数据结构与算法期末复习
Homework

尹超

中国科学院大学，北京 100049

Carter Yin

University of Chinese Academy of Sciences, Beijing 100049, China

2024.8 – 2025.1

序言

本文为笔者数据结构与算法的期末复习笔记。
望老师批评指正。

目录

题目 第十章图应用

169

> 如果把朋友圈视为一无向图，那么即使 A 君看不到你给 B 点的赞，你们仍可能属于同一双连通分量。

Solution 1. 这个说法是**正确**的。

详细分析：

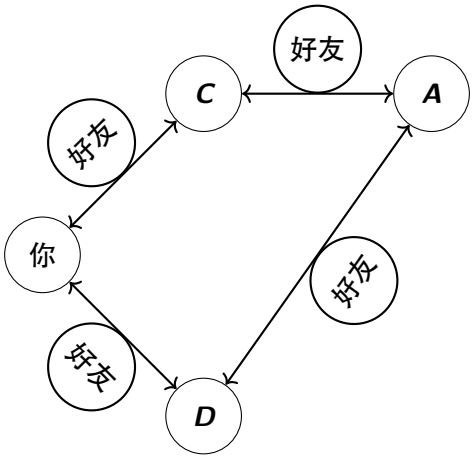
***(1)* 图模型解释：**

- **朋友圈 = 无向图**：每个用户是一个顶点，好友关系是一条无向边。
- ***A* 君看不到你的赞**：这通常意味着你和 *A* 君不是直接好友。在图模型中，即顶点“你”和顶点“*A*”之间没有直接的边。
- **双连通分量 *(Biconnected Component)***：这是图的一个子图，其中任意两个顶点之间都至少存在两条**顶点不相交**的路径。通俗地讲，这意味着移除分量中的任意一个“中间人”（顶点），你和 *A* 君之间仍然是连通的。这是一种非常“紧密”和“鲁棒”的社群结构。

***(2)* 举例说明：** 你和 *A* 君不直接是好友，但你们有两个或更多的共同好友。例如，*C* 和 *D*。

- 你是 *C* 的好友，*A* 也是 *C* 的好友。
- 你是 *D* 的好友，*A* 也是 *D* 的好友。

这就构成了一个环路：你 – *C* – *A* – *D* – 你。



在这个结构中：

- 你和 *A* 之间没有直接的边。
- 存在路径 *1*：你 → *C* → *A*。
- 存在路径 *2*：你 → *D* → *A*。

这两条路径除了起点（你）和终点（*A*）之外，没有任何共同的顶点。因此，你和 *A* 处于同一个双连通分量中。即使你们的共同好友 *C*（或 *D*）注销了账号，你们依然可以通过另一位共同好友 *D*（或 *C*）保持联系。

结论： 没有直接联系（边）并不妨碍两个顶点同属于一个双连通分量。只要它们之间存在至少两条独立的路径，它们的关系就是双连通的。

170

> 对于同一无向图，起始于顶点 s 的 DFS 尽管可能得到结构不同的 DFS 树，但 s 在树中的度数必然固定。

Solution 2. 这个说法是**正确**的。

详细分析：

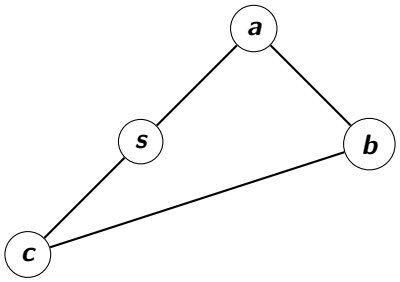
(1) DFS 树中顶点的度： 在 *DFS* 树中，一个顶点 *'u'* 的度数等于它作为父节点所拥有的子节点数量。一个邻居 *'v'* 成为 *'u'* 的子节点，当且仅当 *DFS* 通过边 *'(u, v)'* 第一次访问到 *'v'* 时（即 *'v'* 尚未被访问过）。

(2) 关键思想：邻居的连通性

- 设 *'s'* 在原图中的所有邻居构成集合 *'N(s)'*。
- 考虑将顶点 *'s'* 从原图中移除后，由 *'N(s)'* 中的顶点所构成的子图。这个子图可能会被分为一个或多个连通分量。
- 当 *DFS* 从 *'s'* 开始，并选择其第一个邻居（比如 *'v_1'*）进行深入访问时，这次递归调用将会访问到与 *'v_1'* 在 *'N(s)'* 的子图中属于同一个连通分量的所有其他邻居。
- 因此，当 *DFS* 从 *'v_1'* 的递归中返回到 *'s'* 时，*'s'* 再去检查 *'N(s)'* 中与 *'v_1'* 属于同一连通分量的其他邻居时，会发现它们都已经被访问过了，于是不会再与它们形成树边。

(3) 结论： *'s'* 在 *DFS* 树中的度数，精确地等于 ****'s'*** 的邻居集 *'N(s)'* 在移除 *'s'* 后的子图中所形成的连通分量的数量 **。这个数量是一个图的结构性属性，它不随 *DFS* 遍历邻居的顺序而改变。因此，无论 *DFS* 树的结构如何变化，*'s'* 在树中的度数是固定的。

(4) 示例： 考虑一个环 *'s-a-b-c-s'*。



- *'s'* 的邻居是 *'a'* 和 *'c'*。
- 移除 *'s'* 后，*'a'* 和 *'c'* 通过路径 *'a-b-c'* 仍然是连通的。它们属于同一个连通分量。
- 因此，*'s'* 在 *DFS* 树中的度数必然是 *1*。
- **验证：**
  - 若先访问 *'a'*：*DFS* 路径为 *'s->a->b->c'*。*'c'* 被访问后，返回到 *'s'* 时发现 *'c'* 已被访问。*'s'* 的孩子只有 *'a'*，度为 *1*。
  - 若先访问 *'c'*：*DFS* 路径为 *'s->c->b->a'*。*'a'* 被访问后，返回到 *'s'* 时发现 *'a'* 已被访问。*'s'* 的孩子只有 *'c'*，度为 *1*。

171

> 已知有向图 G=(V, E)，其中 V = {v1, v2,v3, v4, v5, v6}，
> E = {<v1,v2>, <v1,v4>,<v2,v6>, <v3,v1>, <v3,v4>, <v4,v5>,<v5,v2>, <v5,v6>}。
> G 的拓扑序列是:
> A. v1, v3, v4, v5, v2, v6
> B. v3, v4, v1, v5, v2, v6
> C. v1, v4, v3, v5, v2, v6
> D. v3, v1, v4, v5, v2, v6

Solution 3. 正确答案是 *D*。

详细分析:

拓扑排序的核心规则是：对于图中的任意一条有向边 *'<u, v>'*，顶点 *'u'* 必须出现在顶点 *'v'* 之前。我们可以逐一检查每个选项是否违反了这个规则。

图中的边为:

- *<v1,v2>, <v1,v4>*
- *<v2,v6>*
- *<v3,v1>, <v3,v4>*
- *<v4,v5>*
- *<v5,v2>, <v5,v6>*

检查各个选项:

- ***A. v1, v3, v4, v5, v2, v6***
  检查边 *'<v3, v1>'*。在这个序列中，*'v3'* 出现在 *'v1'* 之后，违反了规则。因此 *A* 是错误的。
- ***B. v3, v4, v1, v5, v2, v6***
  检查边 *'<v1, v4>'*。在这个序列中，*'v1'* 出现在 *'v4'* 之后，违反了规则。因此 *B* 是错误的。
- ***C. v1, v4, v3, v5, v2, v6***
  检查边 *'<v3, v1>'*。在这个序列中，*'v3'* 出现在 *'v1'* 之后，违反了规则。因此 *C* 是错误的。
- ***D. v3, v1, v4, v5, v2, v6***
  我们来检查所有边:
  - *'<v1,v2>': v1 在 v2 之前*，正确。
  - *'<v1,v4>': v1 在 v4 之前*，正确。
  - *'<v2,v6>': v2 在 v6 之前*，正确。
  - *'<v3,v1>': v3 在 v1 之前*，正确。
  - *'<v3,v4>': v3 在 v4 之前*，正确。
  - *'<v4,v5>': v4 在 v5 之前*，正确。
  - *'<v5,v2>': v5 在 v2 之前*，正确。
  - *'<v5,v6>': v5 在 v6 之前*，正确。

  所有边的约束都得到了满足。因此 *D* 是一个正确的拓扑序列。

另一种方法：使用 *Kahn* 算法（基于入度）

(1) 计算所有顶点的入度：*in(v1)=1, in(v2)=2, in(v3)=0, in(v4)=2, in(v5)=1, in(v6)=2*。

(2) 将所有入度为 *0* 的顶点（只有 *v3*）放入队列。队列：*'[v3]'*。

(3) 从队列中取出 *v3*，加入拓扑序列。序列：*'[v3]'*。更新 *v3* 的邻居（*v1, v4*）的入度：*in(v1)=0, in(v4)=1*。
将新的入度为 *0* 的顶点（*v1*）入队。队列：*'[v1]'*。

(4) 从队列中取出 *v1*，加入拓扑序列。序列：*'[v3, v1]'*。更新 *v1* 的邻居（*v2, v4*）的入度：*in(v2)=1, in(v4)=0*。将新的入度为 *0* 的顶点（*v4*）入队。队列：*'[v4]'*。

(5) 从队列中取出 *v4*，加入拓扑序列。序列：*'[v3, v1, v4]'*。更新 *v4* 的邻居（*v5*）的入度：*in(v5)=0*。将新的入度为 *0* 的顶点（*v5*）入队。队列：*'[v5]'*。

(6) 从队列中取出 *v5*，加入拓扑序列。序列：*'[v3, v1, v4, v5]'*。更新 *v5* 的邻居（*v2, v6*）的入度：*in(v2)=0, in(v6)=1*。将新的入度为 *0* 的顶点（*v2*）入队。队列：*'[v2]'*。

(7) 从队列中取出 *v2*，加入拓扑序列。序列：*'[v3, v1, v4, v5, v2]'*。更新 *v2* 的邻居（*v6*）的入度：*in(v6)=0*。将新的入度为 *0* 的顶点（*v6*）入队。队列：*'[v6]'*。

(8) 从队列中取出 *v6*，加入拓扑序列。序列：*'[v3, v1, v4, v5, v2, v6]'*。

算法得到的拓扑序列与选项 *D* 完全一致。

172

> 在拓扑排序算法中用堆栈和用队列产生的结果会不同吗？A. 有可能会不同 B. 肯定是相同的 C. 以上全不对 D. 是的肯定不同

Solution 4. 正确答案是 *A*。

详细分析：

对于一个有向无环图（*DAG*），其拓扑序列通常不是唯一的。使用不同的数据结构（队列或栈）来实现拓扑排序，会影响在遇到多个选择时处理顶点的顺序，因此**有可能**产生不同的、但同样有效的拓扑序列。

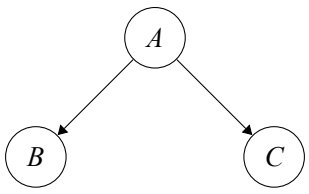
***(1)* 使用队列（*Kahn* 算法）：**

- 该算法首先找到所有入度为 *0* 的顶点，并将它们放入一个队列中。
- 然后，它不断地从队列头部取出一个顶点，加入到拓扑序列中，并将其所有出边“删除”（即将其邻居的入度减 *1*）。如果邻居的入度变为 *0*，则将其加入队列尾部。
- 这种方法是按“层”处理的，具有广度优先（*BFS*）的特点。

***(2)* 使用栈（基于 *DFS* 的算法）：**

- 该算法对图进行深度优先搜索（*DFS*）。当一个顶点的所有后代都被访问完毕后（即该顶点即将完成时），将该顶点压入一个栈中。
- 整个 *DFS* 结束后，从栈中依次弹出所有顶点，得到的序列就是一个拓扑序列。
- 这种方法具有深度优先（*DFS*）的特点，会沿着一条路径走到尽头再回溯。

示例：考虑以下简单的有向无环图：



- **用队列：**
  - ***(1)*** 初始时，只有 *A* 的入度为 *0*。队列：*'[A]'*。
  - ***(2)*** *A* 出队。序列：*'[A]'*。*B* 和 *C* 的入度都变为 *0*。
  - ***(3)*** 将 *B* 和 *C* 入队。如果先 *B* 后 *C*，队列为 *'[B, C]'*。最终序列为 ***A, B, C***。
  - ***(4)*** 如果先 *C* 后 *B*，队列为 *'[C, B]'*。最终序列为 ***A, C, B***。
- **用栈（*DFS*）：**

(1) 从 *A* 开始 *DFS*。

(2) 访问 *A* 的邻居，如果先访问 *B*，则 *DFS* 路径为 *A -> B*。*B* 没有后代，*B* 完成，*B* 入栈。

(3) 回溯到 *A*，访问另一个邻居 *C*。*DFS* 路径为 *A -> C*。*C* 没有后代，*C* 完成，*C* 入栈。

(4) 回溯到 *A*，*A* 的所有邻居都已访问，*A* 完成，*A* 入栈。

(5) 栈中内容（从栈顶到栈底）：*A, C, B*。

(6) 依次弹出，得到序列 ***A, C, B***。

在这个例子中，队列可以产生 *'A, B, C'*，而栈（*DFS*）可以产生 *'A, C, B'*。由于可以产生不同的结果，所以答案是"有可能会不同"。

173

> 若将 n 个顶点 e 条弧的有向图采用邻接表存储，则拓扑排序算法的时间复杂度是：A. O(n×e) B. O(n^2) C. O(n+e) D. O(n)

Solution 5. 正确答案是 *C*。

详细分析：

拓扑排序主要有两种经典的实现算法：基于 *Kahn* 算法（入度）和基于 *DFS*。我们来分析这两种算法在使用邻接表存储时的复杂度。

1. 基于 Kahn 算法（使用队列和入度）

(1) **计算所有顶点的入度：**需要遍历整个邻接表一次。遍历所有顶点的邻接表，总共会访问 *'n'* 个顶点和 *'e'* 条边。因此，这部分的时间复杂度是 $O(n+e)$。

(2) **初始化队列：**将所有入度为 *0* 的顶点入队。这需要扫描所有 *'n'* 个顶点一次，时间复杂度为 $O(n)$。

(3) **主循环：**循环执行直到队列为空。

- 每个顶点最多入队一次，出队一次。所以对顶点的操作总共是 $O(n)$。
- 当一个顶点出队时，会遍历其所有的出边（即其在邻接表中的邻居列表）。在整个算法的生命周期中，每条边都只会被访问一次。所以对边的操作总共是 $O(e)$。

综合以上步骤，*Kahn* 算法的总时间复杂度为 $O(n+e)+O(n)+O(n)+O(e)=O(n+e)$。

2. 基于深度优先搜索（DFS）

(1) 拓扑排序可以通过 *DFS* 实现：对图进行 *DFS*，当一个顶点完成搜索后（即其所有后代都已被访问），将其加入一个列表的前端（或压入一个栈）。

(2) 对一个用邻接表表示的图进行完整的 *DFS* 遍历，需要访问每个顶点和每条边一次。

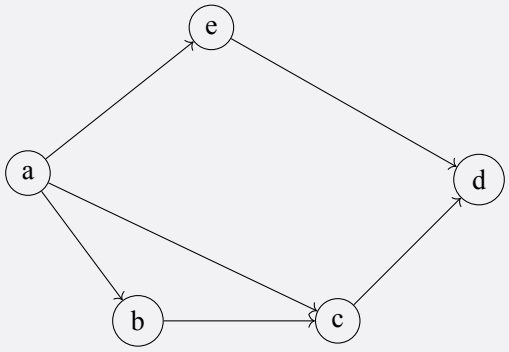
(3) 因此，*DFS* 本身的时间复杂度就是 $O(n+e)$。

(4) 将顶点加入列表或压栈的操作是 $O(1)$，对所有顶点操作总共是 $O(n)$。

DFS 算法的总时间复杂度由 *DFS* 遍历主导，因此也是 $O(n+e)$。

结论：无论使用哪种主流算法，在邻接表存储结构下，拓扑排序的时间复杂度都是 $O(n+e)$。

174

> 对下图进行拓扑排序，可以得到不同的拓扑序列的个数是：
>
> 
>
>
> A. 1
> B. 2
> C. 3
> D. 4

Solution 6. 正确答案是 *C*。

详细分析：

我们使用 *Kahn* 算法（基于入度）来找出所有可能的拓扑序列。

***(1)* 计算初始入度：**

- *in(a) = 0*
- *in(b) = 1 (*来自 *a)*
- *in(c) = 2 (*来自 *a, b)*
- *in(d) = 2 (*来自 *e, c)*
- *in(e) = 1 (*来自 *a)*

***(2)* 第一步：**

- 只有顶点 ***a*** 的入度为 *0*。所以任何拓扑序列都必须以 ***a*** 开头。
- 序列：*'[a]'*
- "移除"*a* 和它的出边 *(a,b), (a,c), (a,e)*。更新邻居的入度：
  - *in(b)* 变为 *0*
  - *in(c)* 变为 *1*
  - *in(e)* 变为 *0*

***(3)* 第二步：**

- 现在入度为 *0* 的顶点是 ***b*** 和 ***e***。这里出现了分支，我们可以选择先处理 *b*，也可以选择先处理 *e*。

(4)* 分支 *1*：先选择 *b

- 序列：*'[a, b]'*
- "移除"*b* 和它的出边 *(b,c)*。更新邻居的入度：*in(c)* 变为 *0*。
- 此时，入度为 *0* 的顶点是 ***c*** 和 ***e***。又出现了一个分支。
  - **分支 *1.1*：先选择 *c***
    - 序列：*'[a, b, c]'*

        * 移除 *c*，更新 *d* 的入度：*in(d)* 变为 *1*。
        * 此时只有 ***e*** 的入度为 *0*。
        * 序列：*'[a, b, c, e]'*
        * 移除 *e*，更新 *d* 的入度：*in(d)* 变为 *0*。
        * 最后选择 ***d***。
        * **得到序列 *1*：*a, b, c, e, d***
    - **分支 *1.2*：先选择 *e***
        * 序列：*'[a, b, e]'*
        * 移除 *e*，更新 *d* 的入度：*in(d)* 变为 *1*。
        * 此时只有 ***c*** 的入度为 *0*。
        * 序列：*'[a, b, e, c]'*
        * 移除 *c*，更新 *d* 的入度：*in(d)* 变为 *0*。
        * 最后选择 ***d***。
        * **得到序列 *2*：*a, b, e, c, d***

(5)* 分支 *2*：先选择 *e

- 序列：*'[a, e]'*
- "移除"*e* 和它的出边 *(e,d)*。更新邻居的入度：*in(d)* 变为 *1*。
- 此时，入度为 *0* 的顶点只有 ***b***。没有分支。
- 序列：*'[a, e, b]'*
- 移除 *b*，更新 *c* 的入度：*in(c)* 变为 *0*。
- 此时只有 ***c*** 的入度为 *0*。
- 序列：*'[a, e, b, c]'*
- 移除 *c*，更新 *d* 的入度：*in(d)* 变为 *0*。
- 最后选择 ***d***。
- **得到序列 *3*：*a, e, b, c, d***

综上所述，总共有 *3* 个不同的拓扑序列。

175

> Prim 算法是通过每步添加一条边及其相连的顶点到一棵树，从而逐步生成最小生成树。

Solution 7. 这个说法是**正确**的。

详细分析：

Prim 算法是一种用于在加权无向图中寻找最小生成树（*MST*）的贪心算法。其核心思想可以概括为"加点法"，具体步骤如下：

***(1)* 初始化：**

- 创建一个集合 *'U'*，用于存放已经在最小生成树中的顶点。
- 任意选择一个起始顶点 *'s'*，将其加入集合 *'U'*。
- 创建一个集合 *'T'*，用于存放最小生成树的边，初始为空。

***(2)* 循环生长：**重复以下步骤 *'n-1'* 次（*'n'* 是顶点总数）：

- 在所有连接集合 *'U'* 内的顶点与集合 *'V-U'*（即尚未加入树的顶点）外的顶点的边中，寻找一条权重最小的边 *'(u, v)'*，其中 *'u'* ∈ *'U'* 且 *'v'* ∈ *'V-U'*。

- 将这条权重最小的边 *'(u, v)'* 加入到集合 *'T'* 中。
- 将顶点 *'v'* 加入到集合 *'U'* 中。

***(3)* 结束：** 当所有顶点都加入到集合 *'U'* 中时，集合 *'T'* 中的所有边就构成了图的最小生成树。

结论： 这个过程完全符合题目的描述。*Prim* 算法在每一步都选择一条最"划算"的边，将一个新顶点"拉入"到正在构建的树中，从而使这棵树不断"生长"，直到覆盖所有顶点。它始终维护着一个连通的树结构。

176

> 数据结构中 Dijkstra 算法用来解决哪个问题？A. 字符串匹配 B. 最短路径 C. 拓扑排序 D. 关键路径

Solution 8. 正确答案是 *B*。

详细分析：

- ***B.* 最短路径 *(Shortest Path):*** *Dijkstra* 算法是解决**单源最短路径问题**的经典算法。它用于计算一个加权图中，从一个指定的起始顶点（源点）到所有其他顶点的最短路径。一个关键的限制是，*Dijkstra* 算法要求图中所有的边的权重都必须是**非负数**。
- ***A.* 字符串匹配 *(String Matching):*** 这是在一段文本中查找一个特定模式串的问题。常用的算法有 *KMP* 算法、*Boyer-Moore* 算法等，与图论无关。
- ***C.* 拓扑排序 *(Topological Sort):*** 这是对有向无环图（*DAG*）的顶点进行线性排序，使得对于每一条有向边 *'(u, v)'*，*'u'* 都在 *'v'* 的前面。常用的算法是 *Kahn* 算法或基于 *DFS* 的算法。
- ***D.* 关键路径 *(Critical Path):*** 这是在项目管理中，对一个表示工程流程的有向无环图（*AOV* 网）进行分析，找出从开始到结束耗时最长的路径。这条路径就是关键路径。虽然它也涉及路径计算，但它本质上是求解一个**最长路径**问题，并且是在拓扑排序的基础上进行的，与 *Dijkstra* 算法不同。

177

> 若从无向图的任意一个顶点出发进行一次深度优先搜索可以访问图中所有的顶点，则该图一定是 () 图。A. 非连通 B. 连通 C. 强连通 D. 有向

Solution 9. 正确答案是 *B*。

详细分析：

***(1)* 深度优先搜索（*DFS*）的性质：** 在任何图（有向或无向）中，从一个顶点 *'v'* 开始进行一次深度优先搜索，能够访问到的所有顶点集合，恰好是包含顶点 *'v'* 的那个**连通分量**中的所有顶点。

***(2)* 题目条件分析：** 题目指出"从**任意一个顶点**出发进行**一次**深度优先搜索可以访问图中**所有的顶点**"。

- "一次"*DFS* 能访问所有顶点，意味着这个图只有一个连通分量。
- "从任意一个顶点出发"都成立，这加强了上述结论。

***(3)* 连通图的定义：** 一个无向图被称为**连通图**，如果图中任意两个顶点之间都存在一条路径。这等价于说，这个图只有一个连通分量。

***(4)* 结论：** 题目所描述的性质，正是连通图的定义。因此，该图一定是连通图。

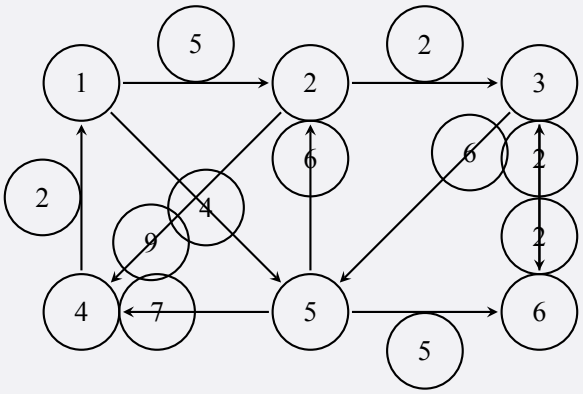
其他选项分析：

- ***A.* 非连通：** 如果图是非连通的，它至少有两个连通分量。从一个分量的顶点出发，一次 *DFS* 无法访问到另一个分量的顶点。

- ***C.* 强连通:** “强连通”是**有向图**的概念，指图中任意两个顶点之间都互相可达。对于无向图，其对应的概念就是“连通”。
- ***D.* 有向:** 题目明确说明是“无向图”。

178

使用 Dijkstra 算法求下图中从顶点 1 到其他各顶点的最短路径，依次得到的各最短路径的目标顶点是:



A. 5, 2, 6, 3, 4
B. 5, 2, 3, 6, 4
C. 5, 2, 4, 3, 6
D. 5, 2, 3, 4, 6

Solution 10. 正确答案是 *B*。

详细分析:

Dijkstra 算法的步骤如下:

- *‘S‘:* 已找到最短路径的顶点集合。
- *‘dist[v]‘:* 从源点 *1* 到顶点 *v* 的当前最短路径长度。

我们用一个表格来追踪算法的执行过程。

步骤	S (已确定集合)	dist 数组 (对未确定顶点)	选定顶点
初始	{1}	dist[2]=5, dist[4]=∞, dist[5]=4, dist[3,6]=∞	-
1	{1, **5**}	dist[2]=5, dist[4]=11, dist[6]=9, dist[3]=∞	**5** (dist=4)
2	{1, 5, **2**}	dist[4]=11, dist[6]=9, dist[3]=7	**2** (dist=5)
3	{1, 5, 2, **3**}	dist[4]=11, dist[6]=9	**3** (dist=7)
4	{1, 5, 2, 3, **6**}	dist[4]=11	**6** (dist=9)
5	{1, 5, 2, 3, 6, **4**}	-	**4** (dist=11)

过程分解:

(1) 初始化: *‘S = 1‘。‘dist[1]=0‘。1* 的邻居是 *2* 和 *5*，所以 *‘dist[2]=5‘, ‘dist[5]=4‘*。

(2) 第 1 轮:

- 在未确定的顶点中，*5* 的距离最短（*‘dist[5]=4‘*）。将 ***5*** 加入 *‘S‘*。
- 更新 *5* 的邻居:
  - 到 *2: ‘dist[5]+d(5,2) = 4+6=10‘ > ‘dist[2]=5‘*，不更新。

- 到 *4: 'dist[5]+d(5,4) = 4+7=11'*。更新 *'dist[4]=11'*。
- 到 *6: 'dist[5]+d(5,6) = 4+5=9'*。更新 *'dist[6]=9'*。

(3) 第 *2* 轮:

- 在未确定的顶点 *2,3,4,6* 中，*2* 的距离最短（*'dist[2]=5'*）。将 ***2*** 加入 *'S'*。
- 更新 *2* 的邻居:
  - 到 *3: 'dist[2]+d(2,3) = 5+2=7'*。更新 *'dist[3]=7'*。
  - 到 *4: 'dist[2]+d(2,4) = 5+9=14' > 'dist[4]=11'*，不更新。

(4) 第 *3* 轮:

- 在未确定的顶点 *3,4,6* 中，*3* 的距离最短（*'dist[3]=7'*）。将 ***3*** 加入 *'S'*。
- 更新 *3* 的邻居:
  - 到 *6: 'dist[3]+d(3,6) = 7+2=9' = 'dist[6]=9'*，不更新。

(5) 第 *4* 轮:

- 在未确定的顶点 *4,6* 中，*6* 的距离最短（*'dist[6]=9'*）。将 ***6*** 加入 *'S'*。
- *6* 的邻居 *3* 已在 *S* 中，无更新。

(6) 第 *5* 轮:

- 只剩下顶点 *4*。将 ***4*** 加入 *'S'*。

算法依次找到最短路径的目标顶点顺序为：***5, 2, 3, 6, 4***。

179

> Dijkstra 算法是按路径长度递增的顺序依次产生从某一固定源点到其他各顶点之间的最短路径。

Solution 11. 这个说法是**正确**的。

详细分析:

Dijkstra 算法的本质是一个贪心算法。它维护两个顶点集合:

- ***S***: 已经找到从源点出发的最短路径的顶点集合。
- ***V-S***: 尚未找到从源点出发的最短路径的顶点集合。

算法的核心步骤如下:

(1) 初始化：*'S'* 中只包含源点 *'s'*。

(2) 循环执行：在每一步中，算法从集合 *'V-S'* 中选择一个顶点 *'u'*，该顶点 *'u'* 满足一个条件：从源点 *'s'* 到 *'u'* 的当前已知路径长度是所有 *'V-S'* 中顶点里最短的。

(3) 更新：将选出的顶点 *'u'* 从 *'V-S'* 移动到 *'S'* 中。此时，可以断定从 *'s'* 到 *'u'* 的这条路径就是最终的最短路径。

(4) 松弛：更新 *'u'* 的所有邻居的路径信息。

关键点在于第 *2* 步的贪心选择：算法总是选择当前看起来“最近”的顶点进行处理。由于图中没有负权边，这个贪心选择是正确的。当一个顶点 *'u'* 被选中并加入集合 *'S'* 时，它的最短路径长度就已经确定了。因为算法总是选择路径长度最小的顶点，所以被加入到 *'S'* 的顶点序列，其对应的最短路径长度必然是**非递减**（即递增或相等）的。

因此，*Dijkstra* 算法确实是按照路径长度递增的顺序，一步步地确定从源点到其他各顶点的最短路径。

180

> 在对有向无环图执行拓扑排序算法之后，入度数组中所有元素的值均为 0。

Solution 12. 这个说法是**正确**的。

详细分析：

我们以基于入度的 *Kahn* 算法为例来分析拓扑排序的过程：

***(1)* 初始化：** 算法开始时，会计算图中每个顶点的初始入度，并存储在一个入度数组 *'in_degree'* 中。

***(2)* 算法核心循环：**

- 算法将所有初始入度为 *0* 的顶点放入一个队列（或栈）中。
- 接着，算法从队列中取出一个顶点 *'u'*，并将其加入到拓扑序列中。
- 对于顶点 *'u'* 的每一个邻接点 *'v'*（即存在边 *'<u, v>'*），算法会将 *'v'* 的入度减 *1*（*'in_degree[v]--'*）。这个操作可以理解为在逻辑上"删除"了边 *'<u, v>'*。
- 如果 *'v'* 的入度在减 *1* 后变为 *0*，就将 *'v'* 加入队列。

***(3)* 算法结束条件：** 对于一个有向无环图（*DAG*），这个过程会持续进行，直到图中所有的顶点都被加入到拓扑序列中。

***(4)* 最终状态：**

- 当算法成功结束时，意味着图中的每一条边 *'<u, v>'* 都被"删除"了一次。
- 每删除一条边 *'<u, v>'*，顶点 *'v'* 的入度就会减 *1*。
- 既然所有的边都被删除了，那么每个顶点的入度都从其初始值减少了相应的值，最终必然会归零。

结论： 拓扑排序算法的执行过程，在逻辑上等同于不断地移除图中的边，直到图变为空图（没有边）。当算法完成时，所有顶点的入度自然都变为了 *0*。