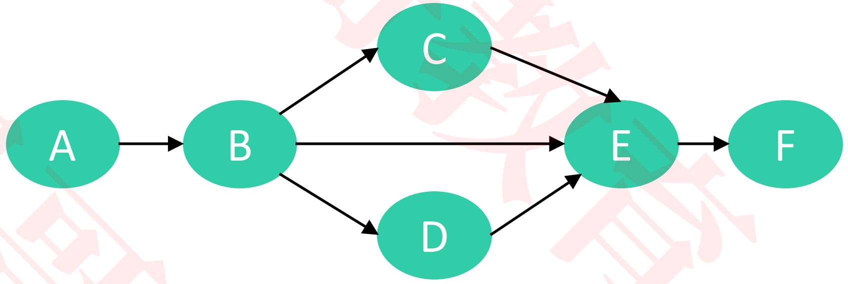
AOV网（Activity On Vertex Network）

- 一项大的工程常被分为多个小的子工程
  - ✓ 子工程之间可能存在一定的先后顺序，即某些子工程必须在其他的一些子工程完成后才能开始

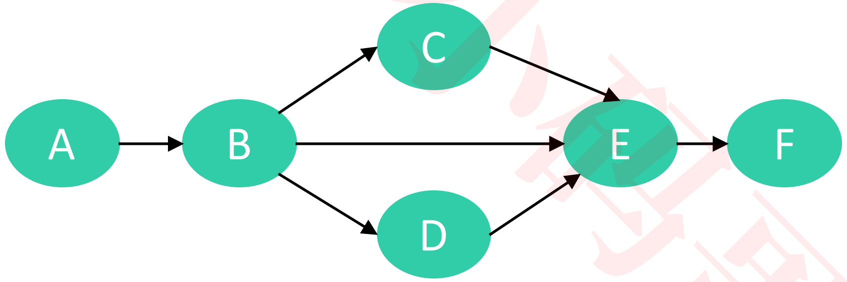
- 在现代化管理中，人们常用有向图来描述和分析一项工程的计划和实施过程，子工程被称为活动（Activity）
  - ✓ 以顶点表示活动、有向边表示活动之间的先后关系，这样的图简称为 AOV 网

- 标准的AOV网必须是一个**有向无环图**（Directed Acyclic Graph，简称 DAG）



- B依赖于A
- C依赖于B
- D依赖于B
- E依赖于B、C、D
- F依赖于E

拓扑排序（Topological Sort）



- 前驱活动：有向边起点的活动称为终点的前驱活动
  - 只有当一个活动的前驱全部都完成后，这个活动才能进行
- 后继活动：有向边终点的活动称为起点的后继活动

- 什么是拓扑排序?
  - 将 AOV 网中所有活动排成一个序列，使得每个活动的前驱活动都排在该活动的前面
  - 比如上图的拓扑排序结果是：A、B、C、D、E、F 或者 A、B、D、C、E、F （结果并不一定是唯一的）

拓扑排序 – 思路

■ 可以使用卡恩算法（Kahn于1962年提出）完成拓扑排序

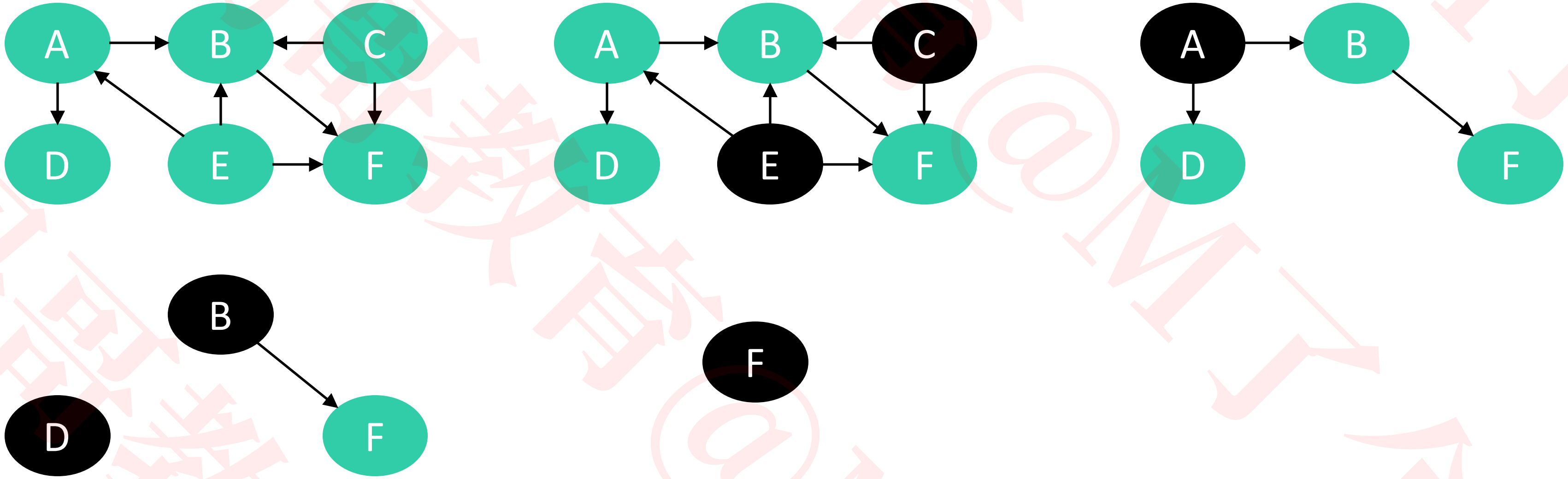
□ 假设 L 是存放拓扑排序结果的列表

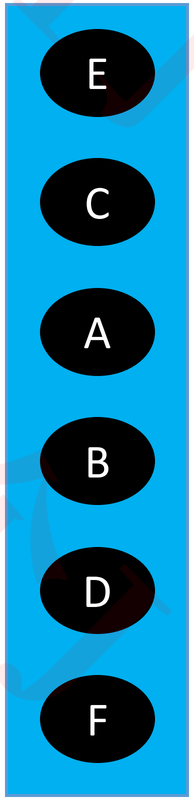
① 把所有入度为 0 的顶点放入 L 中，然后把这些顶点从图中去掉

② 重复操作 ①，直到找不到入度为 0 的顶点

□ 如果此时 L 中的元素个数和顶点总数相同，说明拓扑排序完成

□ 如果此时 L 中的元素个数少于顶点总数，说明原图中存在环，无法进行拓扑排序





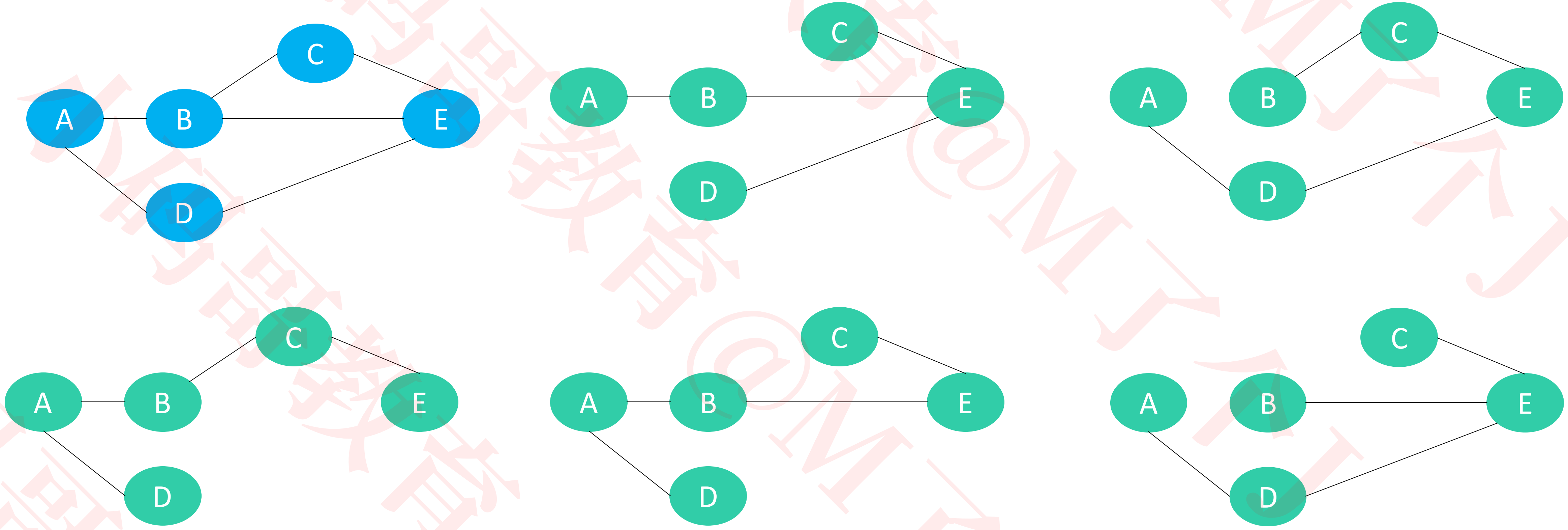
拓扑排序 – 实现

```java
List<V> list = new ArrayList<>();
Queue<Vertex<V, E>> queue = new LinkedList<>();
Map<Vertex<V, E>, Integer> ins = new HashMap<>();
vertices.forEach((V key, Vertex<V, E> vertex) -> {
    Integer in = vertex.inEdges.size();
    if (in == 0) {
        queue.offer(vertex);
    } else {
        ins.put(vertex, in);
    }
});
```

```java
while (!queue.isEmpty()) {
    Vertex<V, E> vertex = queue.poll();
    list.add(vertex.value);
    for (Edge<V, E> edge : vertex.outEdges) {
        Integer in = ins.get(edge.to) - 1;
        if (in == 0) {
            queue.offer(edge.to);
        } else {
            ins.put(edge.to, in);
        }
    }
}
```

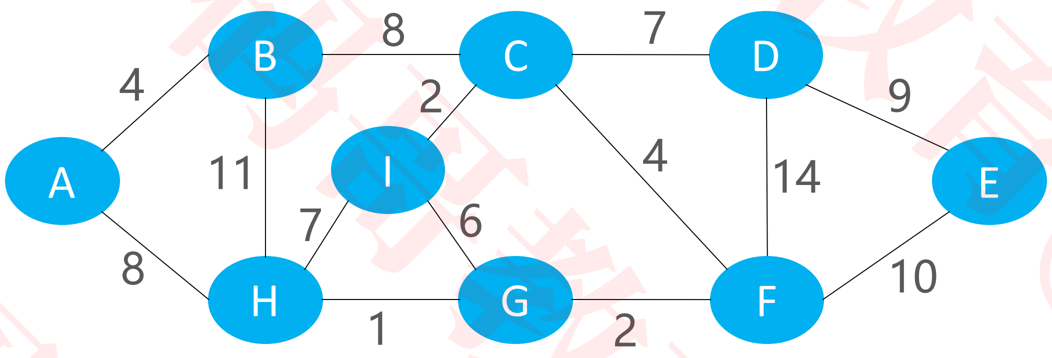
生成树（Spanning Tree）

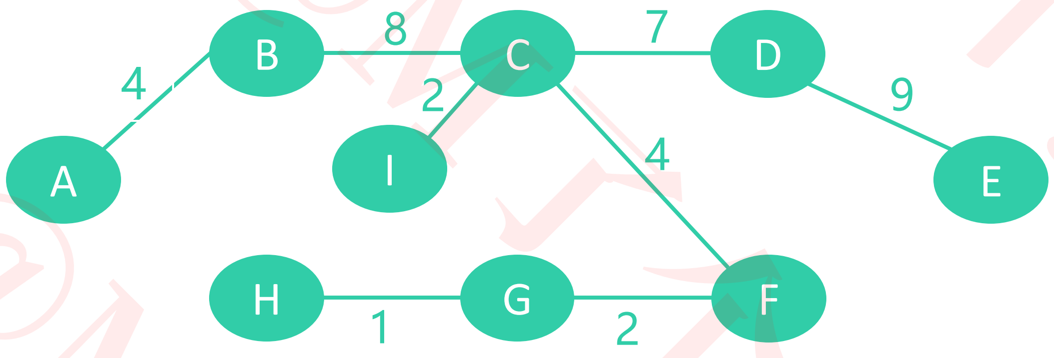
- 生成树（Spanning Tree），也称为支撑树
  - 连通图的极小连通子图，它含有图中全部的 n 个顶点，恰好只有 n – 1 条边



最小生成树（Minimum Spanning Tree）

- 最小生成树（Minimum Spanning Tree，简称MST）
  - 也称为最小权重生成树（Minimum Weight Spanning Tree）、最小支撑树
  - 是所有生成树中，总权值最小的那棵
  - 适用于有权的连通图（无向）





最小生成树

- 最小生成树在许多领域都有重要的作用，例如
  - 要在 n 个城市之间铺设光缆，使它们都可以通信
  - 铺设光缆的费用很高，且各个城市之间因为距离不同等因素，铺设光缆的费用也不同
  - 如何使铺设光缆的总费用最低?

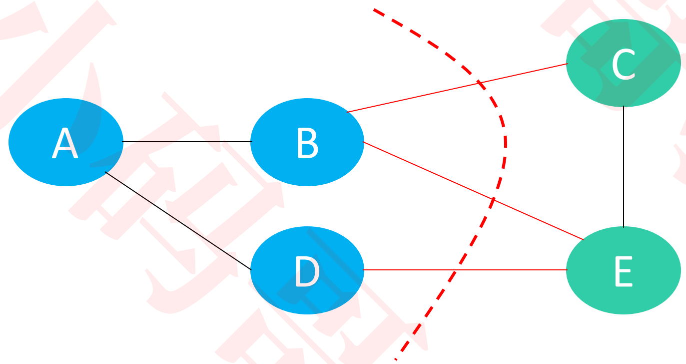
- 如果图的每一条边的权值都互不相同，那么最小生成树将只有一个，否则可能会有多个最小生成树

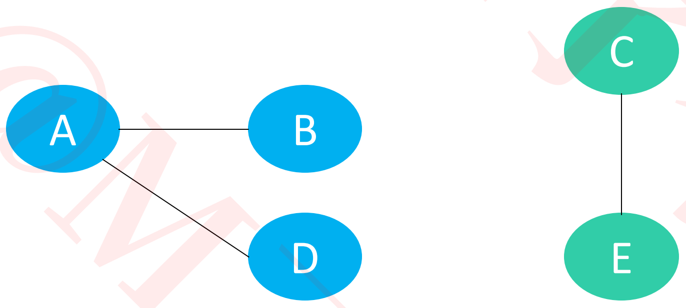
- 求最小生成树的2个经典算法
  - Prim（普里姆算法）
  - Kruskal（克鲁斯克尔算法）

切分定理

- 切分（Cut）：把图中的节点分为两部分，称为一个切分

- 下图有个切分 C = (S, T)，S = {A, B, D}，T = {C, E}





- 横切边（Crossing Edge）：如果一个边的两个顶点，分别属于切分的两部分，这个边称为横切边
  - 比如上图的边 BC、BE、DE 就是横切边

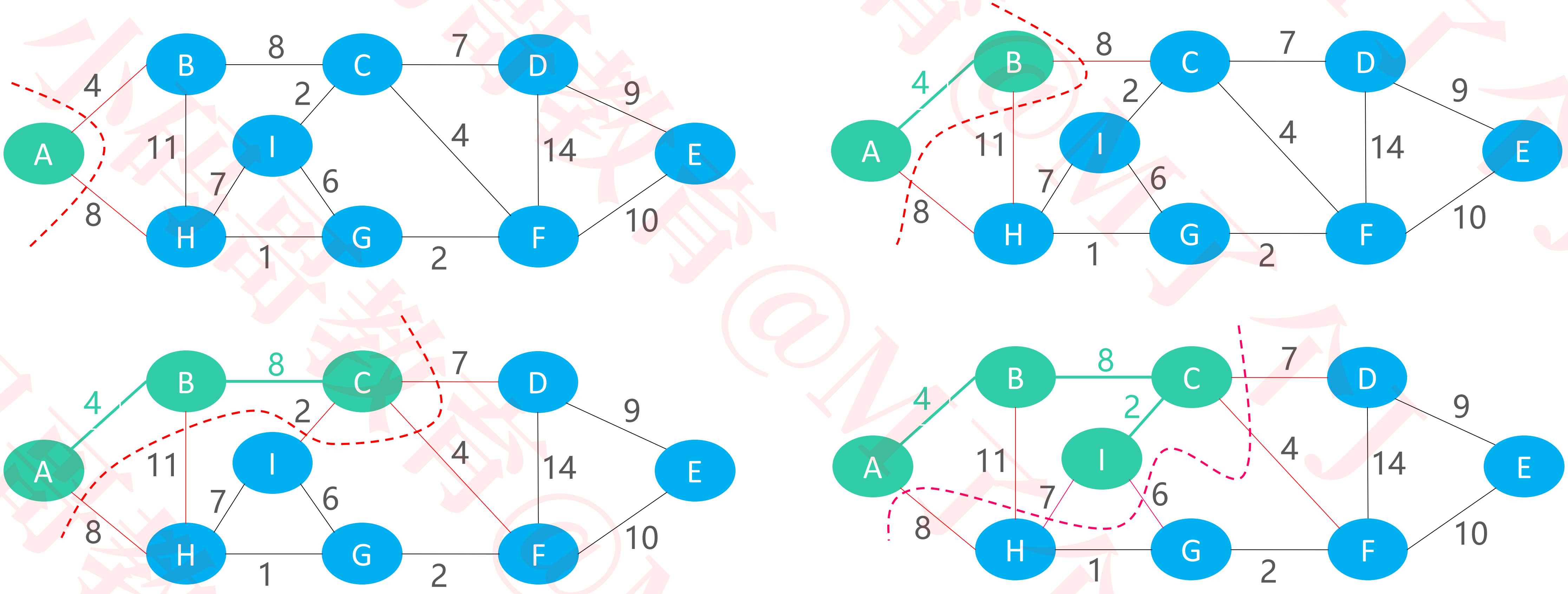
- 切分定理：给定任意切分，横切边中权值最小的边必然属于最小生成树

Prim算法 – 执行过程

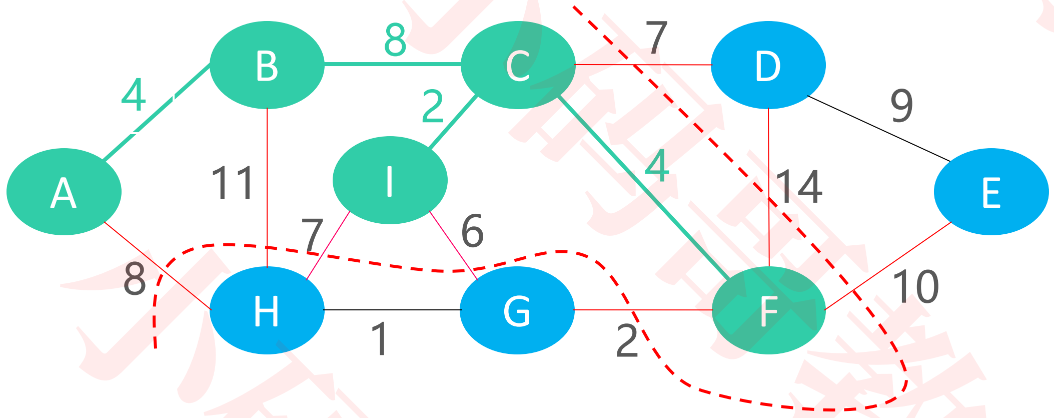
■ 假设 G = (V，E) 是有权的连通图（无向），A 是 G 中最小生成树的边集

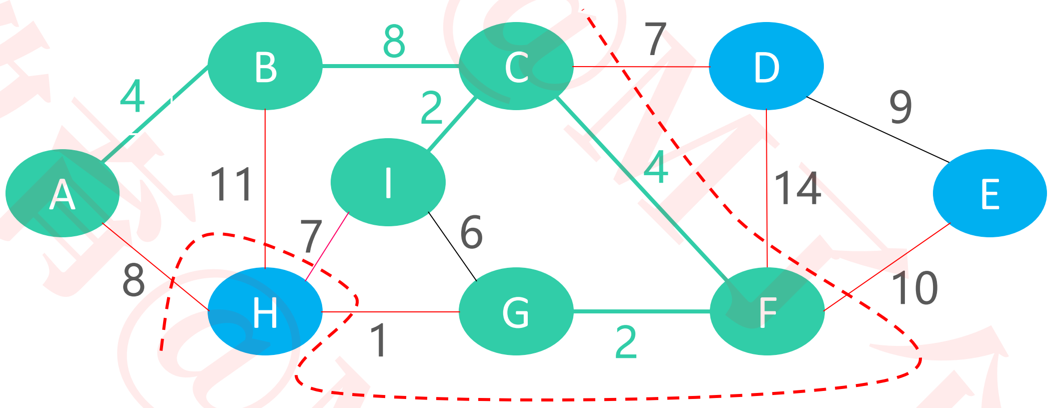
□ 算法从 S = { u_0 }（u_0 ∈ V），A = { } 开始，重复执行下述操作，直到 S = V 为止

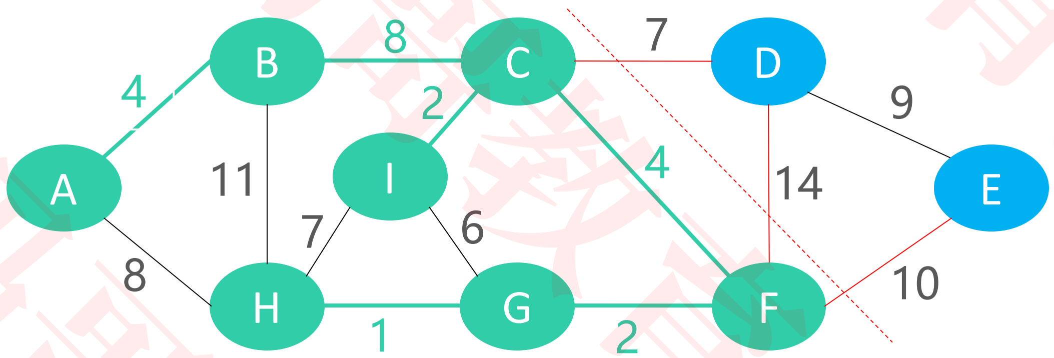
✓ 找到切分 C = (S，V – S) 的最小横切边 (u_0，v_0) 并入集合 A，同时将 v_0 并入集合 S

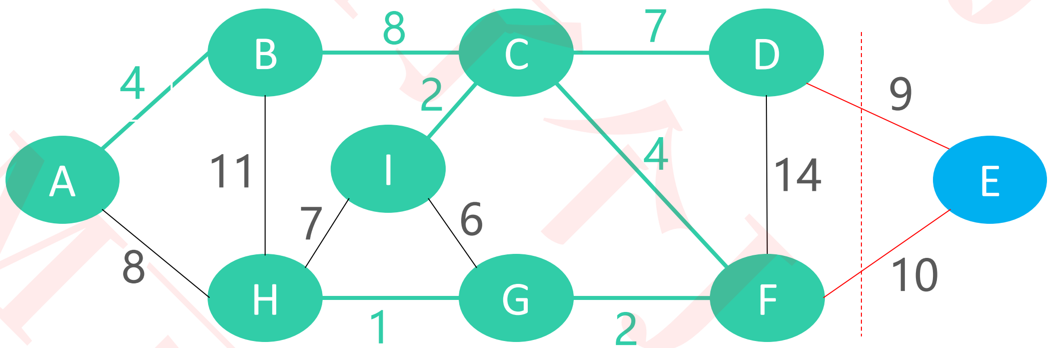


Prim算法 – 执行过程

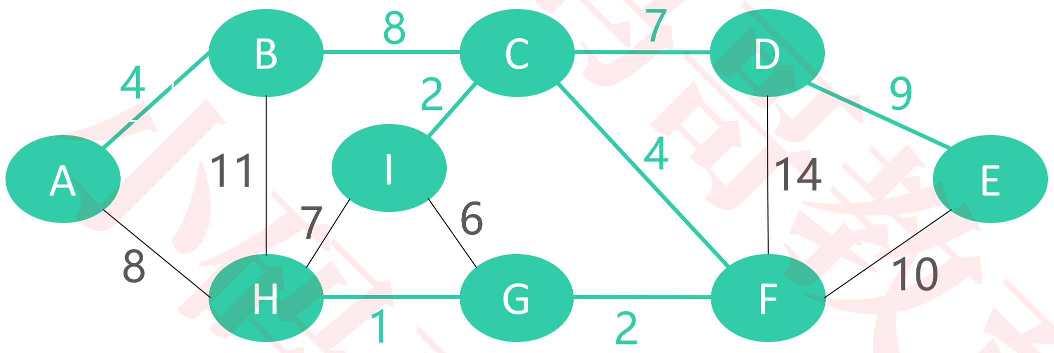

A
B
C
D
E
F
G
H
I
4
8
7
9
2
4
14
11
7
6
8
1
2
10

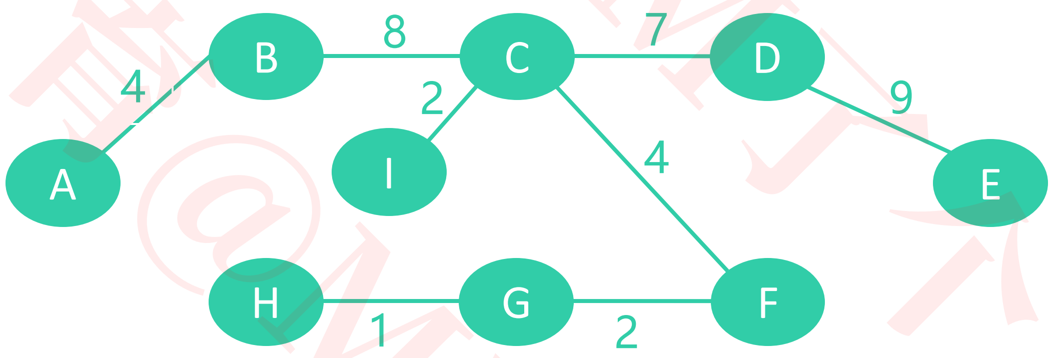

A
B
C
D
E
F
G
H
I
4
8
7
9
2
4
14
11
7
6
8
1
2
10


A
B
C
D
E
F
G
H
I
4
8
7
9
2
4
14
11
7
6
8
1
2
10


A
B
C
D
E
F
G
H
I
4
8
7
9
2
4
14
11
7
6
8
1
2
10

Prim算法 – 执行过程





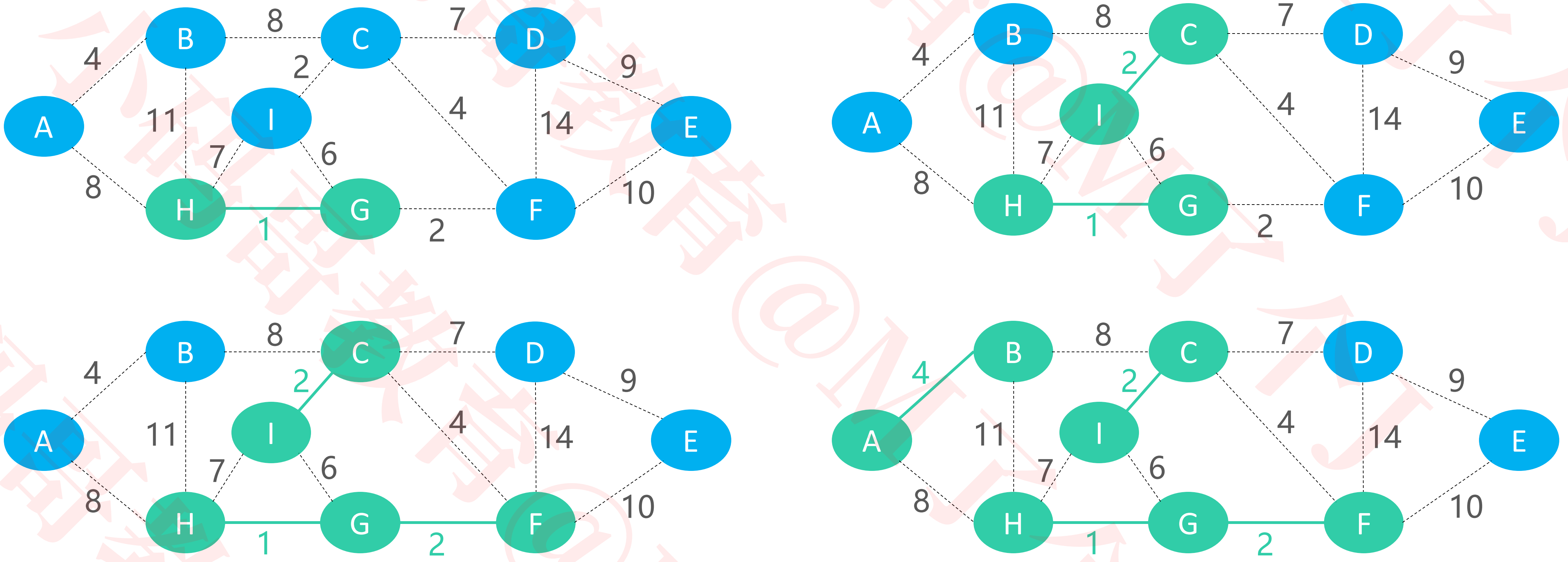
Prim算法 – 实现

```java
private Set<EdgeInfo<V, E>> prim() {
    Iterator<Vertex<V, E>> it = vertices.values().iterator();
    if (!it.hasNext()) return null;
    Vertex<V, E> vertex = it.next();
    Set<EdgeInfo<V, E>> edgeInfos = new HashSet<>();
    Set<Vertex<V, E>> addedVertices = new HashSet<>();
    addedVertices.add(vertex);
    MinHeap<Edge<V, E>> heap = new MinHeap<>(vertex.outEdges, edgeComparator);
    int verticesSize = vertices.size();
    while (!heap.isEmpty() && addedVertices.size() < verticesSize) {
        Edge<V, E> edge = heap.remove();
        if (addedVertices.contains(edge.to)) continue;
        edgeInfos.add(edge.info());
        addedVertices.add(edge.to);
        heap.addAll(edge.to.outEdges);
    }
    return edgeInfos;
}
```

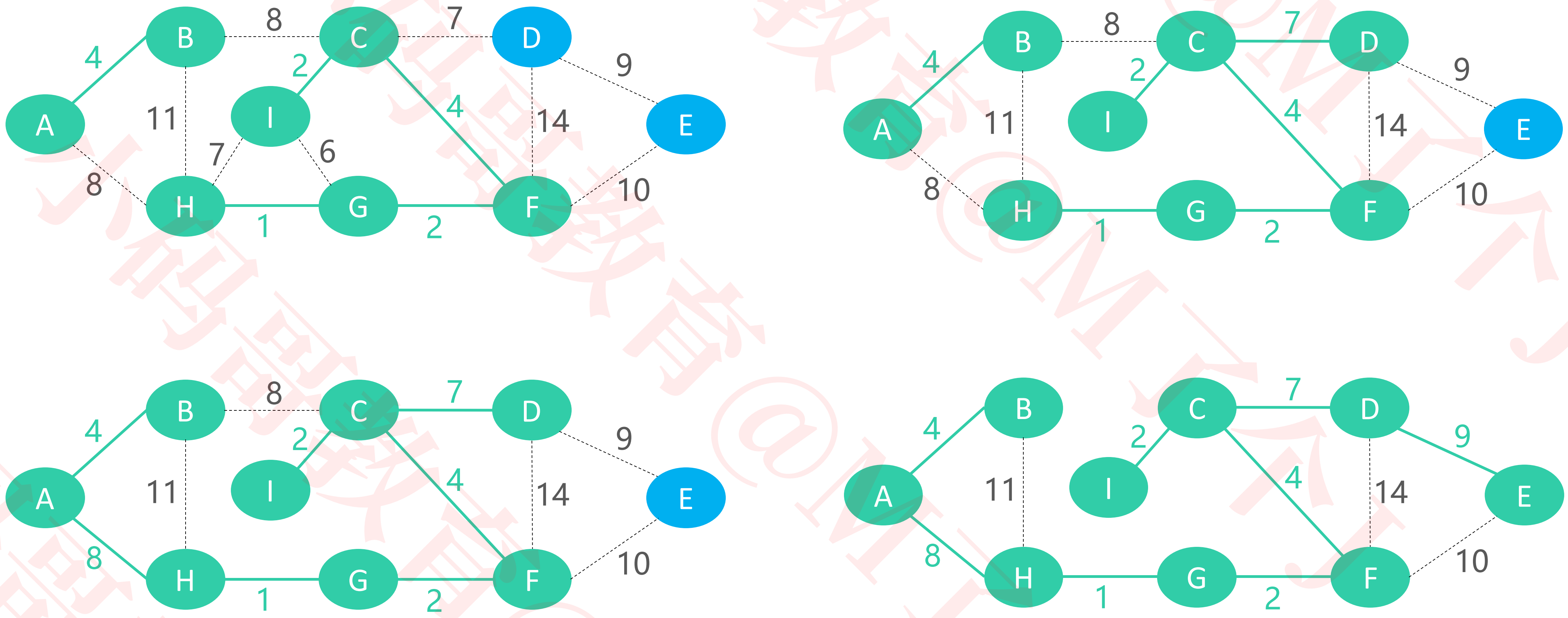
小码哥教育 SEEMYGO

Kruskal算法 – 执行过程

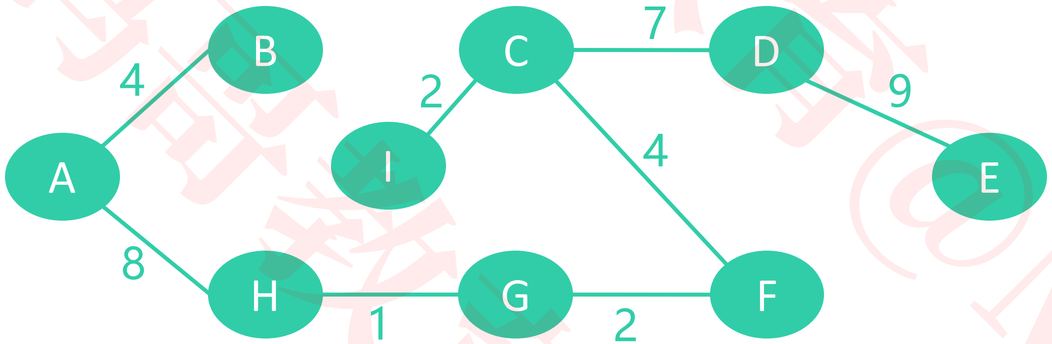
- ■ 按照边的权重顺序（从小到大）将边加入生成树中，直到生成树中含有 V – 1 条边为止（ V 是顶点数量）
  - ◻ 若加入该边会与生成树形成环，则不加入该边
  - ◻ 从第3条边开始，可能会与生成树形成环



Kruskal算法 – 执行过程



Kruskal算法 – 执行过程



Kruskal算法 – 实现

```java
private Set<EdgeInfo<V, E>> kruskal() {
    int edgeSize = vertices.size() - 1;
    if (edgeSize == -1) return null;
    Set<EdgeInfo<V, E>> edgeInfos = new HashSet<>();
    MinHeap<Edge<V, E>> heap = new MinHeap<>(edges, edgeComparator);
    UnionFind<Vertex<V, E>> uf = new UnionFind<>();
    vertices.forEach((V v, Vertex<V, E> vertex) -> {
        uf.makeSet(vertex);
    });
    while (!heap.isEmpty() && edgeInfos.size() < edgeSize) {
        Edge<V, E> edge = heap.remove();
        if (uf.isSame(edge.from, edge.to)) continue;
        edgeInfos.add(edge.info());
        uf.union(edge.from, edge.to);
    }
    return edgeInfos;
}
```

- 时间复杂度：O(ElogE)