07 - Minimum Spanning Tree

最小生成树（Minimun Spanning Tree）

最小生成树问题

T 是一棵生成树，这个问题被称为**最小生成树问题**：

- 考虑一个连通图 $G = (V, E)$，对于每条边 $e = (v, w) \in E$，都有一个相关的正成本 c_e
- 我们的目标是：找到 E 的一个子集 T，使得图 $G' = (V, T)$ 是连通的，并且总成本 $\sum_{e \in T} c_e$ 最小化

值得注意的是

T 是一棵树
根据定义，(V, T) 是连通的
我们假设它包含一个环（这样就不是树了），e 是这个环上的一条边
考虑 $(V, T - \{e\})$，其仍然是联通的：使用 e 的所有路径都可以通过其他方向重新路由
因此 $(V, T - \{e\})$ 是一个有效的解决方案，并且总成本更小，矛盾

这就是说，在一个最小生成树中不可能存在环

切分定理(Cut Property)

假设所有边的成本都是不同的

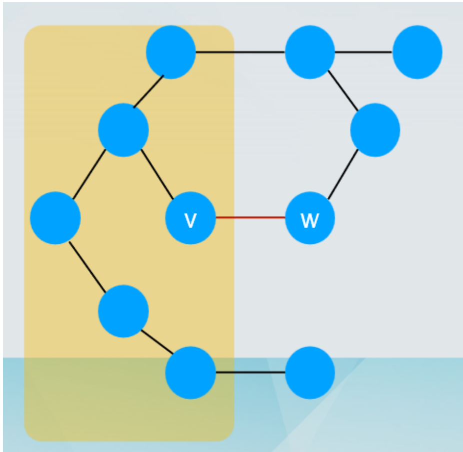
- 令 S 是 V 的任意**非空真子集**
- 令 $e = (w, v)$ 是 S 和 $V - S$ 之间成本最小的边

那么 e 包含在每一颗最小生成树中

证明

假设某棵生成树 T 不包含 e
由于它是一棵生成树，它一定包含一些其他的跨越 S 到 $V - S$ 的边 f
但是 $c_e \le c_f$，所以 $T - \{f\} \cup \{e\}$ 是一颗成本更小的生成树

这对吗？这不对
因为 $T - \{f\} \cup \{e\}$ 可能不是一个生成树



在此图中，$e=(v,w)$，f 是 e 下面那条。我们先假装看不见那条红线
将 f 替换为 e 之后，出现了环

我们应该选择这样的一条边 e'

- 比 e 更贵
- 如果用 e' 代替 e，仍然会生成一颗生成树

因此
令 T 是一颗不包含 $e=(v,w)$ 的最小生成树
由于 T 是一颗生成树，从 v 到 w 有一条路径
令 w' 是 $V-S$ 中遇到的第一个节点，令 v' 是之前的一个节点
令 $e'=(v',w')$
考虑 $T'=T-\{e'\}\cup\{e\}$

环的性质

假设所有边的成本都是不同的

- 令 C 是图 G 的任意一个环
- 令 $e=(w,v)$ 是环 C 中成本最大的边

那么 e 不包含在图 G 的任何最小生成树中

证明

设 T 是包含边 e 的一棵生成树，我们将证明它不是最小成本的
我们将边 e 替换为另一条边 e'，得到一个成本更低的生成树。如何找到这个 e'？

- 我们从 T 中删除边 e
- 这会将节点划分为两个部分：
  - S（包含 u）
  - $V-S$（包含 w）
- 我们沿着环中的另一条路径从 u 到 w
  - 沿着边 e'，在某个点我们会从 S 交叉到 $V-S$
- 结果图是一棵成本更低的树

最小生成树算法

Kruskal 算法

1. 输入：图的边集合 E，其中每个元素是 (u,v,w)，表示在 u 和 v 之间有一条权重为 w 的边
2. 输出：输入图的最小生成树的边集合
3. 方法：
   1. $\text{result} \leftarrow \emptyset$
   2. 将 E 按权重 w 的非递减顺序排序
   3. 对于排序后的 E 中的每个 (u,v,w)
      1. 如果 u 和 v 在并查集中不连通
         1. 在并查集中连接 u 和 v
         2. $\text{result} \leftarrow \text{result} \cup (u,v,w)$

4. 返回 result

其中，并查集（Union-Find）是一种数据结构，用于处理一些不交集（Disjoint Set）的合并及查询问题，它支持：

- 查找（Find）：确定元素属于哪个子集。它可以被用来确定两个元素是否属于同一个子集
- 合并（Union）：将两个子集合并成一个集合

（其实看下面这个就行了，我嫌弃 ppt 上面的不正规）

- 从一个空的边集合 T 开始。
- 向 T 中添加一条边
  - 添加哪一条？
  - 添加成本 c_e 最小的那一条边
- 我们继续这样做
- 我们是否总是将新的边 e 添加到 T 中？
  - 只有在不引入任何环的情况下才添加

Kruskal 算法是最优的

- 考虑 Kruskal 算法在某一步添加到输出中的任意边 $e = (u, w)$
- 设 S 为在添加边 e 之前从 u 可达的节点集合
- 由此可得，u 在 S 中，w 在 $V - S$ 中
  - 因为否则添加边 e 会形成一个环
- 算法还没有找到任何跨越 S 和 $V - S$ 的边
  - 这样的边已经被算法添加到输出中
- 边 e 必须是跨越 S 和 $V - S$ 的最便宜的边
- 根据割边性质，它属于每一棵最小生成树

Kruskal 算法是可行的

它是否总是生成一棵生成树？

该算法明确避免了环（Kruskal 算法在选择边时，始终选择不会形成环的边。这是通过并查集结构来实现的，它可以高效地检测和避免环的形成）

- 输出 T 是一片森林

它是一棵树吗？

- 它是连通的吗？

- G 是连通的
- 假设 T 不是连通的
  - 算法会添加一条跨越两个组件的边，矛盾，因此 T 是连通的

Prim 算法

1. **输入**：图的节点集合 V；函数 $g(u, v)$，表示边 (u, v) 的权重；函数 $\mathrm{adj}(v)$，表示与节点 v 相邻的节点
2. **输出**：输入图的最小生成树的权重总和

3. 方法:
    1. $\text{result } \leftarrow 0$
    2. 从 V 中选择一个任意节点作为 root
    3. $\text{dis}(\text{root}) \leftarrow 0$
    4. 对于每个节点 $v \in (V - \{\text{root}\})$
        1. $\text{dis}(\text{root}) \leftarrow \infty$
    5. $\text{rest} \leftarrow V$
    6. 当 $\text{rest} \neq \emptyset$
        1. $\text{cur} \leftarrow \text{rest}$ 中具有最小 dis 的节点
        2. $\text{result} \leftarrow \text{result} + \text{dis}(\text{cur})$
        3. $\text{rest} \leftarrow \text{rest} - \{\text{cur}\}$
        4. 对于每个节点 $v \in \text{adj}(\text{cur})$
            1. $\text{dis}(v) \leftarrow \min\left(\text{dis}(v), g(\text{cur}, v)\right)$
    7. 返回 result

(还是一样，看下面的就行)

- 从一个空的边集合 T 开始
- 从一个节点 s 开始
    - 将边 $e = (s, w)$ 添加到 T 中
    - 添加哪一条边?
    - 添加成本 c_e 最小的那一条边
- 我们继续这样做
    - 我们只考虑与生成树中不在树中的邻居相连的边

Prim算法是最优的

在 Prim 算法的每次迭代中，算法会从部分生成树的节点集中选择一条边来扩展生成树。选择的这条边具有最小的权重，并且连接了部分生成树中的节点和部分生成树外的一个节点

根据切分定理，对于任何跨越割集（一个部分生成树节点集和剩余节点集）的最小成本边，该边必须包含在最小生成树中。因为在每次迭代中，Prim算法选择的边正是这种最小成本边，因此这些边必须包含在最终的最小生成树中

Prim 算法的时间复杂度

朴素情况：对于每一步，我们遍历所有候选节点
因此，复杂度为 $O(n^2)$

我们可以用 Priority Queue 来进行优化

Prim in Priority Queue

维护:

- 一个元素集合 S
- 每个元素 v 在 S 中有一个键 $key(v)$，这个键表示 v 的优先级

操作:

- $Add(v)$：将元素 v 及其优先级键添加到优先队列中
- $Delete(v)$：从优先队列中删除元素 v
- $Extract\_Min(v)$：提取并删除优先级最小的元素
- $Change\_key(v)$：修改元素 v 的优先级键

我们可以用堆来实现有限队列对Prim算法的优化：

- 我们将节点添加到扩展的生成树 S 中。
- 我们需要找出下一个要添加的节点。
- 我们需要知道每个节点的附加成本：

$$a(v) = \min_{e=(u,v):u \text{ 在 } S} c_e$$

- PQ解法：将节点插入优先队列（PQ），以负的附加成本作为键。
  - 运行 $Extract\_Min(v)$ 来找到下一个节点。
    - 运行 $n-1$ 次。
  - 运行 $Change\_key(v)$ 来更新附加成本。
    - 每条边最多运行一次。
- 运行时间：$O(m\log n)$。

运行时间分析

- $Extract\_Min(v)$ **操作**：
  - 这是从优先队列中提取优先级最小的元素，在整个算法中需要运行 $n-1$ 次，每次操作的时间复杂度是 $O(\log n)$。
- $Change\_key(v)$ **操作**：
  - 这是更新优先队列中某个元素的优先级，每条边最多进行一次更新操作，总共进行 m 次，每次操作的时间复杂度是 $O(\log n)$。

综合以上操作，Prim算法的总体运行时间是 $O(m\log n)$，其中 m 是图中的边数，n 是节点数。这使得Prim算法在处理大型图时依然保持高效。

反向删除算法（Reverse-Delete Algorithm）

- 从完整的图 $G=(V,E)$ 开始
- 从图中删除一条边
  - 删除哪一条边?
  - 删除成本 c_e 最大的那一条边
- 我们继续这样做
- 我们是否总是删除考虑的边 e
  - 只要不使图断开连接

反向删除算法是最优的

- 考虑任何一条由反向删除算法移除的边 $e=(v,w)$

- 在删除之前，它位于某个环 C 上
- 它在环上的所有边中具有最大的成本，因此它不能是任何最小生成树的一部分

反向删除算法是可行的

它是否总是生成一棵生成树?

它是连通的吗?

- 该算法不会断开图的连通性

它是一棵树吗?

- 假设它不是
- 那么它包含某个环 C
- 考虑环上成本最大的边 e
- 该算法会删除那条边

边权重不唯一（Non-distinct costs）

之前我们仅仅讨论了边权重唯一的情况，对于边权重唯一的情况：

- 通过在成本上添加小数 ϵ 来打破平局，使成本唯一化
- 获得一个扰动实例
- 在扰动实例上运行算法
- 输出最小生成树 T
- T 是原始实例上的一棵最小生成树

正确性

- 假设在原始实例上存在一棵成本更低的生成树 T^*
- 如果 T^* 包含具有相同成本的不同边，则它在原始实例上不比 T 更便宜
- 如果 T 包含具有不同成本的不同边，我们可以使 ϵ 足够小，以确保我们选择的边仍然是更便宜的