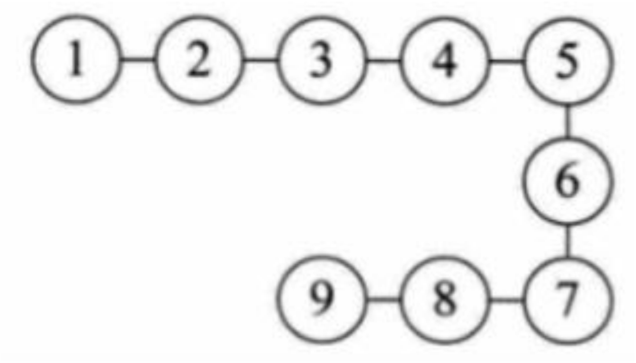
图（Graph）

数据结构回顾

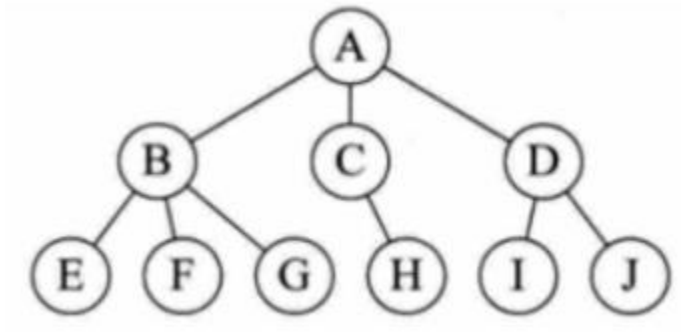


线性结构

数组、链表、

栈、队列、

哈希表

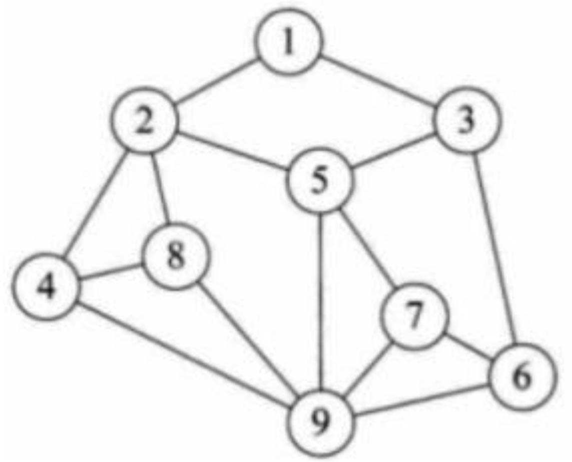


树形结构

二叉树、B树、

堆、Trie、

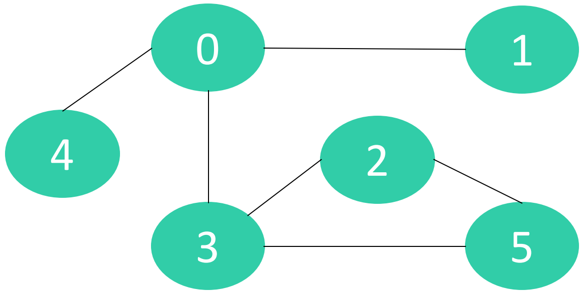
哈夫曼树、并查集

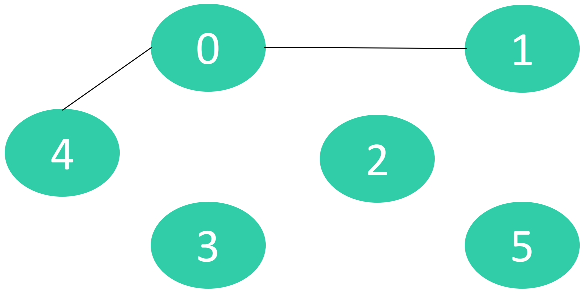


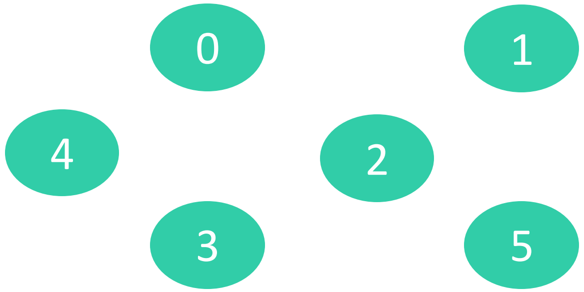
图形结构

图（Graph）

- 图由顶点（vertex）和边（edge）组成，通常表示为 G = (V, E)
  - G表示一个图，V是顶点集，E是边集
  - 顶点集V有穷且非空
  - 任意两个顶点之间都可以用边来表示它们之间的关系，边集E可以是空的

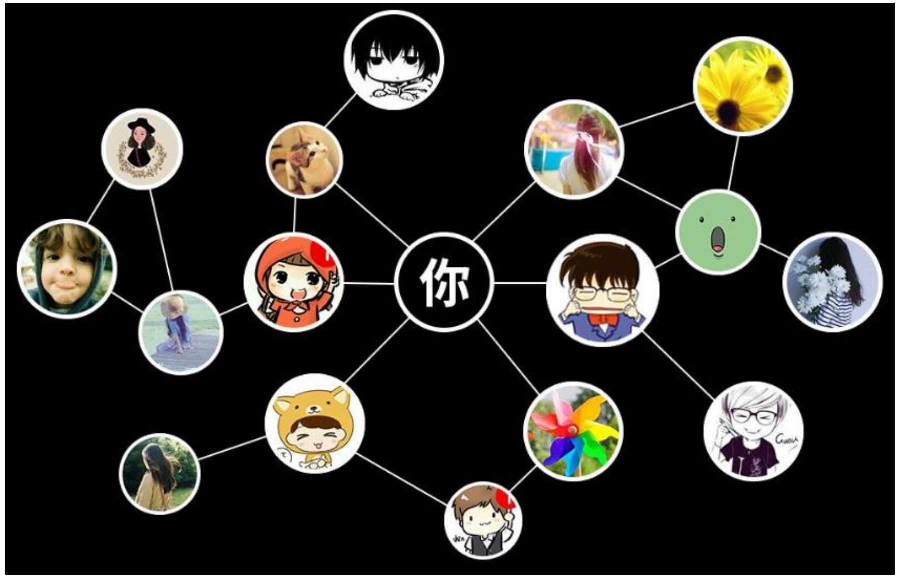






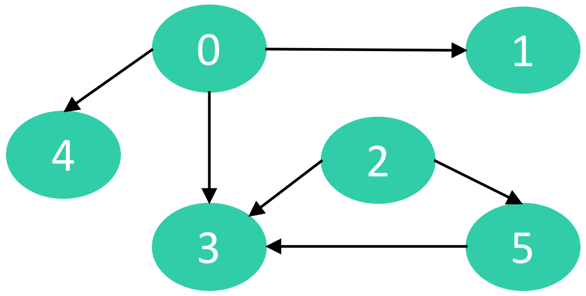
图的应用举例

- 图结构的应用极其广泛
  - 社交网络
  - 地图导航
  - 游戏开发
  - ……



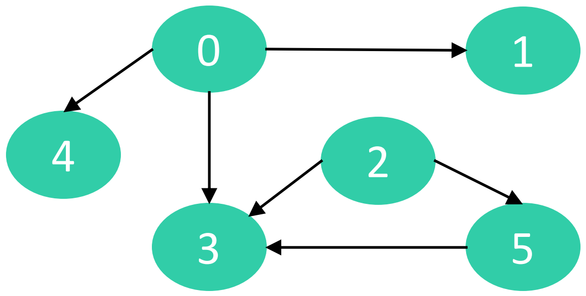
有向图（Directed Graph）

■ 有向图的边是有明确方向的

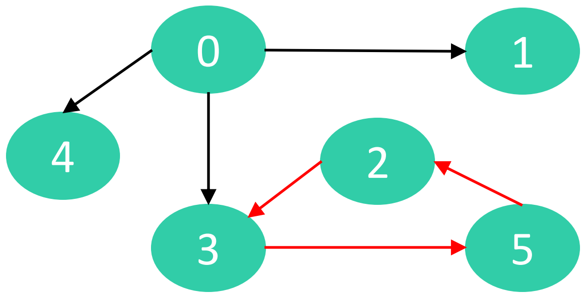


■ 有向无环图（Directed Acyclic Graph，简称 DAG）

□ 如果一个有向图，从任意顶点出发无法经过若干条边回到该顶点，那么它就是一个有向无环图



无环



有环

出度、入度

■ 出度、入度适用于有向图

■ 出度（Out-degree）

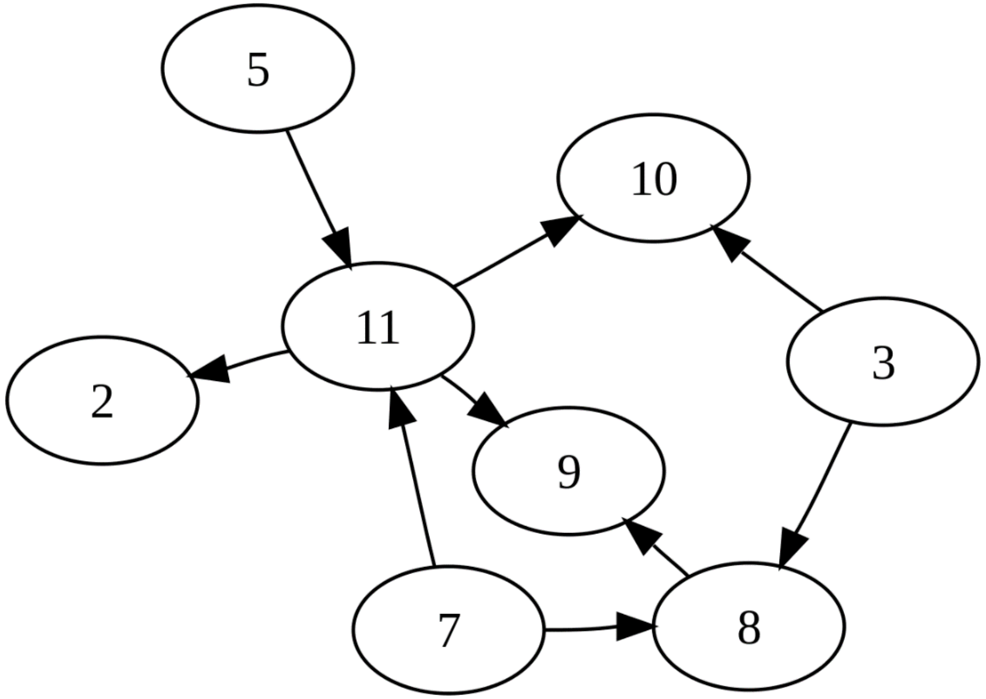
□一个顶点的出度为 x，是指有 x 条边以该顶点为起点

□顶点11的出度是3

■ 入度（In-degree）

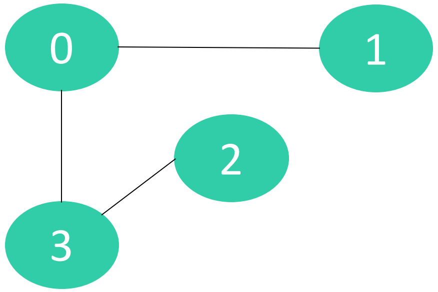
□一个顶点的入度为 x，是指有 x 条边以该顶点为终点

□顶点11的入度是2

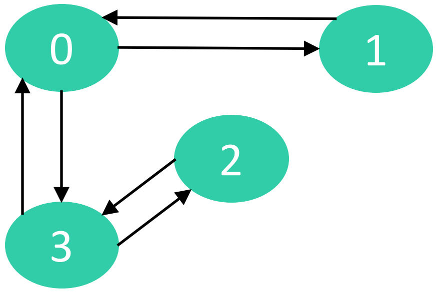


无向图（Undirected Graph）

- 无向图的边是无方向的

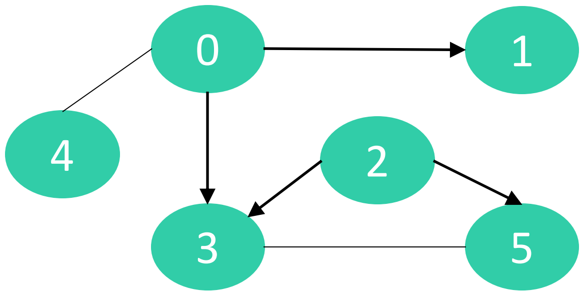


- 效果类似于下面的有向图



混合图（Mixed Graph）

- 混合图的边可能是无向的，也可能是有向的



简单图、多重图

■ 平行边

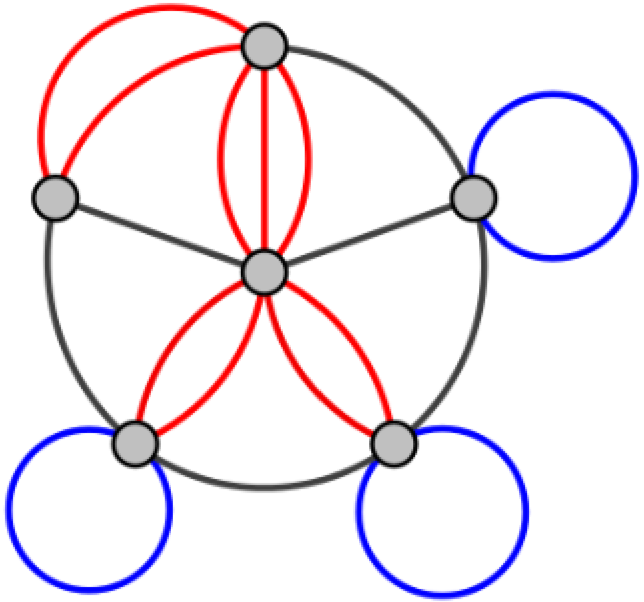
- 在无向图中，关联一对顶点的无向边如果多于1条，则称这些边为平行边
- 在有向图中，关联一对顶点的有向边如果多于1条，并且它们的的方向相同，则称这些边为平行边

■ 多重图（Multigraph）

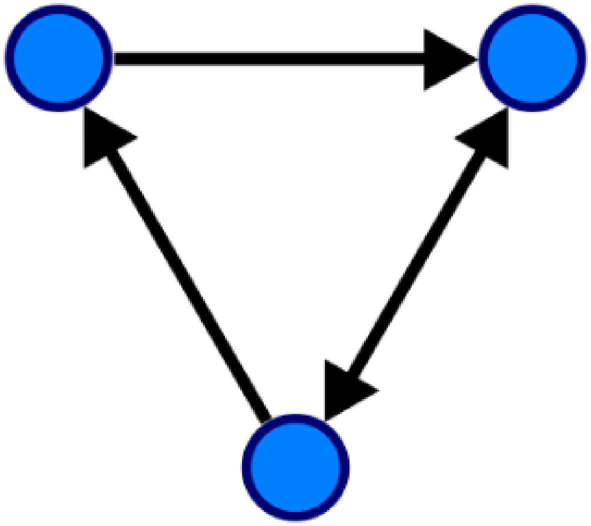
- 有平行边或者有自环的图

■ 简单图（Simple Graph）

- 既没有平行边也不没有自环的图
- 课程中讨论的基本都是简单图



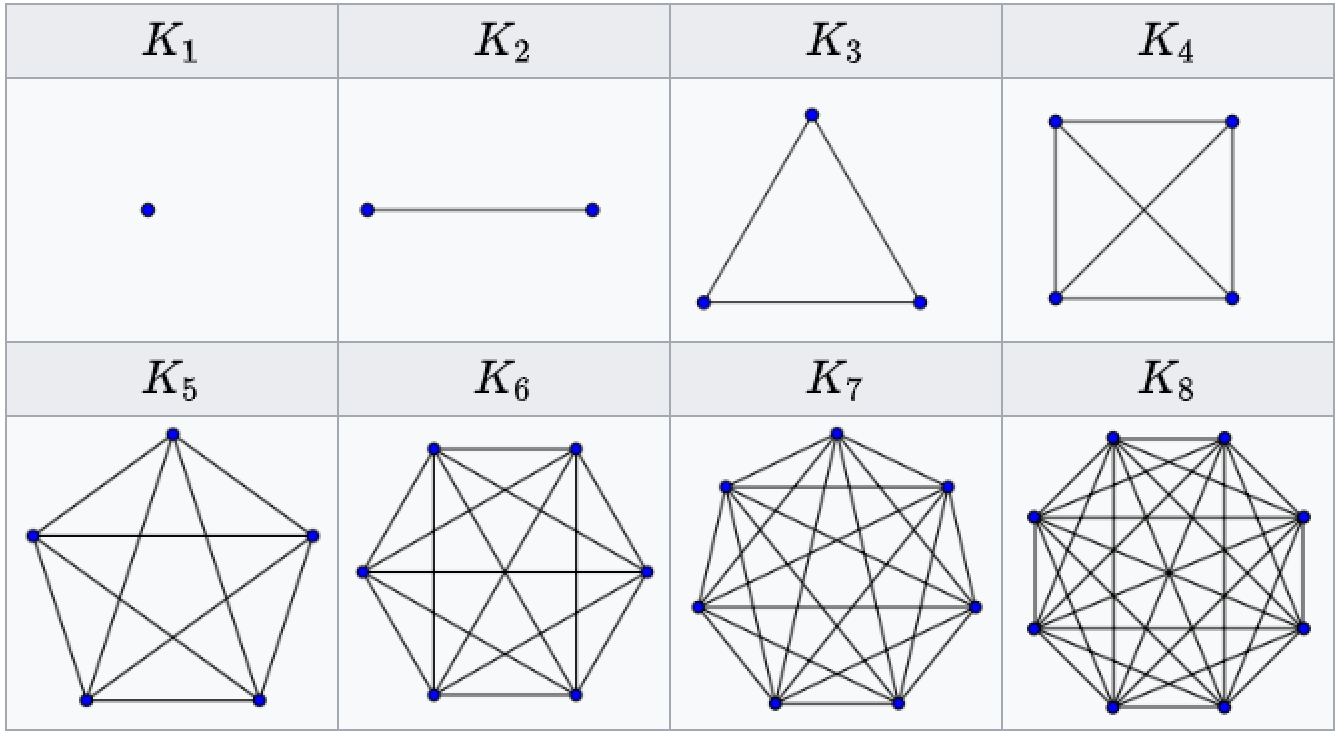
多重图



简单图

无向完全图（Undirected Complete Graph）

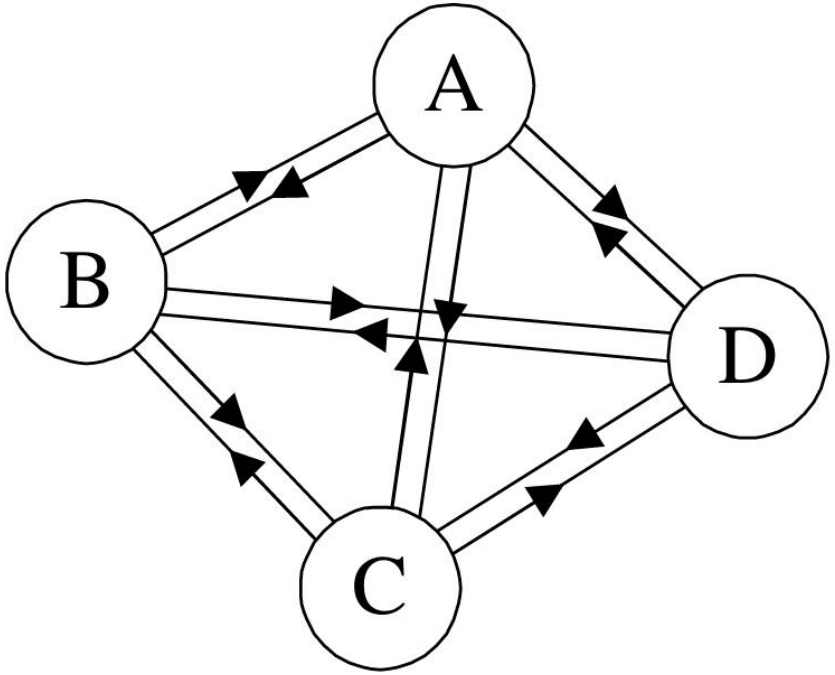
- ■ 无向完全图的任意两个顶点之间都存在边
- ◻ n 个顶点的无向完全图有 $n(n-1)/2$ 条边
- ✓ $(n-1)+(n-2)+(n-3)+\cdots+3+2+1$



有向完全图（Directed Complete Graph）

■ 有向完全图的任意两个顶点之间都存在方向相反的两条边

□ n 个顶点的有向完全图有 $n(n-1)$ 条边

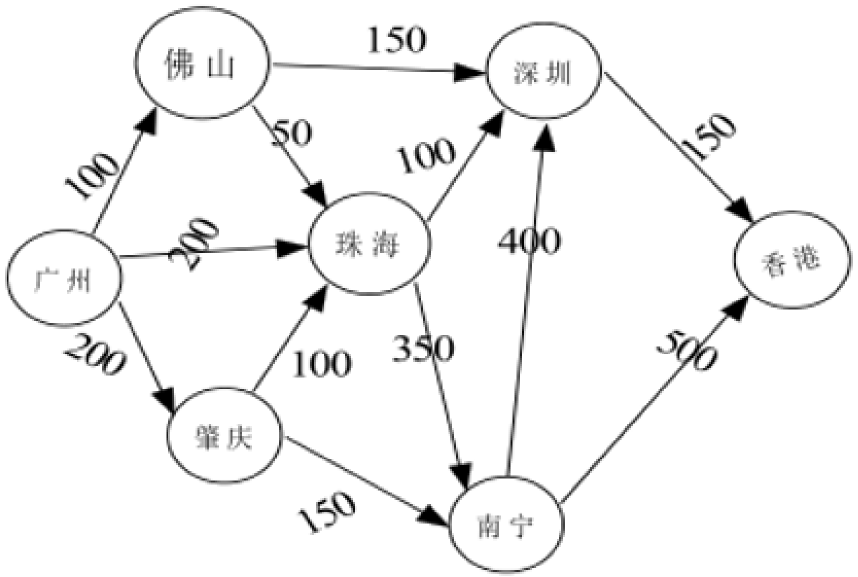


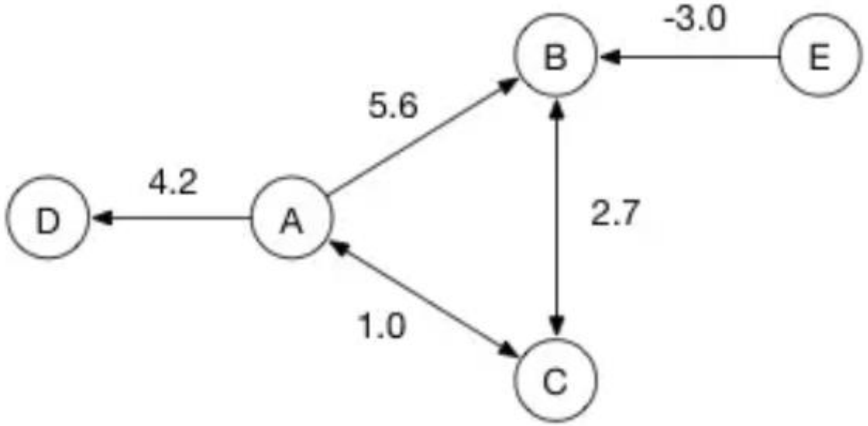
■ 稠密图（Dense Graph）： 边数接近于或等于完全图

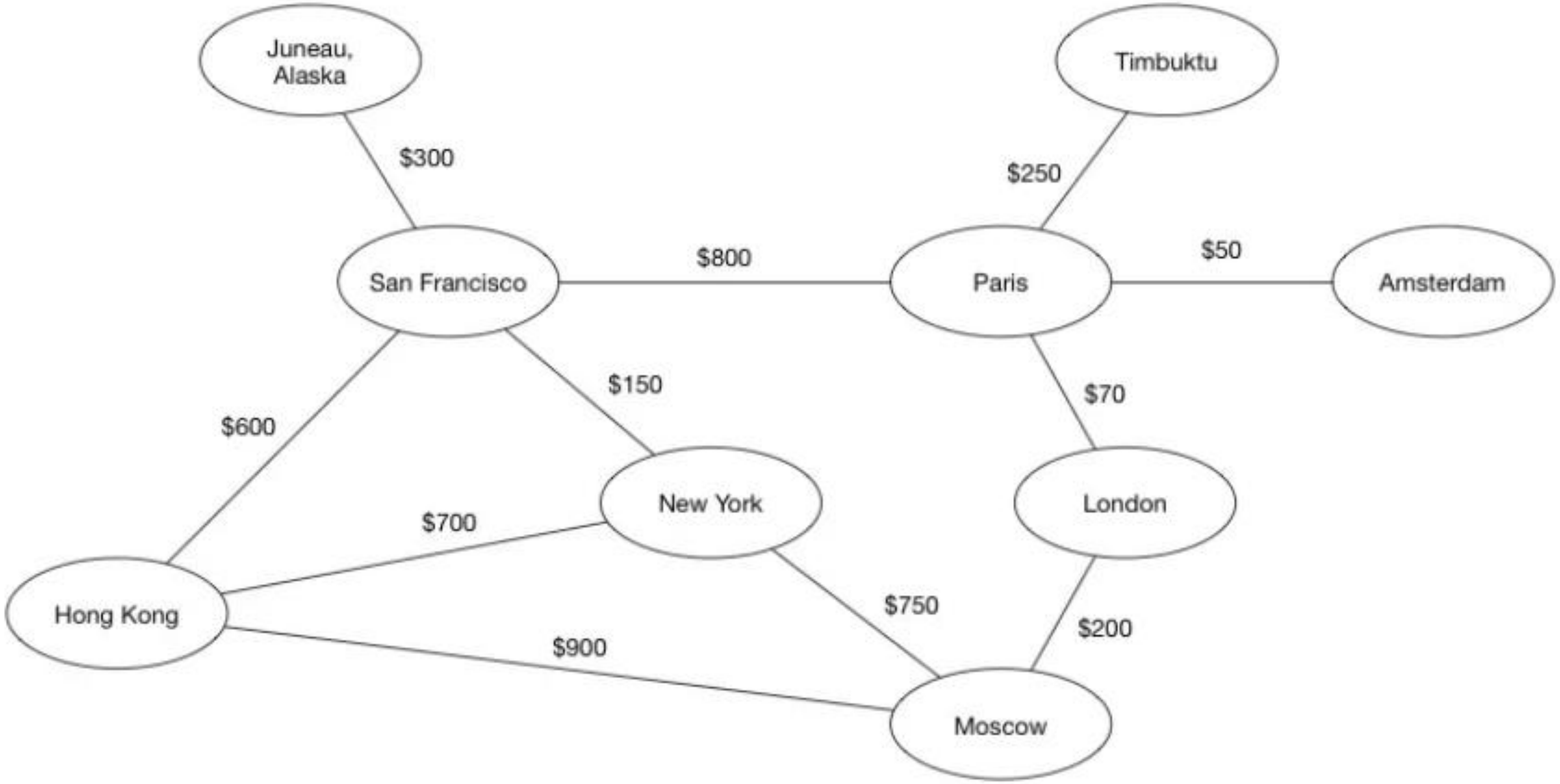
■ 稀疏图（Sparse Graph）： 边数远远少于完全图

有权图（Weighted Graph）

- 有权图的边可以拥有权值（Weight）



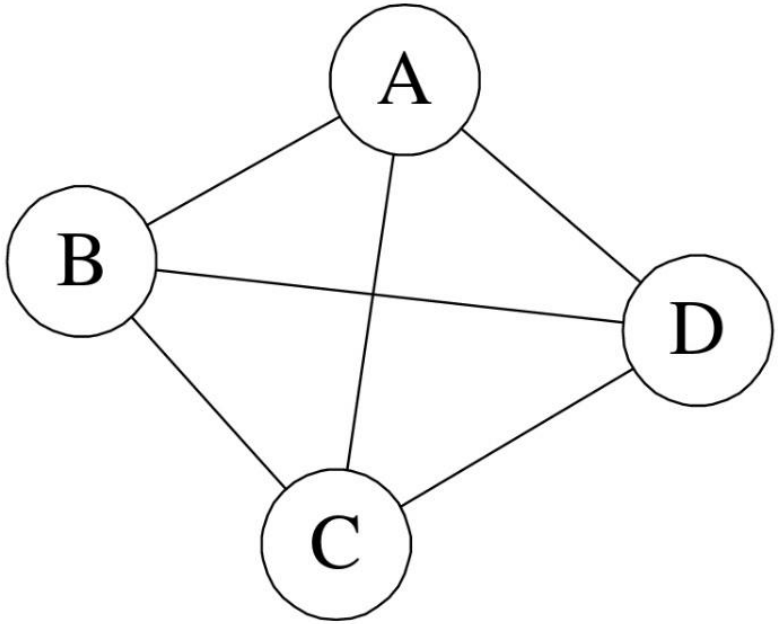


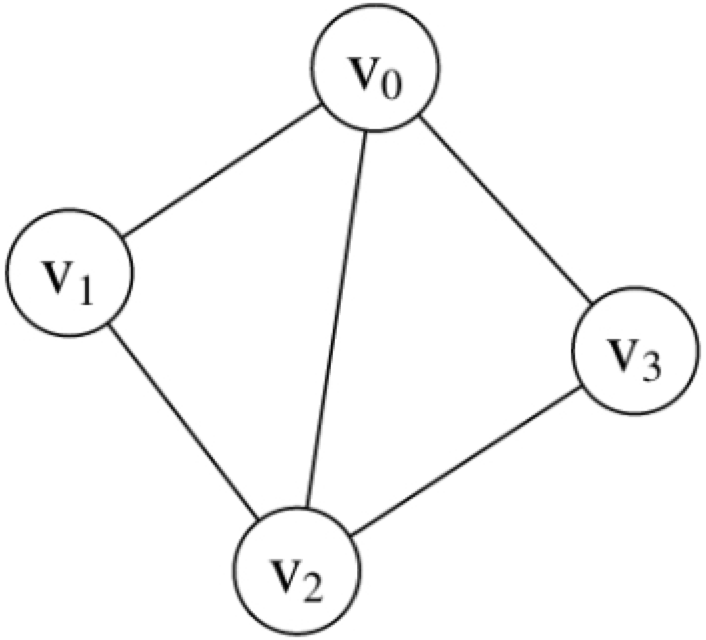


连通图（Connected Graph）

- 如果顶点 x 和 y 之间存在可相互抵达的路径（直接或间接的路径），则称 x 和 y 是连通的

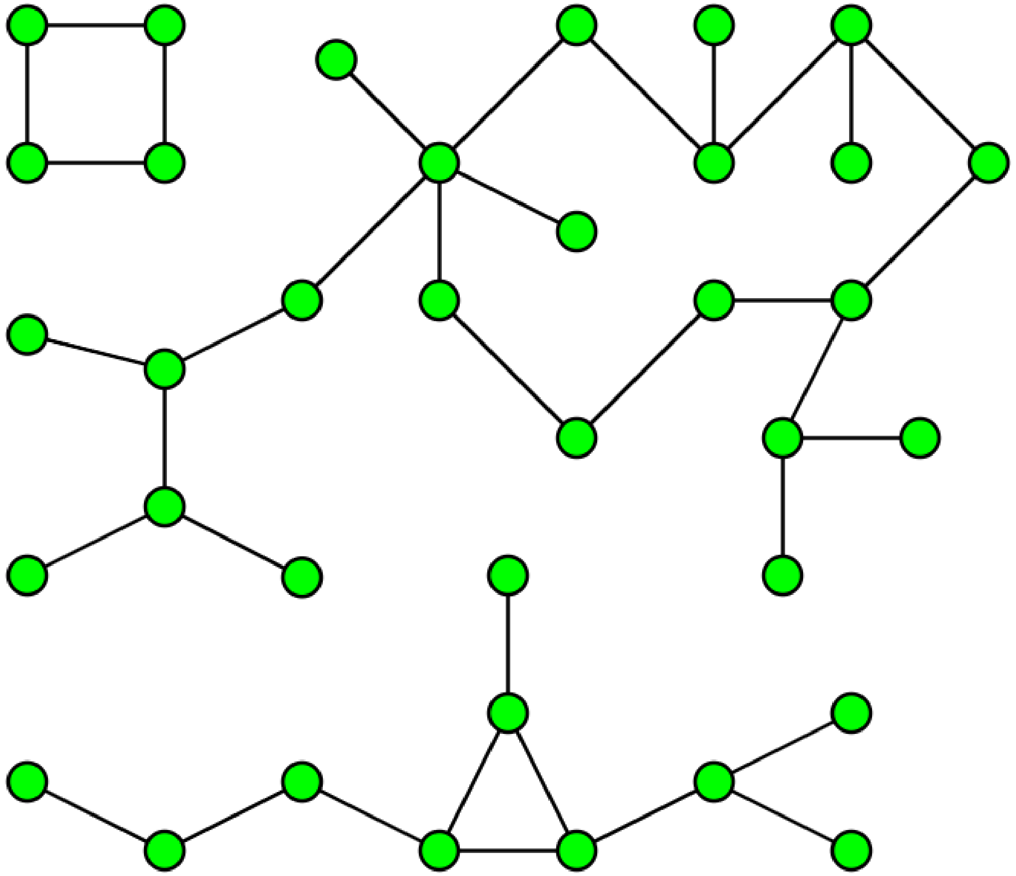
- 如果无向图 G 中任意2个顶点都是连通的，则称G为连通图





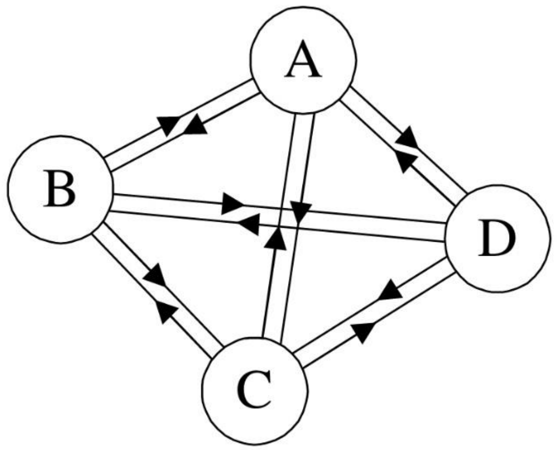
连通分量（Connected Component）

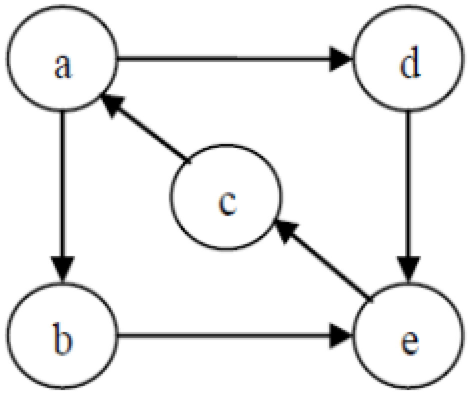
- 连通分量：无向图的极大连通子图
  - 连通图只有一个连通分量，即其自身；非连通的无向图有多个连通分量
- 下面的无向图有3个连通分量

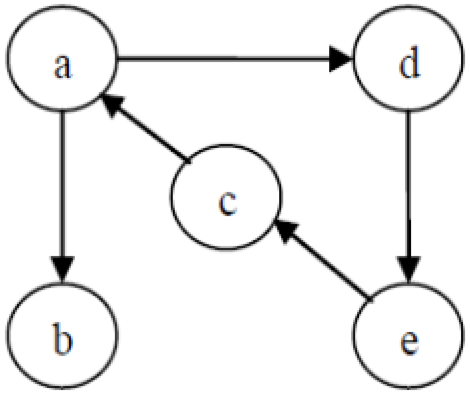


强连通图（Strongly Connected Graph）

- 如果有向图 G 中任意2个顶点都是连通的，则称G为强连通图



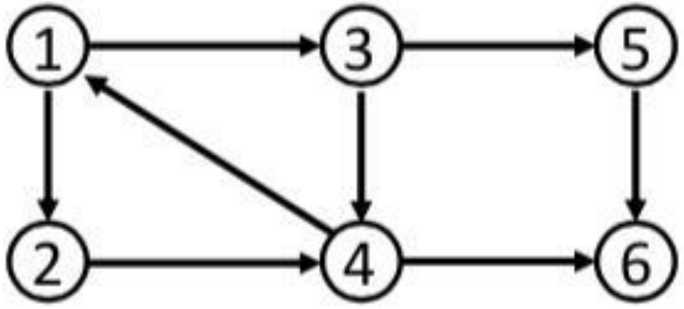




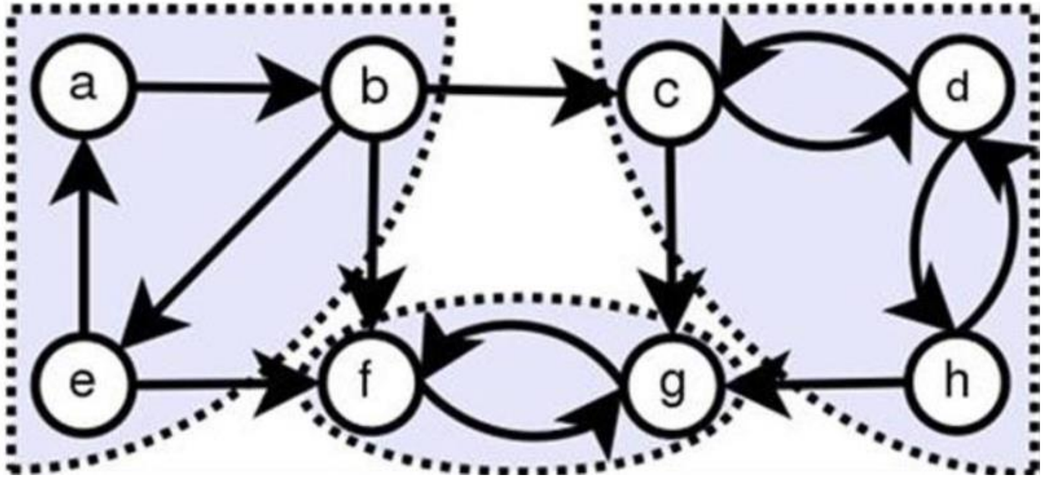
不是强连通图

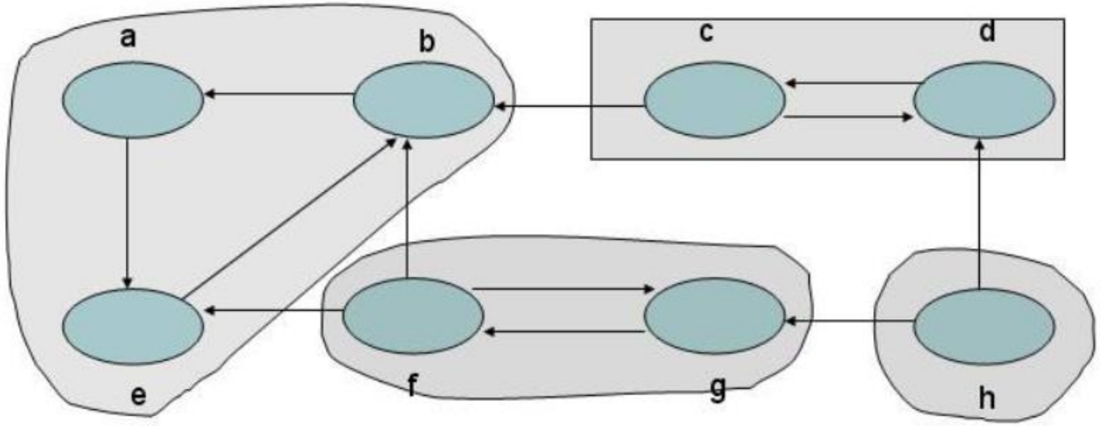
强连通分量（Strongly Connected Component）

- ■ 强连通分量：有向图的极大强连通子图
- □ 强连通图只有一个强连通分量，即其自身；非强连通的有向图有多个强连通分量



强连通分量：{1,2,3,4}、{5}、{6}





图的实现方案

- 图有2种常见的实现方案
- 邻接矩阵（Adjacency Matrix）

- 邻接表（Adjacency List）

邻接矩阵（Adjacency Matrix）

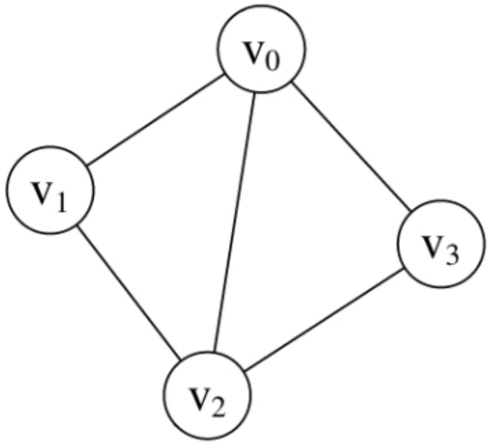
■ 邻接矩阵的存储方式

□ 一维数组存放顶点信息

□ 二维数组存放边信息

■ 邻接矩阵比较适合稠密图

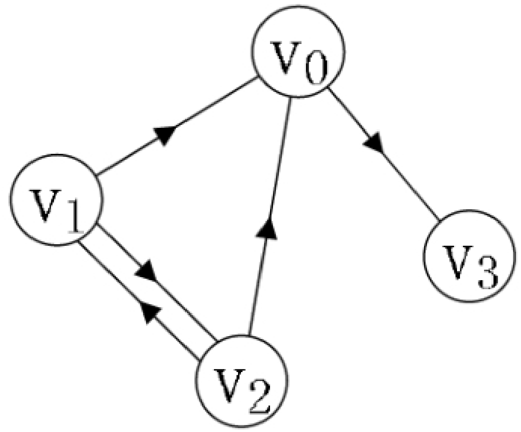
□ 不然会比较浪费内存



顶点数组			
v_0	v_1	v_2	v_3

边数组

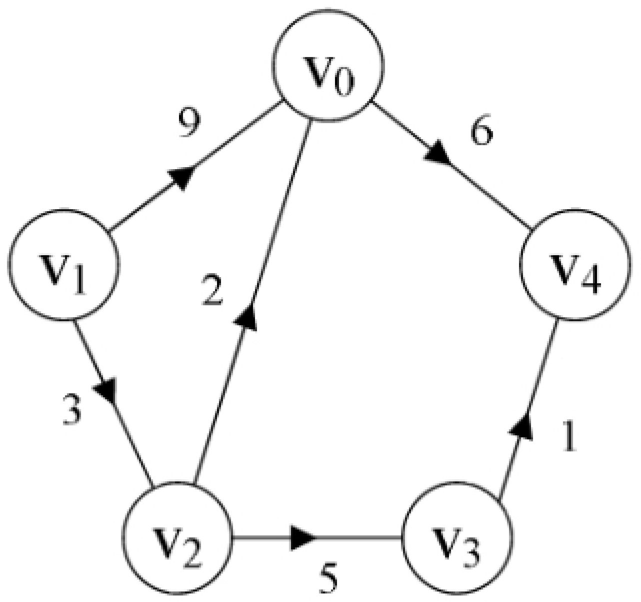
	v_0	v_1	v_2	v_3
v_0	0	1	1	1
v_1	1	0	1	0
v_2	1	1	0	1
v_3	1	0	1	0



边数组

	v_0	v_1	v_2	v_3
v_0	0	0	0	1
v_1	1	0	1	0
v_2	1	1	0	0
v_3	0	0	0	0

邻接矩阵 – 有权图

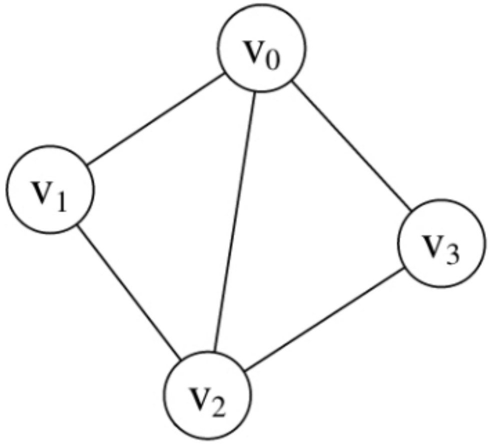


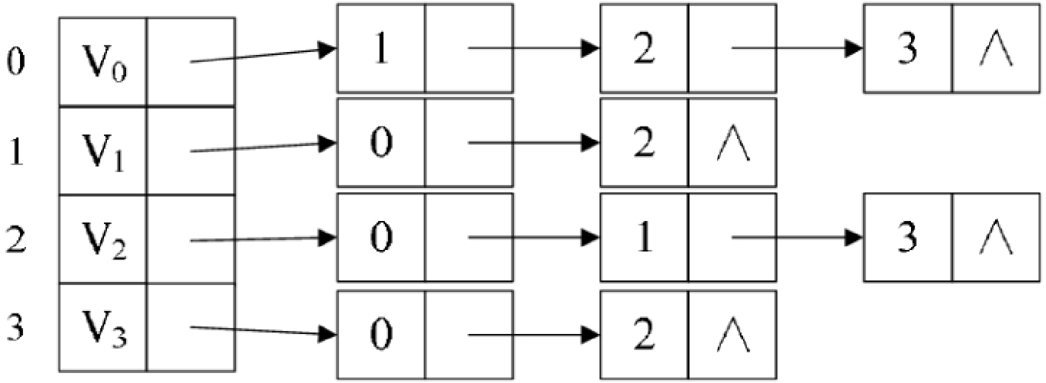
顶点数组				
v_0	v_1	v_2	v_3	v_4

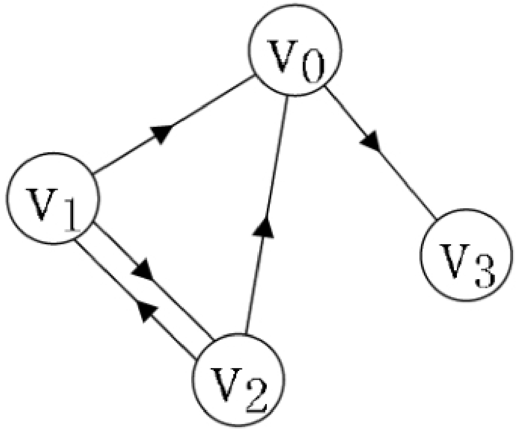
边数组

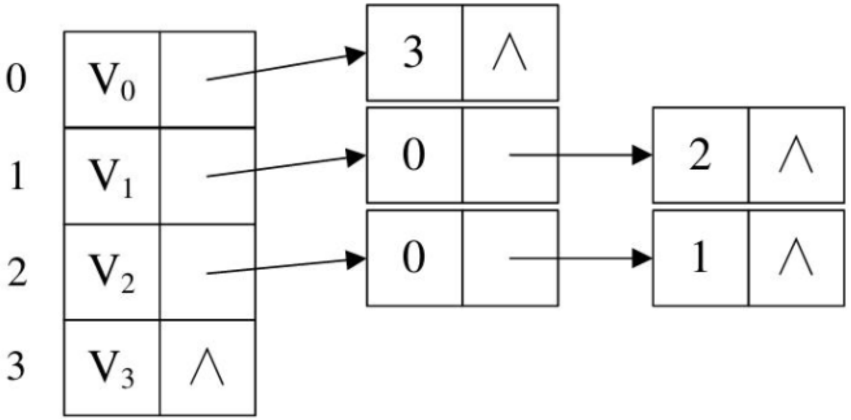
	v_0	v_1	v_2	v_3	v_4
v_0	∞	∞	∞	∞	6
v_1	9	∞	3	∞	∞
v_2	2	∞	∞	5	∞
v_3	∞	∞	∞	∞	1
v_4	∞	∞	∞	∞	∞

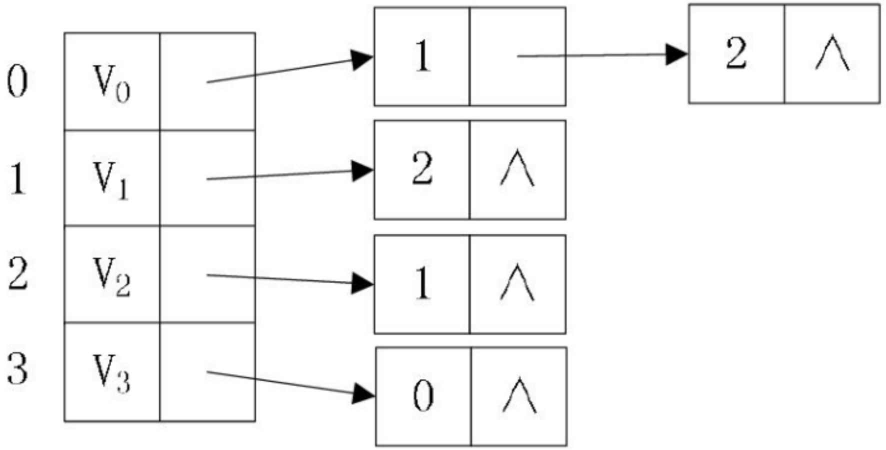
邻接表（Adjacency List）





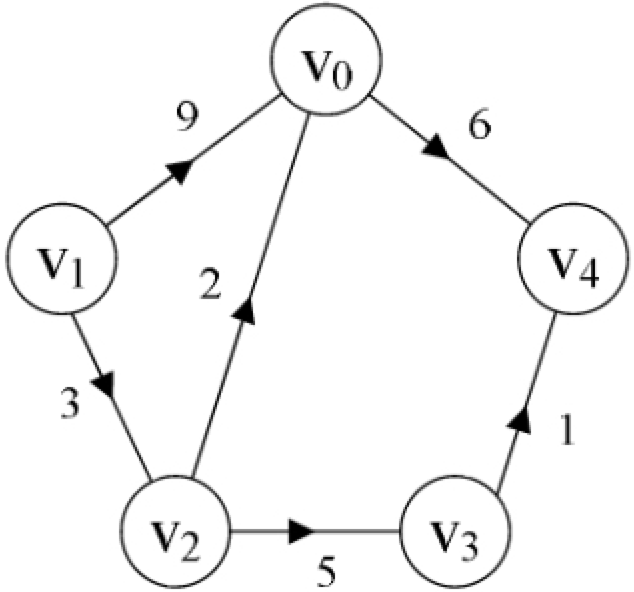


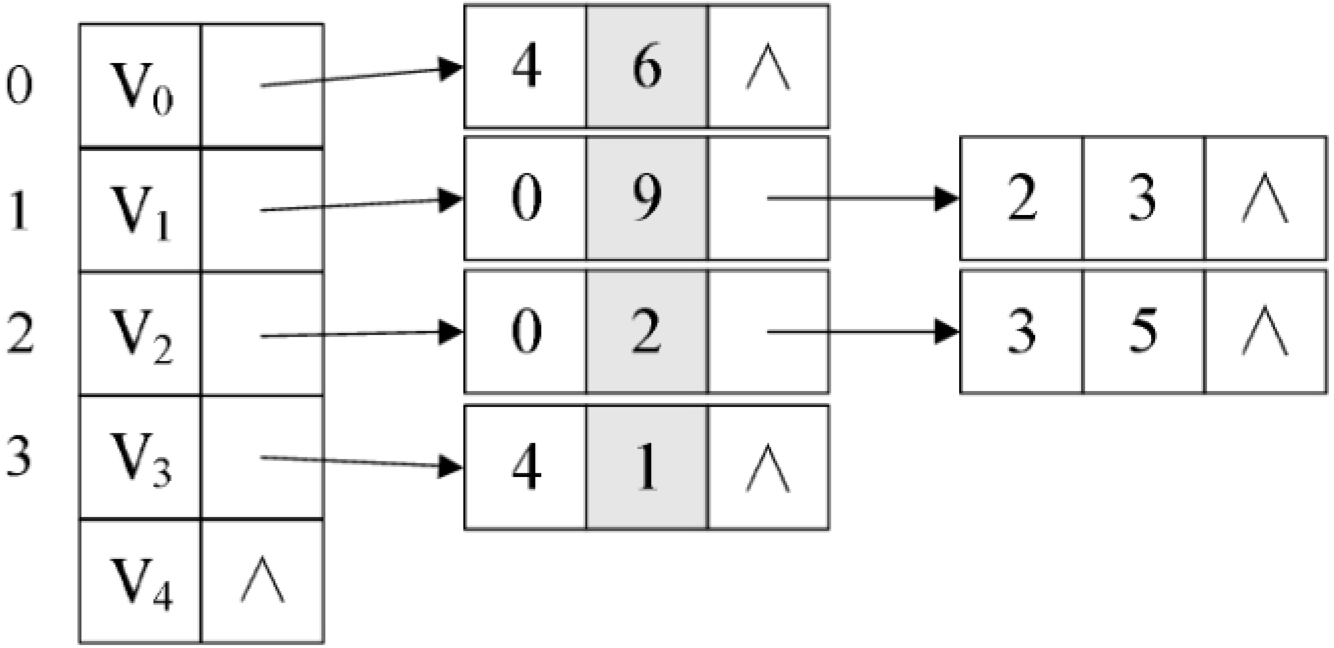




逆邻接表

邻接表 – 有权图





图的基础接口

```
int verticesSize();
int edgesSize();

void addVertex(V v);
void removeVertex(V v);

void addEdge(V fromV, V toV);
void addEdge(V fromV, V toV, E weight);
void removeEdge(V fromV, V toV);
```

顶点的定义

```java
private static class Vertex<V, E> {
    V value;
    Set<Edge<V, E>> inEdges = new HashSet<>();
    Set<Edge<V, E>> outEdges = new HashSet<>();
    Vertex(V value) {
        this.value = value;
    }
    @Override
    public boolean equals(Object obj) {
        return Objects.equals(value, ((Vertex<V, E>) obj).value);
    }
    @Override
    public int hashCode() {
        return value == null ? 0 : value.hashCode();
    }
}
```

边的定义

```java
private static class Edge<V, E> {
    Vertex<V, E> from;
    Vertex<V, E> to;
    E weight;
    public boolean equals(Object obj) {
        Edge<V, E> edge = (Edge<V, E>) obj;
        return from.equals(edge.from) && to.equals(edge.to);
    }
    public int hashCode() {
        return from.hashCode() * 31 + to.hashCode();
    }
}
```

遍历

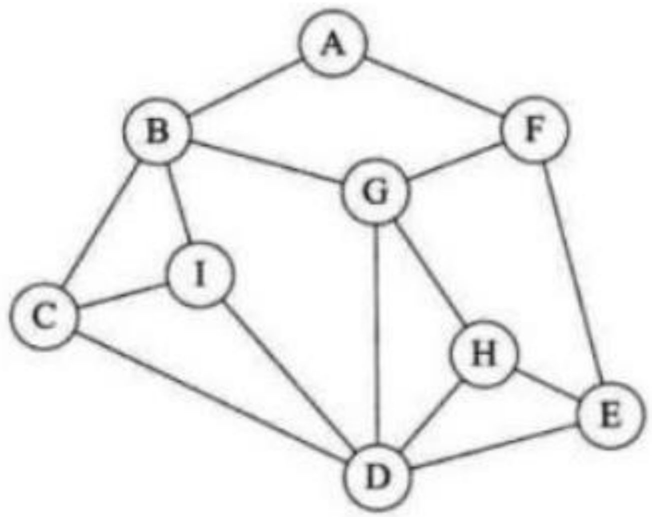
- ■ 图的遍历
- ◻ 从图中某一顶点出发访问图中其余顶点，且每一个顶点仅被访问一次

- ■ 图有2种常见的遍历方式（有向图、无向图都适用）
- ◻ 广度优先搜索（Breadth First Search，BFS），又称为宽度优先搜索、横向优先搜索

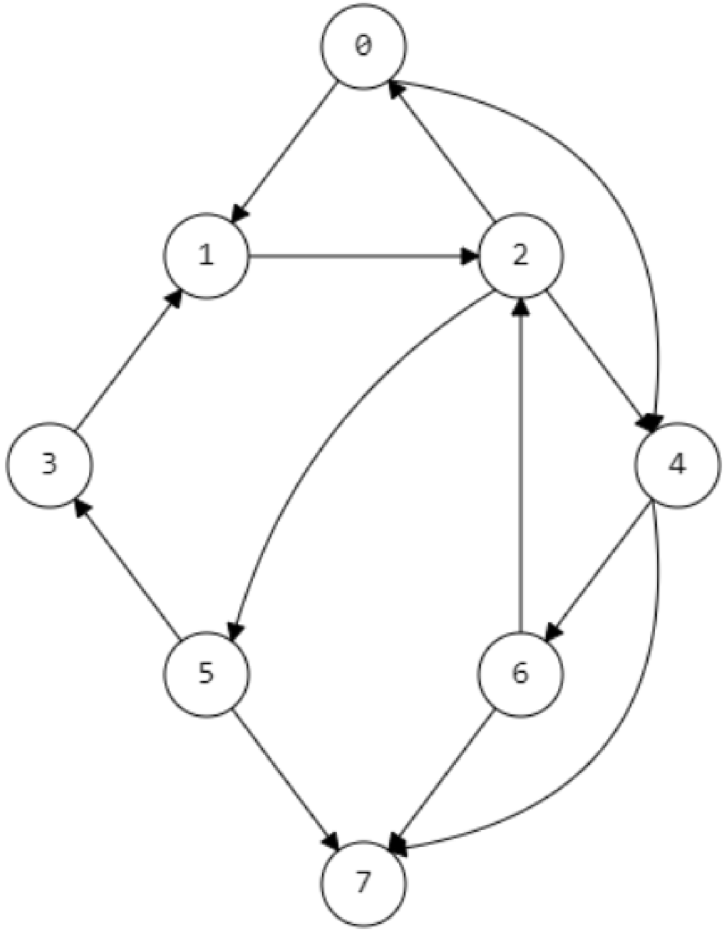
- ◻ 深度优先搜索（Depth First Search，DFS）
- ✓ 发明“深度优先搜索”算法的2位科学家在1986年共同获得计算机领域的最高奖：图灵奖

广度优先搜索（Breadth First Search）

- 之前所学的二叉树层序遍历就是一种广度优先搜索

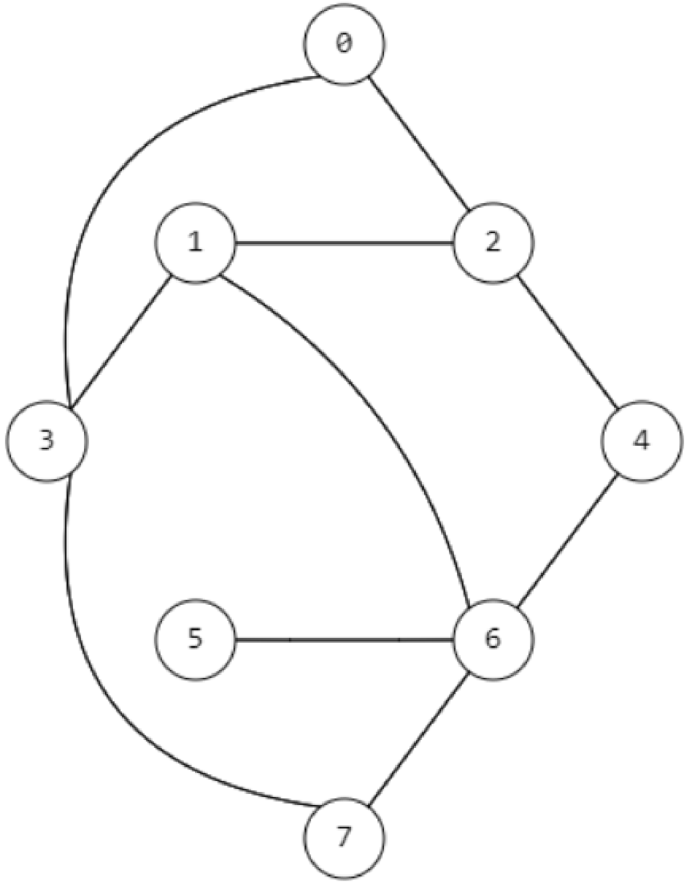


第1层	A
第2层	B、F
第3层	C、I、G、E
第4层	D、H

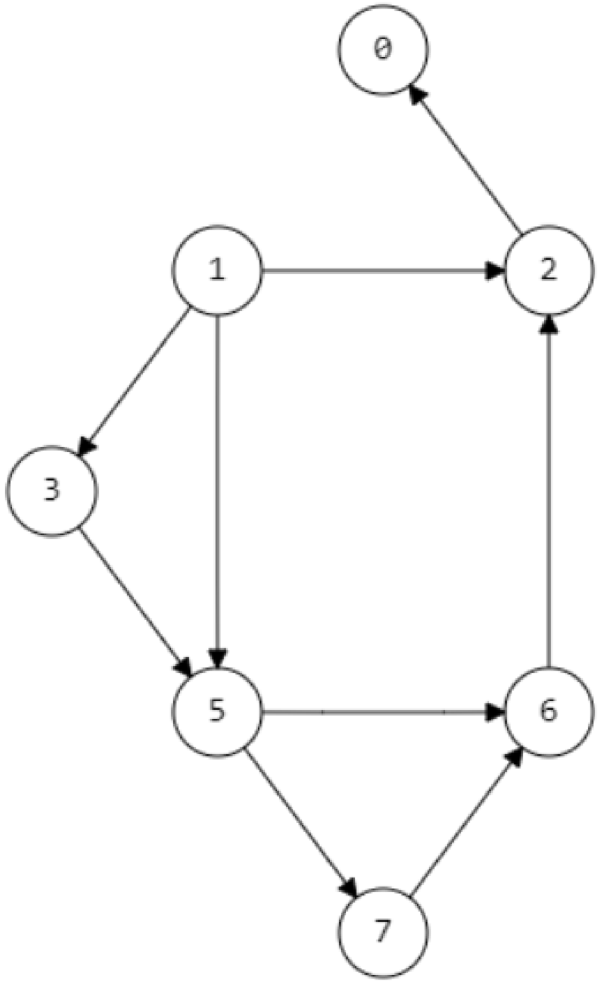


第1层	0
第2层	1、4
第3层	2、6、7
第4层	5
第5层	3

广度优先搜索

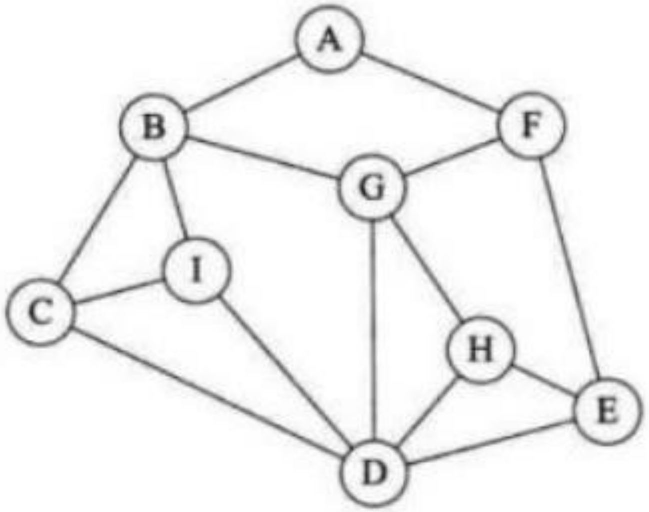


第1层	0
第2层	3、2
第3层	7、1、4
第4层	6
第5层	5



第1层	5
第2层	7、6
第3层	2
第4层	0

广度优先搜索 – 思路



← A ←

← B、F ←

← F、C、I、G ←

← C、I、G、E ←

← I、G、E、D ←

← G、E、D ←

← E、D、H ←

← D、H ←

← H ←

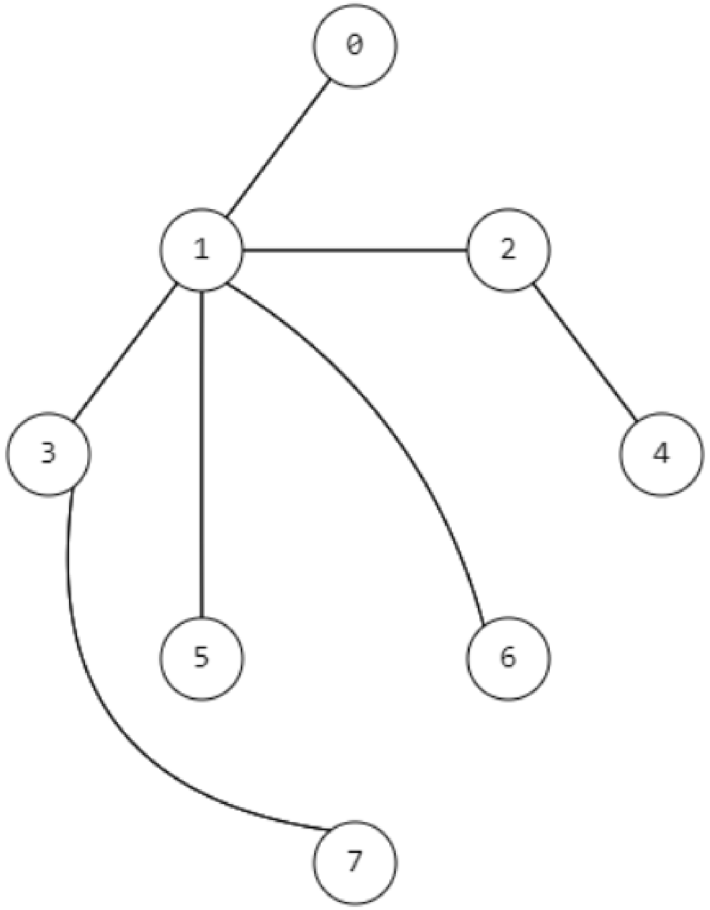
← ←

广度优先搜索 – 实现

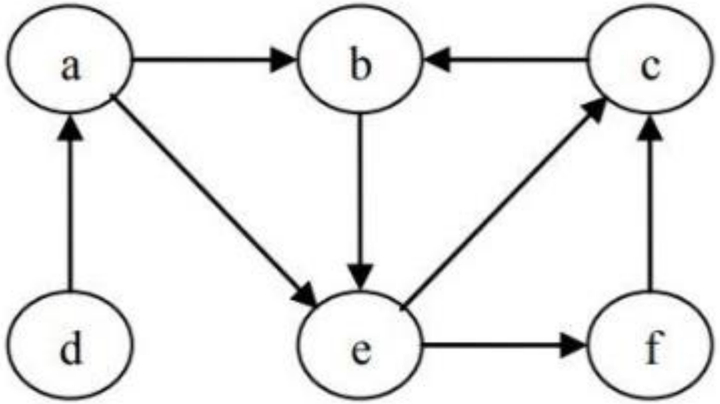
```java
private void bfs(Vertex<V, E> beginVertex) {
    Set<Vertex<V, E>> visitedVertices = new HashSet<>();
    Queue<Vertex<V, E>> queue = new LinkedList<>();
    queue.offer(beginVertex);
    visitedVertices.add(beginVertex);
    while (!queue.isEmpty()) {
        Vertex<V, E> vertex = queue.poll();
        System.out.println(vertex.value);

        for (Edge<V, E> edge : vertex.outEdges) {
            if (visitedVertices.contains(edge.to)) continue;
            queue.offer(edge.to);
            visitedVertices.add(edge.to);
        }
    }
}
```

深度优先搜索（Depth First Search）

■ 之前所学的二叉树前序遍历就是一种深度优先搜索



1、3、7、5、6、2、4、0



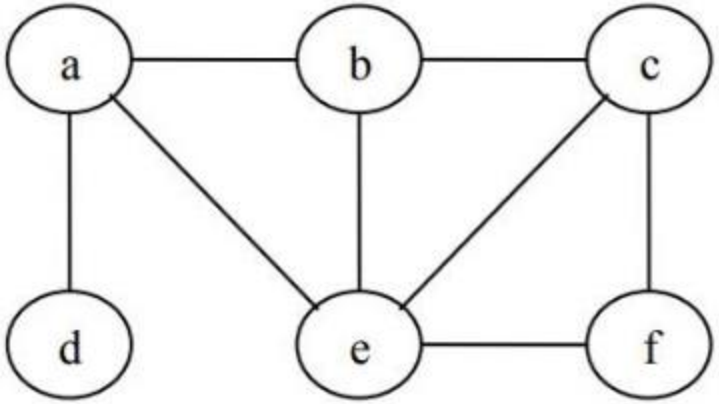
a、e、f、c、b

a、e、c、b、f

a、b、e、f、c

a、b、e、c、f

深度优先搜索



e、f、c、b、a、d

e、c、f、b、a、d

e、c、b、a、d、f

e、b、c、f、a、d

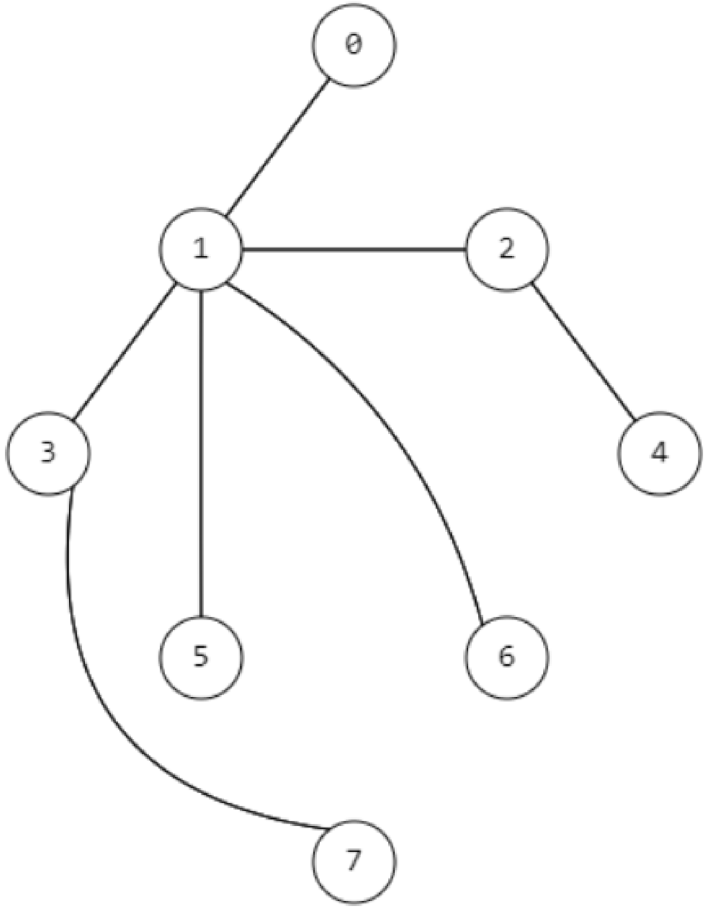
e、a、b、c、f、d

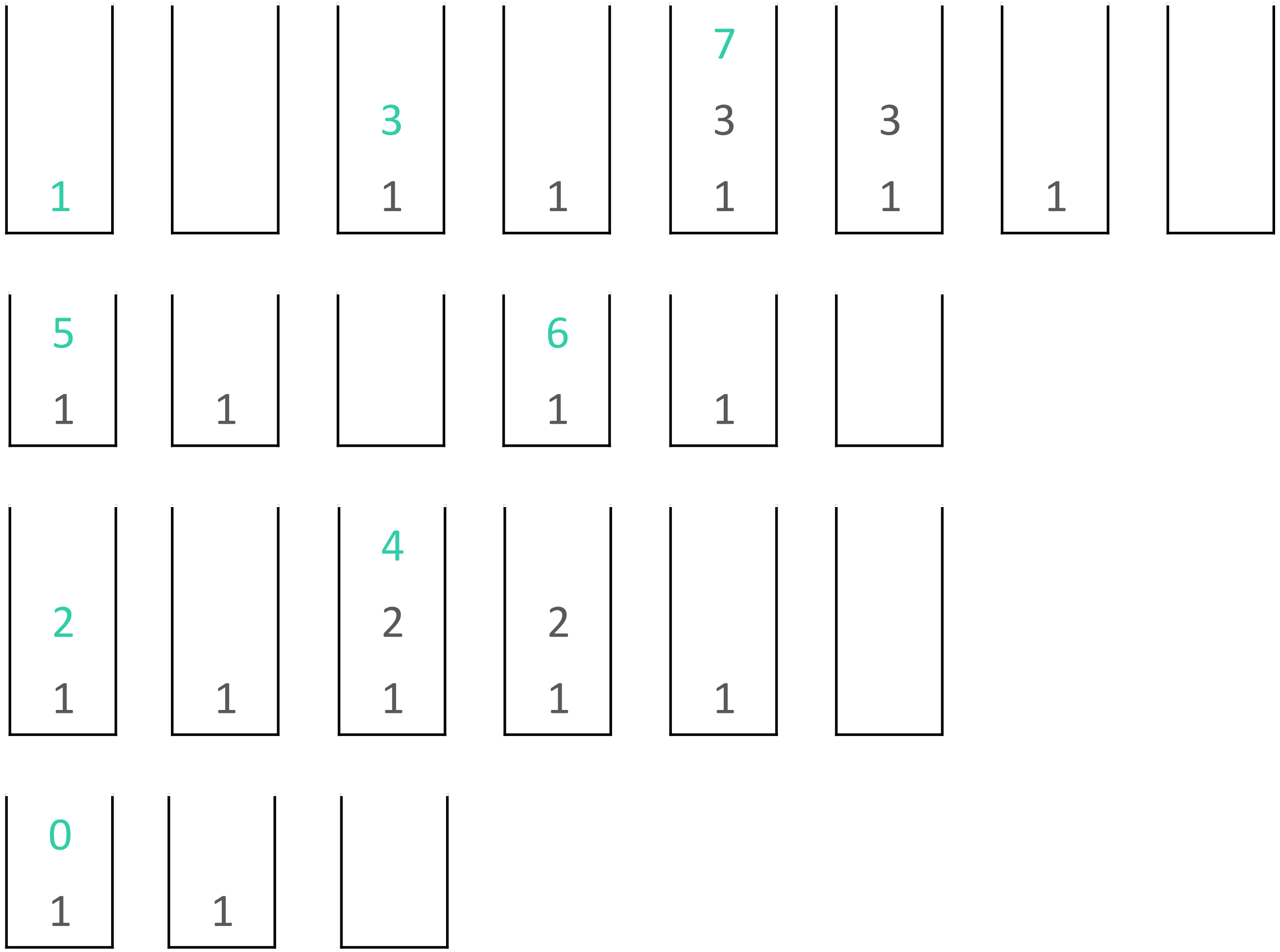
e、a、d、b、c、f

深度优先搜索 – 递归实现

```java
private void dfs(Vertex<V, E> vertex, Set<Vertex<V, E>> visitedVertices) {
    System.out.println(vertex.value);
    visitedVertices.add(vertex);

    for (Edge<V, E> edge : vertex.outEdges) {
        if (visitedVertices.contains(edge.to)) continue;
        dfs(edge.to, visitedVertices);
    }
}
```

深度优先搜索 – 非递归思路





深度优先搜索 – 非递归实现

```java
private void dfs(Vertex<V, E> beginVertex) {
    Set<Vertex<V, E>> visitedVertices = new HashSet<>();
    Stack<Vertex<V, E>> stack = new Stack<>();
    stack.push(beginVertex);
    visitedVertices.add(beginVertex);
    System.out.println(beginVertex.value);

    while (!stack.isEmpty()) {
        Vertex<V, E> vertex = stack.pop();
        for (Edge<V, E> edge : vertex.outEdges) {
            if (visitedVertices.contains(edge.to)) continue;
            stack.push(edge.from);
            stack.push(edge.to);
            visitedVertices.add(edge.to);
            System.out.println(edge.to.value);
            break;
        }
    }
}
```

AOV网（Activity On Vertex Network）

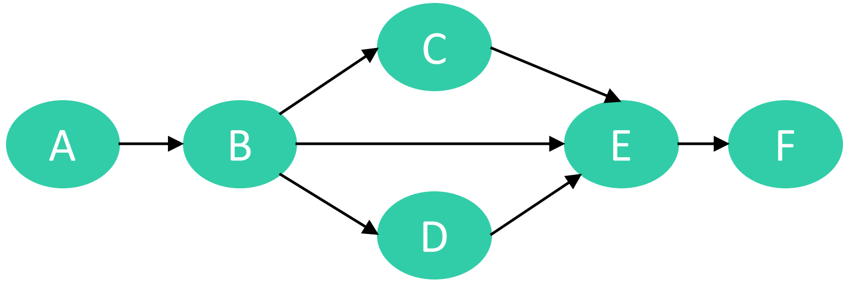
■ 一项大的工程常被分为多个小的子工程

✓ 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

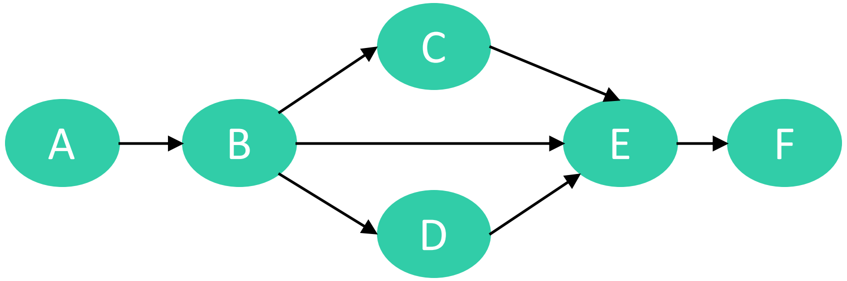
■ C依赖于B

■ 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）

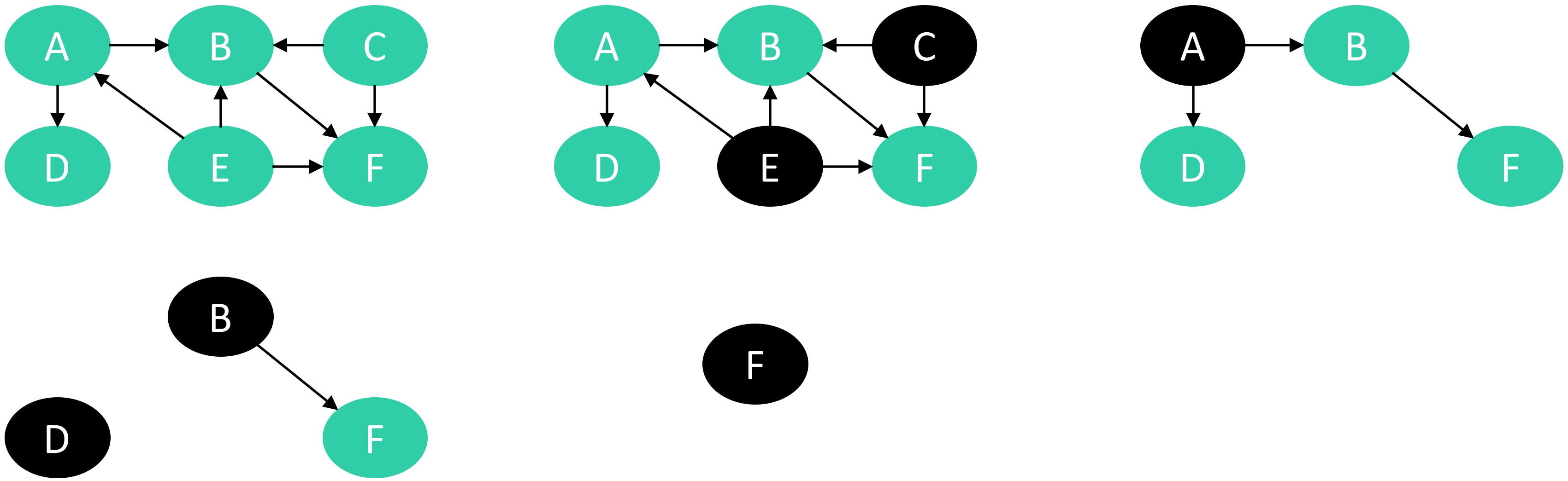
拓扑排序 – 思路

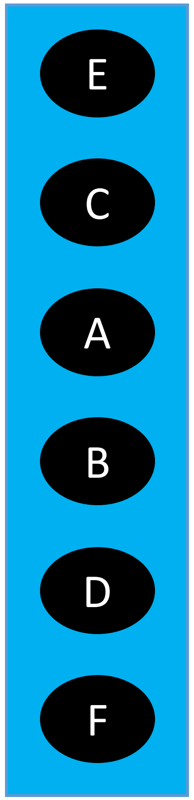
- 可以使用卡恩算法（Kahn于1962年提出）完成拓扑排序
  - 假设 L 是存放拓扑排序结果的列表

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);
        }
    }
}
```

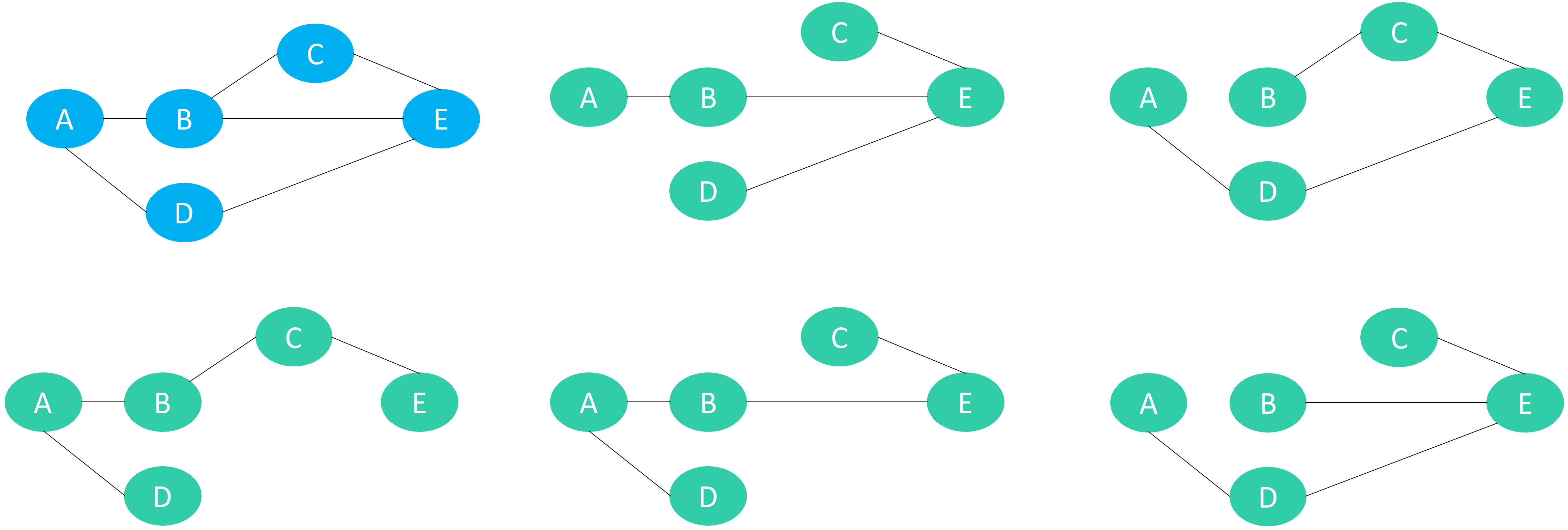
作业

- 自学《AOE网络》

- 课程表 II
  - https://leetcode-cn.com/problems/course-schedule-ii/

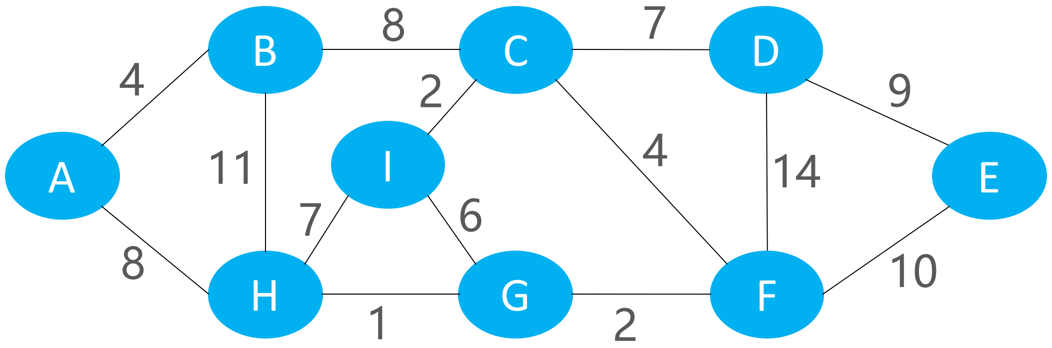
生成树（Spanning Tree）

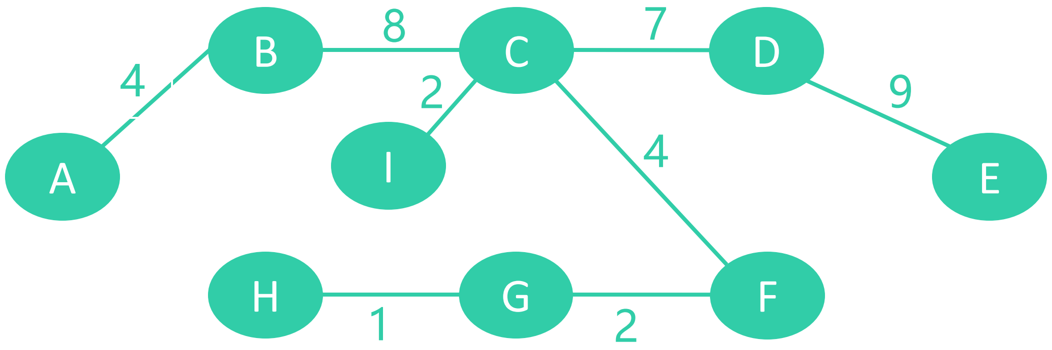
- 生成树（Spanning Tree），也称为支撑树
  - 连通图的极小连通子图，它含有图中全部的 n 个顶点，恰好只有 n – 1 条边



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?

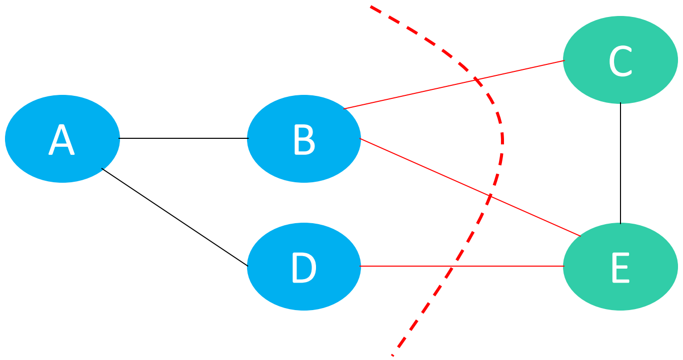
- ■ 如果图的每一条边的权值都互不相同，那么最小生成树将只有一个，否则可能会有多个最小生成树

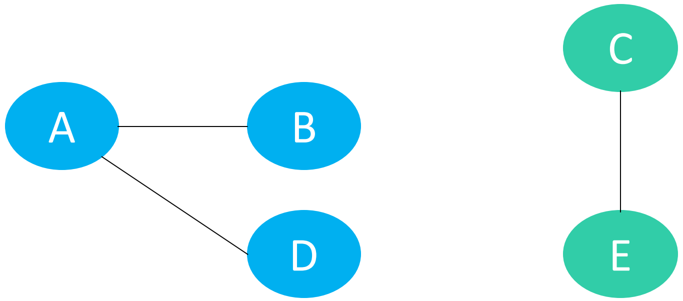
- ■ 求最小生成树的2个经典算法
- □ Prim（普里姆算法）
- □ Kruskal（克鲁斯克尔算法）

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

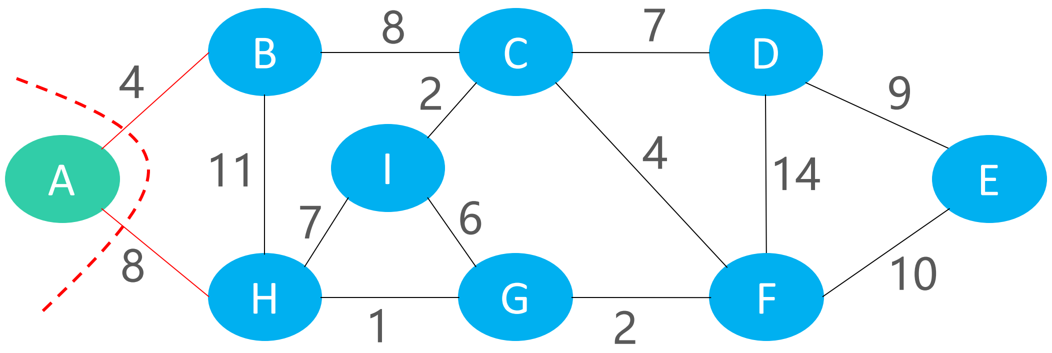
■ 切分定理：给定任意切分，横切边中权值最小的边必然属于最小生成树

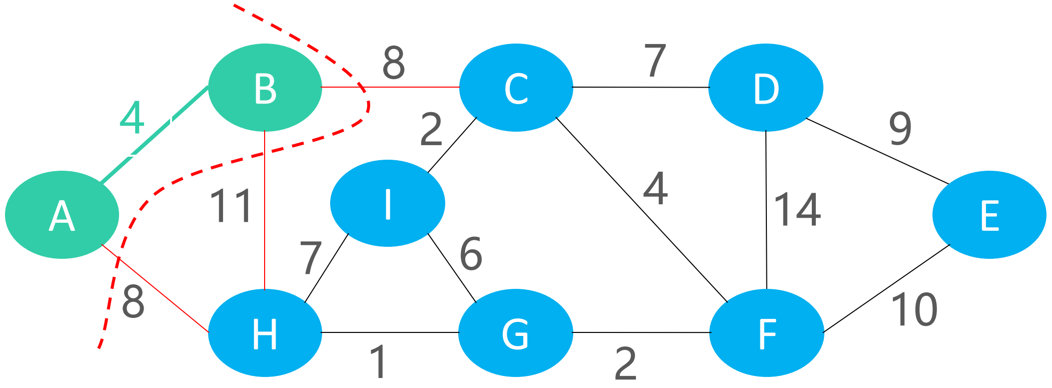
Prim算法 – 执行过程

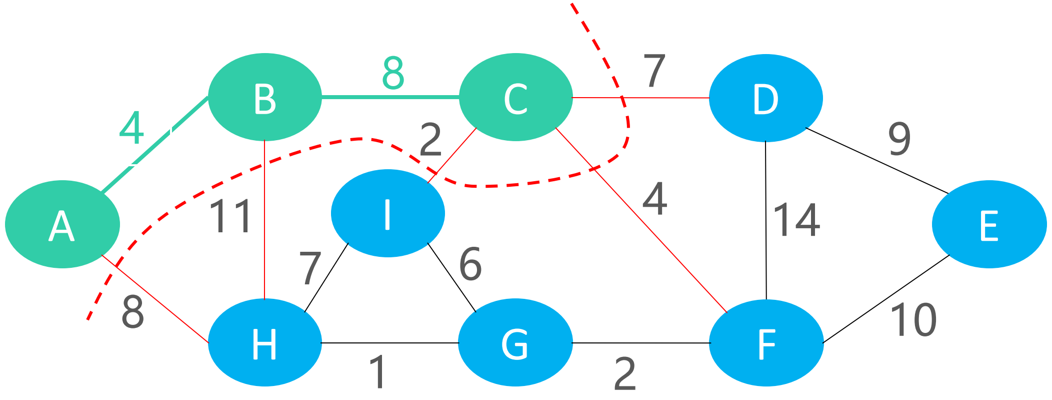
■ 假设 G = (V，E) 是有权的连通图（无向），A 是 G 中最小生成树的边集

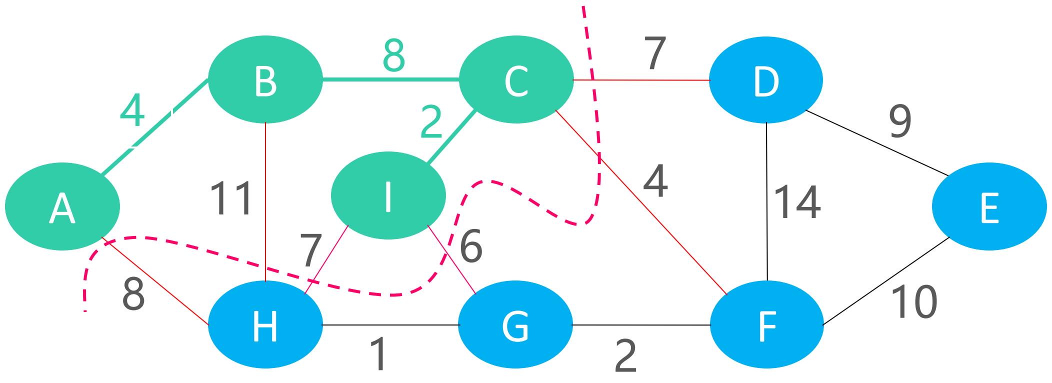
□ 算法从 S = { u_0 }（$u_0 \in$ V），A = { } 开始，重复执行下述操作，直到 S = V 为止

✓ 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

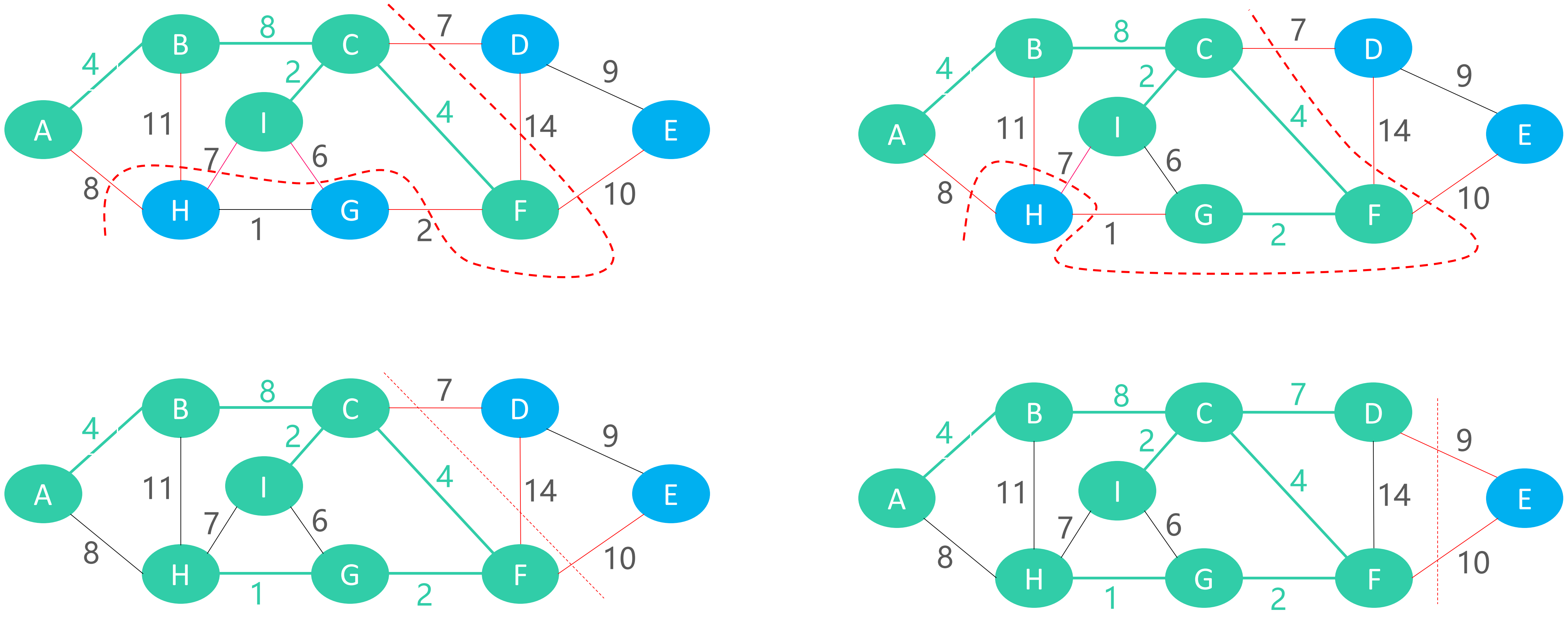




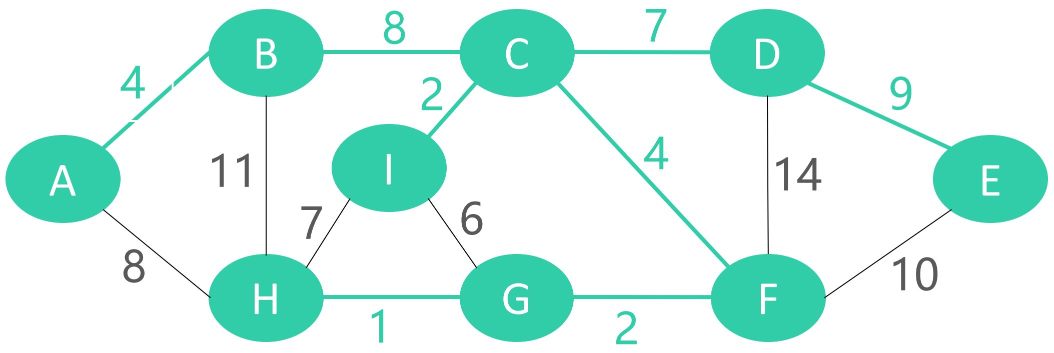


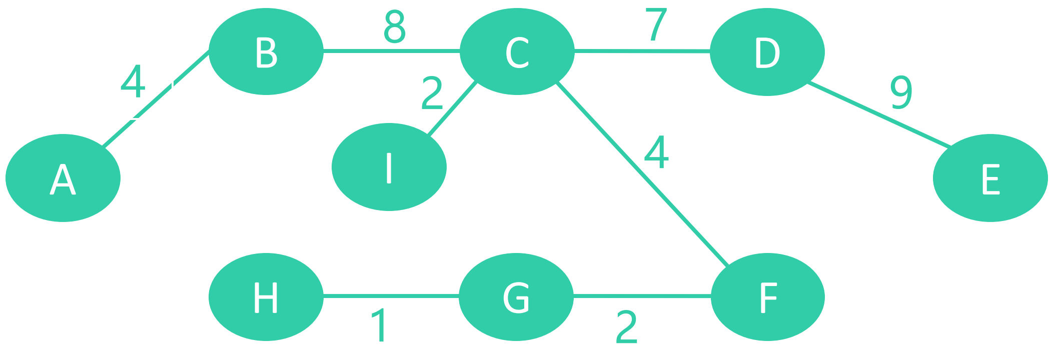


Prim算法 – 执行过程



Prim算法 – 执行过程



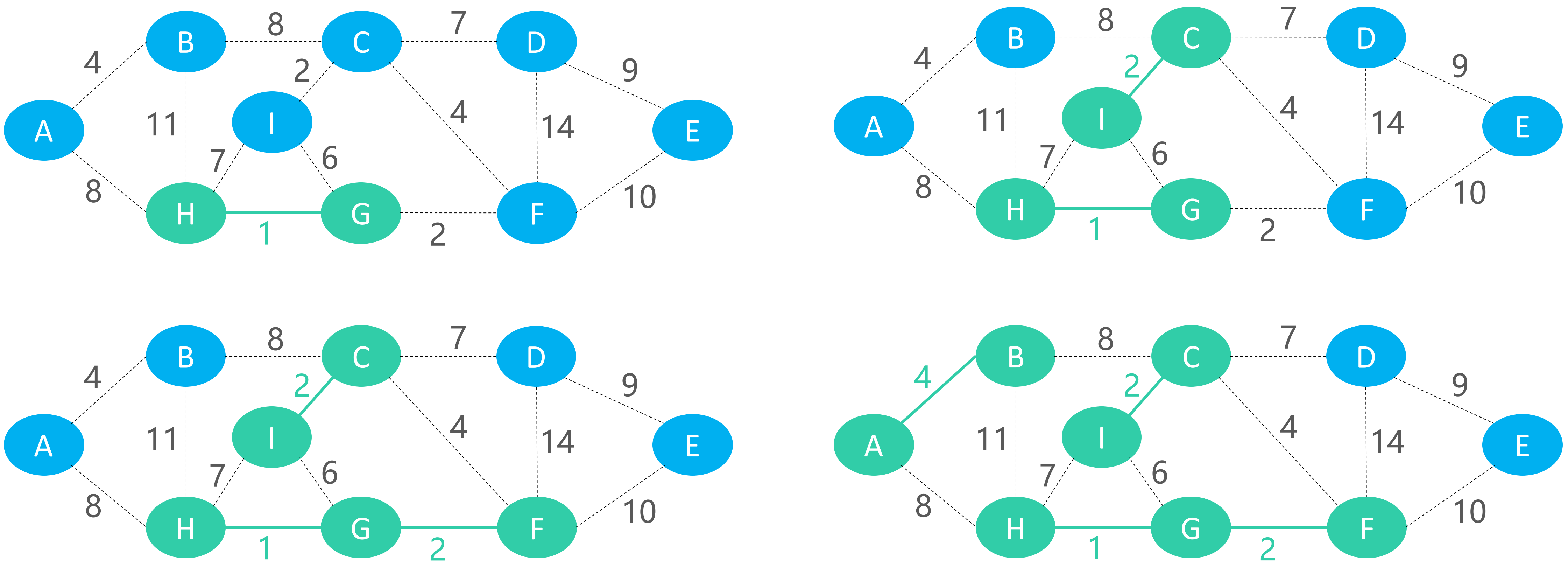


Prim算法 – 实现

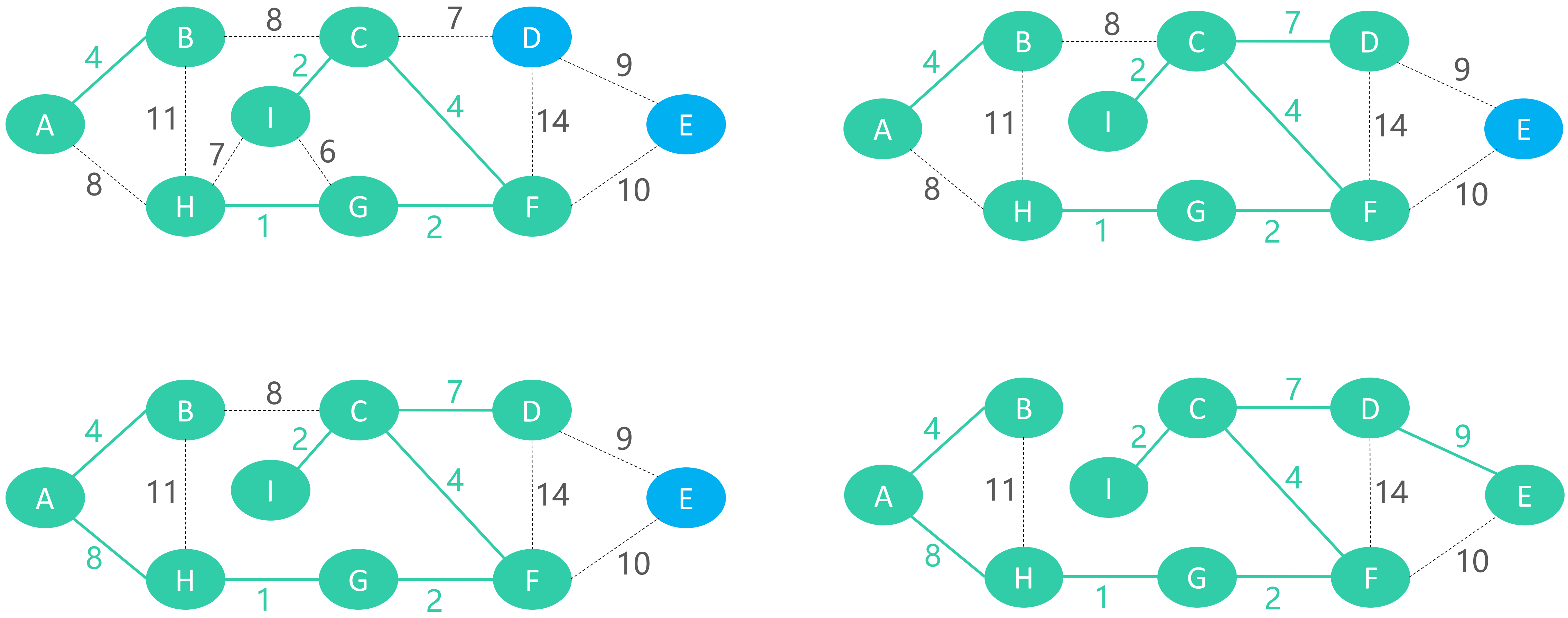
```java
private Set<EdgeInfo<V, E>> prim() {
    Iterator<Vertex<V, E>> it = vertices.values().iterator();
    if (!it.hasNext()) return null;
    Vertex<V, E> vertex = it.next();
    Set<EdgeInfo<V, E>> edgeInfos = new HashSet<>();
    Set<Vertex<V, E>> addedVertices = new HashSet<>();
    addedVertices.add(vertex);
    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;
}
```

Kruskal算法 – 执行过程

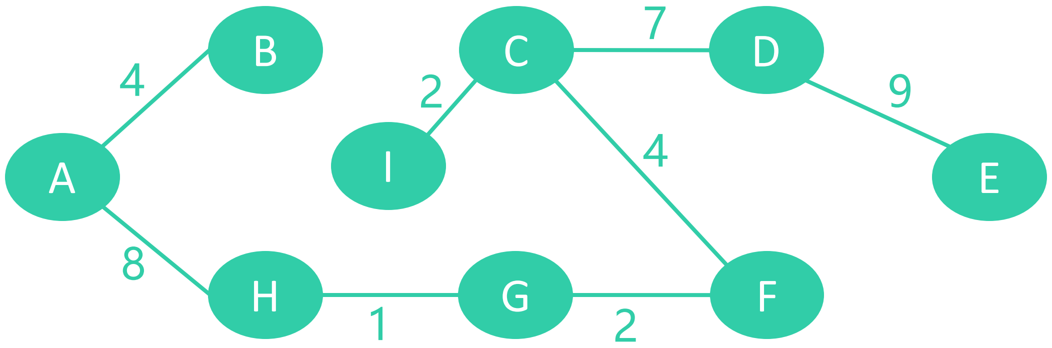
- 按照边的权重顺序（从小到大）将边加入生成树中，直到生成树中含有 V – 1 条边为止（ V 是顶点数量）
  - 若加入该边会与生成树形成环，则不加入该边
  - 从第3条边开始，可能会与生成树形成环



Kruskal算法 – 执行过程



Kruskal算法 – 执行过程



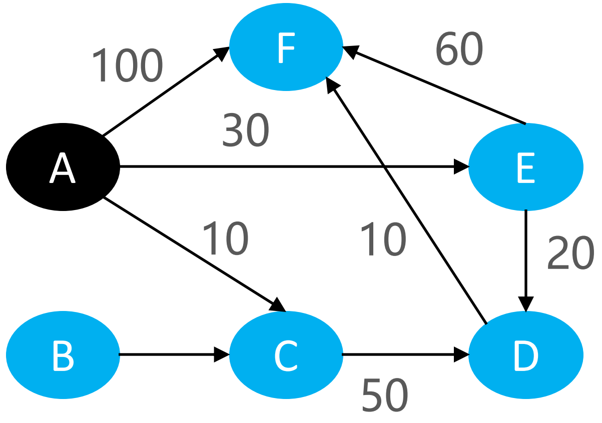
Kruskal算法 – 实现

```
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;
}
```

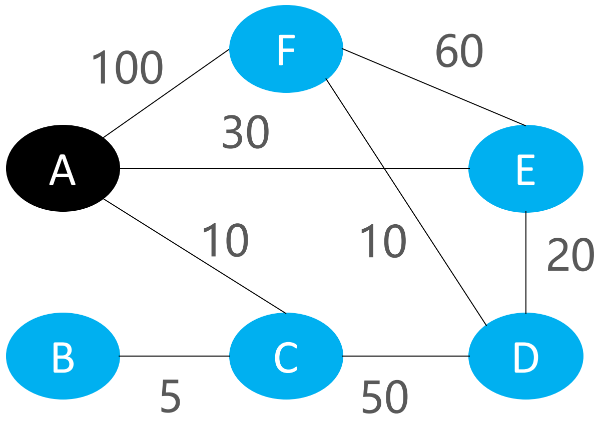
- 时间复杂度：O(ElogE)

最短路径（Shortest Path）

- 最短路径是指两顶点之间权值之和最小的路径（有向图、无向图均适用，不能有负权环）



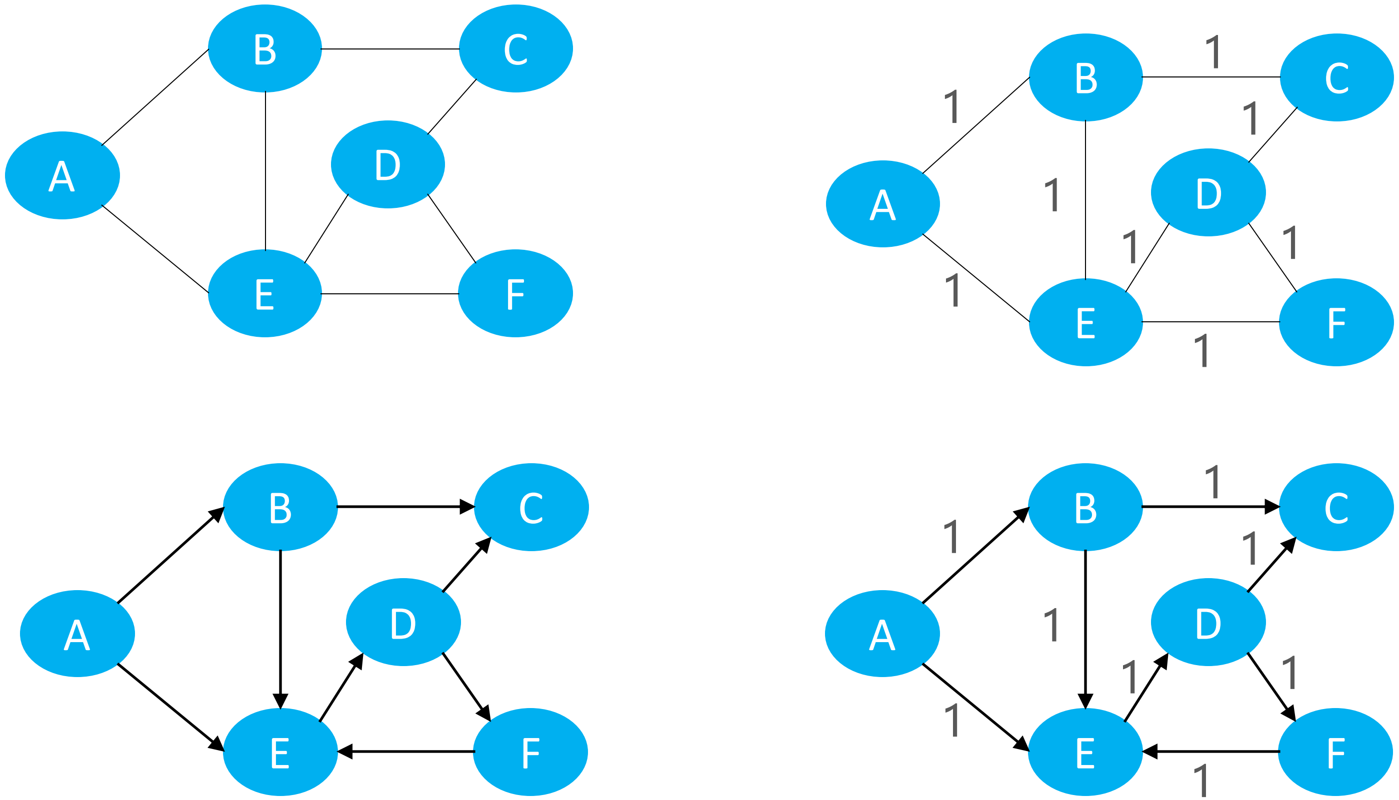
源点	终点	最短路径	路径长度
A	B		∞
	C	A → C	10
	D	A → E → D	50
	E	A → E	30
	F	A → E → D → F	60



源点	终点	最短路径	路径长度
A	B	A → C → B	15
	C	A → C	10
	D	A → E → D	50
	E	A → E	30
	F	A → E → D → F	60

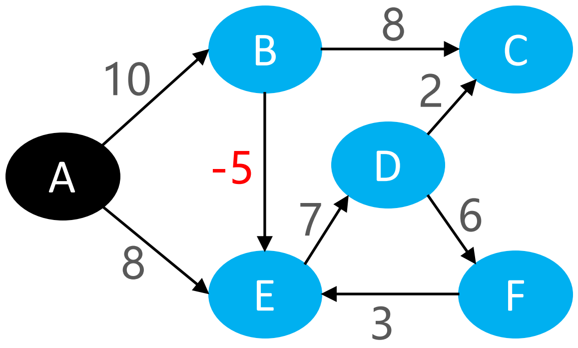
最短路径 – 无权图

- 无权图相当于是全部边权值为1的有权图



最短路径 – 负权边

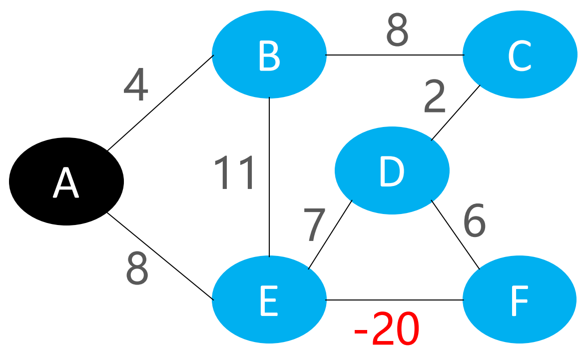
■ 有负权边，但没有负权环时，存在最短路径

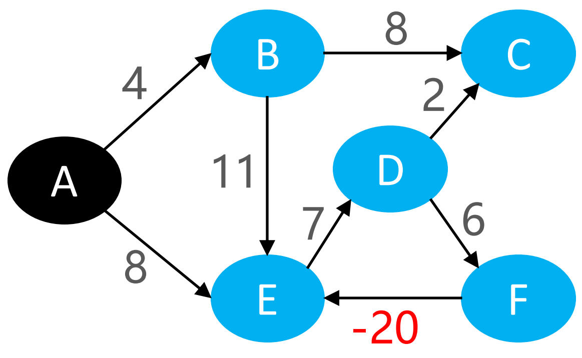


■ A到E的最短路径是：A → B → E

最短路径 – 负权环

■ 有负权环时，不存在最短路径





■ 通过负权环， A到E的路径可以无限短

□ A → E → D → F → E → D → F → E → D → F → E → D → F → E → ......

最短路径

■ 最短路径的典型应用之一：路径规划问题

■ 求解最短路径的3个经典算法

□ 单源最短路径算法

✓ Dijkstra（迪杰斯特拉算法）

✓ Bellman-Ford（贝尔曼-福特算法）

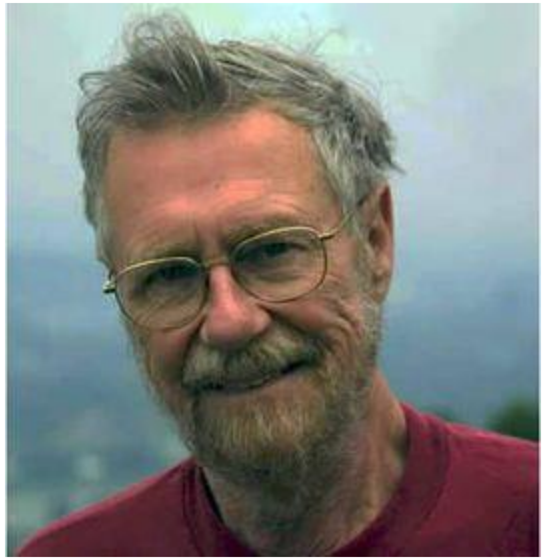
□ 多源最短路径算法

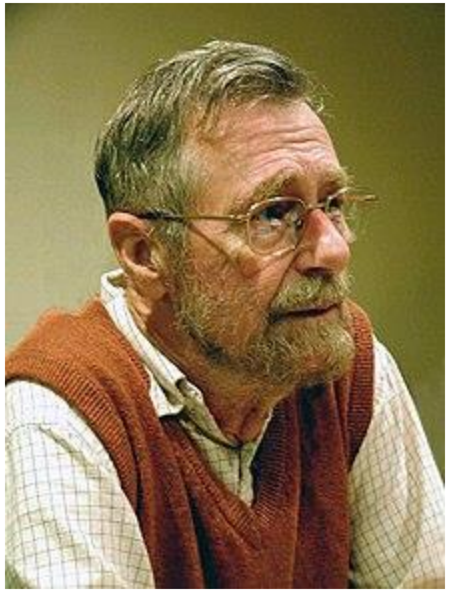
✓ Floyd（弗洛伊德算法）

Dijkstra

- Dijkstra 属于单源最短路径算法，用于计算一个顶点到其他所有顶点的最短路径
- 使用前提：不能有负权边
- 时间复杂度：可优化至 O(ElogV)，E 是边数量，V 是节点数量

- 由荷兰的科学家 Edsger Wybe Dijkstra 发明，曾在1972年获得图灵奖





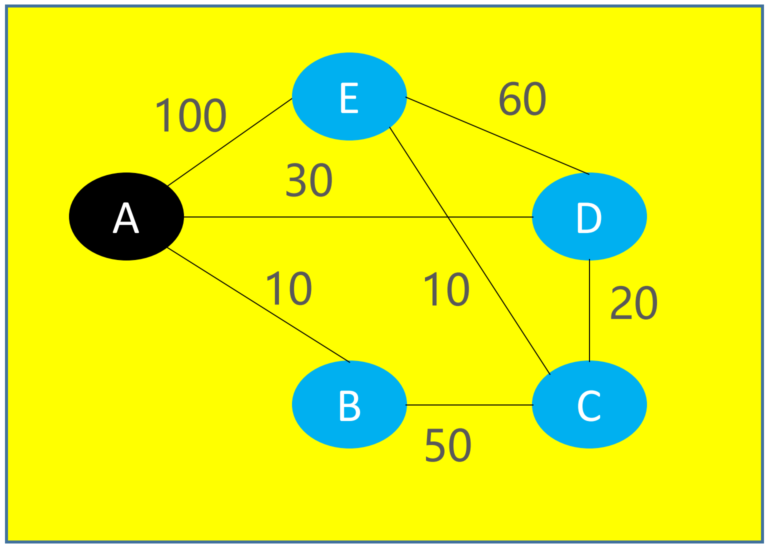
Dijkstra – 等价思考

■ Dijkstra 的原理其实跟生活中的一些自然现象完全一样

□ 把每1个顶点想象成是1块小石头

□ 每1条边想象成是1条绳子，每一条绳子都连接着2块小石头，边的权值就是绳子的长度

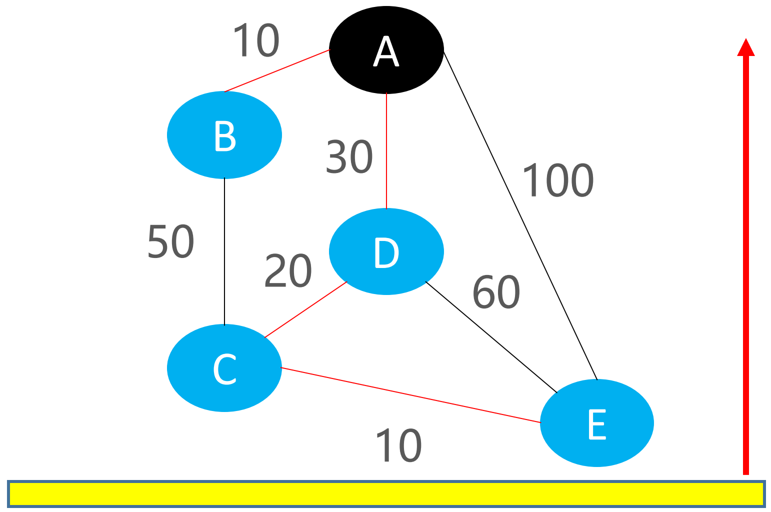
□ 将小石头和绳子平放在一张桌子上（下图是一张俯视图，图中黄颜色的是桌子）



■ 接下来想象一下，手拽着小石头A，慢慢地向上提起来，远离桌面

□ B、D、C、E会依次离开桌面

□ 最后绷直的绳子就是A到其他小石头的最短路径

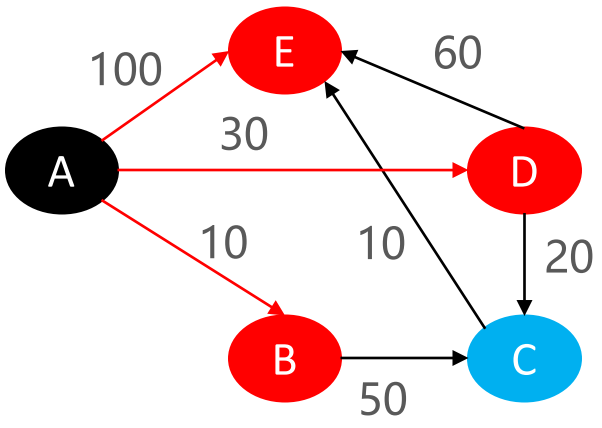


■ 有一个很关键的信息

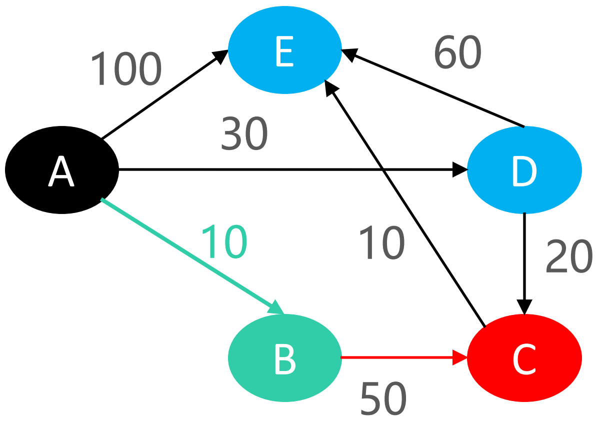
□ 后离开桌面的小石头

✓ 都是被先离开桌面的小石头拉起来的

Dijkstra – 执行过程



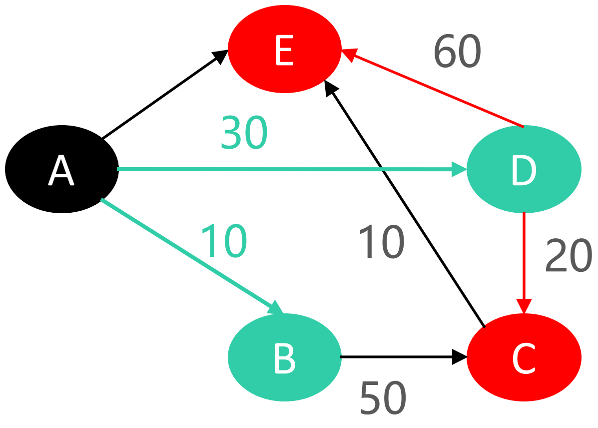
源点	终点	最短路径	路径长度
A	B	A → B	10
	C		∞
	D	A → D	30
	E	A → E	100



源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → B → C	60
	D	A → D	30
	E	A → E	100

- 绿色
  - 已经“离开桌面”
  - 已经确定了最终的最短路径
- 红色：更新了最短路径信息

Dijkstra – 执行过程



源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → D → C	50
	D	A → D	30
	E	A → D → E	90

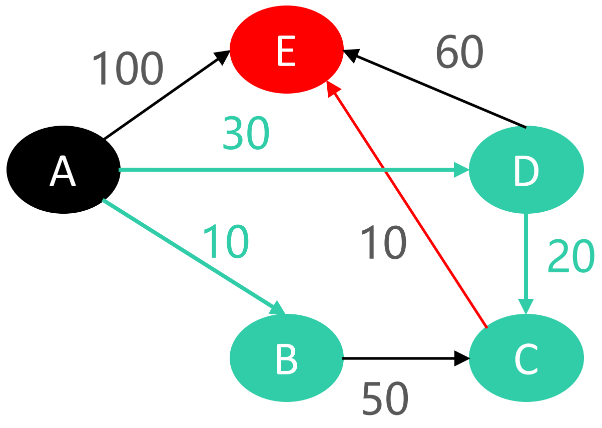
■ 松弛操作（Relaxation）：更新2个顶点之间的最短路径

□ 这里一般是指：更新源点到另一个点的最短路径

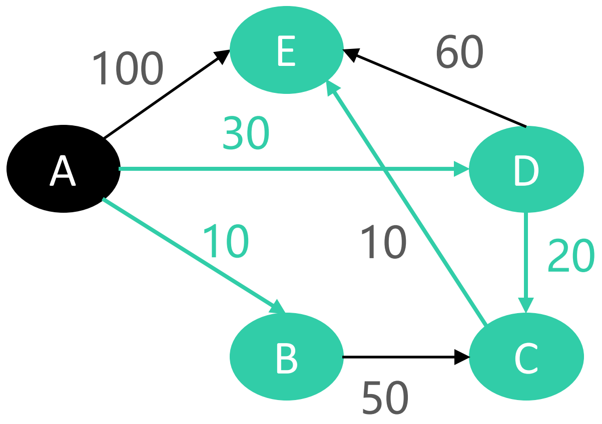
□ 松弛操作的意义：尝试找出更短的最短路径

■ 确定A到D的最短路径后，对DC、DE边进行松弛操作，更新了A到C、A到E的最短路径

Dijkstra – 执行过程



源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → D → C	50
	D	A → D	30
	E	A → D → C → E	60



源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → D → C	50
	D	A → D	30
	E	A → D → C → E	60

Bellman-Ford

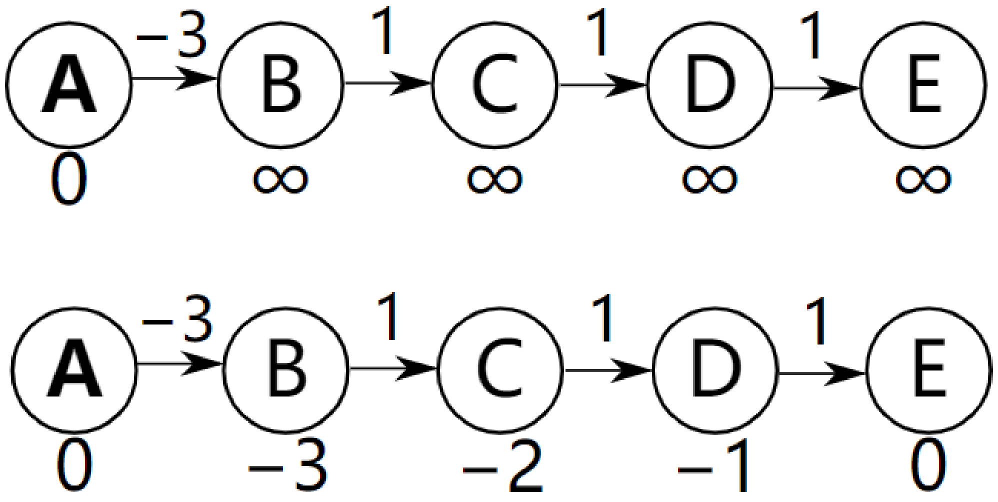
■ Bellman-Ford 也属于单源最短路径算法，支持负权边，还能检测出是否有负权环

□ 算法原理：对所有的边进行 V – 1 次松弛操作（ V 是节点数量），得到所有可能的最短路径

□ 时间复杂度：O(EV)，E 是边数量，V 是节点数量

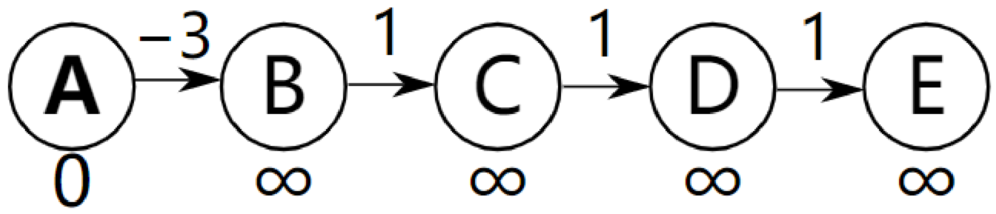
■ 下图的最好情况是恰好从左到右的顺序对边进行松弛操作

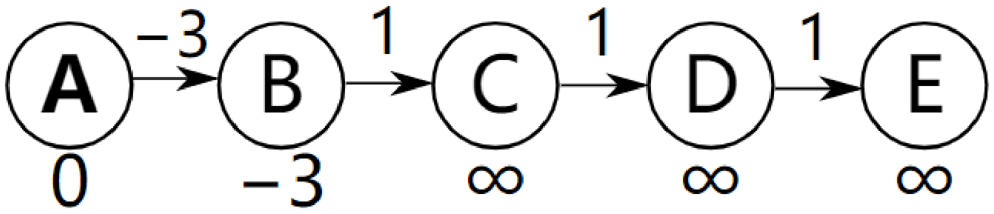
□ 对所有边仅需进行 1 次松弛操作就能计算出A到达其他所有顶点的最短路径

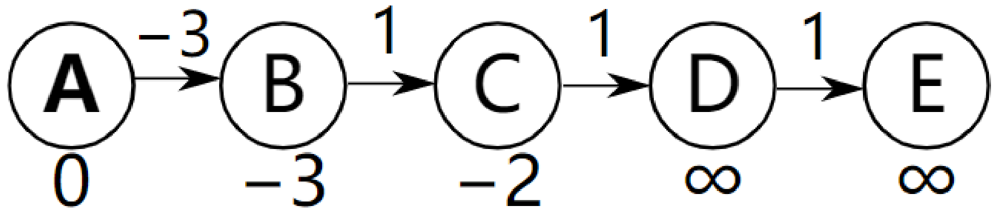


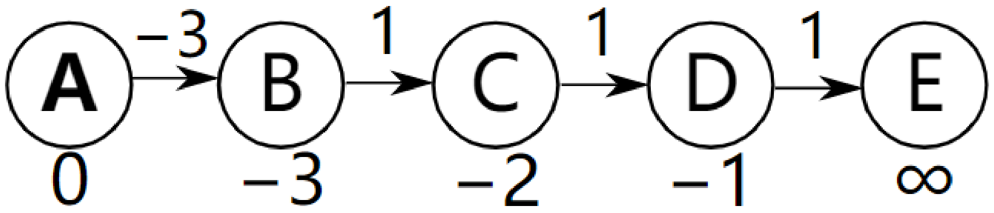
Bellman-Ford

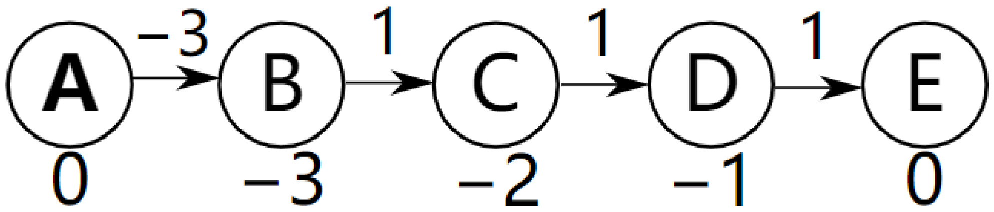
- 最坏情况是恰好每次都从右到左的顺序对边进行松弛操作
- 对所有边需进行 V – 1 次松弛操作才能计算出A到达其他所有顶点的最短路径



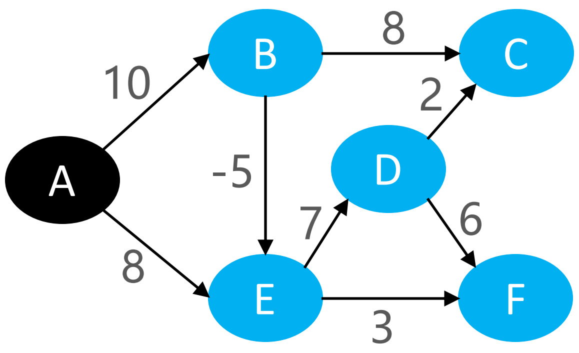








Bellman-Ford – 实例

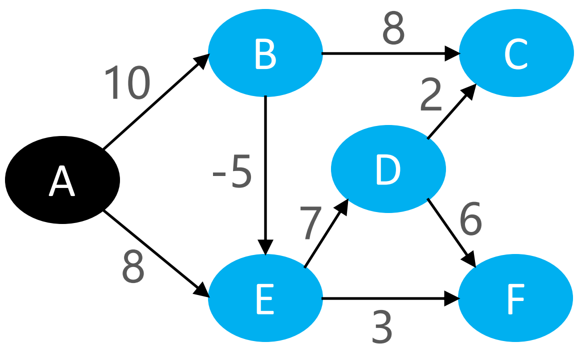


- 一共8条边
- 假设每次松弛操作的顺序是：DC、DF、BC、ED、EF、BE、AE、AB

第1次松弛操作			
源点	终点	最短路径	路径长度
A	B	A → B	10
	C		∞
	D		∞
	E	A → E	8
	F		∞

第2次松弛操作			
源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → B → C	18
	D	A → E → D	15
	E	A → B → E	5
	F	A → E → F	11

Bellman-Ford – 实例

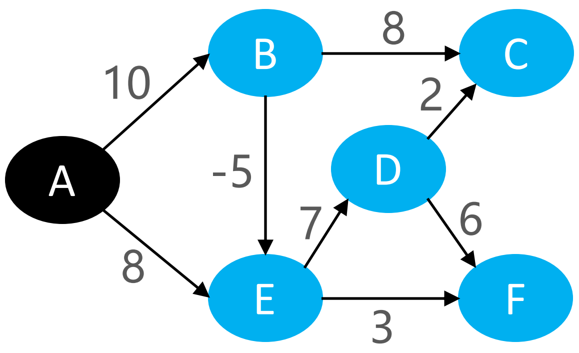


- 每次松弛操作的顺序是：DC、DF、BC、ED、EF、BE、AE、AB

第2次松弛操作			
源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → B → C	18
	D	A → E → D	15
	E	A → B → E	5
	F	A → E → F	11

第3次松弛操作			
源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → E → D→ C	17
	D	A → B → E → D	12
	E	A → B → E	5
	F	A → B → E → F	8

Bellman-Ford – 实例



■ 每次松弛操作的顺序是：DC、DF、BC、ED、EF、BE、AE、AB

第3次松弛操作			
源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → E → D→ C	17
	D	A → B → E → D	12
	E	A → B → E	5
	F	A → B → E → F	8

第4次松弛操作			
源点	终点	最短路径	路径长度
A	B	A → B	10
	C	A → B → E → D → C	14
	D	A → B → E → D	12
	E	A → B → E	5
	F	A → B → E → F	8

■ 不难分析出，经过4次松弛操作之后，已经计算出了A到其他所有顶点的最短路径

Floyd

■ Floyd 属于多源最短路径算法，能够求出任意2个顶点之间的最短路径，支持负权边

□ 时间复杂度：$O(V^3)$，效率比执行 V 次 Dijkstra 算法要好（ V 是顶点数量）

```
for (int k = 0; k < V; k++) {
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++) {
      if (dist(i, k) + dist(k, j) < dist(i, j)) {
        dist(i, j) = dist(i, k) + dist(k, j);
      }
    }
  }
}
```

■ 算法原理

□ 从任意顶点 i 到任意顶点 j 的最短路径不外乎两种可能

① 直接从 i 到 j

② 从 i 经过若干个顶点到 j

□ 假设 dist(i，j) 为顶点 i 到顶点 j 的最短路径的距离

□ 对于每一个顶点 k，检查 dist(i，k) + dist(k，j) < dist(i，j) 是否成立

✓ 如果成立，证明从 i 到 k 再到 j 的路径比 i 直接到 j 的路径短，设置 dist(i，j) = dist(i，k) + dist(k，j)

✓ 当我们遍历完所有结点 k，dist(i，j) 中记录的便是 i 到 j 的最短路径的距离