数据结构与算法B

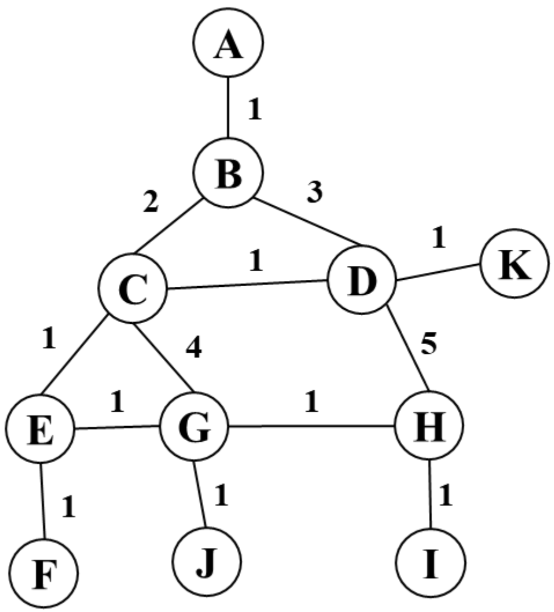
13-最小生成树

目录

13.1 图的最小生成树

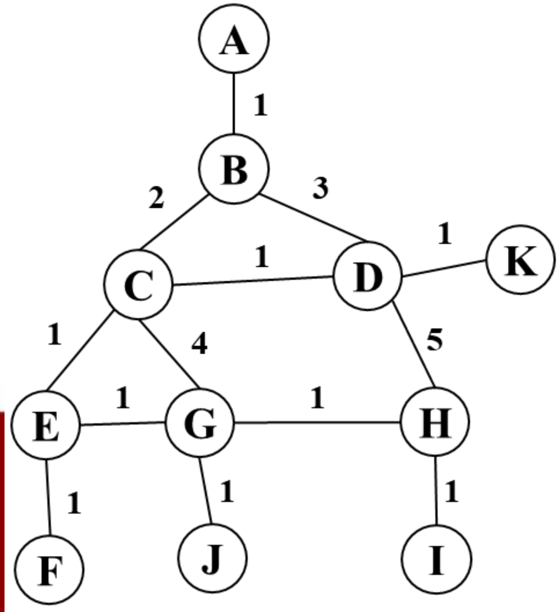
问题引入

- 先来看一个实际场景：信息广播问题
  - 下图展示了一个通信网络：表示为带权无向图，每个结点表示一个用户，相邻的结点可以互相通信，边权代表通信代价
  - 当前只有用户 A 具有信息，A 希望将信息传递至网络内的所有用户
  - 通信过程中，每沿着一条边传递一次数据，就会产生相应的通信代价。需要设计一种方案，使得总的代价之和最小



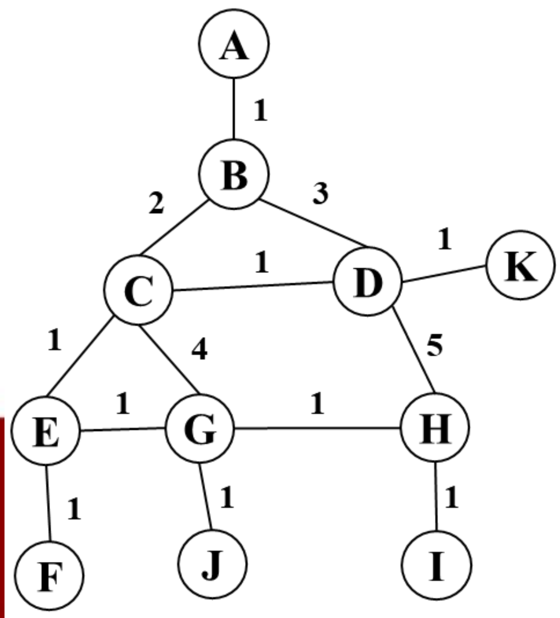
问题引入

- 洪泛解法：
  - 每个用户收到信息后，都将信息转发给相邻的所有用户
  - 信息会占据所有可能的路径，如同洪水泛滥，因此称洪泛
  - 洪泛能够保证信息传递到所有用户，但无法使转发过程停止
  - 因此，需要为数据包附加生命值（Time To Live, TTL）：每经过一次转发就减少 1，减少到 0 时路由器不再转发，初始设为最远距离
  - 可以想象，洪泛解法会产生大量重复的流量



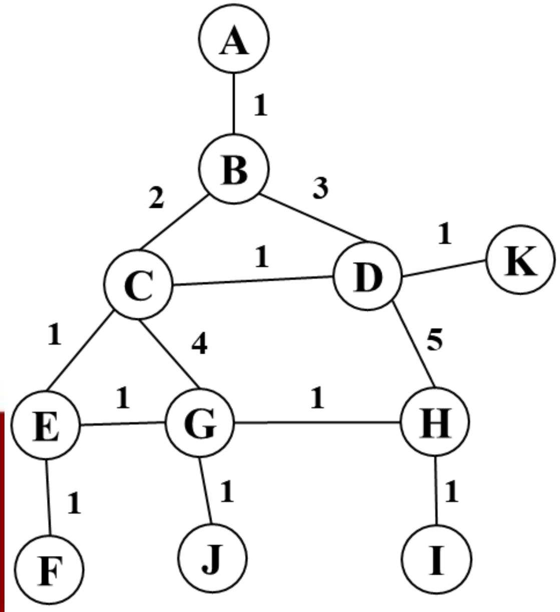
问题引入

- 单播解法：
  – 用户 A 负责通知网络内的每一个用户，即分别发送一条消息给用户 B, C, D, ……, K
  – 假设对于每一次发送，用户 A 都设法计算出最短带权路径，并沿着最短带权路径发送
  – 但这仍会产生大量重复流量，靠近 A 的边上会发生多次重复转发
  – 例如，A – B 一定位于所有最短带权路径上，会被重复 10 次



问题引入

- 最优解法：
  - 首先，网络中的每条边上最多只进行一次转发
  - 发生了转发的边不应该构成回路，否则就可以去除冗余边，这意味着，最优解一定构成了一个树
    - 在图论中，不含回路的无向图称为树
    - 注意区分：与数据结构中的树是完全不同的概念
  - 在原无向图中，构建边权和最小的一棵树，即本节课要研究的最小生成树问题



最小生成树

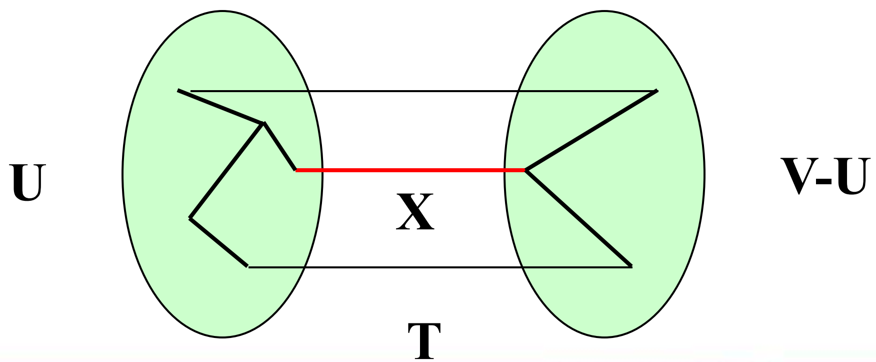
- 定义：生成树（Spanning Tree）
  - 对于带权无向连通图 G=(V, E)，以及 G 的一个子图 G'=(V', E')，若 V=V'，且 G' 是不含回路的连通图，则称 G' 是 G 的生成树
  - 连通图的生成树是连通图的一个极小连通子图，它含有图中的全部 n 个顶点，以及足以构成一棵树的 n-1 条边。
    - 增加一条边，则必定构成环；
    - 去掉一条边，则连通图变为不连通的。
  - 基于 DFS 以及 BFS 得到的搜索树，就是图的生成树，因此也称为 DFS 生成树、BFS 生成树
- 对于非连通图，从任一顶点出发无法访问到所有的顶点，只能得到各连通分量的生成树所组成的生成森林

最小生成树

- 定义：最小生成树（Minimum Spanning Tree, MST）
  - 定义生成树的权值为其中所有边的权值之和，则 G 的所有生成树中，权值最小的生成树称为图 G 的最小生成树
- 根据定义，最小生成树并不唯一
- 类似地，可以定义图的最大生成树
  - 求解最小生成树与最大生成树的本质是相同的，只需要将图中的所有边权取相反数即可转换二者
- 为了研究如何求解一个连通无向图的最小生成树，我们先来介绍最小生成树的一条重要性质

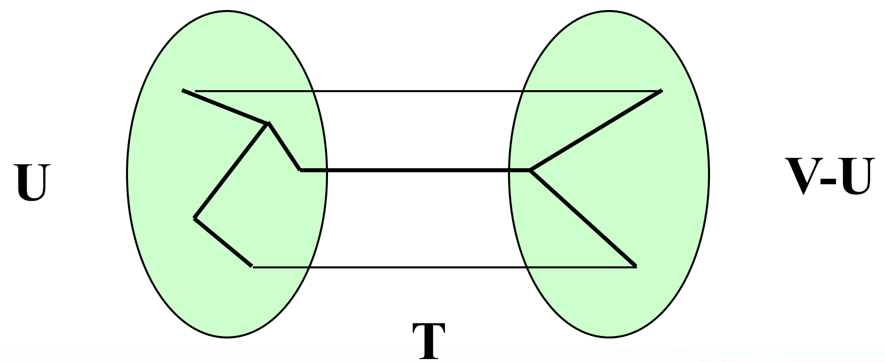
最小生成树的性质

- G=(V, E) 是带权连通无向图，U 是 V 的任意非空子集，考虑所有一端在 U 中而另一端不在 U 中的边，其中权值最小的边 X 一定属于 G 的一棵最小生成树 T。
- 直观理解：
  - 将图 G 任意地划分成两部分
  - 由于 G 是连通的，这两部分之间必然存在边作为“桥梁”
  - 任意一个权值最小的“桥梁”，一定属于某棵最小生成树 T



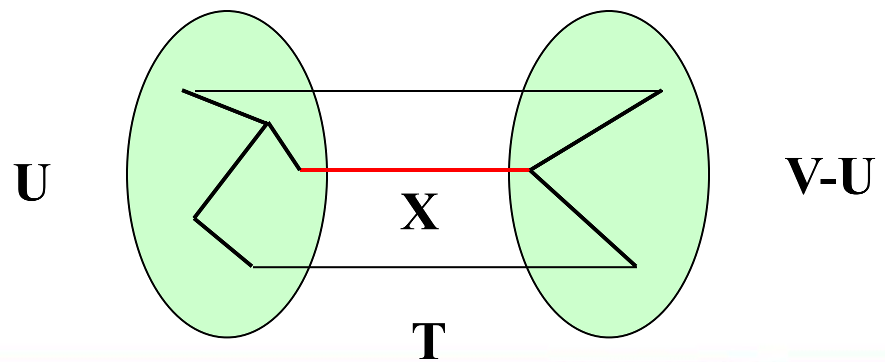
最小生成树的性质

- 性质的证明：
  – 考虑任意一棵最小生成树 T，其中必然存在边跨越了 U 与 V – U。



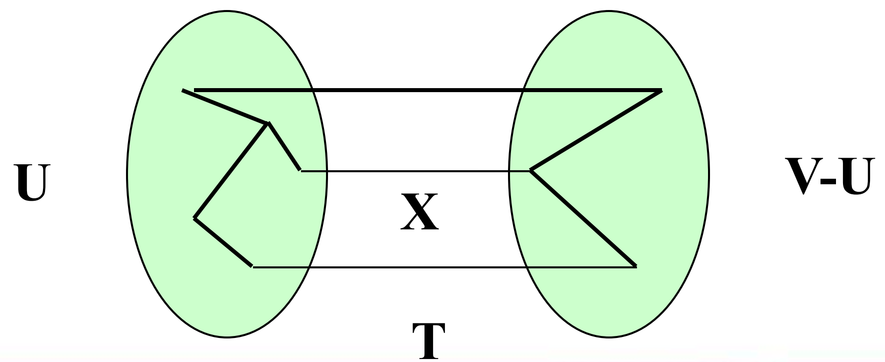
最小生成树的性质

- 性质的证明：
  - 考虑任意一棵最小生成树 T，其中必然存在边跨越了 U 与 V – U。
  - 若其中包括 X，则证明结束；



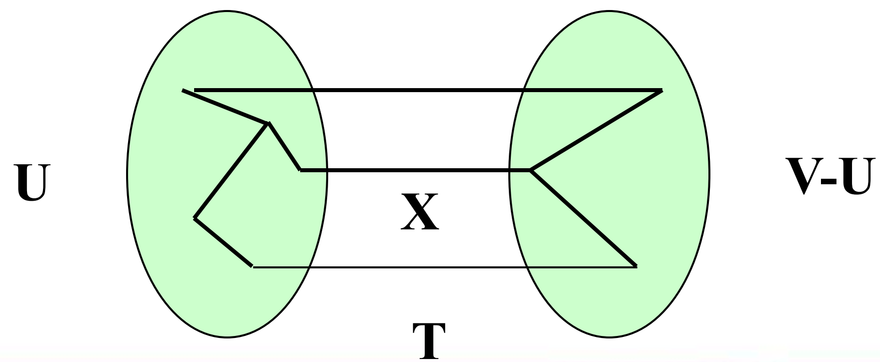
最小生成树的性质

- 性质的证明：
  - 考虑任意一棵最小生成树 T，其中必然存在边跨越了 U 与 V – U。
  - 若其中包括 X，则证明结束；
  - 下面假设其中不包括 X



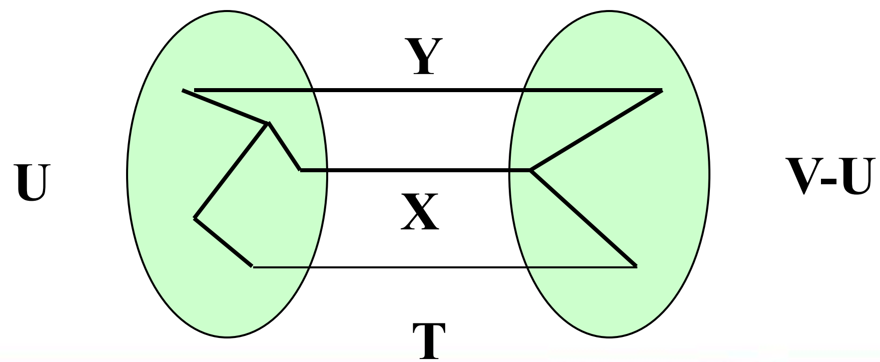
最小生成树的性质

- 性质的证明：
  – 考虑任意一棵最小生成树 T，其中必然存在边跨越了 U 与 V – U。
  – 若其中包括 X，则证明结束；下面假设其中不包括 X
  – 将 X 加入 T 中，必然会形成一个回路



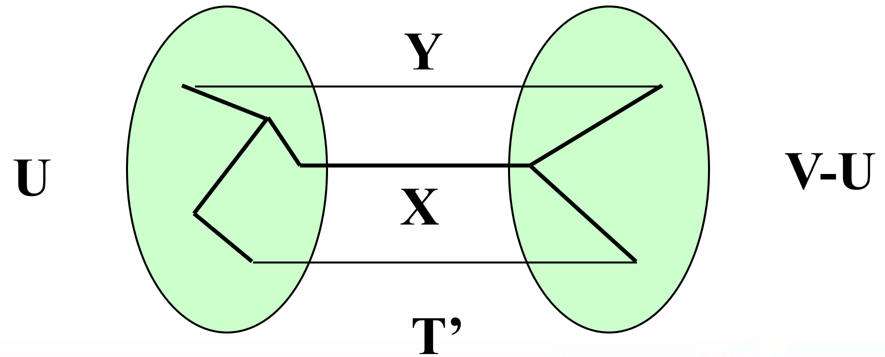
最小生成树的性质

- 性质的证明：
  - 考虑任意一棵最小生成树 T，其中必然存在边跨越了 U 与 V – U。
  - 若其中包括 X，则证明结束；下面假设其中不包括 X
  - 将 X 加入 T 中，必然会形成一个回路
  - 该回路中，跨越 U 与 V – U 的边，至少包括 X 与另一条边 Y



最小生成树的性质

- 性质的证明：
  - 考虑任意一棵最小生成树 T，其中必然存在边跨越了 U 与 V – U。
  - 若其中包括 X，则证明结束；下面假设其中不包括 X
  - 将 X 加入 T 中，必然会形成一个回路
  - 该回路中，跨越 U 与 V – U 的边，至少包括 X 与另一条边 Y
  - 删除 Y 得到 T’。由于 X 的权值不大于 Y，因此 T’ 权值也不大于 T
  - 就构造出了包含边 X 的最小生成树



13.2 Prim算法

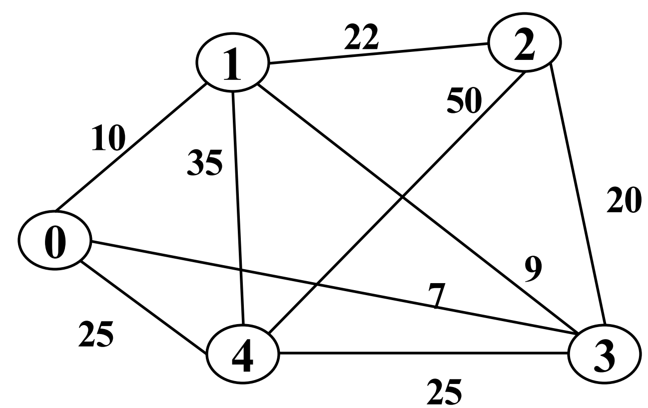
Prim算法

- Prim 算法是求解最小生成树问题的贪心算法，其基本思路如下：
  - 假设G = (V, E)是有n个节点的带权连通图，为构建 G = (V, E) 的一棵最小生成树 T = (V’, E’)，将 V’ 初始化为包含V中一个顶点（任意一个顶点）的集合，将 E’ 初始化为空集
  - 每次从 V – V’ （即在T外的顶点）中，寻找一个“距离 V’ 最近”的结点，加入 V’ 中；对应关联的边加入 E’ 中
    - 距离 V’ 最近的含义为，与 V’ 中的结点相邻，且边权最小
    - 可以理解为，每次从跨越了 V’ 与 V – V’ 的边中，寻找权值最小的，将其关联的顶点加入 V’ 中，将这条边本身加入 E’ 中
  - 重复该步骤 n – 1次，就得到了最小生成树 T

Prim算法：示例

- 用 Dist 列表表示结点到 V’ 的“距离”，用 Prev列表表示结点到 V’ 中的哪一个结点距离最近
  - Dist[i]表示顶点i到T的距离，即i和T中的顶点的所有连边的最小权值，初始时，Dist[i]均为无穷大
  - 顶点i不在T中时，Pred[i]表示T中和顶点i有边连接，且边最短的那个顶点，Prev[i]为None表示尚未发现这样的顶点。初始时，对所有顶点i, Prev[i]均为None.
  - 选取 0 作为初始节点，初始化 Dist, Prev

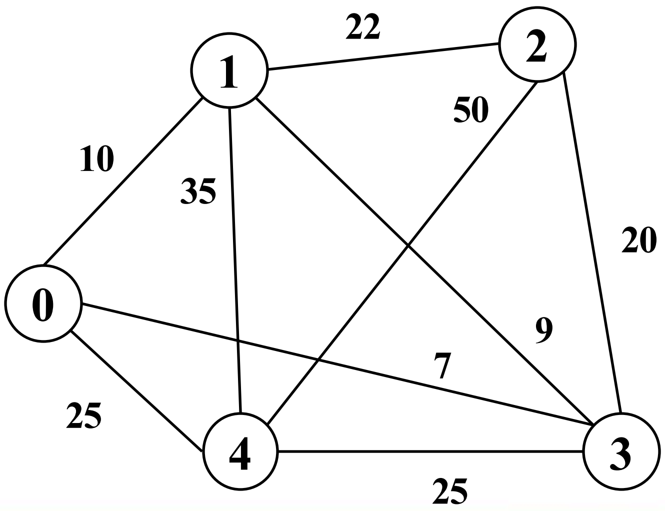
Dist	0	∞	∞	∞	∞
Prev	None	None	None	None	None



Prim算法：示例

- 用 Dist 列表表示结点到 V’ 的“距离”，用 Prev 列表表示结点到 V’ 中的哪一个结点距离最近。
  - 依据结点 0 关联的边，更新信息（结点 1, 3, 4）

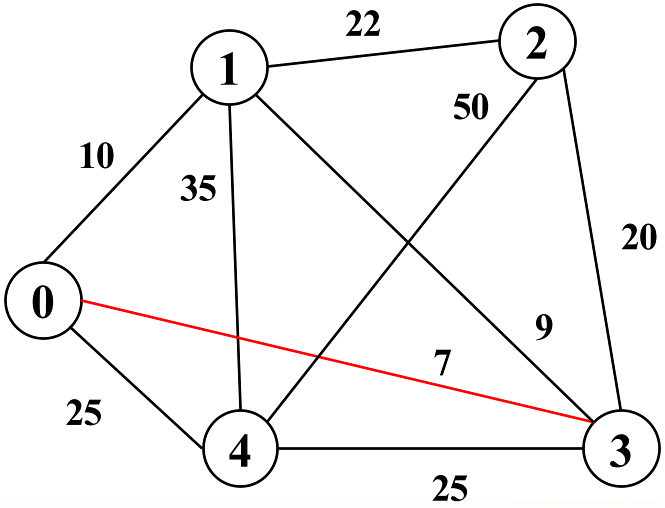
Dist	0	10	∞	7	25
Prev	None	0	None	0	0



Prim算法：示例

- 用 Dist 列表表示结点到 V’ 的“距离”，用 Prev 列表表示结点到 V’ 中的哪一个结点距离最近。
  - 选取结点 3 加入 V’，边（0，3）加入E’中

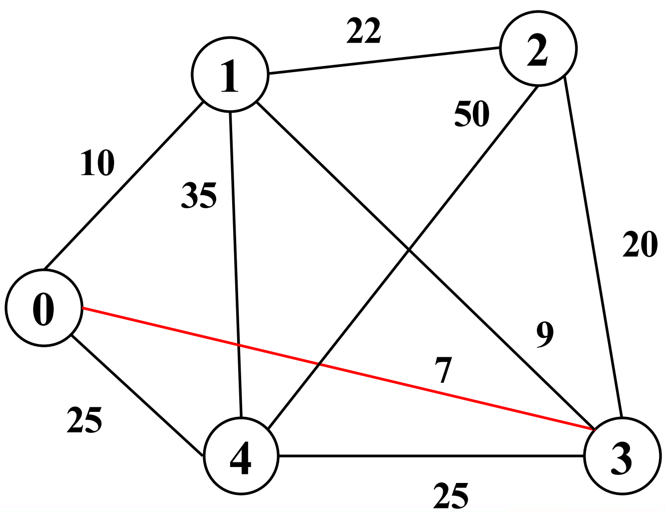
Dist	0	10	∞	0	25
Prev	None	0	None	0	0



Prim算法：示例

- 用 Dist 列表表示结点到 V’ 的“距离”，用 Prev 列表表示结点到 V’ 中的哪一个结点距离最近。
  - 依据结点 3 关联的边，更新信息（结点 1, 2）

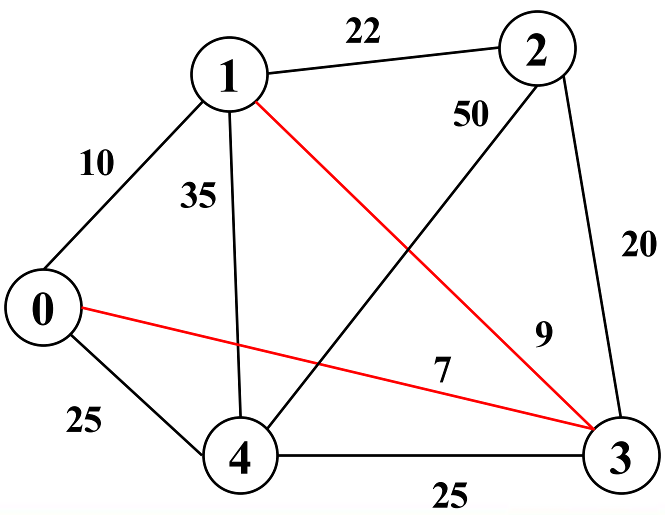
Dist	0	9	20	0	25
Prev	None	3	3	0	0



Prim算法：示例

- 用 Dist列表表示结点到 V’ 的“距离”，用 Prev 列表表示结点到 V’ 中的哪一个结点距离最近。
  - 选取结点 1 加入 V’，边（3，1）加入E’

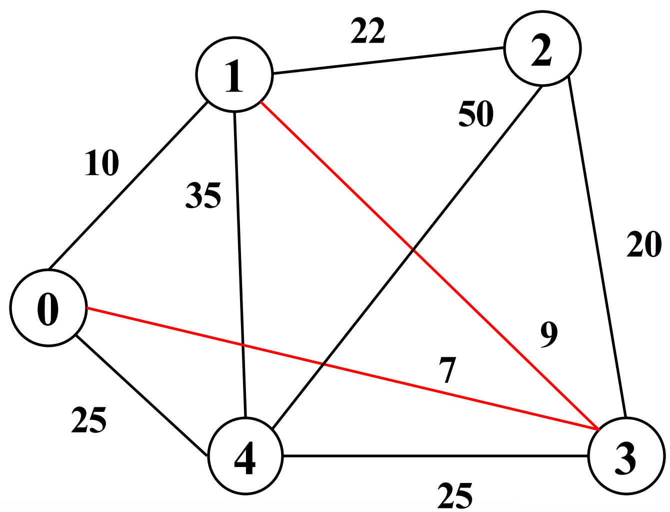
Dist	0	0	20	0	25
Prev	None	3	3	0	0



Prim算法：示例

- 用 Dist 表示结点到 V’ 的“距离”，用 Prev 表示该结点到 V’ 中的哪一个结点距离最近。
  – 依据结点 1 关联的边，更新信息（没有更新发生）

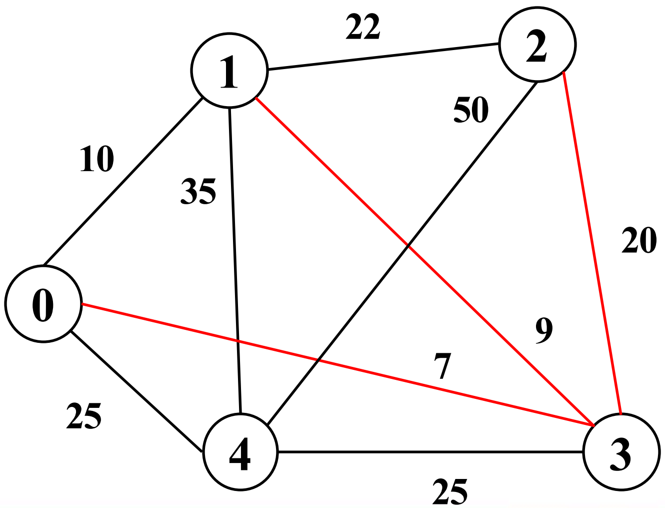
Dist	0	0	20	0	25
Prev	None	3	3	0	0



Prim算法：示例

- 用 Dist 列表表示结点到 V’ 的“距离”，用 Prev 列表表示结点到 V’ 中的哪一个结点距离最近。
  - 选取结点 2 加入 V’，边（3，2）加入E’

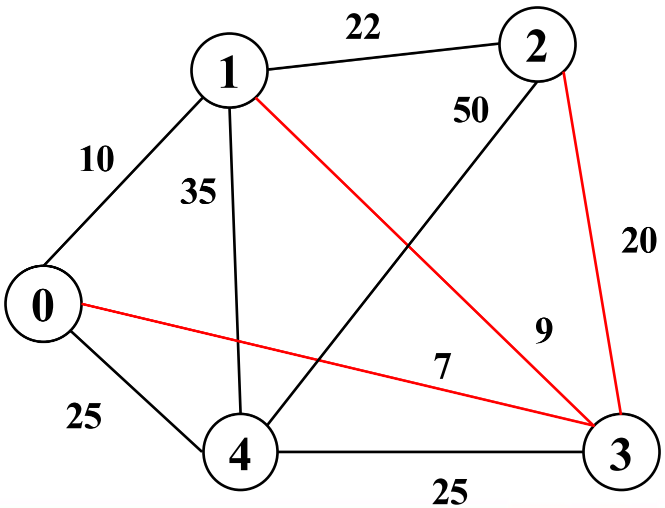
Dist	0	0	0	0	25
Prev	None	3	3	0	0



Prim算法：示例

- 用 Dist 列表表示结点到 V’ 的“距离”，用 Prev 列表表示结点到 V’ 中的哪一个结点距离最近。
  - 依据结点 2 关联的边，更新信息（没有更新发生）

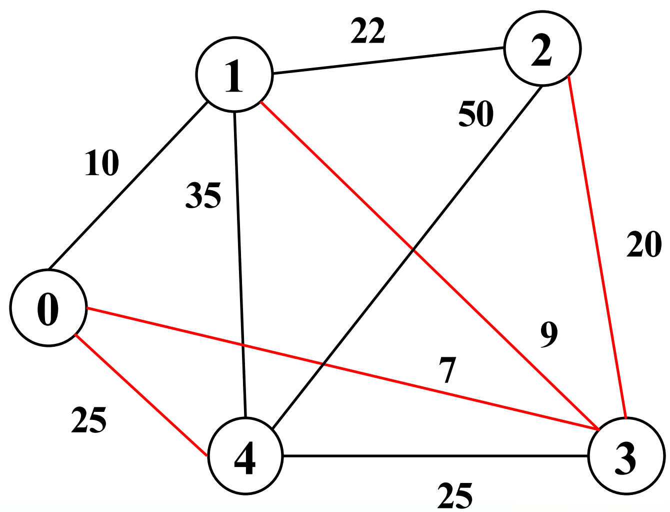
Dist	0	0	0	0	25
Prev	None	3	3	0	0



Prim算法：示例

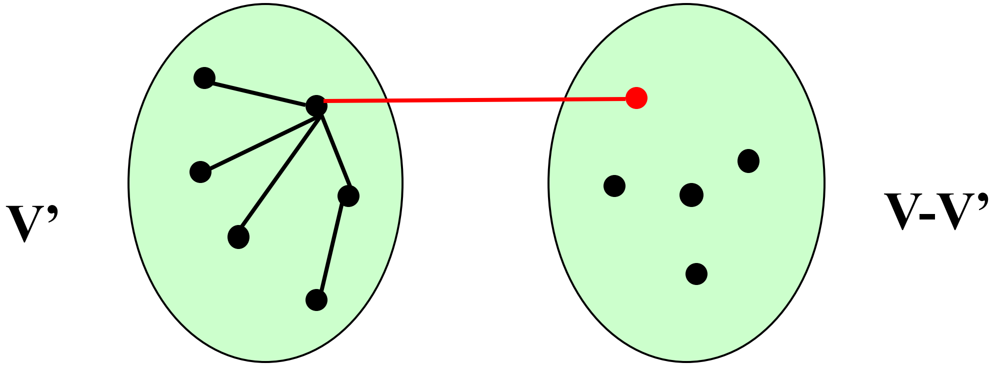
- 用 Dist列表表示结点到 V’ 的“距离”，用 Prev 列表表示结点到 V’ 中的哪一个结点距离最近。
  - 选取结点 4 加入 V’，边（0，4）加入E’，T构建完成

Dist	0	0	0	0	0
Prev	None	3	3	0	0



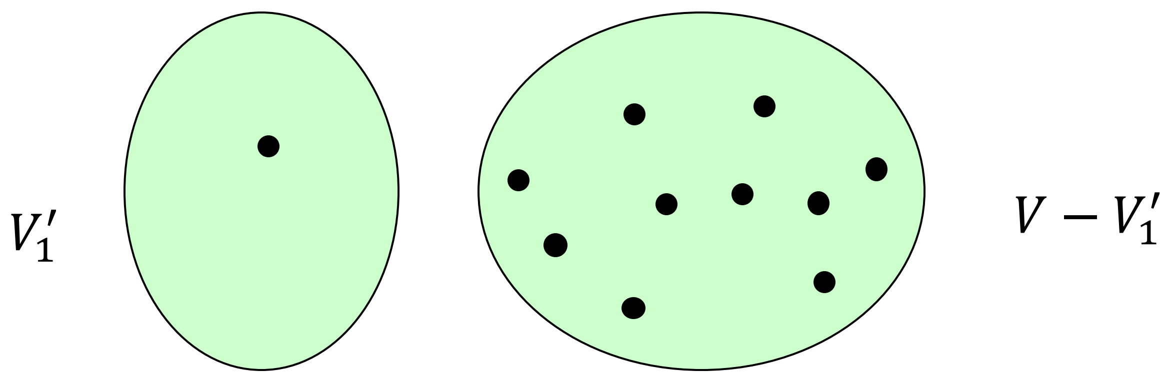
Prim算法的正确性

- 归纳证明：每次向 V’ 中加入结点，并将对应的边加入 E’ 后，总是存在一棵最小生成树 T，使得 G’=(V’, E’) 为 T 的子图。从而最终构建的 G’ 是一棵最小生成树



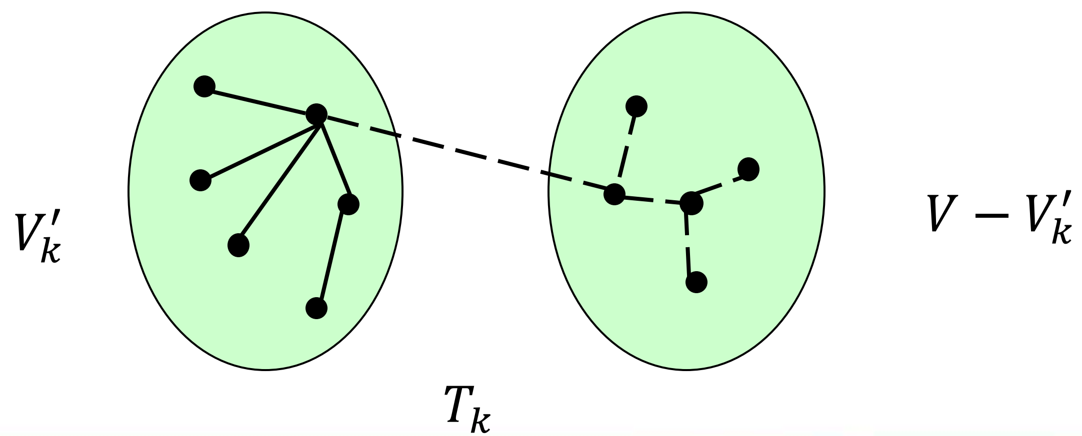
Prim算法的正确性

- 归纳证明：每次向 V’ 中加入结点，并将对应的边加入 E’ 后，总是存在一棵最小生成树 T，使得 G’=(V’, E’) 为 T 的子图。从而最终构建的 G’ 是一棵最小生成树
  - 归纳基础：构建开始时，$V_1' = \{v\}, E_1' = \{\}$ 显然成立
  - 由于 E_1' 为空，G_1' 一定是任何最小生成树的子图



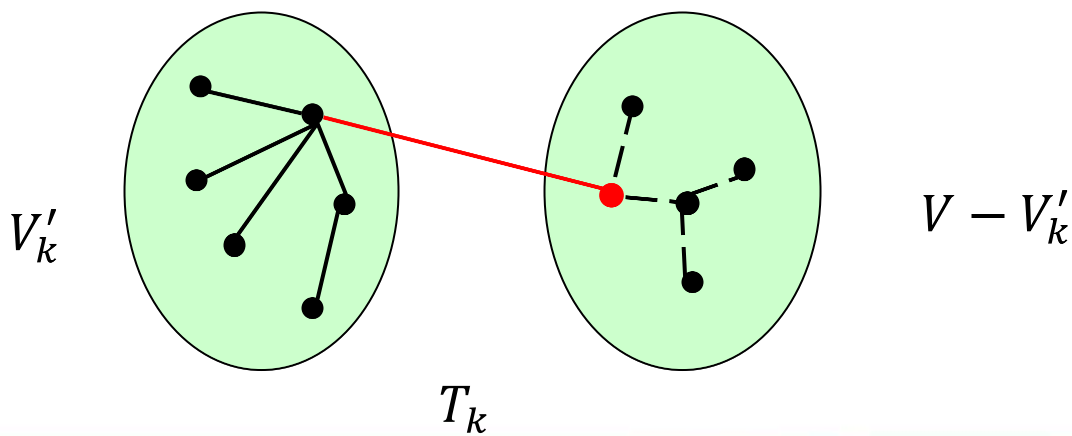
Prim算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G_k' = (V_k', E_k')$ 是最小生成树 T_k 的子图



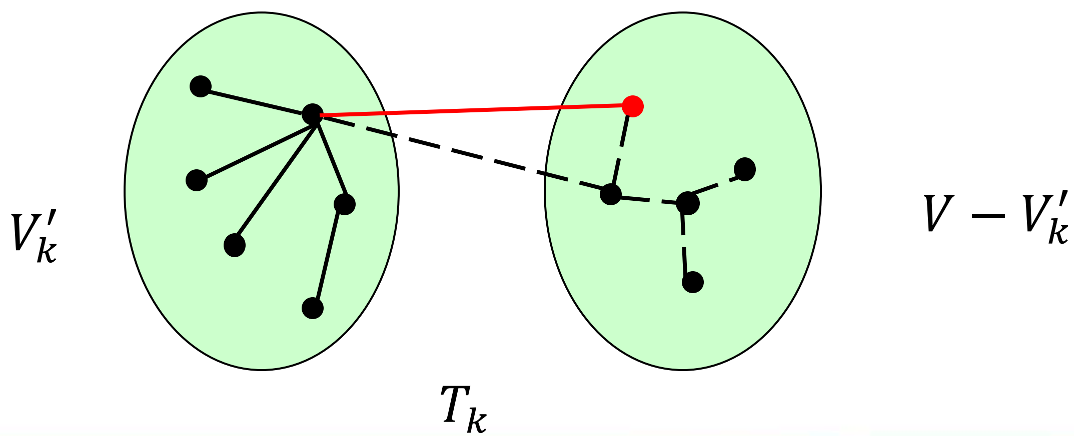
Prim算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G'_k = (V'_k, E'_k)$ 是最小生成树 T_k 的子图
  - 加入第 k+1 个结点 v_{k+1} 以及边 e_{k+1}后，得到 $G'_{k+1} = (V'_{k+1}, E'_{k+1})$
  - 欲证明 G'_{k+1} 也是某棵最小生成树 T_{k+1} 的子图
  - 若 e_{k+1} 本身就在 T_k 中，则取 $T_{k+1} = T_k$ 即可



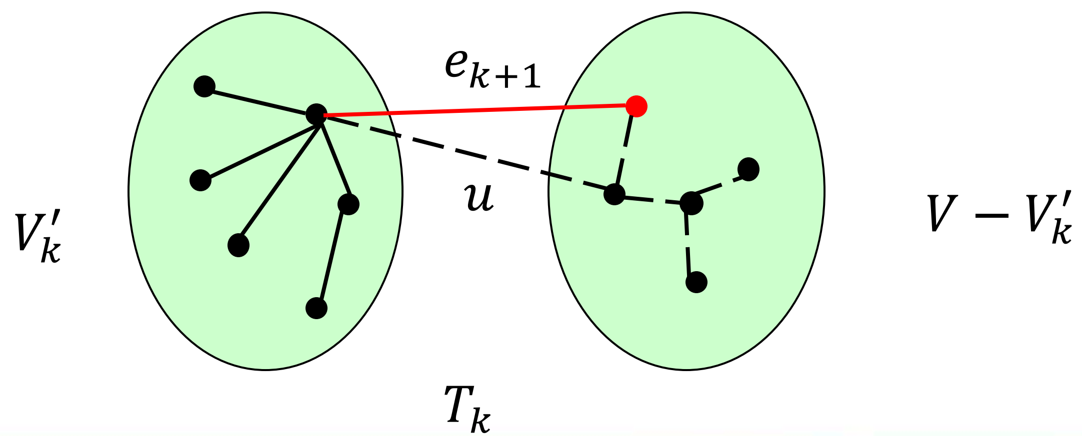
Prim算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G'_k = (V'_k, E'_k)$ 是最小生成树 T_k 的子图
  - 若 e_{k+1} 不在 T_k 中



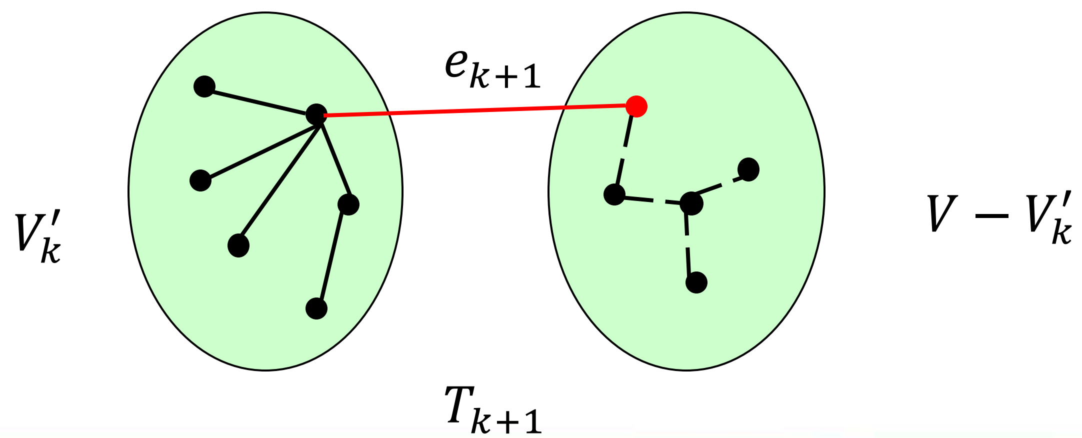
Prim算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G'_k = (V'_k, E'_k)$ 是最小生成树 T_k 的子图
  - 若 e_{k+1} 不在 T_k 中
  - 将 e_{k+1} 加入到 T_k 中，形成一个回路；考虑其中跨越 V'_k 与 $V - V'_k$ 的边，至少包括 e_{k+1} 与另一条边 u



Prim算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G_k' = (V_k', E_k')$ 是最小生成树 T_k 的子图
  - 由于 e_{k+1} 是跨越 V_k' 与 $V - V_k'$ 中权重最小的边，用 e_{k+1} 替换边 u，得到的 T_{k+1} 必然也是最小生成树
  - 实际上，可以推出 e_{k+1} 的权重与边 u 必定相等

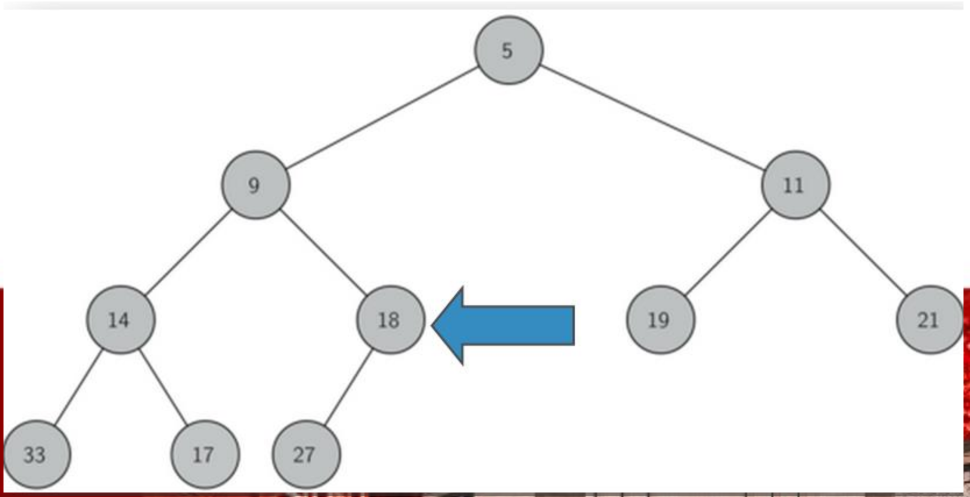


Prim算法：伪代码

```
# 输入一个带权连通无向图 G，输出 G 的一棵最小生成树
def Prim(G):
    任选一个顶点 v 作为构建 MST 的起点，初始化生成树边集 T = []
    依据 v 关联的边，初始化 Dist, Pred 列表，
    创建一个最小堆 Heap，插入所有结点，键值为 Dist 值
    while 堆非空：
        取出堆顶元素 u，将 (u, Pred[u]) 加入 T 中
        依据 u 关联的所有边，更新 Dist, Pred 列表
        对于发生更新的元素，相应调整堆的结构（DECREASE-KEY）
    返回 T
```

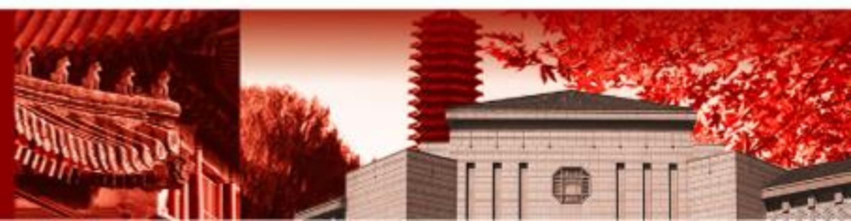
Prim算法：伪代码

- 上述伪代码中使用的最小堆这一数据结构，需要支持 DECREASE-KEY 操作
  - 即，将最小堆中某个元素的 key 减小，并重新调整堆，使其仍然满足堆的性质
  - 每次更新结点的 Dist 值，一定只会减小，不会增加
- 该操作可以在 O(log n) 的代价内完成
  - key 值减小之后，以该结点为根的子树仍然可以保持堆性质
  - 只需要将该结点上浮，不断与父结点比较；若小于父结点，则交换二者，直至根结点



Prim算法的时间复杂度

- 记 $n = |V|, e = |E|$
- 算法的主要代价在于，不断从堆中取出最小元素，以及调整堆的操作
  - 要进行 $O(n)$ 次取出最小元素操作，代价为 $O(n\log n)$
  - 每个结点加入 V’ 后都需要更新其全部邻居结点（DECREASE-KEY）
  - 最坏情况下，要进行 $O(e)$ 次更新，代价为 $O(e\log n)$
- 基于堆实现的 Prim 算法复杂度为 $O(n\log n + e\log n)$
  - 由于通常要求输入的图必须是连通图，即 $e \geq n - 1$
  - 因此，可以认为基于堆实现的 Prim 算法的时间复杂度为 $O(e\log n)$



Prim算法的时间复杂度

- 如果每次寻找最小值都使用复杂度为 O(n) 的简单遍历，则可以避免维护堆所产生的代价。
  - 对于 $n-1$ 个结点，需要进行寻找最小的操作
- 总的复杂度将变为 $O(n^2)$
  - 对于稠密图，即 e 达到 n^2 水平，n^2 的复杂度低于 $e\log n$，此时适合采用基于遍历的方法
  - 对于稀疏图，则 $e\log n$ 远小于 n^2，此时适合采用基于堆的方法

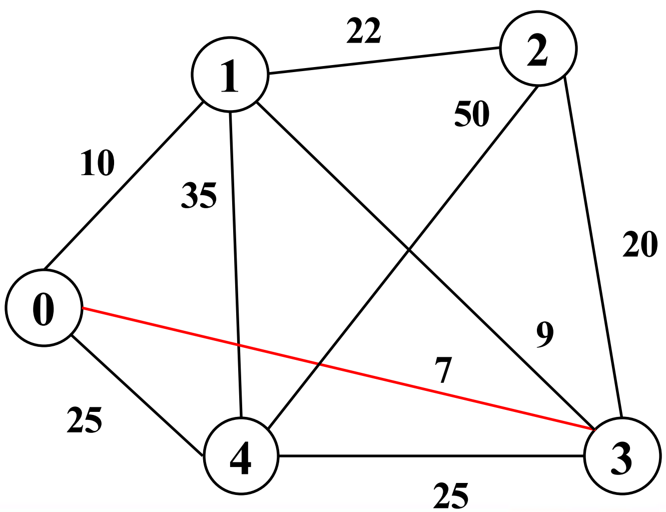
13.3 Kruskal算法

Kruskal算法

- Kruskal 算法同样是贪心算法，其基本思路如下
  - 为构建 G = (V, E) 的一棵最小生成树 T = (V’, E’)，将 V’ 初始化为 V，将 E’ 初始化为空集，即 G’ 初始时具有 n 个连通分量
  - 每次从边集 E 中取出权值最小的边 e 。如果 e 关联的两个顶点在 G’ 中属于不同的连通分量，就将 e 加入到 E’ 中
  - 不断重复上述步骤，直到图 G’ 成为连通图

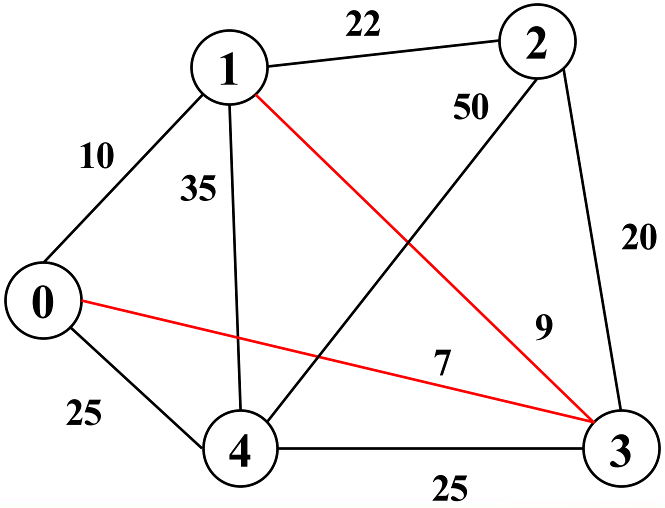
Kruskal算法：示例

- 初始时，5 个结点分别属于 5 个连通分量
- 当前权值最小的边：(0, 3)
- 顶点 0，3 属于不同的连通分量
- 将该边加入 E’ 中



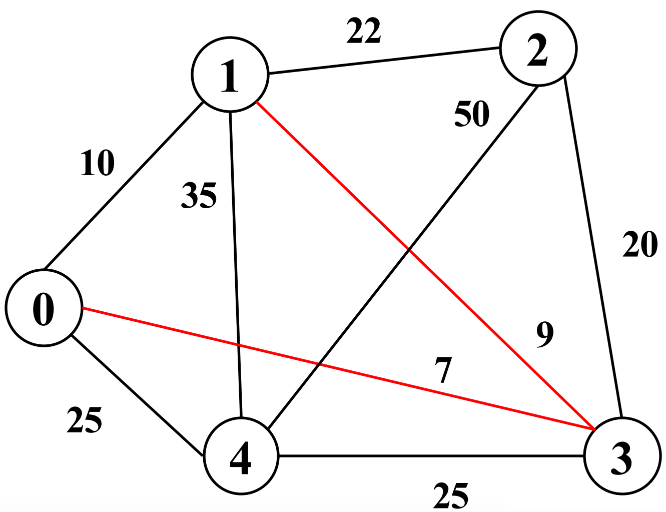
Kruskal算法：示例

- 当前权值最小的边：(1, 3)
- 顶点 1，3 属于不同的连通分量
- 将该边加入 E’ 中



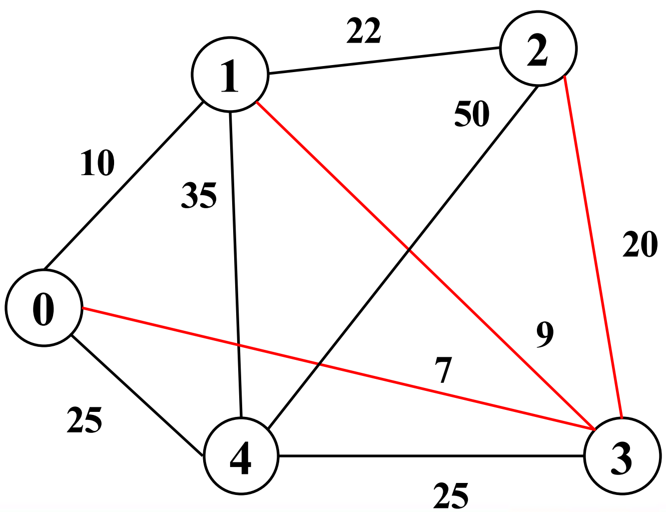
Kruskal算法：示例

- 当前权值最小的边：(0, 1)
- 顶点 0，1 属于相同的连通分量



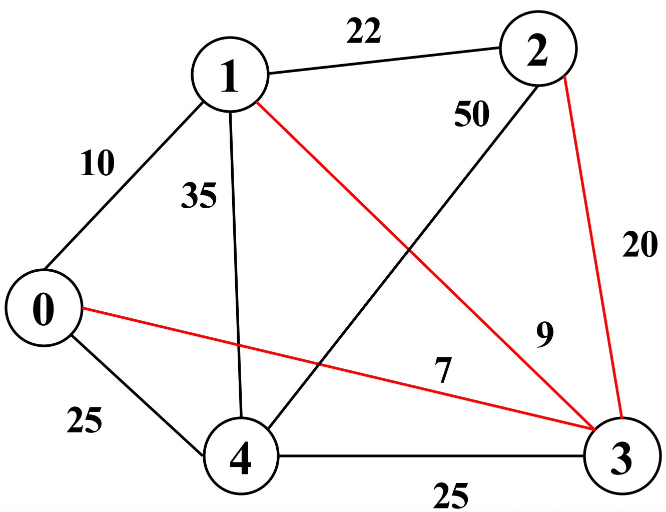
Kruskal算法：示例

- 当前权值最小的边：(2, 3)
- 顶点 2，3 属于不同的连通分量
- 将该边加入 E’ 中



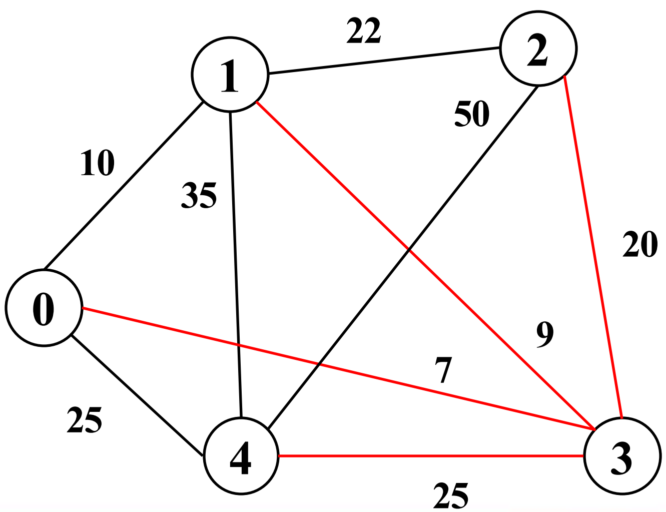
Kruskal算法：示例

- 当前权值最小的边：(1, 2)
- 顶点 1，2 属于相同的连通分量



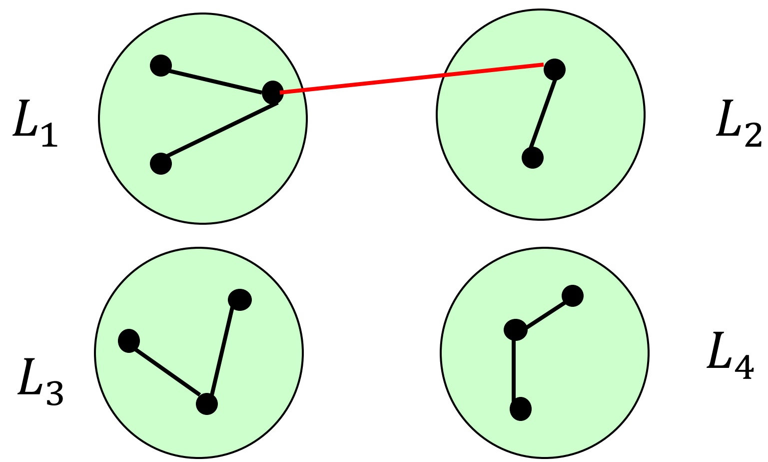
Kruskal算法：示例

- 当前权值最小的边：(3, 4)
- 顶点 3，4 属于不同的连通分量
- 将该边加入 E’ 中，构建完成
- 注：这一步也可能选择加入边 (0, 4)，即 MST 不是唯一的



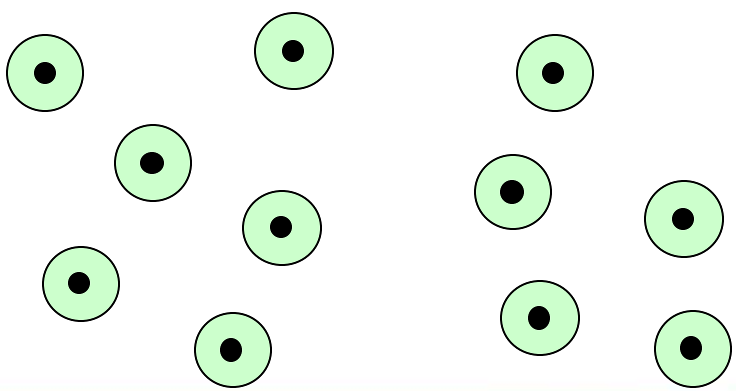
Kruskal算法的正确性

- 同样采取归纳证明：每次向 E’ 中加入边后，总是存在一棵最小生成树 T，使得 G’=(V, E’) 为 T 的子图。从而最终构建的 G’ 是一棵最小生成树



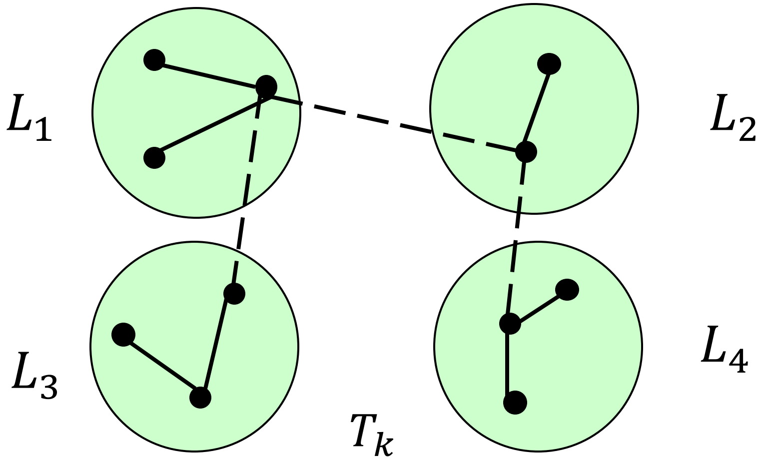
Kruskal算法的正确性

- 同样采取归纳证明：每次向 E’ 中加入边后，总是存在一棵最小生成树 T，使得 G’=(V, E’) 为 T 的子图。从而最终构建的 G’ 是一棵最小生成树
  - 归纳基础：构建开始时，每个顶点自成一个连通分量，
  - $V'_0 = V, E'_0 = \{\}$，显然成立



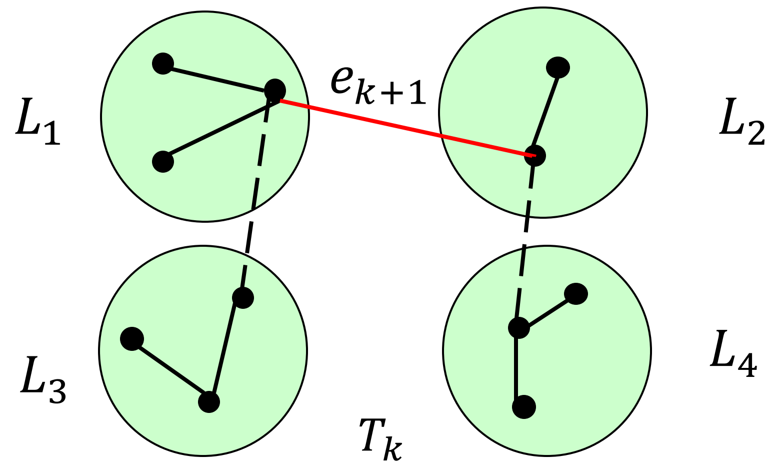
Kruskal算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G_k' = (V, E_k')$ 是最小生成树 T_k 的子图



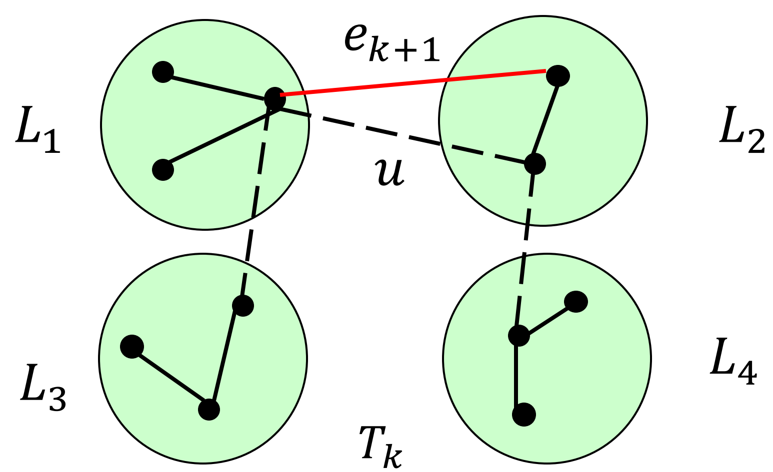
Kruskal算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G'_k = (V, E'_k)$ 是最小生成树 T_k 的子图
  - 加入第 k+1 个边 e_{k+1}后，得到 $G'_{k+1} = (V, E'_{k+1})$，其中 e_{k+1} 连通了两个连通分量 L_1, L_2
  - 欲证明 G'_{k+1} 也是某棵最小生成树 T_{k+1} 的子图
  - 若 e_{k+1} 本身就在 T_k 中，则取 $T_{k+1} = T_k$ 即可



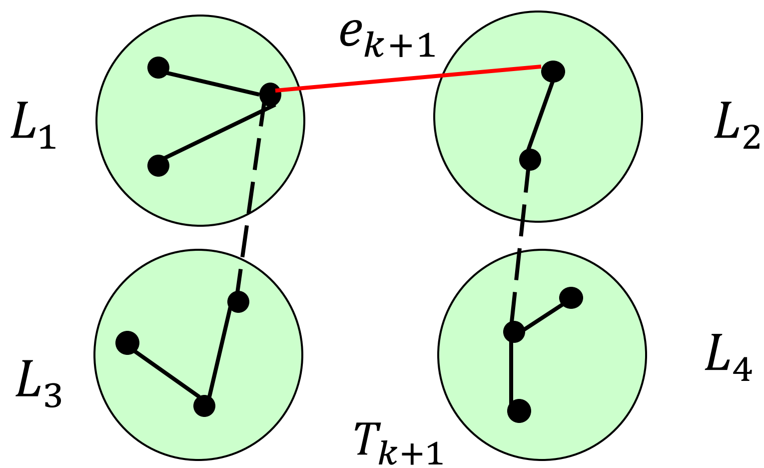
Kruskal算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G_k' = (V, E_k')$ 是最小生成树 T_k 的子图
  - 若 e_{k+1} 不在 T_k 中，将 e_{k+1} 加入到 T_k 中，形成一个回路；
  - 考虑回路中跨越 $V(L_1)$ 与 $V - V(L_1)$ 的边，至少包括 e_{k+1} 与另一条边 u



Kruskal算法的正确性

- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G_k' = (V, E_k')$ 是最小生成树 T_k 的子图
  - 由于 e_{k+1} 是跨越任意两个连通分量的边中权重最小的，用 e_{k+1} 替换边 u，得到的 T_{k+1} 也一定是最小生成树
  - 同样可以推出， e_{k+1} 的权重与边 u 必定相等



Kruskal算法：伪代码

```
# 输入一个带权连通无向图 G，输出 G 的一棵最小生成树T
def Kruskal(G):
    初始化生成树边集 T = []
    将 G 中的所有边按照权值升序排序
    创建并查集，G 中的每个结点自成一个集合
    按权值升序，遍历 G 中的每一条边 (u, v):
        如果 u, v 在并查集中属于不同集合（不同连通分量）:
            将 (u, v) 加入 T
            在并查集中将 u, v 合并
    返回 T
```

Kruskal算法的时间复杂度

- 记 n = |V|, e = |E|
- 初始时可以应用堆排序、快速排序和二路归并排序对图中所有的边进行排序，代价为 O(elog e)
- 初始化并查集的代价为 O(n)
- 并查集上的 Union 操作代价为 O(1)，Find 操作的均摊代价也可以认为是 O(1)
  - 执行 O(e) 次 Find 操作
  - 执行 O(n) 次 Union 操作
  - 代价为 O(e + n)
- 对于连通图，有 $n - 1 \le e \le n(n-1)/2$，则总代价即为 O(elog e)
  - 也可以写为等价的 O(elog n)，与基于堆实现的 Prim 算法相同

Kruskal算法的时间复杂度

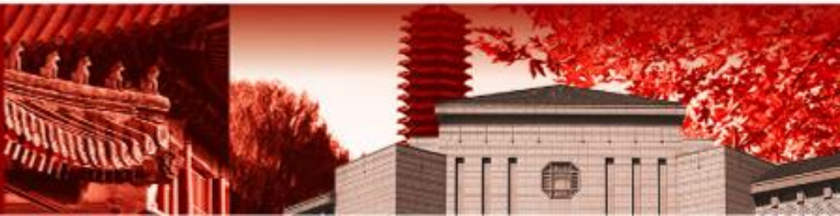
- Kruskal 算法的优化方法：
  - Kruskal 算法不断取出权重最短的边，若先对所有边进行排序，然后在依次遍历，复杂度为 O(elog e)
- 针对这一操作，也可以采用堆数据结构进行优化
  - 对所有边建堆，时间复杂度为 O(e)
  - 每次取出权值最小的边，复杂度为 O(log e)
  - 如果执行每条边都被取出一次，最坏情况下的复杂度同样为 O(elog e)
- 但实际构建过程中，只需要取权值较小的一部分边进行判断
  - 尤其是对于稠密图，实际执行次数将远小于 e

总结：Prim 算法 & Kruskal 算法

- 是求解最小生成树问题的两种贪心算法，通过不断选取边来构建生成树
  - Prim 算法：选取跨越 V’ 与 V – V’ 的权重最小的边
  - Kruskal 算法：选取跨域两个连通分量的权重最小的边
- 要求输入为连通的带权无向图
- 时间复杂度
  - Prim 算法（堆实现）：O(elog n)，适用于稀疏图
  - Prim 算法（遍历实现）：$O(n^2)$，适用于稠密图
  - Kruskal 算法（并查集实现）：O(elog n)，适用于稀疏图

课堂练习

- 口袋的天空：小杉坐在教室里，透过口袋一样的窗户看口袋一样的天空。有很多云飘在那里，看起来很漂亮，小杉想摘下那样美的几朵云，做成棉花糖。
- 问题：给你云朵的个数 N，再给你 M 个关系，表示哪些云朵可以连在一起。现在小杉要把所有云朵连成 K 个棉花糖，一个棉花糖最少要用掉一朵云，小杉想知道他怎么连，花费的代价最小。如果无法做到，算法应该识别
- 建模为图论的语言：给定无向图 G，包含 N 个顶点与 M 条边。求图 G 的一个最小权重和的子图，使得其恰好包含 K 个连通分量。
  - 如果 K = 1，就是求 G 的最小生成树

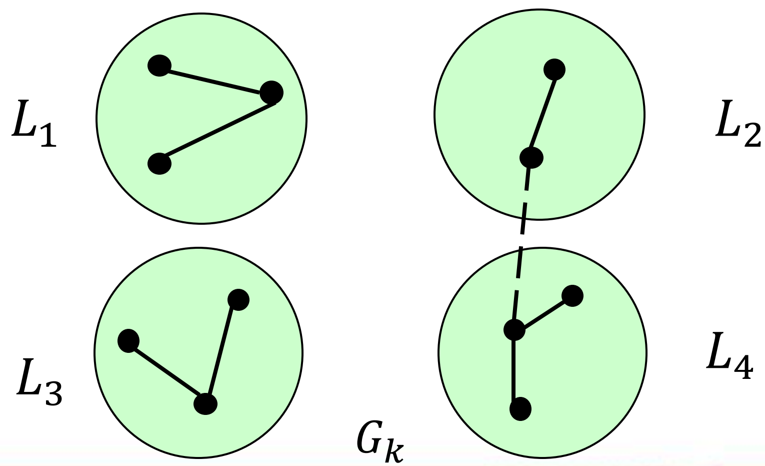


解答

- 该问题与最小生成树问题非常相似
  - 最小生成树问题要求构建一个连通分量
  - 该问题要求构建 K 个连通分量
- 一个自然的猜测：仍然在原图上运行 Kruskal 算法，但是少循环 K–1次，这样得到的连通分量数量就多了 K-1
- 实际上，这种贪心方法是正确的：
  - 即，每次都选取从连通两个连通分量的边中选择边权最小的，直到剩下 K 个连通分量
  - 如果算法运行过程中找不到连通两个连通分量的边了，就表明无法构建
- 如何仿照 Kruskal 算法的正确性证明，来说明这种贪心策略的正确性？

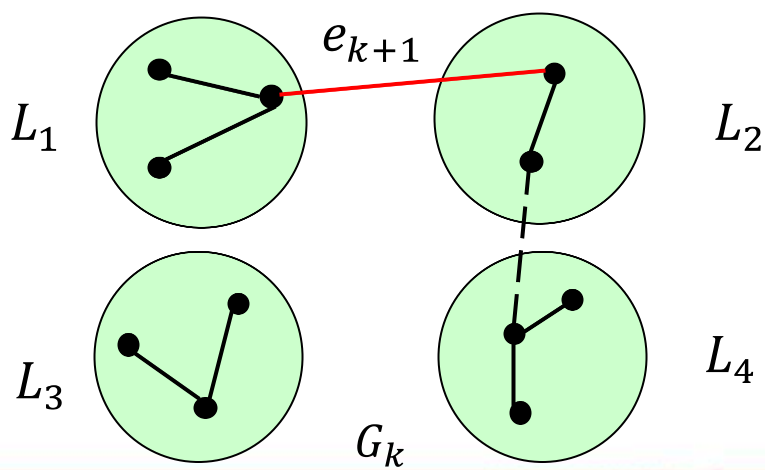
解答

- 归纳证明：每次选取边后，总是存在最优解 G，使得当前构建的图 G’ 是图 G 的子图。
- 同样，归纳基础是显然的。
  - 选取边之前的图不包含边，一定是 G 的子图
- 归纳假设：假设加入第 k 个结点后成立该性质，即 $G_k' = (V, E_k')$ 是某个最优解 G_k 的子图



解答

- 加入第 k+1 个边 e_{k+1}后，得到 $G'_{k+1}=(V,E'_{k+1})$，其中 e_{k+1} 连通了两个连通分量 L_1,L_2
- 若 e_{k+1} 本身就在 G_k 中，证明结束；因此不妨假设不在 G_k 中
- 将 e_{k+1} 加入到 G_k 中，此时，要考虑在 G_k 中 L_1,L_2 是否连通
  - 若连通，则 e_{k+1} 的加入导致回路的产生，可以用类似的方式替换
  - 若不连通，将导致 G_k 的连通分量数减一，为了还原，应该删去任意一条边
  - 而 e_{k+1} 是当前未添加的边中的边权是最小的，替换其他当前未添加的边，不会使边权增加

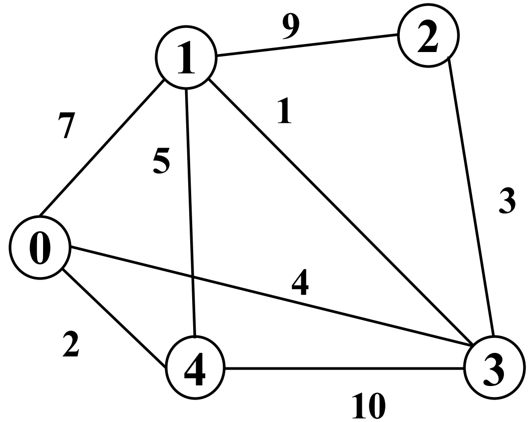


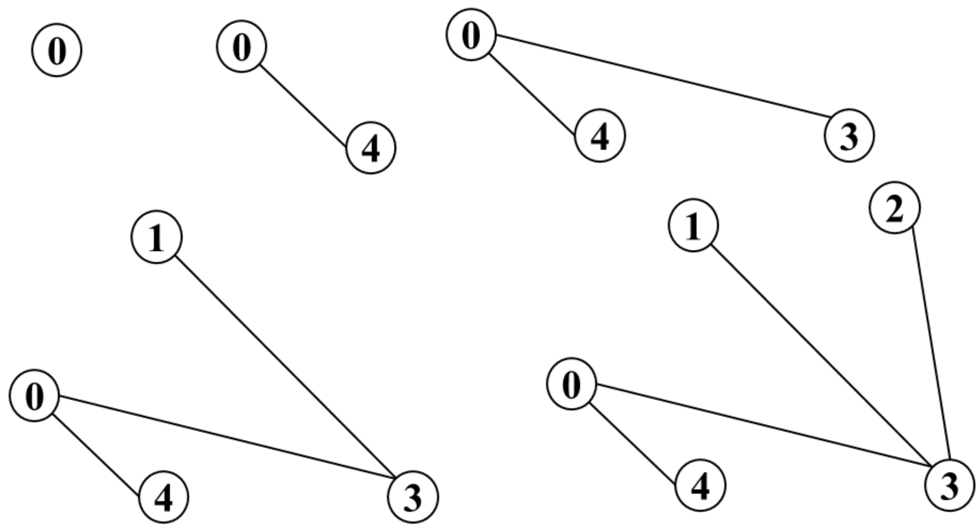
Quiz

- 已知图 G 的顶点集合 V={V_0, V_1, V_2, V_3, V_4}，邻接矩阵如下图所示
  - (1) 画出图 G
  - (2) 以 V_0 起始作为顶点，运行 Prim 算法，画出每一步的构建过程
  - (3) 运行 Kruskal 算法，画出每一步的构建过程

$$\begin{bmatrix} 0 & 7 & \infty & 4 & 2 \\ 7 & 0 & 9 & 1 & 5 \\ \infty & 9 & 0 & 3 & \infty \\ 4 & 1 & 3 & 0 & 10 \\ 2 & 5 & \infty & 10 & 0 \end{bmatrix}$$

Prim 算法过程





Kruskal 算法过程