arXiv:2305.03240v1 [cs.DS] 5 May 2023

基于局部效应求和的可分图数据结构*

Xing Lyu[1], Travis Gagie[2][0000−0003−3689−327X]、Meng He[2][0000−0003−0358−7102]、YakovNekrich[3][0000−0003−3771−5088]和 Norbert Zeh[2][0000−0002−−0562−1629]

[1] 加拿大哈利法克斯的 Halifax West High School
lyuxing1006@gmail.com[2] 加拿大哈利法克斯的达尔豪斯大学
{firstname.lastname}@dal.ca[3] 密歇根理工大学 yakov@mtu.edu

摘要。 很容易想到一些应用场景：这些场景可以建模为图问题，其中，在某个顶点放置设施或商品，会对一定距离范围内所有顶点的值产生正面或负面影响；我们希望能够快速计算任意时刻任一顶点值所受的累积影响，或找出对该顶点影响最有利或最不利的因素列表。本文展示了，对于具有常数规模分隔集的边加权图，我们如何支持以下操作，使其运行时间关于顶点数量和已放置在顶点上的设施数量呈多对数级；顶点间的距离根据边权重计算：

添加(v, f, w, d) 将权重为 w、影响半径为 d 的设施放置在顶点 v上。

删除(v, f) 从 v 中删除先前使用 Add 放置在 v 上的设施 f。

求和(v) 或 求和(v, d) 返回影响v 的所有设施的总权重；或者，给定距离参数 d，返回其影响区域与以 v 为中心、半径为 d 的“圆”相交的所有设施的总权重。

Top(v, k) 或 Top(v, k, d) 返回影响 v 的权重最高的 k 个设施；或者，给定距离参数 d，返回其影响区域与以 v 为中心、半径为 d 的“圆”相交的权重最高的 k 个设施。
设施的权重以及 Sum 用于对这些权重进行“求和”的运算必须构成一个半群。对于 Top 查询，权重必须取自一个全序集。

关键词： 图数据结构 · 树宽 · 分支宽度 · 图分解 · 树分解 · 局部效应之和

1 引言

即使从未听说过 Baron Samuel of Wych Cross，人们也可能听过一句通常被认为出自他之口的名言：有三件事至关重要，那就是.....

*由加拿大自然科学与工程研究理事会探索资助项目 RGPIN-07185-2020 和 RGPIN-05435-2018 资助。

房地产：位置，位置，还是位置。这意味着，房产的价值可能会因其是否靠近公交车站、优质学校、超市或垃圾填埋场等地点而上涨或下跌。当然，对于公交车站而言，“近”与对于垃圾填埋场而言的“近”可能并不意味着同一回事，而且前者带来的积极影响可能无法抵消后者带来的负面影响。事实上，“近”可能并非指欧几里得距离，因为沿街步行五分钟即可到达的公交车站，比穿过垃圾填埋场步行五分钟才能到达的公交车站更值得选择。为了对具有此类加性局部效应且局部性的定义采用非欧几里得方式的应用进行建模，本文提出了一种适用于图 G 的数据结构，支持以下操作：

添加(v, f, w, d) 将权重为 w 的设施放置在顶点 v 上。 f 的 影响区域 是一个以 v 为圆心、半径为 d 的圆。
删除(v, f) 会从 v 中删除先前使用 Add 放置在 v 上的设施 f。**求和(v) 或 求和(v, d)** 返回影响 v 的所有设施的总权重，或者在给定距离参数 d 时，返回影响区域与以 v 为圆心、半径为 d 的“圆”相交的所有设施的总权重。

Top(v, k) 或 Top(v, k, d) 返回权重最大的 k 设施，这些设施会影响 v ，其距离参数为 d，且其影响区域与半径为 d 、以 v为中心的“圆”相交。

我们假设每条边 $e \in G$ 都有非负长度 $\ell(e)$，并且顶点之间的距离是两顶点之间任意路径上所有边的总长度的最小值。以某个顶点 v 为圆心、半径为 d 的圆，包含距离 d 不超过 v的所有顶点和边（或边的一部分）。更确切地说，如果将影响半径为 d' 的设施 f 放置在某个顶点 u上，那么当且仅当 dist$(u, v) \leq d + d'$时，我们认为 f的影响区域与以另一个顶点 v 为圆心、半径为 d 的圆相交。

设施的权重以及 Sum 用于对其进行“求和”的运算必须构成一个半群（参见附录 A）。对于 Top 查询，权重必须取自一个全序。请注意，Sum(v) 和 Top(v, k) 可以视为“图上的范围刺探查询”，而 Sum(v, d) 和 Top(v, k, d) with $d > 0$ 则是“图上的范围相交查询”；其中范围是设施的影响区域，查询对象可以是单个顶点，也可以是某个顶点周围半径为 d 的区域。

我们将这种数据结构称为局部效应求和（SOLE）数据结构。在第 2 节中，我们证明，当 G 是一个含有 n 个顶点的树时，存在一种 SOLE 数据结构，支持 Add、Remove 和 Sum 操作，耗时为 $O(\lg n \lg m)$，并支持 Top 查询，耗时为 $O(k \lg n \lg m)$，其中 m 是当前放置在 G 各顶点上的设施总数。在第 3 节中，我们将这一结果推广到任意常数 t 的 t-可分图，其中包括串并联图（$t \leq 2$）、树宽为常数 w 的图（$t \leq w + 1$）以及分支宽度为常数 b 的图（$t \leq b$）。我们证明，当 G 是 t-可分图时，存在一种 SOLE 数据结构，支持 Add、Remove 和 Sum 操作，耗时为$O(\lg n \lg^t m)$，并支持 Top 查询，耗时为 $O(k \lg n \lg^t m)$。在这种情况下，Add 和 Remove 操作的开销为摊销开销。

我们相信，所有结果都可以推广到有向图 G，并且我们的数据结构可以支持在 G 中删除顶点和边。我们将在本文未来的版本中研究这一推广。

2 树的 SOLE 数据结构

在本节中，我们证明

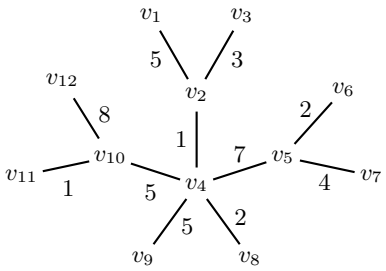
定理 1. 如果 G 是一棵含有 n 个顶点的树，那么存在一种 *SOLE* 数据结构，支持 添加、删除和求和 操作，耗时为 $O(\lg n \lg m)$，并支持$\textsc{Top}(v,k,d)$ 操作，耗时为 $O(k \lg n \lg m)$，其中 m 是当前位于 G 顶点上的设施数量。该数据结构的大小为 $O(n + m \lg n)$。

暂且假设 G 的所有顶点的度数至多为 3。我们任意选择 G 的一个顶点 ρ，并用它到 ρ 的距离 dist(ρ, v) 为 G 中的每个顶点 v 标记。G 的质心边是指一条边 (u,v)，删除该边后会将 G 分成两个子树 G_u 和 G_v，每个子树至多包含 $2n/3$ 个顶点。由于 G 的所有顶点的度数至多为 3，这样的边一定存在。G 的质心分解是一棵二叉树 T，按如下方式归纳定义（见图1c）：如果 G 只有一个顶点 v，那么 T 仅有一个节点 v。否则，令 (u,v) 为 G 的任意一条质心边。那么 T 的根为 (u,v)，而 (u,v) 的两个子节点分别是 G_u 和 G_v 的质心分解的根。对于 T 的每条边 e，令 T_e 为 T 中位于 e 下方的子树，并令 V_e 为 G 中对应于 T_e 的叶节点的顶点集合。T 的高度至多为 $\log_{3/2} n = O(\lg n)$。

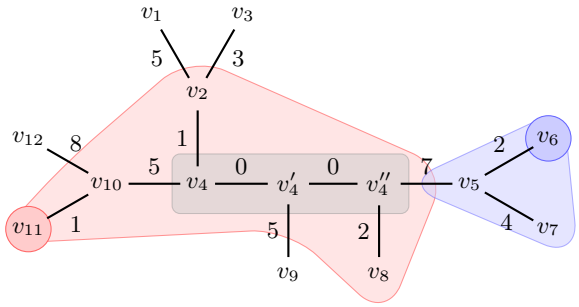
我们的 G SOLE 数据结构由 G 的质心分解 T 构成，其中 T 的每条边 e 都关联一个数据结构 W_e，用于存储可能影响 V_e 中顶点的设施。T 的每个叶节点 v（对应于 G 中的顶点 v）也关联一个数据结构 W_v，用于存储放置在其自身 v 上的设施。W_e 或 W_v 中每个权重为 w 的设施 f 都关联一个半径 r，并作为三元组 (r, f, w) 存储在 W_e 中。

我们将每个数据结构 W_x 表示为两棵搜索树 R_x 和 F_x，其中 x 可以是 T 的一条边或一个叶节点。R_x 是一棵优先搜索树 [4]（见附录 B.1），其存储了 W_x 中的三元组 (r, f, w)，使用半径 r 作为 x 坐标，并使用权重 w 作为 y 坐标。R_x 的每个节点 v 都附加了其下方子树 v 中所有三元组的总权重（见附录 B.2）。我们在附录 B.4 中将优先搜索树与范围和数据结构的这种组合称为“范围和优先搜索树”。F_x 是一棵标准搜索树，存储了 W_x 中的三元组 (r, f, w)，并以设施的标识符 f 作为键。R_x 和 F_x 中的两份 (r, f, w) 通过交叉指针相连。因此，W_x 支持以下操作，耗时 O 中的 O(lg m)：插入新的三元组 (r, f, w)、删除与设施 f 关联的三元组，以及报告满足 $q \le r$ 的所有三元组 (r, f, w) 的总权重，其中 q 为查询半径。此外，它还支持在 $O(k \lg m)$ 时间内，报告满足 $q \le r$ 的所有三元组 (r, f, w) 中权重最大的 k 个三元组。

一次 Add(v, f, w, d) 操作会从 T 的根开始，沿着路径 $P_v = \langle x_1, \ldots, x_h, v \rangle$ 遍历到 v（它是 T 的一个叶节点）。我们将三元组 (d, f, w) 插入 W_v。P_v 中的每个节点 x_i 都表示 G 的一条边 (x, y)，并且有两条子边 e_x

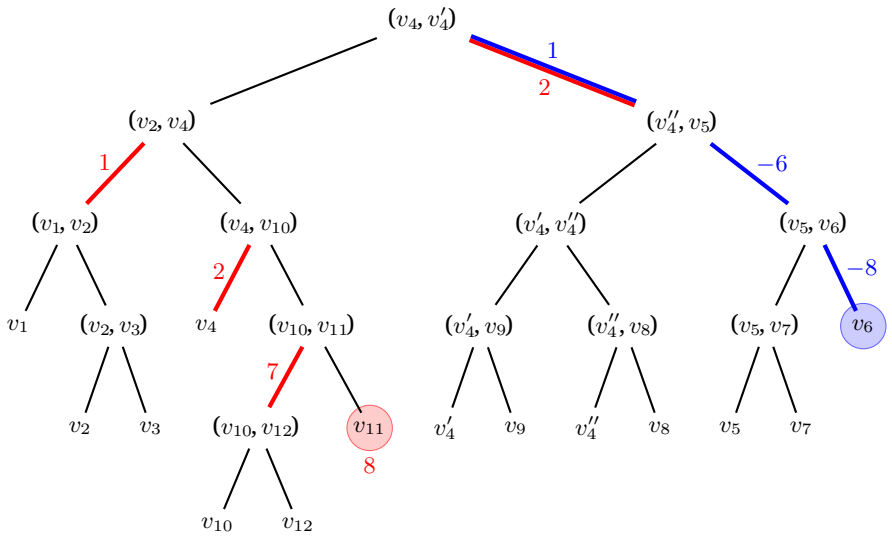


(a)



(b)

图1：树 (a)、其度为 3 的版本 (b)，以及 (b) 中树的重心分解 (c，见下一页)。(b) 中阴影子树是 (a) 中高阶节点 v_4 的度为 3 的树替代结构。一个位于 v_{11}、影响半径为 8 的设施 f 在 (b) 中产生了粉色影响区域。该区域与围绕 v_6、半径为 8 的蓝色查询区域重叠。在重心分解 (c) 中，f 存储在粗红色边的边数据结构以及 v_{11} 的节点数据结构中，半径以红色显示。围绕 v_6 、半径为 8 的查询会查询 v_6 的节点数据结构和粗蓝色边的边数据结构，查询半径以蓝色显示。特别地，f 作为对与 (v_4, v_4') 的高亮子边关联的数据结构 W_e 的查询结果的一部分被报告。



(c)

图1 续

且 e_y使得 $x \in V_{e_x}$和 $y \in V_{e_y}$。不失一般性地，假设 $v \in V_{e_x}$，并令 $d' = |\text{dist}(\rho, y) - \text{dist}(\rho, v)|$。我们将三元组 $(d - d', f, w)$ 插入与 e_y 关联的数据结构 W_{e_y} 中。如图 1c 所示。对 P_v中的每个顶点，这需要 $O(\lg m)$ 时间；总共需要 $O(\lg n \lg m)$ 时间。

Remove(v, f) 操作会遍历路径 P_v，并从 P_v 中每个节点 x_i 的每条子边 e 的数据结构 W_e 中删除与 f 关联的三元组，同时也会从 W_v 中删除该三元组。通过与 Add(v, f, w, d) 操作类似的分析可知，该操作需要 $O(\lg n \lg m)$ 时间。

一个 Sum(v, d) 查询会遍历路径 P_v。对于该路径上每条顶端端点为 x_i 的边 e，x_i 表示一条边 $(x, y) \in G$，其中不失一般性地，$e = e_y$。我们查询 W_e，以报告 W_e 中所有满足 $r \geq |\text{dist}(\rho, y) - \text{dist}(\rho, v)| - d$ 的三元组 (r, f, w) 的总权重。我们还查询 W_v，以报告直接放置在 v 上的所有设施的总权重。如图1c所示。我们将从 W_v 和沿着 P_v 的所有边数据结构 W_e 中获取的这些权重相加，并报告最终总和。由于我们通过查询 O 个数据结构 W_e 和 W_v 来回答 Sum(v, d) 查询，因此时间开销为 $O(\lg n \lg m)$。Sum(v) 查询与 Sum$(v, 0)$ 查询相同。

Top(v, k, d) 查询遍历 P_v。对于此路径上每条顶端点为 x_i 的边 e，x_i 表示一条边 $(x, y) \in G$，且不失一般性地，$e = e_y$。我们查询W_e，以检索W_e中所有满足 $r \geq |\text{dist}(\rho, y) - \text{dist}(\rho, v)| - d$ 的三元组 (r, f, w) 中权重最大的 k 个三元组。对于 P_v 上的所有边，这需要 $O(k \lg n \lg m)$ 时间。我们还从 W_v 中检索权重最大的 k 个设施，这需要 $O(k \lg m)$ 时间。影响顶点的 k 个最大权重设施

距离 d 以内的 v 位于这些查询检索到的 $O(k \lg n)$ 个设施之中，并且可使用线性时间选择 [2] 在 $O(k \lg n)$ 时间内找到。因此，Top(v, k, d) 查询需要 $O(k \lg n \lg m)$ 时间。Top(v, k) 查询等同于 Top$(v, k, 0)$ 查询。

为证明 Sum(v, d) 和 Top(v, k, d) 查询的正确性，请注意，这两个查询查询的是相同的数据结构 W_e，且查询区域也相同。Sum 查询返回这些查询区域内所有设施的总权重。Top 查询返回这些区域内权重最大的 k 个查询结果。如果我们能够证明：假设任一查询不是对这些查询区域内设施的权重求和或挑选权重最大的 k 个设施，而是 报告 这些区域内的所有设施，那么，放置在某个顶点 u 上的任意设施当且仅当其影响半径 d' 满足 $d + d' \geq \text{dist}(u, v)$ 时才会被报告，并且任何此类设施恰好被报告一次，则这两个查询都是正确的。

因此，考虑一个影响半径为 d' 、放置在某个顶点 $u \in G$ 上的设施 f，并令 v 为另一个顶点 $v \in G$。如果 $v = u$，则必须报告 f，因为无论查询半径 d如何，它都会影响 v。设施 f 不属于路径 $P_u = P_v$ 上的任何数据结构 W_e。因此，如果要报告 f，就必须由查询 W_v 报告。由于将 f 放置在 u 上会将 f 添加到 $W_u = W_v$，Sum(v, d) 查询会返回 W_v 中所有设施的总权重，而 Top(v, k, d) 查询会返回 W_v 中权重最大的 k 个设施，因此，相应的报告查询会报告 W_v 中的所有设施，包括 f。

若 $v \neq u$，则令 x_i 为 T 中从 u 到 v 的路径上的最高顶点。该顶点表示一条边 (x, y)，不失一般性，设 $u \in V_{e_x}$ 且 $v \in V_{e_y}$。在这种情况下，f 未存储在 W_v 中，而 e_y 是路径 P_v 上唯一一条属于 P_u 的悬挂边。因此，W_{e_y} 是唯一一个在存储 f 的 Sum(v, d) 或 Top(v, k, d) 查询中会被考虑的数据结构。在 W_{e_y} 中，f 的存储半径为 $r = d' - |\text{dist}(\rho, y) - \text{dist}(\rho, u)| = d' - \text{dist}(u, y)$。$G$ 中从 u 到 v 的路径经过 y，因此 $\text{dist}(u, v) = \text{dist}(u, y) + \text{dist}(y, v)$。因此，$\text{dist}(u, v) \leq d + d'$ 当且仅当 $q = |\text{dist}(\rho, v) - \text{dist}(\rho, y)| - d = \text{dist}(v, y) - d \leq d' - \text{dist}(u, y) = r$。Sum$(v, d)$ 或 Top(v, k, d) 查询的报告版本会报告 W_{e_y} 中满足 $r \geq q$ 的所有三元组 (r, f, w)。因此，当且仅当 $d + d' \geq \text{dist}(u, v)$ 时，f 会被报告。

为了界定数据结构的大小，注意到 T 的大小为 $O(n)$；每个数据结构 W_x，其中 x 是 T的叶节点或边，其大小与所存储的三元组数量呈线性关系；而放置在某个顶点 v 上的每个设施，都存储在 W_v 中，以及路径 P_v上每条边 e 所关联的数据结构 W_e 中。由于这条路径的长度为 $O(\lg n)$，每个设施会被存储 $O(\lg n)$次。因此，度数至多为 3 的树的 SOLE 数据结构大小为 $O(n + m \lg n)$。

为了获得适用于任意树（度大于 3）的 SOLE 数据结构，我们可以将任意此类树 G 转换为一棵节点度数至多为 3 的树 G'：将 G 中每个高阶顶点 u 替换为一个度数为 3 的子树 G'_u，且其所有边的长度均为 0（见图 1b、c）。我们在 G'_u 中任意选择一个顶点，作为 G'中 u 的代表。这确保了 G 中顶点之间的距离与 G' 中对应代表之间的距离相同。因此，我们可以改为在 G' 上构建 SOLE 数据结构，以支持对 G 进行添加、删除、求和和 Top 查询。至此，定理 1 的证明完成。

3 可分图的 SOLE 数据结构

本节将第 2 节的结果推广至 t-可分图。对于某个常数 t，如果图 G t-可分具有以下结构的 t-分隔符分解 C （见图 2a、b），则称其为 G t-可分：

– C 是一棵非根树，具有 $O(n)$个节点，且所有节点的度数均不超过 3。– C 的每条边 e 都关联着 G 的一个顶点子集 $S_e \subseteq V$，其大小为$|S_e| \leq t$。我们将与 e关联的 S_e 称为（边）袋。– G 的每个顶点至少属于 C的一个袋。– 设 C_1 和 C_2 是从 C 中移除任意一条边 e 后得到的子树，并设 V_i、 $i \in \{1, 2\}$分别为 C_i中所有边的袋的并集。那么，从 V_1 中的一个顶点到 V_2 中的一个顶点的任意路径，至少包含 S_e中的一个顶点。换言之， S_e 将 V_1 中的顶点与 V_2中的顶点分隔开。

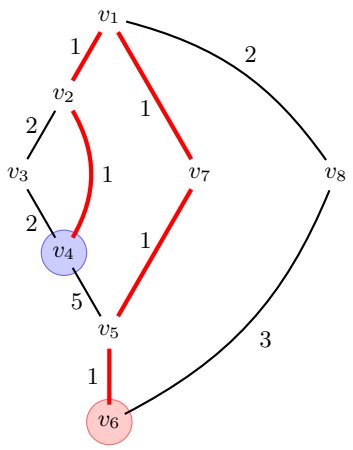
t-分隔分解的定义与树分解 [5] 和分支分解 [5], 都类似，但 t-分隔分解的性质比树分解和分支分解的性质都弱。特别是，容易看出，宽度为 b 且度数为 3 的分支分解是一个 b-分隔分解；宽度为 w 的优良树分解 C [3] 则可通过如下方式得到一个（$w+1$）-分隔分解：将每条边（v， w） $\in C$ 关联的（边）袋定义为与 v 和 w关联的（节点）袋的并集。然而， t-分隔分解不要求 G 的边与 C的叶节点之间存在双射，而分支分解要求这一点。树分解要求，对于 G的每条边（v， w），都存在一个同时包含 v 和 w的（节点）袋；而 t-分隔分解并不要求满足这一条件。因此，每个分支宽度为 b的图都有一个 t-分隔分解，其 $t \leq b$; 每个树宽为 w 的图都有一个t-分隔分解，其 $t \leq w+1$；但可能存在使这些不等式严格成立的图。

在本节中，我们证明：

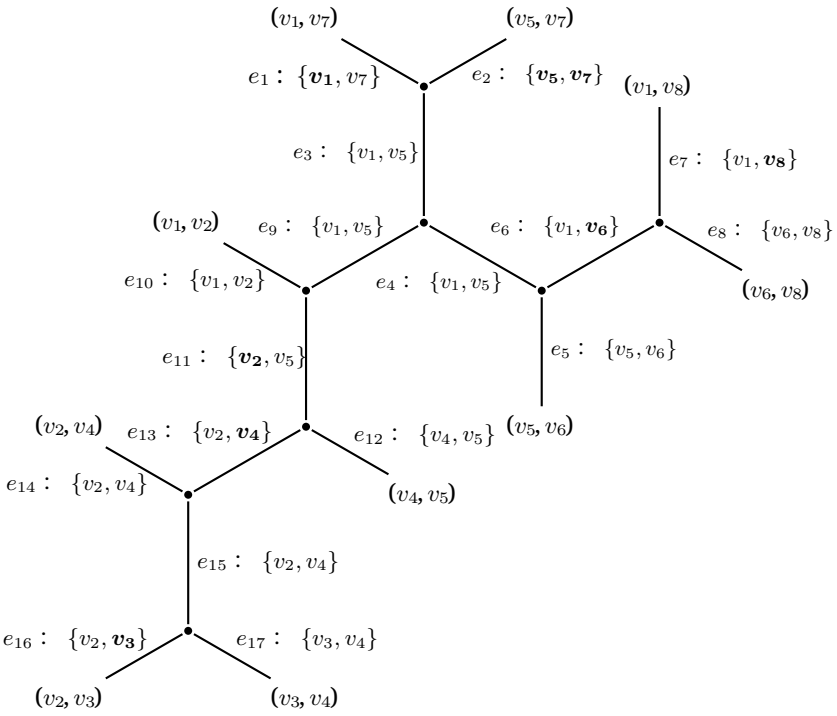
定理 2. 如果 G 是一个具有 n 个顶点的 t-可分图，其中 t 为某个常数，则存在一种 ***SOLE*** 数据结构，支持 ***Add***、***Remove*** 和 ***Sum*** 操作，所需时间为 $O(\lg n \lg^t m)$，并支持 ***Top(v, k, d)*** 操作，所需时间为 $O(k \lg n \lg^t m)$。该数据结构的空间大小为 $O(tn \lg n + m \lg^{t-1} m \lg n)$。***Add*** 和 ***Remove*** 操作的代价为摊销代价。

令 C 是针对 G的 t-分隔集分解。由于 C 是一个无根树，其节点的度数至多为3，我们可以再次构造其重心分解 T （见图2c）。 T 的每个叶节点对应于 C 的一个节点，而 T 的每个内部节点对应于 C的一条边 e 。由于 C 的大小为 O（n）， T 的高度为 O（lg n）。由于 G 的一个顶点 v 可能包含在 C的多个袋中，我们任意选择包含 v 的 C 中的一个袋 S_e ，并将 e称为 e_v，如图2b中加粗的顶点标签所示。

我们通过为 T 的每个内部节点e 增设两个数据结构 W_e 和 D_e ，并为 T 的每条边 a 增设数据结构，得到一个用于 G 的 SOLE 数据结构e T W_e D_e a T

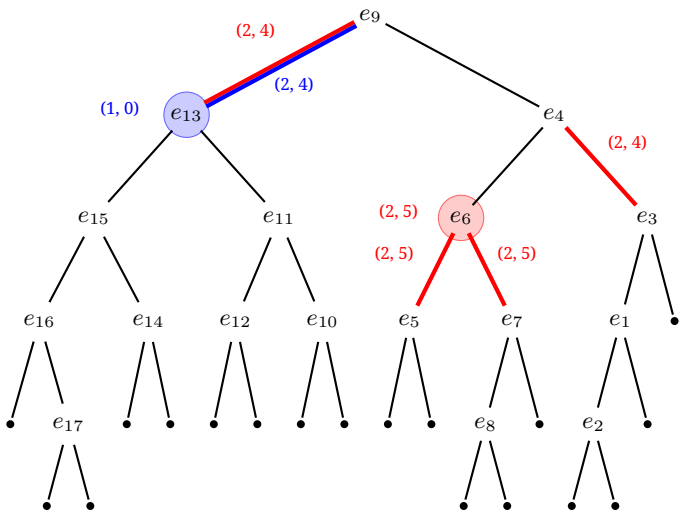


(a)



(b)

图2：说明文字见下一页。



(c)

图 2（续）：一个串并联图 G (a)、一个宽度为 2 的 G 的分支分解 C (b)，它同时也是 G 的 2-分隔集分解，以及 C 的质心分解 T (c)。(a) 中的每条边都标注了其长度。(b) 中的每条边都标注了其名称及对应的袋 S_e。一个影响半径为 5 的设施放置在顶点 v_6 上，会影响顶点 v_4，因为 (a) 中以红色标出的路径长度为 5。假设 $e_{v_6} = e_6$ 和 $e_{v_4} = e_{13}$，如 (b) 中粗体顶点所示，则 f 存储在 e_6 的节点数据结构中，以及 (c) 中红色边的边数据结构中，所示半径对以红色标出。Sum(v_4) 或 Top(v_4, k) 查询会查询 e_{13} 的顶点数据结构，以及 (c) 中蓝色边的边数据结构，所示半径对以蓝色标出。设施 f 会被添加到查询结果中，因为 $2 \leq 2$ 和 $4 \leq 4$。这两个条件满足其一即可。

使用数据结构 W_a。我们将 W_e 称为节点数据结构，将 W_a称为边数据结构。每个数据结构 W_x（其中 x 是 T中的一个节点或边）将若干设施 f存储为元组 $(r_1, \ldots, r_{t'}, f, w)$。如果 $x = e$是 T 中的一个节点，或者 $x = a$ 是 T 中顶端点为 e的一条边，那么 $t' = |S_e| \leq t$。同样， W_x 由两棵树 R_x 和 F_x 组成，它们基于存储在 W_x中的元组集合构建。F_x 将这些元组存储为二叉搜索树，并以设施 f 作为 W_x中每个元组 $(r_1, \ldots, r_{t'}, f, w)$ 的键。 R_x 是一个“t'维范围和优先搜索树”（参见附录 B.4），构建于由坐标 $(r_1, \ldots, r_{t'})$ 定义的点集之上。这是一棵 t'维范围树 [1,6]，经过增强后可支持 t'维范围和查询，耗时 $O(\lg^{t'} m)$；并支持 t'维范围 Top-k 查询，耗时$O(k\lg^{t'} m)$。

与 T 的每个内部节点 e 相关联的数据结构 D_e，存储了从每个满足以下条件的顶点 v 到 S_e中所有顶点的距离：e_v 是 T中 e 的后代。

如附录 B.4 所述，一个 t' 维范围和优先搜索树支持在 $O(\lg^{t'} m)$ 摊销时间内进行插入和删除。因此，每个数据结构 W_x 都支持在 $O(\lg^{t'} m) = O(\lg^t m)$ 摊销时间内插入一个新元组 $(r_1, \ldots, r_{t'}, f, w)$，以及删除与设施 f 关联的元组。

为支持求和和 Top 查询，我们需要支持对 W_x 进行范围和查询以及范围 Top-k 查询，但查询范围较为特殊。查询点 $q = (q_1, \ldots, q_{t'})$ 定义了一个查询范围 $\mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$，即 t' 边范围查询的补集。我们需要支持对任意此类 t' 边范围查询的补集进行范围和查询和范围 Top-k 查询。

支持此类查询的一种简单方法是将其分解为 t' 个“普通”范围查询，其查询范围

- $[q_1, \infty) \times (-\infty, \infty) \times \cdots \times (-\infty, \infty)$,
- $(-\infty, q_1) \times [q_2, \infty) \times (-\infty, \infty) \times \cdots \times (-\infty, \infty)$,
- $(-\infty, q_1) \times (-\infty, q_2) \times [q_3, \infty) \times (-\infty, \infty) \times \cdots \times (-\infty, \infty)$,
- $\cdots$
- $(-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'-1}) \times [q_{t'}, \infty)$.

这使我们能够分别在 $O(t'\lg^{t'} m)$ 和 $O(t'k\lg^{t'} m)$ 时间内支持范围求和和范围 Top-k 查询；由于 $t' \le t$ 和 t 是常数，因此时间复杂度分别为 $O(\lg^{t'} m)$ 和 $O(k\lg^{t'} m)$。

一种更好的方法是在 t' 维范围和优先搜索树上直接实现范围查询，从而支持形如 $\mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$ 的查询范围，且不产生 t' 的额外开销。要回答具有此类查询范围的范围求和查询，我们在 R_x 的第 1 层树上回答查询范围为 $[q_1, \infty)$ 的一维范围求和查询。该查询会遍历 R_x 的第 1 层树中对应于 q_1 的路径。对于这条路径左侧每个子树的根，我们在与该根关联的 $(t'-1)$ 维范围和优先搜索树上，回答查询范围为 $\mathbb{R}^{t'-1} \setminus ((-\infty, q_2) \times (-\infty, q_3) \times \cdots \times (-\infty, q_{t'}))$ 的 $(t'-1)$ 维范围求和查询。最终结果是这些查询所得总和的总和，其中包括在 R_x 的第 1 层树上执行的一维范围求和查询。因此，以 t' 面范围查询的补集作为查询范围的范围求和查询，其代价与“常规”的正交范围求和查询相同，即 $O(\lg^{t'} m)$。

类似地，为了支持一个 t' 维范围 top-k 查询，其查询范围为 $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$，我们在 R_x 的第 1 层树上回答一个 1 维范围 top-k 查询，查询范围为 $[q_1, \infty)$。此查询会遍历 R_x 的第 1 层树中与 q_1 对应的路径。对于这条路径左侧每棵子树的根，我们在与该根关联的 $(t'-1)$ 维范围和优先搜索树上，回答一个 $(t'-1)$ 维范围 top-k 查询，其查询范围为 $\mathbb{R}^{t'-1} \setminus ((-\infty, q_2) \times (-\infty, q_3) \times \cdots \times (-\infty, q_{t'}))$。很容易看出，查询范围 Q 内排名前 k 的元组属于这些 $O(k \log n)$

个由这些 $(t'-1)$ 维范围 top-k 查询以及在第 1 层树上进行的 1 维范围 top-k 查询报告的元素。现在可以使用线性时间选择 [2]，在 $O(k \lg n)$ 时间内找到排名前 k 的元组。因此，以一个 t' 边范围查询的补集作为查询范围的范围 top-k 查询需要 $O\left(k\lg^{t'} m\right)$ 时间，与“普通”的正交范围 top-k 查询相同。

我们现在可以讨论如何在针对 t-可分图的 SOLE 数据结构中支持 Add、Remove、Sum 和 Top 操作。

Add(v, f, w, d) 操作会在 T 中，从 T 的根到节点 e_v 遍历路径 $P_v = \langle e_1, \ldots, e_h = e_v \rangle$。对于 P_v 中的每个节点 e_i，令 $S_{e_i} = \{v_1, \ldots, v_{t'}\}$，并令 $d_1, \ldots, d_{t'}$ 为从 v 到 $v_1, \ldots, v_{t'}$ 的距离。然后，我们将元组 $(r_1, \ldots, r_{t'}, f, w)$ 添加到 W_{e_i}，其中对于所有 $1 \le j \le t'$，$r_j = d - \text{dist}(v, v_j)$。如果 $i = h$，我们还会将元组 $(r_1, \ldots, r_{t'}, f, w)$ 添加到与 T 中 e_i 的子边相关联的边数据结构 W_a 中。如果 $i < h$，则仅将 $(r_1, \ldots, r_{t'}, f, w)$ 添加到 e_i 的不属于路径 P_v 的子边 a 的数据结构 W_a 中。如图2c所示。由于每个数据结构 W_x 都支持在摊销 $O(\lg^t m)$ 时间内执行插入操作，因此 Add 操作的摊销成本为 $O(\lg\ n \lg^t m)$。

Remove(v, f) 操作会遍历路径 $P_v = \langle e_1, \ldots, e_k = e_v \rangle$，并从每个数据结构 W_{e_i} 以及与 P_v 中每个节点 e_i 的两条子边相关联的数据结构 W_a 中删除 f。由于每个数据结构 W_x 支持以摊销 $O(\lg^t m)$ 时间执行删除操作，因此 Remove 操作的摊销开销为 $O(\lg\ n \lg^t m)$。

Sum(v, d) 查询会遍历路径 $P_v = \langle e_1, \ldots, e_h = e_v \rangle$。对于 P_v 中的每个节点 e_i，令 $S_{e_i} = \{v_1, \ldots, v_{t'}\}$，并令 $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$，其中对于所有 $1 \le j \le t'$，$q_j = \text{dist}(v, v_i) - d$。如果 $1 \le i < h$，则在边数据结构 W_a 上以 Q 为查询范围执行范围求和查询，其中 $a = (e_i, e_{i+1})$ 是属于 P_v 的 e_i 的子边。如果 $i = h$，则在 W_{e_i} 上以 Q 为查询范围执行范围求和查询。如图2c所示。Sum(v, d) 查询的结果是所有这些范围求和查询所返回结果的总和。由于每个数据结构 W_x 上的范围求和查询都可在 $O\left(\lg^{t'} m\right) = O(\lg^t m)$ 时间内完成，因此 Sum(v, d) 查询的开销为 $O(\lg\ n \lg^t m)$。Sum(v) 查询等同于 Sum$(v, 0)$ 查询。

Top(v, k, d) 查询遍历路径 $P_v = \langle e_1, \ldots, e_h = e_v \rangle$。对于 P_v 中的每个节点 e_i，令 $S_{e_i} = \{v_1, \ldots, v_{t'}\}$，并令 $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times (-\infty, q_2) \times \cdots \times (-\infty, q_{t'}))$，其中对于所有 $1 \le j \le t'$，$q_j = \text{dist}(v, v_i) - d$。若 $1 \le i < h$，则我们在边数据结构 W_a 上执行查询范围为 Q 的范围 top-k 查询，其中 $a = (e_i, e_{i+1})$ 是属于 P_v 的 e_i 的子边。若 $i = h$，则我们在 W_{e_i} 上执行查询范围为 Q 的范围 top-k 查询。Top(v, k, d) 查询的结果是所有这些范围 top-k 查询所报告的 O 个设施中权重最高的 k 个设施组成的列表。这些 k 个设施可以使用线性时间选择 [2] 在 $O(k \lg n)$ 时间内找到。对数据结构 W_x 的每次查询耗时 $O(k \lg^t m)$。因此，Top(v, k, d) 查询的总成本为 $O(k \lg\ n \lg^t m)$。Top(v, k) 查询等同于 Top$(v, k, 0)$ 查询。

为证明 Sum(v, d) 和 Top(v, k, d) 查询的正确性，请注意，与第 2 节类似，这两个查询查询的是相同的节点数据和边数据

结构，且查询范围相同。因此，只需证明：如果任一查询报告了这些查询范围内的所有设施，那么对于放置在某个顶点 u 上、影响半径为 d' 的任意设施，当且仅当 $d + d' \geq \text{dist}(u, v)$ 时，该设施都会被报告，并且每个此类设施恰好报告一次。

因此，设 f 是一个影响半径为 d'、放置在某个顶点 $u \in G$ 上的设施，并设 v 是任意其他顶点 $v \in G$。设 e 是 e_u 和 e_v 在 T 中的最近公共祖先（LCA），并设 $S_e = \{v_1, \ldots, v_{t'}\}$。我们区分以下两种情况：

如果 e_v 是 e 的真后代，那么 f 不会存储在 W_{e_v} 中，而 P_v 中存储 f 的唯一边数据结构，是与 e 中从 e 到 e_v 的路径上通向 a 的子边相对应的数据结构 W_a。因此，$\text{Sum}(v, d)$ 或 $\text{Top}(v, k, d)$ 查询的报告版本至多报告一次 f。此查询在 W_a 上查询，查询区域为 $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times \cdots (-\infty, q_{t'}))$，其中对于所有 $1 \leq j \leq t'$，$q_j = \text{dist}(v_j) - d$。由于 e 是 T 中 e_u 和 e_v 的 LCA，因此 G 中从 u 到 v 的任何路径都必须经过 S_e 中至少一个顶点。不失一般性，假设 v_1 是其中一个顶点。那么对于所有 $1 < j \leq t'$，$\text{dist}(u, v) = \text{dist}(v, v_1) + \text{dist}(u, v_1)$，且 $\text{dist}(u, v) \leq \text{dist}(v, v_j) + \text{dist}(u, v_j)$。设施 f 在 W_a 中存储为元组 $(r_1, \ldots, r_{t'}, f, w)$，其中对于所有 $1 \leq j \leq t'$，$r_j = d' - \text{dist}(u, v_j)$。因此，$(r_1, \ldots, r_{t'}) \in Q$ 当且仅当存在某个索引 $1 \leq j \leq t'$，使得 $d' - \text{dist}(u, v_j) \geq \text{dist}(v, v_j) - d$，也就是说，$d + d' \geq \text{dist}(u, v_j) + \text{dist}(v, v_j)$。由于对于所有 $1 \leq j \leq t'$，$\text{dist}(u, v_1) + \text{dist}(v, v_1) = \text{dist}(u, v)$，且 $\text{dist}(u, v_j) + \text{dist}(v, v_j) \geq \text{dist}(u, v)$，因此当且仅当 $d + d' \geq \text{dist}(u, v)$ 时，此条件成立。因此，当且仅当 $d + d' \geq \text{dist}(u, v)$ 时，才会报告 f。

如果 e_v 不是 e 的真后代，那么 $e_v = e$ 和 $P_v \subseteq P_u$。因此，f 不存储在沿着 P_v 的任何边数据结构中，而是存储在 $W_e = W_{e_v}$ 中，其形式为元组 $(r_1, \ldots, r_{t'}, f, w)$，其中对于所有 $1 \leq j \leq t'$，$r_j = d' - \text{dist}(u, v_j)$。因此，$\text{Sum}(v, d)$ 或 $\text{Top}(v, k, d)$ 查询的报告版本最多只会报告 f 一次。该查询使用查询区域 $Q = \mathbb{R}^{t'} \setminus ((-\infty, q_1) \times \cdots (-\infty, q_{t'}))$ 查询 W_{e_v}，其中对于所有 $1 \leq j \leq t'$，$q_j = \text{dist}(v_j) - d$。由于 $v \in S_{e_v}$，我们可以不失一般性地假设 $v = v_1$。于是，对于所有 $1 < j \leq t'$，$\text{dist}(u, v) = \text{dist}(v, v_1) + \text{dist}(u, v_1)$，且 $\text{dist}(u, v) \leq \text{dist}(v, v_j) + \text{dist}(u, v_j)$。与前一种情况相同的分析现在表明，当且仅当 $d + d' \geq \text{dist}(u, v)$ 时，查询才会在 W_{e_v} 上报告 f。

接下来需要界定 SOLE 数据结构的大小。再次，树 T 的大小为 $O(n)$，高度为 $O(\lg n)$。对于每个顶点 $v \in G$，D_e 中每个祖先节点 e 的距离数据结构都会存储从v 到 S_e中所有 $t' \leq t$ 顶点的距离。因此，每个顶点 v 对 $O(\lg n)$ 个距离数据结构中的每一个贡献的大小至多为 t。因此，距离数据结构的总大小为 $O(tn \lg n)$。每个放置在某个顶点 u 上的设施 f，都会存储在路径 P_u 上 $O(\lg n)$ 个节点的节点数据结构中，并存储在这些节点的子边对应的一个或两个边数据结构中。因此，每个设施都会存储在 $O(\lg n)$ 个数据结构 W_x中。每个这样的数据结构都是一个t'维范围和优先搜索树，其中 $t' \leq t$。因此，若其中存储了s 个设施，其大小为 $O(s \lg^{t-1} s)$。因此，所有节点和边数据结构的总大小为 $O(m \lg^{t-1} m \lg n)$。定理 2 的证明至此完成。

参考文献

1. Bentley, J.L.：可分解搜索问题。《信息处理快报》**8**(5), 244–251 (1979) 2. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.：选择问题的时间界。《计算机与系统科学杂志》**7**(4), 448–461 (1973) 3. Bodlaender, H.L., Kloks：图的路径宽度和树宽的高效构造算法。《算法杂志》**21**(2), 358–402 (1996) 4. McCreight, E.M.：优先搜索树。《SIAM计算杂志》**14**(2), 257– 276 (1985) 5. Robertson, N., Seymour, P.D.：图小子。X. 树分解的障碍。《组合理论杂志》B辑**52**(2), 153–190 (1991) 6. Willard, D.E., Lueker, G.S.：为动态数据结构添加范围限制能力。《ACM杂志》**32**(3), 597–617 (1985)

A 半群

半群是一个集合 S，配备了加法运算 $+$：$S \times S \to S$，且该运算满足结合律：$x+(y+z) = (x+y)+z$对所有 $x, y, z \in S$ 都成立。不 要求存在一个单位元 $0 \in S$使得 $0+x = x = x+0$; 也不要求每个元素 $x \in S$ 都具有一个逆元 $-x \in S$ 使得 $x+(-x) = 0$。

半群的例子包括采用加法或乘法的标准数类型、任意带有 $x + y = \min(x, y)$ 或 $x + y = \max(x, y)$ 的全序集 S、由序列组成且采用 x 与 y 拼接的集合 $x+y$，以及其他许多例子。

B 标准数据结构

我们需要一些标准数据结构，作为构建 SOLE 数据结构的基础组件。下面我们将对它们进行回顾。这些数据结构所基于的所有二叉搜索树都是“叶导向”的，也就是说，它们存储的数据项位于树的叶节点处（或至少与叶节点相关联）；内部节点仅存储搜索信息，用于查找每个数据项所在的叶节点。

B.1 优先搜索树

二维空间中，点集 P 上的优先搜索树 [4] 是一种二叉搜索树T，它按照 x 坐标存储 P 中的点。 P 中的每个点都对应一个叶节点。然而，点不一定存储在与其对应的叶节点中；每个点都存储在其对应叶节点的某个祖先节点处，具体定义如下：具有最大 y 坐标的点存储在 T 的根节点处。根节点的左子节点存储与左子树中的叶节点相关联的点中具有最大y坐标的点，但如果已经存储在根节点处的点来自左子树，则排除该点。根节点的右子节点所存储的点以类似方式定义，树中所有其他节点所存储的点也遵循相同规则。

如果点已按照其 x 坐标排序，则可以在线性时间内构建优先搜索树。如果底层二叉搜索树结构采用红黑树，则插入和删除操作的最坏情况时间复杂度为 $O(\lg n)$。

我们关注的优先搜索树查询操作是范围 Top-k 查询：给定一个 x坐标范围 $[x_1, x_2]$，报告 x坐标位于区间$[x_1, x_2]$内的点中，具有最大 y坐标的 k 点。该操作可在 $O(\lg\ n + k(\lg\lg\ n + \lg\ k))\ = O(k \lg n)$ 时间内完成：遍历树 T中分别对应坐标 x_1 和 x_2的两条路径 P_1 和 P_2。两条路径的长度均为 $O(\lg n)$。这两条路径之间的部分可分解为 $O(\lg\ n)$ 棵子树 $T_1, \ldots, T_h$，其中 P_1 或 P_2中的每个节点至多对应一棵子树。区间 $[x_1, x_2]$ 中的点是沿着 P_1 和 P_2 存储的点，以及 $T_1, \ldots, T_h$中所有点的子集。每棵树 T_i 都是其所存储点上的一个堆。特别地，范围 $[x_1, x_2]$ 内具有最大 y坐标的点，存储在沿着 P_1、沿着 P_2 的位置，或某棵子树 T_i的根节点处。

要回答区间为 $[x_1, x_2]$ 的范围 Top-k 查询，我们将存储在 P_1 或 P_2 上且 x 坐标位于区间 $[x_1, x_2]$ 内的所有点，以及存储在每个子树 T_i 根上的点，插入二叉堆 Q 中。总共向 Q 中插入 $O(\lg n)$ 个点。现在重复以下操作 k 次。我们使用 DeleteMax 操作，从 Q 中取出具有最大 y 坐标的点 p 。设 p' 为 $[x_1, x_2]$ 中具有次大 y 坐标的点。若 p 存储在 P_1 或 P_2 上，那么 p' 就是沿 P_1 存储的其余点之一、沿 P_2 存储的点之一，或存储在子树 T_1、.....、T_h 的根上的点之一。若 p 存储在某个子树 T_i 中，那么 p' 就是 Q 中的其余点之一，或者是存储 T_i 的节点的两个子节点上所存储的两个点之一。我们将这两个点插入 Q ，以确保下一次 DeleteMax 操作取出 p'。此过程持续进行，直到我们取出所有 x 范围内具有最大 y 坐标的 k 个点。

遍历 P_1 和 P_2 并构建 Q 需要 $O(\lg n)$ 时间，因为这两条路径的长度为 $O(\lg n)$，而二叉堆可以在线性时间内根据其初始存储的元素数量构建。DeleteMax 操作需要 $O(\lg m)$ 时间，其中 m 是堆中的元素数量。初始时，$m = O(\lg n)$。对于从 Q中取出的每个点，我们最多向 Q中插入两个点。由于我们从 Q中取出 k 个点，m 始终不超过 $k + O(\lg n)$，因此每次 DeleteMax 操作的开销为 $O(\lg(k + \lg\ n))\ = O(\lg\lg\ n + \lg\ k)$。因此，整个范围 Top-$k$ 查询的开销为 $O(\lg\ n + k(\lg\lg\ n + \lg k))\ = O(k \lg\ n)$。

B.2 范围和数据结构

范围和数据结构维护一组按 x-坐标排序的点对 (x, w)。我们关注的查询如下：给定一个区间$[x_1, x_2]$，报告所有满足 $x_1 \le x \le x_2$的点对 (x, w) 的 w-坐标之和。

为支持此操作，范围和数据结构将这些数对按照其 x 坐标存储在二叉搜索树 T 中。T 的每个节点都存储与其后代叶节点相关联的所有数对的 w 坐标之和。要回答区间为 $[x_1, x_2]$ 的范围求和查询，我们沿着对应于 x_1 和 x_2 的两条路径 P_1 和 P_2 进行遍历，并对存储在子树根节点处的总和

介于 P_1 和 P_2 之间的值求和。如果位于 P_1 和 P_2 底部的叶节点所关联的数对，其 x 坐标在查询范围内，我们还会将这些数对的 w 坐标加上。由于 P_1 和 P_2 的长度为 $O(\lg n)$，因此需要对 $O(\lg\ n)$ 个值求和，耗时为 $O(\lg n)$。

范围和数据结构还支持在 $O(\lg n)$ 时间内进行插入和删除。具体细节很简单。

请注意，维护范围和数据结构以及回答范围和查询时，只需要数据结构中存储的各对元素的 w-坐标取自一个半群（见附录 A）。

B.3 范围树

一个 d 维范围树 [1] 是一种用于存储点集 $S \subset \mathbb{R}^d$ 的树 T，其定义如下：如果 $d = 1$，则 T 是关于 P的二叉搜索树。如果 $d > 1$，则将待存储在 T 中的点的坐标称为其 x_i坐标，其中 $1 \leq i \leq d$。T 是关于 S中各点的 x_1坐标的二叉搜索树。对于 T的每个内部节点 v，令 S_v 表示存储在 v的后代叶节点处的点的子集。然后，我们为每个节点 v关联一棵 $(d-1)$ 维范围树，该树基于 S_v中各点的第 x_2 至 x_d 个坐标构建。

在讨论范围树时，我们将 x_1 坐标上的树称为第 1层树，将与第 1 层树的节点相关联的树称为第 2层树，将与这些树的节点相关联的树称为第 3层树，依此类推。一般而言，第 i 层树是以其存储的点的 x_i 坐标为键的二叉搜索树。

Bentley [1] 证明，一个 d 维范围树需要 $O\big(n\lg^{d-1} n\big)$ 的空间，并且可以在 $O\big(n\lg^{d-1} n\big)$ 时间内构建。给定一个 d 维查询范围 $[l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_d, r_d]$，可以在 $O\big(\lg^d n + k\big)$ 时间内报告属于该范围的所有点 S，其中 k 是报告的点数。Willard 和 Lueker [6] 证明，d 维范围树支持插入和删除，且摊销时间为 $O\big(\lg^d n\big)$。他们的证明仅依赖于以下性质：二叉搜索树支持在 $O(\lg n)$ 时间内进行插入和删除，并且只要待存入树中的点已经排好序，就可以在线性时间内构建。

B.4 组合

作为我们 SOLE 数据结构的构建模块，我们需要将优先级搜索树、范围和数据结构以及范围树进行两种简单的组合。

第一种是“范围和优先搜索树”。它是定义在一组对 $(x, w) \in \mathbb{R} \times S$上的优先搜索树 T，其中 S 是任意全序半群。这些对的 w分量称为权重，并被视为 y 坐标。T中的每个节点 v 还存储了存放在 v 及其后代中的所有对的总权重。显然，这种数据结构可以在每次插入和删除所需的 $O(\lg\ n)$ 时间内维护；它还支持范围和查询，并能在 $O(\lg\ n)$ 时间内报告在某个查询范围 $[x_1, x_2]$内、x坐标对应的对中权重最大的 k 对；两者所需时间分别为 $O(k \lg\ n)$。

我们需要的第二种组合是“d 维范围和优先搜索树”。它存储一组 $(d+1)$ 元组 $(x_1, \ldots, x_d, w) \in \mathbb{R}^d \times S$，其中$S$ 再次是任意全序半群。其底层结构是这些元组前 d 个坐标上的d维范围树。每个层级为 i 的树，在任意层级 $1 \leq i \leq d$，都组织成“范围和优先搜索树”，并使用所有元组的 w 坐标作为优先级。换句话说，与每棵此类树的叶节点相关联的点会像在优先搜索树中那样向上传递，从而使该树成为以这些点的 w 坐标为堆序的堆。此类树的每个节点还存储与其后代叶节点相关联的所有元组的总权重。由于范围和优先搜索树支持在 $O(\lg n)$ 时间内更新，并且给定按排序顺序提供的待存储点，可以在线性时间内构建，因此这个“d 维范围和优先搜索树”支持以摊销 $O\left(\lg^d n\right)$ 时间进行插入和删除，采用 Willard 和 Lueker [6] 所描述的相同方法。

给定这样一棵树，我们现在可以支持 d 维范围求和和 d 维范围 Top-k 查询。查询范围为 $[l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_d, r_d]$ 的 d 维范围求和查询，会遍历第 1 层树，找出位于分别对应于 l_1 和 r_1 的两条路径 P_1 和 P_2 之间的所有子树。它会在与这些子树的根相关联的 $(d-1)$ 维范围树上，递归地回答 $(d-1)$ 维范围求和查询。结果是这些查询返回的总和，再加上沿 P_1 和 P_2 存储且落在给定查询范围内的那些元组的权重。简单的归纳论证表明，这种查询的代价为 $O\left(\lg^d n\right)$。

查询范围为 $q = [l_1, r_1] \times [l_2, r_2] \times \cdots \times [l_d, r_d]$ 的 d 维范围 Top-k 查询，会遍历分别对应于 l_1 和 r_1 的路径 P_1 和 P_2，以在第 1 层树中找出这两条路径之间的子树。它会在与这些子树的根相关联的 $(d-1)$ 维范围树上，递归地回答 $(d-1)$ 维范围 Top-k 查询。q 中排名前 k 的元组来自这些递归查询返回的 $O(k \log n)$ 个元素，以及沿 P_1 和 P_2 存储且落在查询范围 q 内的那些元组。现在可以使用线性选择 [2]，在 $O(k \lg n)$ 时间内找出这 k 个排名靠前的元组。简单的归纳论证表明，d 维范围 Top-k 查询的代价因此为 $O\left(k \lg^d n\right)$。