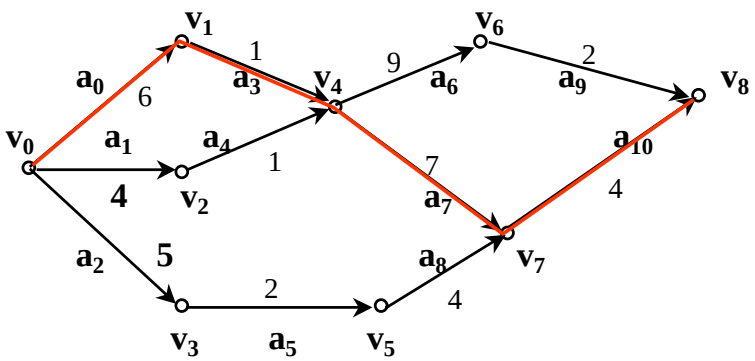
AOE 网

- 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
- 图中显示，活动 a_0 需要 6 个单位时间完成，活动 a_1 需要 4 个单位时间完成，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



关键路径

- AOE 网中的活动可以并行进行
  - 只要一项活动（边）的前提事件均已发生（以该边的始点为终点的所有活动都完成），这项活动就可以开始
  - 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
  - 在 AOE 网上最重要的一项计算是找出其中的关键路径



v_0, v_1, v_4, v_7, v_8 是一条关键路径，长度为18，所以，完成整个工程至少需要 18 个单位时间

关键路径

- 现在需要开发一个算法，确定AOE网的关键路径
- 下面总假定 v_0 是开始事件，v_{n-1} 是结束事件，$w(<v_i,v_j>)$为 $<v_i,v_j>$ 的权
- 首先定义几组变量，用它们记录关键路径计算中确定的信息

1、事件 v_j 的最早可能发生时间 **ee[j]**，根据它之前的事件和活动确定，不可能更早发生。 **ee[j]** 可以递推计算：

ee[0] = 0

ee[j] = max{ ee[i] + w(<v_i ,v_j>) |　<v_i ,v_j>∈T }，1⩽j⩽n-1

T 是所有以 v_j 为终点的入边集

2、事件 v_i 的最迟允许发生时间 **le[i]**，更晚发生将延误整个工程的进度。可以根据已知的 **ee** 值反向地递推计算：

le[n-1] = ee[n-1]

le[i] = min{ le[j] - w(<v_i, v_j>) | <v_i ,v_j>∈S}，0⩽i⩽n-2

S 是所有以 v_i 为开始顶点的出边集

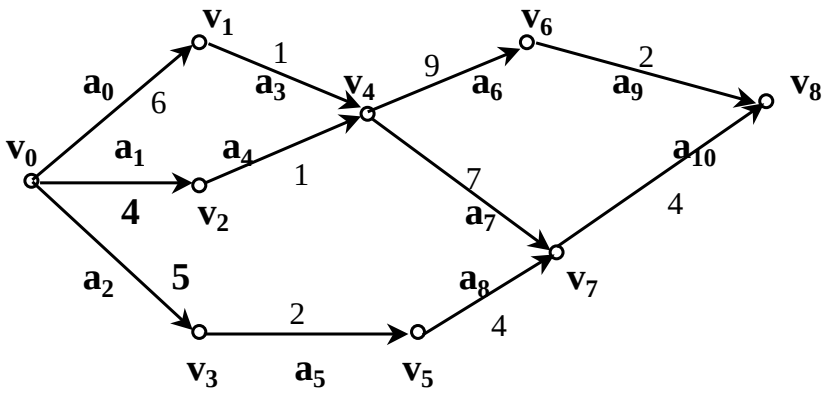
关键路径

3、活动 a_k=< v_i ,v_j> 的最早可能开始时间 **e[k] = ee[i]**，最迟允许开始时间 **l[k] = le[j] - w(<v_i,v_j>)**（在保证整个工程不拖延工期的情况下）

- 所有 **e[k] = l[k]** 的活动 a_k 称为该 **AOE** 网里的关键活动，因为它们中的任何一个推迟开始，都会延误整个工程的工期
- 差 **l[k] - e[k]** 表示完成活动 a_k 的时间余量，这是在不延误整体工期的前提下，活动 a_k 可以推迟的时间量
- 由关键活动构成的从初始点到终点的路径就是关键路径（可能不止一条，可以同时得到）

关键路径：示例

例：求图中的AOE网的关键路径

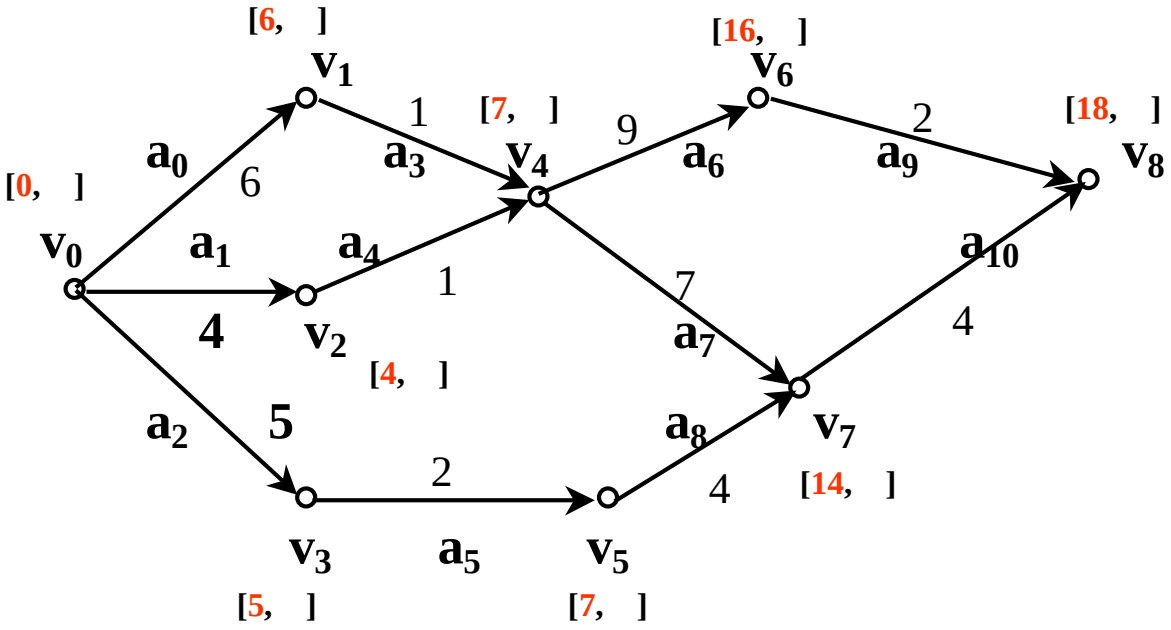


工作过程：按前面公式分别求出事件的最早可能发生时间和最迟允许发生时间，活动的最早可能开始时间和最晚允许开始时间

为求事件的最早发生时间，必须知道其所有前驱事件的最早发生时间；求最迟发生时间，必须知道所有后继事件的最迟发生时间。因此需要图结点的一个拓扑排序，按拓扑序列里的事件顺序计算。下面工作中取拓扑序列 $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

关键路径：示例

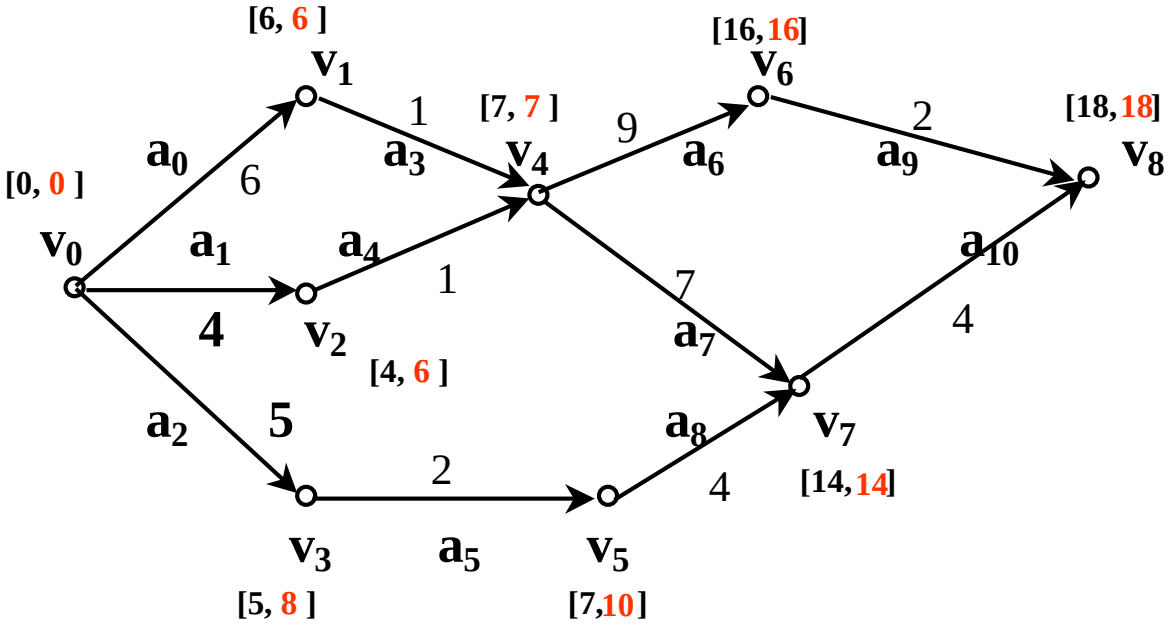
向前，计算事件的最早可能发生时间



拓扑序列：$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

关键路径：示例

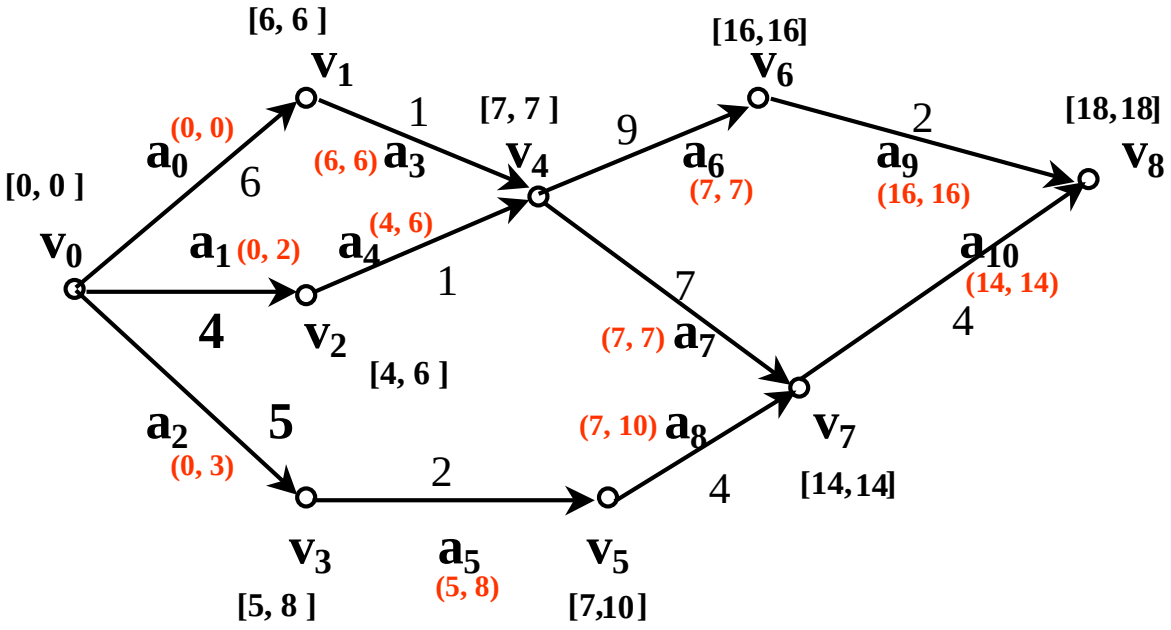
向后，计算事件的最迟允许发生时间



拓扑序列：$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

关键路径：示例

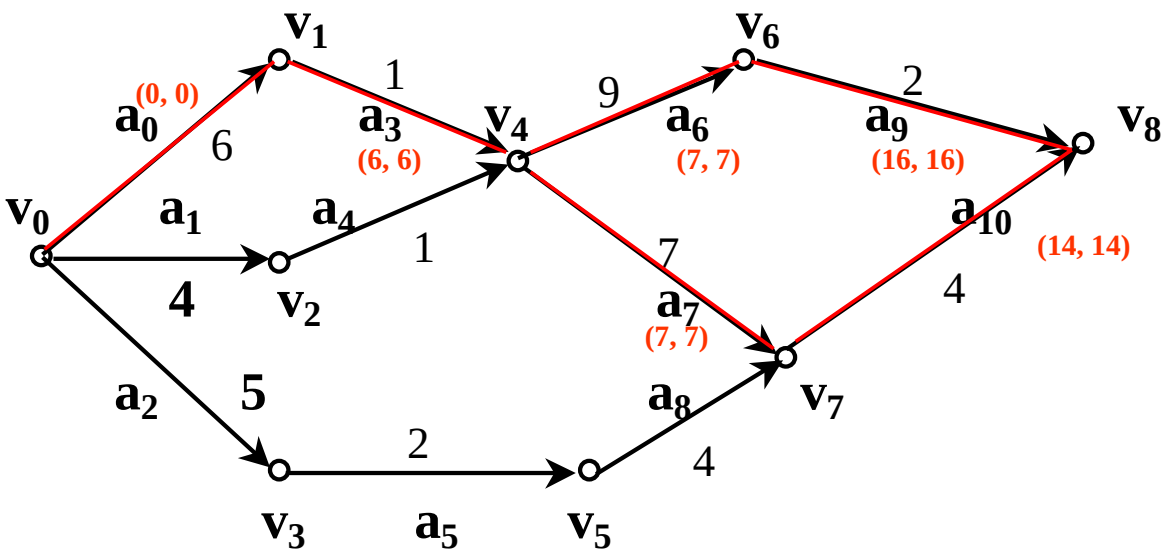
计算活动的最早和最迟开始时间



拓扑序列：$v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$

关键路径：示例

关键活动和关键路径



关键路径：算法

- 算法的实现直截了当，需要一步步计算出有关的信息
- 算法过程分几步
  1. 是生成 AOE 网的一个拓扑序列
  2. 生成 ee 数组的值，应该按拓扑序列的顺序计算
  3. 生成 le 数组的值，应该按拓扑序列的逆序计算
  4. 最后的 e 和 l 数据组可以一起计算（太简单）。如果只需要得到关键路径，也可以不显式表示，而是直接用它们相等求得结果
- 在下面算法里
  - 步骤 1 直接调用前面定义的拓扑排序过程
  - 步骤 2 和 3 定义为独立的过程
  - 最后一步直接计算，收集起确定的关键活动，就能得到 AOE 网的所有关键路径。如前所述，得到的关键路径可能不止一条

关键路径：算法

主函数返回找到的关键活动，可能表示多条关键路径：

```
def critical_path(graph):
    toposeq = toposort(graph)
    if toposeq == False: return False
    vnum = graph.vertex_num()
    ee, le = [0]*vnum, [infinity]*vnum
    crtPath = []
    setEventE(vnum, graph, toposeq, ee)
    setEventL(vnum, graph, toposeq, ee[vnum-1], le)
    for i in range(vnum):
        for j, w in graph.out_edges(i):
            if ee[i] == le[j] - w: # a critical action
                crtPath.append([i, j, ee[i]])
    return crtPath
```


关键路径：算法

建立 ee 数组时其元素都初始化为 0。算法中按照拓扑顺序逐个更新，有路径结束更晚时更新相应的 ee 值（最早可能时间）

```
def setEventE(vnum, graph, toposeq, ee):
    for k in range(vnum-1): # 最后一个顶点不必做
        i = toposeq[k]
        for j, w in graph.out_edges(i):
            if ee[i] + w > ee[j]: ee[j] = ee[i] + w # 事件 j 需更晚结束
```

先把 le 元素都赋为最后顶点的最迟时间，然后按拓扑排序的逆序向前逐个更新最迟允许时间（le 的值）

```
def setEventL(vnum, graph, toposeq, eelast, le):
    for i in range(vnum): le[i] = eelast
    for k in range(vnum-2, -1, -1):# 逆拓扑顺序, 最后顶点不必做
        i = toposeq[k]
        for j, w in graph.out_edges(i):
            if le[j] - w < le[i]: le[i] = le[j] - w  # 事件 i 需更早开始
```

关键路径：算法分析

- 算法复杂度：采用图的邻接表表示
  - 拓扑排序时间复杂度为 **O(|V|+|E|)**
  - 求事件的最早可能时间和允许最迟时间，活动的最早开始时间和最晚开始时间，都要对图中所有顶点及每个顶点边表中所有的边结点各检查一次，时间复杂度为 **O(|V|+|E|)**
  - 因此，求关键路径算法的时间复杂度为 **O(|V|+|E|)**

  如果图用邻接矩阵表示，算法至少需要 $O(|V|^2)$ 时间
- 空间复杂度：保存拓扑序列和 **2** 个保存事件时间的数组，**O(|V|)**
- 总结一下这两个算法：
  - 拓扑序列是有向无环图的一个重要概念，拓扑排序算法的思想简单。但求拓扑序列是许多有向图算法的基础，这个概念很重要
  - 关键路径是带权有向无环图的一种重要概念，广泛用于工程规划领域。算法需要按拓扑顺序遍历结点，计算顶点和边的最早最迟时间

图和图算法：总结

- 图是一种复杂的非线性结构。本章开始介绍了
  - 图的基本概念
  - 两种常用的表示方法（相邻矩阵和邻接表）及其 **Python** 实现
- 图的计算问题和算法
  - 宽度优先和深度优先周游
  - 最小生成树
  - 最短路径（单源点的和任意顶点之间的）
  - 拓扑排序及关键路径
- 本章重点是：
  - 掌握图的概念，性质和存储表示
  - 掌握（除 **Floyd** 算法之外）各种算法的工作过程