UFO 树：实用且可证明高效的并行批量动态树

Quinten DeMan马里兰大学帕克分校，美国
deman@umd.edu

Atharva Sharma马里兰大学帕克分校，美国
asharm07@umd.edu

Kishen N Gowda
马里兰大学帕克分校，美国
kishen19@umd.edu

Laxman Dhulipala
美国马里兰大学帕克分校
laxman@umd.edu

arXiv:2601.10706v1 [cs.DS] 15 Jan 2026

摘要

动态树问题是指在边更新的同时维护一棵树，并支持连通性查询或路径查询等操作。尽管针对这一基础问题的首个数据结构——链剖分树——早在 40 年前就已提出，我们的实验表明，它至今仍是解决该问题最快的顺序数据结构。然而，链剖分树无法支持并行批量动态更新，而且它所支持的查询类型也有限。

本文设计了一种名为 UFO 树的新型并行批量动态树数据结构，它同时支持丰富的查询功能、支持工作高效的并行批量动态更新，并且在顺序执行时可与链剖分树相媲美。我们证明，链剖分树和 UFO 树在实践中表现出色的一个关键原因是，对于低直径树，它们能够在亚对数时间内执行更新和查询。我们对经过优化的 C++ UFO 树实现进行了实验研究，并将其与另外十种动态树实现进行比较，其中几种是全新的；基准测试涵盖了直径和规模各异的合成树与真实世界树。结果表明，无论在顺序还是并行环境下，UFO 树都是支持丰富查询的最快动态树数据结构。我们的全新 UFO 树实现空间使用量低，并且能够轻松扩展至十亿规模的输入，因此有望成为实践中实现更复杂动态图算法的基础组件。

1 引言

这个动态树问题是最基础、最有趣的动态图问题之一，自 Sleator 和 Tarjan 首次提出以来便得到了广泛研究 [40]。该问题要求维护一个表示森林（即树的集合）的数据结构，同时支持边插入、边删除和顶点连通性查询。动态树和批动态树问题是并行批动态图库算法中最基础的问题之一，也是许多更复杂动态和静态图算法的基础，例如连通性 [2, 30], 最小生成森林 [30, 42],层次凝聚聚类 [18, 19],等。Sleator 和 Tarjan 首先使用著名的链接切割树 [40] 数据结构解决了动态树问题。此后，人们设计了多种动态树，以支持不同的应用、查询和权衡，包括欧拉巡游树 [29], 耙压缩树 [3, 4], 拓扑树 [21, 21, 22], 和顶树 [10]。然而，令人惊讶的是，我们的实验结果表明，这些数据结构都无法在顺序执行性能上胜过经过精心优化的链接切割树实现。

遗憾的是，链接切割树存在一些显著的局限：它们只能支持连通性查询和路径查询，而且不支持并行操作。相比之下，较新的动态树数据结构支持范围更广的查询，并能并行处理批次操作。在表 1 中，我们总结了现有动态树数据结构支持的操作及其开销。据我们所知，耙压缩树 [3, 4] 支持的查询种类最为广泛，包括连通性查询、路径查询、子树查询、最近公共祖先查询等。这些查询对于动态树数据结构的许多应用都至关重要 [8, 16, 22, 30, 33, 39, 40]。

对于批量操作，近期关于欧拉巡游树 [29,41] 和耙压缩树 [5, 6] 的研究，设计了这些数据结构的工作高效且具有多对数深度的并行批动态 版本，支持批量更新和查询。并行批动态数据结构能够利用多核快速处理连续的更新流；由于可用于设计可扩展的静态和动态算法 [1, 2, 12, 20, 25, 26, 43, 45]，近期相关研究层出不穷。

从实验结果来看，并行欧拉巡游树（ETT） [41] 是目前速度最快的并行批量动态树数据结构。不过，ETT 只能支持连通性查询和子树查询，因此无法用于需要支持更复杂查询的应用。近期一项关于并行耙压缩（RC）树的实现 [32]

首次支持了并行批次动态设置中的路径查询及许多其他关键查询。然而，我们的基准测试结果表明，从实验表现来看，它的性能明显逊于并行 ETT。造成这一差距的一个主要原因是 RC 树需要进行三元化：由于 RC 树只支持度为常数有界的输入，必须将输入树映射为度数不超过三的树。三元化会带来很大的额外开销，还需要额外付出动态维护的成本。具体而言，三元化后的树顶点数最多可达原树的2 倍，而对原树的一次边更新，可能需要对三元化后的树执行多达7 次边更新（更多详情请参见附录A.1 ）。鉴于目前的情况，一个重要的问题是：我们能否设计出一种并行批量动态树数据结构，既支持广泛的查询，又在实践中速度很快？

在本文中，我们通过设计一种名为无界扇出树（***UFO*** 树）的新型动态树数据结构，对这个问题给出了肯定的回答。据我们所知，UFO 树支持耙压缩树能够支持的所有查询，涵盖了文献中描述的动态树数据结构的所有查询。此外，UFO 树具有很强的并行性，并支持工作高效且深度为多对数级的批量更新算法。

UFO 树。 UFO 树基于并行树收缩 [36]，通过迭代聚类输入，直至森林中的每棵树都只包含一个簇，从而将输入森林映射为平衡树结构。初始时，输入树的顶点就是输入簇。每一轮中，剩余的簇会与一个或多个相邻簇合并，在下一轮形成一个簇。具体而言，UFO 树允许任何度为 3 或更高的节点与其所有 度为 1 的邻居合并。剩余的度为 1 和度为 2 的节点则沿极大匹配进行合并。

在其他基于收缩的动态树数据结构（如 RC 树和拓扑树）中，更新效率依赖于成对合并簇（每次合并最多涉及 2 个簇）。这一限制使三元化成为必要：如果不进行三元化，对于含有高阶顶点的输入，就必须将每个邻居逐一与中心顶点合并，从而产生不平衡的收缩树。UFO 树通过支持在单轮中将高阶簇与多个度为 1 的邻居合并，克服了这一限制。

UFO 树将收缩过程表示为一棵树，其中一个节点的子节点是前一轮中聚类后形成该节点的那些节点。我们证明，这棵树的高度为 $O(\log n)$。为高效执行更新，我们的更新算法本质上会删除所有内容发生变化的节点，然后自底向上重新聚类这棵树。然而，如果删除具有许多子节点或邻居的节点，更新中的重新聚类步骤可能会变得极其昂贵。为避免这一问题，我们证明，无需删除任何具有超过 3 个子节点或超过 3 个邻居的节点，也能正确地对这棵树进行重新聚类。我们还为 UFO 树中的批量插入和批量删除提供了工作高效且具有多对数深度的算法。我们的更新界在最坏情况下是工作高效的（即，工作量与最佳顺序算法相当）；但在树直径较小的情况下，其渐近表现可能更优，正如我们接下来将要讨论的。

以树直径为参数。我们证明，UFO 树和链接切割树的更新与查询复杂度都可以用树直径来表示。具体而言，UFO 树更新可以按顺序在$O(\min\{\log n,\ D\})$时间内完成，而链接切割树更新可以按顺序在 $O(\min\{\log n,\ D^2\})$ 摊还时间内完成。我们的实验结果表明，这一渐近结果在实践中确实有影响，有助于解释这两种树相较于现有动态树数据结构的出色表现。值得注意的是，动态树在实践中的许多应用可能以直径较小的森林作为输入（例如，网络图或社交网络图上的广度优先搜索树）。

其他现有的动态树数据结构在低直径输入上无法获得渐近意义上更快的性能，而我们的实验研究表明，无论输入如何，它们在实践中的性能都相对稳定。因此，在低直径输入上，直径受限树（链接切割树和 UFO 树）与其他数据结构之间的性能差距可能会非常大。

全面支持查询。最后，我们展示了如何修改 UFO 树，以支持（据我们所知）动态树中所有已知的查询，并使其查询能力与 RC 树 [3, 4] 相匹配。这些查询包括子树查询、路径查询以及若干非局部查询。子树查询和路径查询分别返回对输入树某个子树或路径上的所有权重应用某种结合函数所得的结果。其余非局部查询包括最近公共祖先查询、直径查询、中心查询、中位数查询，以及到最近标记顶点的距离查询。重要的是，对于大多数此类查询，尤其是实践中常用的查询（例如路径查询、使用可逆函数的子树查询等），可以轻松修改 UFO 树算法以支持这些查询，同时将每次操作的开销保持在 $O(\min\{\log n,\ D\})$。UFO 树仅在少数查询（例如子树最大值查询）上需要每次操作 $O(\log n)$ 时间；即便直径为常数，这类查询的每次操作时间下界也为 $\Omega(\log n)$。

我们的贡献。 本文的贡献如下：(1) 一种新颖实用的动态树数据结构——UFO 树。该结构基于采用无界扇出合并的并行树收缩，并为 UFO 树提供批量更新算法；与当前最先进的动态树数据结构相比，这些算法具有工作高效性和较低的深度。

表1。 对现有动态树数据结构以及本文理论新结果的总结。带有星号 * 的条目为本文的贡献。此处， n 表示输入树中的顶点数，D 表示输入树的直径。

	顺序开销		Not	并行操作		支持的查询		
	更新	查询	三元化	更新	查询	子树	Path	非局部
链剖分树	$O(\min\{\log n, D^2\})^*$	$O(\min\{\log n, D^2\})^*$	✓				✓	
欧拉游览树	$O(\log n)$	$O(\log n)$	✓	✓	✓	✓		
顶树	$O(\log n)$	$O(\log n)$	✓		✓	✓	✓	✓
耙压缩树	$O(\log n)$	$O(\log n)$		✓	✓	✓	✓	✓
拓扑树	$O(\log n)$	$O(\log n)$		✓*	✓*	✓*	✓	✓*
UFO 树*	$O(\min\{\log n, D\})^*$	$O(\min\{\log n, D\})^*$	✓*	✓*	✓*	✓*	✓*	✓*

(2) 为 UFO 树和拓扑树提供了新的查询算法，涵盖了文献中针对动态树提出的所有已知查询。

(3) 对 UFO 树和链接切割树的分析表明，当输入树的直径较小时，它们执行的工作量为次对数级。

(4) 我们开展了一项实验研究，将自行实现的顺序和并行 UFO 树与另外 10 种动态树进行了比较，其中包括新实现的拓扑树和 RC 树。实验结果表明，与现有最先进的动态树数据结构相比，我们的方法在速度和空间方面均有显著提升，同时也支持复杂查询和批量更新。

2 预备知识

并行模型。我们对并行算法的分析基于嵌套式 fork-join 并行 [17, 24]。具体而言，我们使用基于二叉分叉 [13]的工作量-跨度模型来分析算法。并行算法的工作量是操作总数，而深度是其计算有向无环图的关键路径长度。工作量为W且深度为 D的计算，在实践中可使用随机工作窃取调度器，在 P个处理器上以$W/P+O(D)$时间运行，且以高概率完成 [7, 14]。我们用 $O(f(n))$ 以高概率（whp）表示：对于 $c \geq 1$，以$\geq 1 - n^{-c}$的概率有 $O(cf(n))$。

并行原语。 给定一个包含 A 个 k 条目的序列，每个条目都有一个键，半排序 会重新排列 A，使所有键相同的条目相邻。半排序的期望工作量为 $O(k)$，深度为 $O(\log k)$ 以高概率 [28]。我们还使用并行哈希表 [27], , 它支持在含有 n个元素的表上执行 k 次更新，工作量为 $O(k)$，深度为 $O(\log n)$ 以高概率。

动态树背景。 动态树问题 [40]是指维护一个表示森林的数据结构，同时高效地支持边插入、边删除和顶点连通性查询。

所有动态树数据结构都支持连通性查询，许多还支持更高级的查询。路径查询 和子树查询 分别计算沿特定路径或子树内权重所应用的交换且结合函数。最低公共祖先（最近公共祖先）查询 会根据指定的根顶点，返回两个顶点的最近公共祖先。直径查询 返回最长路径的长度。中心查询 返回使其到任何其他顶点的最大距离最小的顶点。中位数查询 返回使其到所有其他顶点的加权距离之和最小的顶点。最近标记顶点查询 返回给定顶点最近的标记顶点（或到最近标记顶点的距离）。

表 1总结了已知的动态树数据结构及其支持的操作。在本文中，我们展示了新的动态树结构 UFO 树支持上述所有查询操作。我们还证明，我们的查询算法适用于拓扑树 [21–23]。

三元化。 由于拓扑树和耙压缩树是为常度数树定义的，三元化必须在构建动态树之前应用于输入树。三元化将任意度数的树转换为最大度数为≤ 3的树。在批动态设置下，也可以高效地维护这种映射表 [32]。三元化不会影响数据结构的渐近开销，但在实践中可能导致空间和时间开销都很高。我们在附录A.1中详细介绍三元化。

3 重新审视拓扑树

拓扑树 [21–23] 首次提出于四十年前，是首批基于并行树收缩的动态树结构。由于本文的主要贡献——UFO 树——采用了与拓扑树类似的树收缩方式，本节将对拓扑树进行全面且自成一体的介绍。我们还将重新分析拓扑树更新算法的开销，这部分内容在原始论文中有所缺失。最后，我们将展示如何让拓扑树支持路径查询、子树查询、最近公共祖先查询、直径查询、中心查询、中位数查询和最近标记顶点查询。

3.1 拓扑树定义

给定一棵输入树 $T = (V, E)$，拓扑树$\mathcal{T}$是一棵有根二叉树，表示对输入树顶点进行的自底向上层次聚类。图1 展示了一棵输入树以及其一种可能的拓扑树示例。

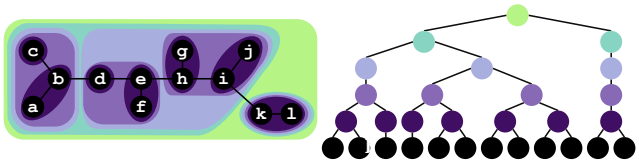

图1。 一个输入树（左）及其拓扑树（右）的示例。最右侧节点链的形成，是因为该簇的唯一邻居会反复与不同的簇合并。

我们将 T 的节点称为簇。T 的叶节点分别表示 V 中的一个顶点，我们称它们为叶簇，也称第0层簇。T 的每个后续层表示一轮并行树收缩，我们将由第 i轮形成的簇集合称为第 i 层簇。

拓扑树所采用的树收缩类型如下：在每一轮中，确定一个层 i 簇对的极大独立集（即一个匹配），然后将每一对中的簇沿着一条公共关联边合并 在一起，在层 $i+1$形成一个新簇；未参与合并的簇则各自形成簇。用于合并簇的层i 边不会出现在层 $i+1$中。未被合并的层 i 边会在层$i+1$ 以一条连接两个由其层 i 端点簇形成的层 $i+1$ 簇的边出现。层 i簇的度 是与该簇关联的层 i 边的数量。拓扑树限制树收缩中允许进行的合并类型。具体而言，允许的合并包括：

- 一个度为 1 的簇与一个度为 1 的簇
- 一个度为 1 的簇与一个度为 2 的簇
- 一个度为 2 的簇与一个度为 2 的簇
- 一个度为 1 的簇与一个度为 3 的簇

这些合并分别产生新的度为 0、度为 1、度为 2 或度为 2 的簇。

对于每一层 i 簇 $\mathcal{T}$，其父节点是树收缩过程中由它形成的层 $i+1$簇。层 $i>0$ 簇的子节点是合并形成该簇的一个或两个层 $i-1$ 簇。层簇的扇出 是指合并形成该层 $i>0$ 簇的数量 $i-1$ 簇。请注意，对于拓扑树，簇的扇出始终为 1 或 2；在接下来的章节中，我们将放宽 UFO 树的这一规则。此外，任何度为 3 的簇的扇出都必须为 1，因为允许的合并不会产生度为 3 的簇。由于每一轮都会合并极大独立集中的簇对，可以证明每轮的簇数量都会几何级数般减少，从而得到定理 3.1。证明见附录B.2。

定理 3.1。 拓扑树的高度为 $O(\log n)$，包含$O(n)$个节点，总共占用 $O(n)$空间，其中 n为输入树中的顶点数。

3.2 拓扑树更新

这里我们对拓扑树的更新算法进行高层概述。详细说明和伪代码见附录 B.3。图 2 展示了拓扑树更新的一个示例。更新算法首先删除 $\mathcal{T}$ 中所有作为与插入或删除边的端点相对应的叶簇的祖先的簇，然后以自底向上方式对所得的拓扑树分量进行重新聚类。删除拓扑树中某些簇的所有祖先，会得到由互不相交的拓扑树分量构成的森林。我们将这些分量根部的簇称为根簇。

拓扑树从第0层的根簇开始自底向上重新聚类。如果所有根簇的度都为 0，则更新完成。算法尝试将每个第 i 层根簇与其某个邻居匹配，前提是根据先前列出的允许的合并，它们可以合并。如果匹配的邻居也是根簇，则会创建一个新的第 $i+1$ 层簇，作为这两个簇的父簇。如果匹配的邻居不是根簇（我们称之为 非根簇），那么只有当它尚未与其他簇合并时，才能与目标邻居簇合并。如果满足这一条件，根簇可以将其父节点设为该邻居现有的父节点。

当根簇与非根簇合并时，必须删除该非根簇现有父节点的祖先。这是因为新的合并会改变父簇及其所有祖先所包含的顶点，因此它们在更高层级参与的合并可能无效或非极大（参见图 2中的示例）。尽管祖先路径可能会被多次删除，但可以证明，每层的根簇数量和已删除簇数量都有常数上界，因此得到定理3.2。证明见附录 B.3。

定理 3.2。 拓扑树更新需要 $O(\log n)$时间，其中 n 是输入树中的顶点数。

3.3 拓扑树查询

先前的理论工作 [21] 已经展示了如何使用拓扑树回答路径查询。本文大幅扩展了拓扑树能够支持的查询类型。我们的结果总结于定理 3.3。我们在附录 C.1中完整说明了如何支持所有这些查询。至关重要的是，所有查询均为只读操作，且可以并行运行任意数量的查询，无需同步。

定理3.3。 拓扑树可以经过修改，以支持路径查询、子树查询、最近公共祖先查询、直径查询、中心查询、中位数查询和最近标记顶点查询。

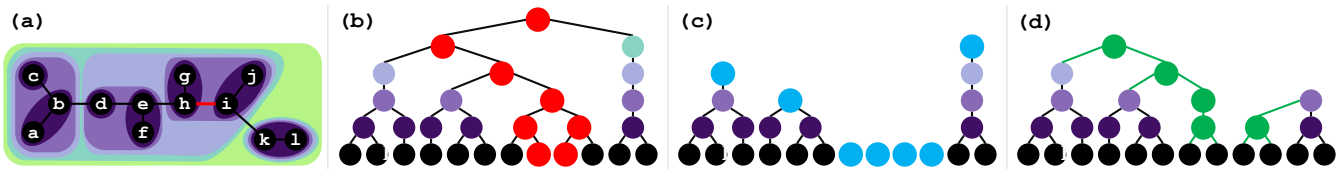

图 2. 拓扑树更新示例。(a) 中展示输入树，已删除的边 $(h,\ i)$以红色标出。(b) 中展示原始拓扑树，并以红色突出显示 h 和 i 的祖先簇。(c) 中展示执行初始的删除祖先步骤后的拓扑树，并以蓝色突出显示根簇。(d) 中展示重新聚类过程后的拓扑树，并以绿色突出显示新创建的簇。在此示例中，一个根簇（i 的父节点）与一个非根簇（k 的父节点）合并，以维持极大匹配。k 的祖父节点的祖先被删除。

4 UFO 树

本节将介绍 无界扇出树（***UFO*** 树），这是一种新的动态树数据结构，能够直接处理具有无界度顶点的输入树，同时支持路径查询以及树上的一系列其他类型的查询。与许多其他动态树数据结构不同，UFO 树不需要进行三元化，因此具有更低的内存使用量和更快的更新速度。我们证明，UFO 树的高度为O（最小值（D，对数 n）），这使得其在所有操作上，对于低直径输入，都比其他动态树数据结构具有渐近更优的性能。图3 展示了一个输入树以及该输入对应的一棵可能的 UFO 树。

UFO 树基于与拓扑树类似的树收缩，并以相同的方式表示顶点的层次聚类：将其表示为一棵树，其中任一节点的子节点代表合并形成该簇的簇。为处理度无界的顶点，UFO 树在簇的合并方式上不同于拓扑树。允许的合并如下：

- 一个度为 1 的簇与一个度为 1 的簇
- 一个度为 1 的簇与一个度为 2 的簇
- 一个度为 2 的簇与一个度为 2 的簇
- 一个 高阶 簇与一个或多个度为 1 的簇。主要区

别在于这第四条合并规则。我们将度为 ≥ 3a高阶 的簇称为高阶簇。允许多个度为 1 的簇与高阶簇合并，可以防止高阶顶点必须一次与其所有邻居逐一合并的问题，而这可能会阻碍树保持对数高度。这一性质意味着 UFO 树中可能存在具有超过 2 个子节点的簇，这给实现更新带来了一些挑战（见第 4.1节）。

在所表示的树收缩过程的每一轮中
通过 UFO 树，会对允许的合并执行极大独立集。具体而言，任何高阶簇都必须与其所有度为 1 的邻居合并。对于剩余的度为 1 和 2 的簇，则沿着极大匹配进行合并。对于高阶簇而言，更强的极大性不变量是必要的，以确保每个连续轮次中的簇数量呈几何级数减少。

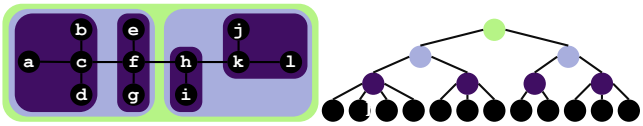

图 3. 输入树（左）及其对应的 UFO 树（右）的示例。高阶顶点 c、 f和 k 在第一轮中分别与其所有度为 1 的邻居合并。

特别地，我们利用极大性证明定理 4.1，该定理表明 UFO 树与拓扑树具有相同的平衡性质。我们进一步证明定理4.2，表明 UFO 树上改进的极大性保证还使得其高度相对于直径具有更强的界，D，其中直径是输入树的直径。因此，UFO 树的高度具有更严格的界 $O(\min\{\log n, D\})$。所有证明均见附录 B.4。

定理 4.1. *UFO* 树的高度为 $O(\log n)$，包含 $O(n)$ 个节点，总计使用 $O(n)$ 空间，其中 n 表示输入树中的顶点数。

定理 4.2. *UFO* 树的高度为 $\leq \lceil D/2 \rceil$，其中 D 表示输入树的直径。

4.1 UFO 树更新

UFO 树的更新总体策略与拓扑树相同：删除内容发生变化的簇的祖先，并自底向上重新构建簇。不过，若仅以朴素方式执行这一操作，就无法维持拓扑树相同的 $O(\log n)$ 更新开销。一个挑战是，删除高阶簇时（例如其度为 $\omega(\log n)$），仅将该簇从其邻居的邻接表中移除就可能开销过高。另一个挑战是，树不再是二叉树：簇的扇出可能达到 $\omega(1)$ （回顾一下，簇的扇出是其子节点的数量）。我们将扇出为高扇出的簇定义为高扇出簇。删除此类簇可能导致某一层出现多于常数个根簇。为避免这些问题，我们的更新算法会谨慎地避免删除高阶簇和高扇出簇。图4 展示了 UFO 树的一次更新示例。算法1 和2 给出了 UFO 树更新的伪代码。

我们首先介绍删除祖先过程，该过程用于在算法2中执行更新。

算法 1： 删除祖先(c，i)

```
1  prev ← c，curr ← c.父节点，next ← curr.parent
2  当curr时执行
3  如果curr.度 < 3且 curr.扇出 < 3则
4  对于 x ∈ curr.邻居 执行
5      x.邻居.删除(curr)
6  对于 y ∈ curr.子节点 执行
7      y.父节点 ←空
8      根簇[i].插入(y)
9  如果 next那么
10     next.子节点.删除(curr)
11 释放簇(当前)
12 否则，如果 前驱.度 ≤ 2 且 ¬前驱.已删除 则
13     前驱.父节点 ← 空
14 前驱 ← 当前，当前 ← 后继，后继 ← 当前.父节点
15 i ← i + 1
```

删除祖先子程序。每当算法尝试删除一条祖先路径时，只会删除那些既非高阶、也非高扇出的簇（算法1，第3–11行）。每个被删除簇的子节点都会成为其所在层的根簇。此外，如果路径上的某个簇未被删除，且其父节点也未被删除，那么仅当该簇的度为 ≤ 2 时，才会将其与父节点断开连接（即不再参与形成父节点的合并）（算法1，第12–13行）。这是必要的，因为度为 1 的簇的内容已经发生变化。如果该簇是高阶簇，且其父节点未被删除，那么它必定是涉及该高阶簇的某次合并中的“中心”，因此仍须参与形成其父节点的合并；否则，父节点剩余的子节点将不再对应有效的合并类型。我们在附录B.5中严格证明了更新算法不会产生无效合并。

更新 算法。更新算法首先尝试删除更新边端点的祖先，u 和 v（算法 2，第 1行）。然后，在树中存在该边的所有层级插入（或删除）更新边（算法2，第 2行）。

接下来，从第 0 层开始，自底向上重新聚类根簇。首先，为高阶根簇指定新的父节点，并将其与所有度为 1 的邻居合并（算法 2，第 4–11 行）。如果其中任何簇不是根簇，算法就会尝试删除它们的祖先，以避免内存泄漏。

剩余未合并的度为 2根簇（算法 2，第 14–22 行）和度为 1 根簇（算法 2，第 23–35 行）随后会尝试与一个邻居合并。度为 2 的根簇会检查它们的两个邻居，寻找尚未与其他簇合并的邻居。度

算法 2： 更新(u，v，删除_标志)

```
1  删除祖先(u,0), 删除祖先(v,0), i ← 0
2  在所有层级插入或删除边 (u，v）。
3  当根簇[i] ≠ ∅ 时
4      对 x ∈ 根簇[i] 执行
5          如果 x.度 ≥ 3则
6              x.父节点 ← 新建簇()
7              对 y ∈ x.邻居 执行
8                  如果 y.度 = 1 则
9                      如果 y.父节点 ≠ 空 则
10                         删除祖先(y, i)
11                     y.父节点 ← x.父节点
12     遍历 x ∈ 根簇[i] 执行
13         如果 x.父节点 ≠ 为空，则继续
14         如果 x.度 = 2，则
15             遍历 y ∈ x.邻居 执行
16                 如果 y.度 <= 2 且 ¬y.合并 则
17                     如果 y.父节点 ≠ 为空 则
18                         删除祖先(y.父节点 i + 1)
19                         x.父节点 ← y.父节点
20                     else
21                         x.父节点 ←新建簇()
22                         y.父节点 ← x.父节点
23         否则如果 x.度 = 1 则
24             y ← x.邻居[0]
25             如果 y.父节点 ≠ 为空且 ¬y.合并 则
26                 删除祖先(y.父节点 i + 1)
27                 x.父节点 ← y.父节点
28             否则如果 y.父节点 ≠ 为空且 y.度 ≥ 3 则
29                 删除祖先(y.父节点 i + 1)
30                 在所有层级 > i 删除边 (y, x)。
31                 x.父节点 ← y.父节点
32             else
33                 x.父节点 ← 新建簇()
34                 如果 y.父节点 =为空 则
35                     y.父节点 ← x.父节点
36         如果 x 无法与任何邻居合并 则
37             x.父节点 ← 新建簇()
38     填充新簇的邻接表。
39     将所有新簇添加到 RootClusters[i + 1]。
40     根簇[i].清除()， i ← i + 1。
```

为 1 的根簇可以与尚未合并的度为 2 的邻居合并，也可以与任何度为 1 或高阶邻居合并。如果一个根簇与一个非根簇合并，就必须尝试删除该非根簇父节点的祖先，因为其内容已经发生变化（算法 2，第 18、26 和 29 行）。

这种更新算法效率很高，因为更新过程中产生的所有根簇的度最多为 4。我们证明这一点，并给出界限，表明每一层的根簇数量和已删除簇数量均有常数上界。结合关于

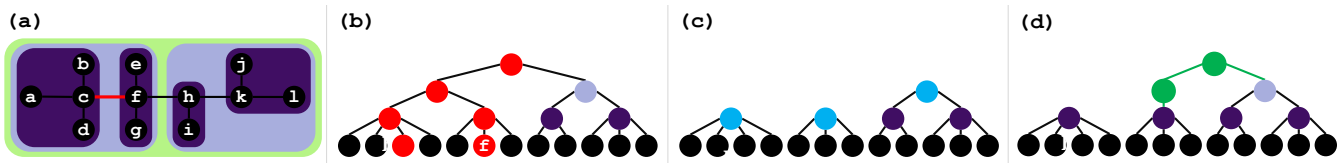

图 4. UFO 树的一次更新示例。子图 (a) 显示输入树，其中已删除的边 $(c,\ f)$以红色标出。在 (b) 中，我们展示原始 UFO 树，并以红色突出显示 c和 f的祖先簇。在 (c) 中，我们展示执行初始祖先移除步骤后的 UFO 树，并以蓝色突出显示根簇。在此示例中，c和 f的父节点因扇出较高而未被删除，但所有其他祖先均已删除。在 (d) 中，我们展示重新聚类过程后的 UFO 树，并以绿色突出显示新建的簇。

UFO 树高度的定理4.1 和4.2中的界限，得到定理4.3。证明见附录B.5。

定理 4.3. *UFO* 树更新所需时间为 $O(\min\{\log n, D\})$，其中 n 是输入树中的顶点数，D 是输入树的直径。

4.2 UFO 树查询

我们展示了 UFO 树如何支持（据我们所知）文献中针对动态树描述的所有知名查询。我们的结果总结于定理4.4 中。附录 C.2 中详细说明了如何支持所有这些查询。所有查询均为只读，且可并行运行任意数量的查询，无需同步。大多数查询都能在不改变 UFO 树每次操作的基本代价的情况下得到支持。其他查询（例如，不可逆子树查询）则需要对高扇出簇的所有子节点重新计算不可逆函数的结果。为了高效支持这类查询，我们使用秩树存储高扇出簇的子节点集合 [44],，其中每个簇的权重是其包含的顶点数。我们采用秩树，是因为其权重偏置性质能使 UFO 树的整体高度和操作开销保持在 $O(\log n)$而非 $O(\log^2 n)$。

定理 4.4。 *UFO* 树可以经过修改，以支持路径查询、使用可逆函数的子树查询，以及每次操作开销为 $O(\min\{\log n, D\})$ 的最近公共祖先查询。*UFO* 树可以经过修改，以支持任意子树查询、直径查询、中心查询、中位数查询和最近标记顶点查询，每次操作的开销为 $O(\log n)$。

在附录C.3 中，我们证明，存在直径为常数的输入树，若要在 $o(\log n)$时间内解决子树最小值或最大值查询，就会突破比较排序下界。这个下界有助于解释为什么我们的数据结构无法针对一般的不可逆子树查询实现每次查询 $o(\log n)$时间的性能。我们认为，类似的下界技术或可用于证明，对于其他几类查询，UFO 树也无法实现每次查询 $o(\log n)$时间的 $O(D)$ 开销上界。

5 并行批量动态树

本节首先介绍一种用于拓扑树的新型批动态更新算法，然后将这些思路扩展到 UFO 树。我们的两种算法都具有工作量高效和低深度的特点。此外，我们证明，在并行批动态 UFO 树上执行 k 次更新需要 $O(kD)$ 的工作量。

我们的算法支持插入和删除组成的混合批次。我们假设输入中每条边至多涉及一次更新，并且更新的任何可能排序都定义了一个有效的顺序更新序列。

5.1 并行批量动态拓扑树

批量更新的基本框架与顺序算法相同：首先，删除每次更新端点对应的叶簇的所有祖先，然后自底向上重新聚类树。与顺序场景不同的是，每层要删除的根簇和簇的数量可能远大于 $O(1)$（我们在附录 $O(k)$中证明其为B.6）。因此，高效的批并行算法应并行地重新聚类并删除簇。算法3 给出了我们的批量并行更新算法的伪代码。

算法 3： 批量更新$((u_1, v_1, \text{删除}_1)\ldots, (u_k, v_k, \text{删除}_k))$

```
1  更新发生变化的叶簇的邻接表。
2  R_0 ← u_1 的叶簇,..., u_k, v_1,..., v_k。
3  D_1 ← R_0.MapToParents()
4  当 |R_i| > 0 或 |D_{i+1}| > 0 时执行
5      D_{i+2} ← D_{i+1}.MapToParents()
6      R_i ← R_i ∪ D_{i+1}.MapToChildren()
7      并行执行 for c ∈ D_{i+1} do
8          从其邻居的邻接表中删除 c。
9          从其父节点的子节点列表中删除 c。
10     并行地对 c ∈ D_{i+1} 执行
11         释放簇(c)
12     [M, NR] ← ParallelMaximalMatching(R_i)
13     并行地对(c_1, c_2) ∈ M 执行
14         如果 c_2 ∈ R_i 那么 c_2.父节点 ← 新建簇()
15         c_1.父节点 ← c_2.父节点
16     填充 R_i的父节点的邻接表。
17     D_{i+2} ← D_{i+2} ∪ NR.MapToParents()
18     R_{i+1} ← R_i.MapToParents()
```

并行更新邻接表。 第一步是更新更新端点对应的叶簇的邻接表（第 1 行）。具体做法是将每次边更新映射为两条有向边，然后使用并行半排序 [28]，按端点对有向边分组。由于拓扑树的输入中，顶点的度为≤ 3，因此任何顶点关联的更新数量最多为6：最多删除 3 条当前关联边，并插入最多 3 条新边。类似的方法可用于高效地从邻居的邻接表中移除所有已删除簇（第 8 行），并填充新簇的邻接表（第 16 行）。同样的方法也可用于更新簇的子节点列表（第9 行），因为拓扑树中簇的扇出为 ≤ 2。

低深度祖先删除。在顺序重聚类步骤中，每当根簇与非根簇合并时，都必须删除该非根簇父节点的祖先。由于这种情况可能在每个 $O(\log n)$ 层发生，而一条路径上的祖先数量为 $O(\log n)$，因此主动删除这些祖先会导致整体深度达到 $O(\log^2 n)$。因此，我们采用惰性方法删除簇。其思路是在当前正在重新聚类的层之上一层，维护一个应被删除的簇集合。我们用 R_i 和 D_i 分别表示层 i 的根簇集合和层 i 中待删除的簇集合。在每一层 i，算法都会维护 R_i 和 D_{i+1}。更新后的叶簇集合将用作 R_0（第 2 行）。由于 R_0 中根簇的所有祖先都应被删除，因此 D_1 初始化为 R_0 中簇的父节点集合（第 3 行）。每当使用 MapToParents 或 MapToChildren 函数将一组簇映射到其父节点或子节点时，我们都会先执行并行映射操作，再执行并行去重操作 [17]。

在每一层 i中，重新聚类 R_i中的簇之前，我们会将 D_{i+1} 中簇的父节点存储为 D_{i+2}（第5行）。随后， D_{i+1} 中的簇全部被删除，其子节点则被添加到 R_i（第6行）。此外，当 R_i 中的根簇与一个非根簇合并时，必须将其祖父节点（如果存在）添加到 D_{i+2}（第17行），这对应于在顺序更新算法中移除该非根簇的父节点的祖先。以下段落将对此进行更详细的说明。

并行重聚类。 顺序拓扑树重新聚类算法使用一种简单的贪心算法，在根簇及其尚未与其他簇合并的非根邻居中寻找极大匹配。对于批量更新，每一层的此类簇数量可能远大于常数（我们在附录 O（k）中证明了这一点，见附录 B.6）。对于度为 3 且只能与相邻的度为 1 簇合并的簇，由于没有其他簇与度为 1的簇相邻，因此可以直接并行完成。其余相关簇构成一组链表。为了计算极大匹配以高效的并行方式处理这些链表，我们应用并行链表排序 [13],，其在二叉分叉-连接模型中需要 $O(\log k)$ 深度，以高概率，以及 $O(k)$ 最坏情况工作量。给定链表排序结果，每个偶数索引处的簇都可以与下一个索引处的邻居合并。

综上所述。算法的主循环会删除第 $i+1$ 层的簇，并在每一层重新聚类第 i 层的根簇，直至树构建完成，且所有应删除的簇都已删除（第4行）。首先，必须在删除 D_{i+1} 中的簇之前，将其中簇的父节点存储在 D_{i+2} 中（第 D_{i+1} 5行）。接下来，将本轮中父节点所在层为 $i+1$ 的簇添加到 R_i（第6行）。最后，从所有邻居的邻接表中删除 D_{i+1} 中的簇，然后删除这些簇（第7–11行）。

接着，算法计算根簇与其尚未和其他簇组合的非根邻居之间的一个极大合并集合（第 12 行）。在伪代码中，ParallelMaximalMatching 函数返回 M 和NR。 M 是合并对的集合。如果找到与某个非根簇的合并，该非根簇就是合并对中的第二个簇。 NR 是与之合并的非根簇的父节点集合。算法会创建新的父簇，并为每次合并适当地设置父指针，同时为新簇填充第$i+1$ 层的邻接表（第 13–16 行）。对于计算所得的匹配中涉及非根簇的任意组合，算法都必须移除现有父簇的所有祖先，因此，NR 中各簇的父节点会被加入 D_{i+2}（第 17 行）。新父簇以及参与组合的任何非根簇的父节点共同构成 R_{i+1}，即下一层的根簇集合（第 18 行）。当不再有根簇或待删除的簇时，批量更新即告完成。

分析。 我们在 B.6 中证明算法 3 的正确性，并分析其开销。我们的结果总结于定理 5.1。

定理 5.1。 拓扑树中的批量并行更新需要$O(k\log(1+n/k))$期望工作量，且深度为 $O(\log n \log k)$ 以高概率，其中 n 为顶点数，k 为批次大小。

5.2 并行批量动态UFO 树

UFO 树中批量并行更新的算法遵循与算法 3 类似的基本流程。这里不再从头解释该算法，而是总结并行批处理动态 UFO 树的关键差异和挑战。

(1) UFO 树中的邻接表和子集的大小不一定是 $O(1)$。
(2) UFO 树更新算法要求祖先路径上的某些簇不得被删除。因此，如果被修改边的两个端点在更高层级仍分别属于不同且未被删除的簇，则必须将较低层级执行的边插入或删除操作向上传播到更高层级。

(3) 在顺序 UFO 树算法中，任何根簇的度都有常数上界。这在批处理设置中不再成立。

算法4 展示了我们在 UFO 树中的批量并行更新算法的伪代码，并着重说明了它与拓扑树算法的不同之处。

算法 4： BatchUpdate($(u_1, v_1,$ 删除$_1) \ldots, (u_k, v_k,$ 删除$_k)$)

```
1  更新已更新叶簇的邻接表。
2  R_0 ← u_1、……、u_k、v_1、……、v_k的叶簇。
3  D_1 ← R_0.MapToParents()
4  E_0^- ← {(u, v, 删除) ∈ U | 删除 = true}
5
6  当 |R_i| > 0 或 |D_{i+1}| > 0 时
7      D_{i+2} ← D_{i+1}.MapToParents()
8      R_i ← R_i ∪ D_{i+1}.MapToChildren()
9      从邻接表中删除 E_{i+1}^- 边。
10     E_{i+2}^- ← E_{i+1}^-.MapToParents()
11     并行执行 c ∈ D_{i+1}
12         如                                  则
13             从 c 的邻居的邻接表中删除 c。
14             从 c 的父节点的子节点列表中删除 c。
               将与 c 相连的所有边添加到 E_{i+1}^-。
15         否则，如                ，则
16             将 c 从其父节点断开，并将 c 添加到 R_{i+1}。
17     并行地，对于 c ∈ D_{i+1}，执行
18         如                              ，则
19             释放簇(c)
20     [M, NR] ← 并行 Ufo 最大匹配(R_i)
21     并行地，对于(c_1, c_2) ∈ M，执行
22         如果 c_2 ∈ R_i 那么 c_2.父节点 ← 新建簇()
23         c_1.父节点 ← c_2.父节点
24     填充 R_i的父节点的邻接表。
25     D_{i+2} ← D_{i+2} ∪ NR.MapToParents()
26     R_{i+1} ← R_i.MapToParents()
```

挑战 1：大型邻接表和子节点集合。我们的批量并行拓扑树更新算法依赖于邻接表和子节点集合的大小有常数上界这一事实，从而实现高效的并行更新。由于 UFO 树中的簇的度和扇出都没有常数上界，我们使用并行哈希表 [27] 来表示每个簇的邻居和子节点。这使我们能够以较低的深度高效地批量插入和删除邻居。为了批量更新某一层的邻接表，我们使用并行半排序 [28] 按端点对更新进行分组，然后将每组作为一次批量更新传递给该端点的邻居集合。更新子节点集合时也采用相同的技术。

挑战 2：有条件地删除簇。顺序 UFO 树算法不会删除高阶或高扇出的簇。情况之所以复杂，是因为删除或断开高扇出簇的某些子节点，可能会使其变为低扇出，随后便应将其删除。通过批量更新，先前度很高的簇可能变为低度，扇出很高的簇也可能变为低扇出。由于我们显式存储并更新每个簇的邻居集合和子节点集合，因此很容易确定簇的度和扇出。因此，在检查是否要删除簇时，位于D_{i+1}（第11–19行），算法只需检查每个簇的邻居集合和子节点集合的大小即可。与顺序算法一样，任何未被删除但度为 ≤ 2 的簇都会与其父节点断开，并在下一层作为根簇加入。

为处理簇因边删除更新而变为低度的情况，在判断是否删除该层的簇之前，我们必须从树的每一层移除这些已删除边。为了在批量并行环境中以低深度完成此操作，我们采用了与批量并行拓扑树更新中的低深度祖先删除类似的方法。我们维护一个集合 E_i^-，其中包含需要在 i层删除的边。我们只针对紧邻当前重新聚类层级之上的那一层维护此集合，而不立即将更新向上传播至所有层。必须先从 E_i^-层删除 i 中的边，然后算法才能判断是否删除 i 层的簇，以处理高阶簇变为低度的情况（第9行）。此外，每当删除一个簇时，其所有关联边也会被删除，因此会被添加到 E_i^-中（第14行）。

挑战 3：高阶簇重新聚类。 对于顺序 UFO 树，我们证明了任意根簇的度 ≤ 4。这在批量算法中并不成立：根簇的度最高可达 $\Omega(k)$。为了对这些高阶根簇重新聚类，我们使用并行哈希表并行遍历它们的邻居，找出所有度为 1 的邻居。与拓扑树的处理方式相同，这些高阶根簇可以彼此独立地并行处理，因为它们必须合并的簇集合互不相交。其余簇构成一组链表，并使用链表排序确定一个极大匹配。尽管某些根簇的度可以达到 $\Omega(k)$，我们在附录 B.6 中证明，任意一层的根簇度数之和为 $O(k)$。因此，我们的算法具有工作量高效性。

分析。 我们在附录 B.6 中详细分析了 UFO 树批量更新算法的正确性和开销，从而得到定理 5.2。

定理 5.2。在 *UFO* 树中，批量并行更新的期望工作量为$O(\min\{k \log(1+n/k)\ kD\})$，深度为 $O(\log n \log k)$，且以高概率成立；其中 n 表示顶点数，D 表示输入森林的直径，k 表示批次大小。

6 实验评估

本文中的所有实验均在一台 96 核机器（192 个超线程）上运行，该机器配备 $4 \times 2$1.1 GHz Intel Xeon(R) Platinum 8160 处理器（每个处理器均有 33MiB L3 缓存）和 1.5TB 主存。我们的实现均使用 C 编写++ ，并以 -O3 优化选项编译。我们的并行实现使用 ParlayLib [11],，这是一个用于共享内存并行编程的 C++ 库。我们使用微软的 mimalloc [35]，并将其与 ParlayLib 的并行分配器组合使用，以实现并行内存分配。有关实现的详细信息，请参阅附录D。

输入。 为了比较不同属性的多种树上的性能，我们在多种合成输入树上测试了实现，包括路径图、完美二叉树、完美 k 叉树、星形树和蒲公英树。我们还测试了随机生成的度为 3、无界度以及优先连接树。

我们还测试了多个真实世界图的生成树，汇总结果见表2。对于每个图，我们分别测试了以下两种生成树：一种是从随机顶点开始构建的 广度优先生成森林（*BFS*），另一种是 随机增量生成森林（*RIS*），其通过以随机顺序插入图的边，并且仅包含在当前生成森林中尚未连通的边来生成。

6.1 顺序性能

我们将 UFO 树与新实现的拓扑树和 RC 树进行比较，这两者仅在必要时使用三元化。我们还与现有的链接切割树 [41], 伸展顶树 [31], 和欧拉游览树（分别使用树堆、伸展树和跳表实现） [41]进行了比较。我们采用了自行实现的顺序 RC 树，因为我们发现，与在单线程上运行最新的并行RC树实现 [32]（批次大小为 $k = 1$ ）相比，它的速度显著更快。

表 2。 我们实验中使用的真实世界图数据集。

Name	缩写	Type	$\|V\|$	$\|E\|$	Cite
USA Roads	USA	Road	23.95M	28.85M	[37]
ENWiki	ENW	Web	4.21M	91.94M	[15]
StackOverflow	SO	时间	6.02M	28.18M	[38]
Twitter	TWIT	社交	41.65M	1.20B	[34]

更新速度。 我们通过测量构建并删除整棵树所需的总时间来研究更新速度；也就是说，对于一棵有 $10^7 - 1$ 条边的树，执行 $10^7 - 1$ 次插入，随后执行$10^7 - 1$ 次删除（两者均按随机顺序进行）。我们在所有前述合成树上运行实验，取 $n = 10^7$，并在真实世界图上运行广度优先搜索树/RIS 树。图5展示了结果。

我们发现，除了链接切割树之外，UFO 树在几乎所有输入上的表现都优于其他数据结构。我们发现，UFO 树的平均速度（几何平均数）仅比链接切割树慢 3.23×（最多慢 5.7× ）。次优的数据结构（伸展顶树）比链接切割树慢至多 18×（平均慢 6.11× ）。我们还发现，对于低直径输入（例如星形、64叉，以及 SO-BFS 等低直径真实世界树），UFO 树和链接切割树都明显快于所有其他实现。与伸展顶树相比，后者支持与 UFO 树相同的一组查询，但显然不易并行化；我们发现，UFO 树的平均速度快 1.9× 。RC 树可以并行化，也支持相同的一组查询，但其表现始终远逊于其他动态树实现，平均而言比 UFO 树慢 24.78× 。总之，我们发现，UFO 树的表现与仅支持特定查询的动态树实现（即链接切割树和欧拉巡游树）不相上下，同时支持范围广得多的查询。

更新和查询的直径扩展。 为了研究本文证明的链接-切割树和 UFO 树的渐近直径界限是否能在实践中改善低直径输入上的表现，我们进行了一项实验，生成直径各异（逐渐减小）的树，并研究性能如何随直径变化。我们通过让节点 i 根据参数为 α 的齐夫分布，在 $[0, i)$ 中选择一个目标，然后随机排列节点 ID，来生成树。随着 α 增大，直径往往会减小。

更新、连通性查询和路径查询的结果如图 6 所示。结果表明，对于更新和两种不同类型的查询，随着输入直径的减小，链接切割树和 UFO 树的速度都会相应提升。此外，随着树直径减小，所有其他实现的运行时间都相对稳定，或变得更慢。我们观察到，在直径较小的输入上，拓扑树和 RC 树实现中的更新速度会显著变慢。这是因为，对于包含许多高阶节点的输入（低直径必然如此），三元化开销会变得更大。有趣的是，UFO 树在路径查询和连通性查询方面都表现最佳，甚至优于链接切割树；这可能是因为链接切割树需要在查询时修改树，而 UFO 树不需要。

6.2 内存使用量

为了比较不同数据结构的内存使用量，我们测量构建包含n 个顶点和 $n - 1$ 条边的完整树后所使用的内存量。我们使用$n = 10^7$ 进行这项实验，并在图7中报告结果。我们发现，不需要三元化时，基于树收缩的数据结构比 ETT 使用的内存更少。需要三元化时，RC 树和拓扑树使用的内存则显著更多。UFO 树是最节省空间的基于树收缩的数据结构，并且在低直径输入下，内存使用量也显著减少。链接-切割

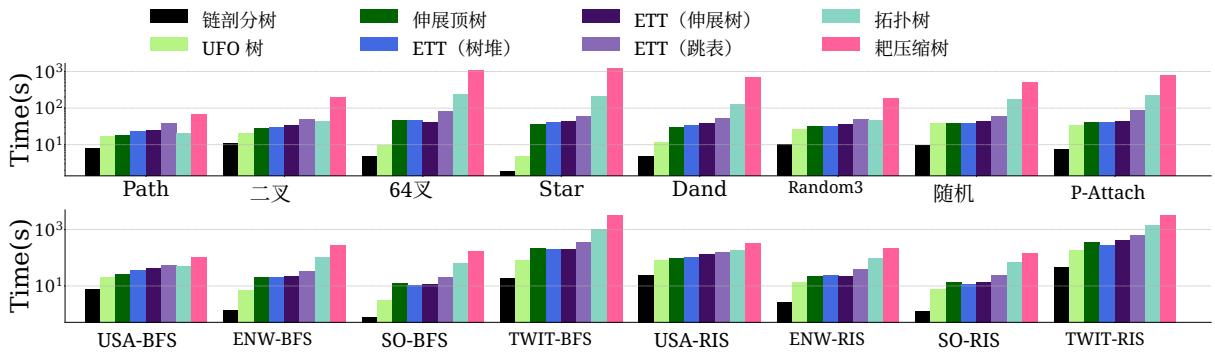

图 5。 我们的**顺序** 动态树数据结构更新速度实验的结果。顶行显示了带有 $n = 10^7$的各种合成树上的结果。底行显示了我们的真实世界图数据集的广度优先搜索森林和随机增量生成森林上的结果。我们报告了以随机顺序插入所有边，然后删除所有边所需的总时间。

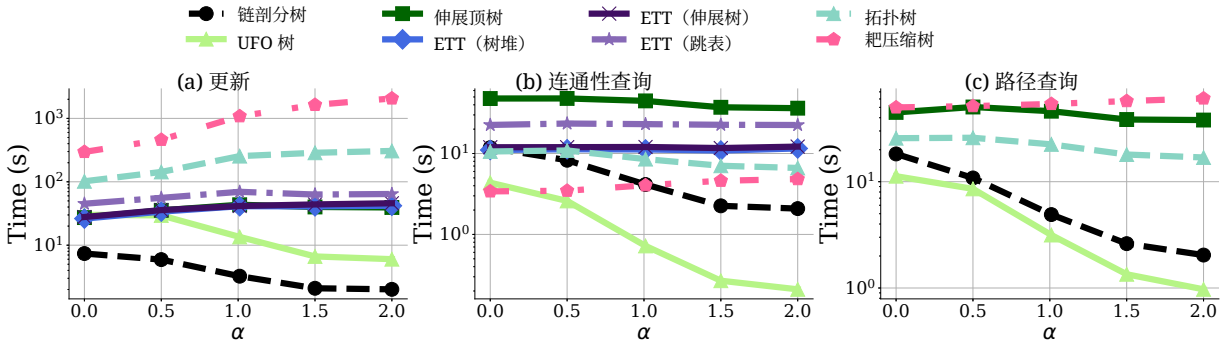

图 6。 我们的**直径扫描** 实验结果。随着 α增加，直径趋于减小。子图 (a) 展示了 $n = 10^7$时的总更新时间结果。子图 (b) 和 (c) 分别展示了连通性查询和路径查询的结果。我们报告了在 $n = 10^6$的完整树上执行 $q = 10^6$ 次查询的总运行时间。

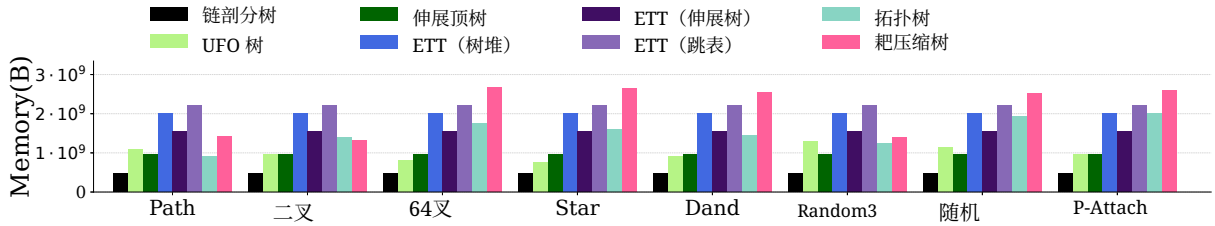

图 7。 我们对动态树数据结构进行的**内存使用量** 实验结果。我们在多种合成树上测量内存使用量，其中包含 $n = 10^7$。我们报告了插入所有边后每种数据结构使用的总字节数。

树在我们的实验中始终使用最少的内存，因为它们恰好只使用 n 个树节点。

6.3 并行批动态更新性能

接下来，我们测量新实现的并行批动态 UFO 树和拓扑树的性能，并与最先进的并行欧拉巡回树实现 [41] 和并行 rake-compress 树实现 [32]进行比较。我们在两种实现中都使用了来自 [32] 的批动态三元化实现 RC 树和我们新实现的拓扑树。由于据我们所知，目前还不存在这些数据结构的高效批量并行算法或实现，因此本节不将其与链接切割树或伸展顶树进行比较。我们使用与顺序更新速度实验相同的合成输入和真实世界输入，但将每个输入划分为批次大小为 $k = 10^6$的更新批次。我们通过随机排列输入树的边，并将边划分为固定批次大小的组来生成批次。我们首先

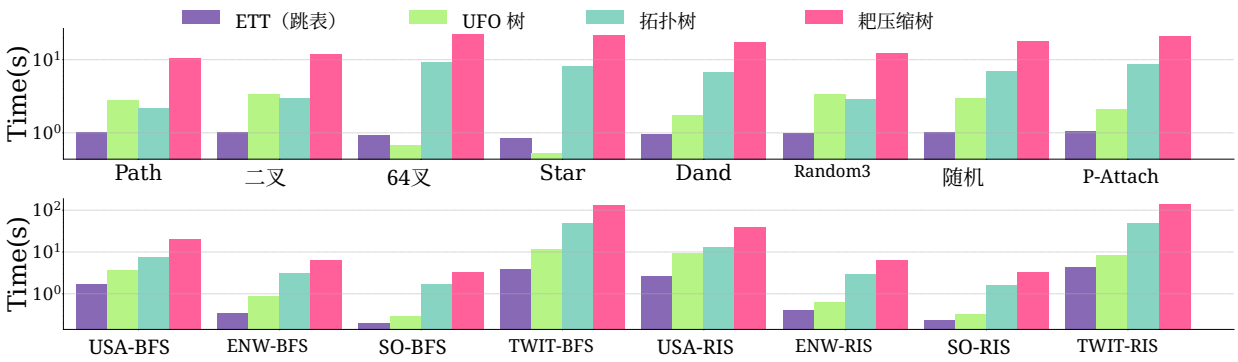

图 8. 我们的**并行**动态树数据结构更新速度实验结果。所有实验的批次大小均固定为 $k = 10^6$。顶行展示了多种合成树上的结果，树的结构为 $n = 10^7$。底行展示了真实世界图数据集的广度优先搜索森林和随机增量生成森林上的结果。我们报告了按随机顺序插入所有边批次、再删除所有边批次所需的总时间。

分批插入所有边，然后再分批删除所有边。图 8展示了结果。

我们观察到，在并行批处理动态设置中，ETT 在我们的各类输入上平均而言是最快的实现。ETT 速度出众的主要原因是，其并行算法比基于树收缩的算法简单得多，而且非常适合并行和阶段并发设置。具体而言，更新算法仅需执行少量常数个并发操作阶段，并且所有节点都预先分配好了，因此更新不需要额外的内存分配。

尽管复杂度更高，UFO 树的表现并不比 ETT 差太多，并且能够支持更多种类的查询。UFO 树的平均速度仅比 ETT 1.95× 慢，在低直径输入上的速度甚至可快 1.62× 。我们发现，拓扑树在某些输入上的表现也很好，但在需要三元化的输入上会受到三元化开销的影响。最后，RC 树在所有输入上的表现都不尽如人意，而且在需要三元化的输入上也会受到三元化开销的影响。RC 树的平均速度比 UFO 树 8.89× 慢，最坏情况下则慢 40× 。我们还在附录 D.3中展示了并行实现的直径扩展结果。

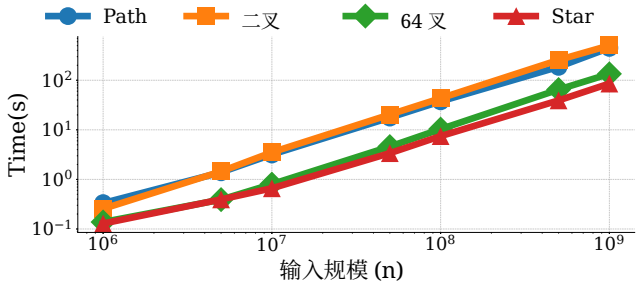

图 9. 在合成输入上，批次大小为 $k = 10^6$ 且 n 各异时，构建（并销毁）UFO 树所需的总时间。

扩展到超大规模输入。 最后，我们首次展示了动态树在十亿规模输入上的实验，并说明了随着输入规模增大，并行 UFO 树的性能如何扩展。我们报告了使用固定批次大小 $n - 1$ 插入边并删除 $n - 1$ 条边所需的总时间，树中的批次大小为 $k = 1M$，变量为 n。结果见图9。所有这些工作负载都在几百秒内完成。值得注意的是，在我们的实验中，只有 UFO 树能够在我们的机器上处理所有输入规模。其他实现则无法做到这一点：例如，并行 ETT 处理十亿规模输入时发生段错误，而并行 RC 树则在我们的机器上发生内存溢出。

7 结论

我们介绍了 UFO 树，这是一种新型并行批量动态树，支持动态树所有已知的查询功能。我们证明，我们的算法在理论和实践中都具有很高的工作效率；当输入树的直径较小时，其性能渐近优于大多数现有算法。实验结果表明，UFO 树能够轻松扩展至十亿级输入，并且是支持广泛查询功能的最快动态树实现。未来，我们计划研究树压缩技术，以设计空间效率更高的动态树实现。

致谢

本研究得到美国国家科学基金会资助，项目编号为 CCF-2403235 和 CNS-2317194。我们感谢匿名评审提供的宝贵意见。

参考文献

[1] Umut Acar 和 Daniel Anderson。2020。通过变化传播实现并行批量动态树。载于欧洲算法研讨会（ESA）。[2] Umut A. Acar、Daniel Anderson、Guy E. Blelloch 和 Laxman Dhulipala。2019。并行批量动态图连通性。载于 ACM算法与体系结构并行性研讨会（SPAA）。ACM。[3] Umut A. Acar、Guy E. Blelloch、Robert Harper、Jorge L. Vittes 和 Shan Leung Maverick Woo。2004。静态算法动态化，并应用于动态树和历史独立性。载于 ACM-SIAM离散算法研讨会（SODA）。531–540。[4] Umut A. Acar、Guy E. Blelloch 和 Jorge L. Vittes。2005。动态树中变化传播的实验分析。载于第七届算法工程与实验研讨会及第二届分析算法学与组合学研讨会论文集（ALENEX /ANALCO）。41–54。[5] Daniel Anderson。2023。并行批量动态算法：动态树、图与自调整计算。博士论文。卡内基梅隆大学。[6] Daniel Anderson 和 Guy E Blelloch。2024。确定性、低跨度且工作高效的并行批量动态树。载于 ACM算法与体系结构并行性研讨会（SPAA）。247–258。[7] N. S. Arora、R. D. Blumofe 和 C. G. Plaxton。2001。多道程序多处理器的线程调度。《计算系统理论》（TOCS）34，2（2001年4月1日）。[8] Vincenzo Auletta、Domenico Parente 和 Giuseppe Persiano。1996。在树中实现资源最优放置的动态与静态算法。《理论计算机科学》165，2（1996），441–461。[9] Abseil C++ 作者。 [无日期。]。abseil.io。abseil.io。在线；访问于2024年9月。[10] Lars Backstrom、Dan Huttenlocher、Jon Kleinberg 和 Xiangyang Lan。2006。大型社交网络中的群体形成：成员资格、增长与演变。载于第12届 ACM SIGKDD知识发现与数据挖掘国际会议论文集（美国宾夕法尼亚州费城）（KDD ’06）。计算机协会，纽约州纽约市，美国，44–54。[11] Guy E. Blelloch、Daniel Anderson 和 Laxman Dhulipala。2020。ParlayLib——面向共享内存多核机器的并行算法工具包。载于第32届 ACM算法与体系结构并行性研讨会论文集（虚拟活动，美国）（SPAA ’20）。计算机协会，纽约州纽约市，美国，507–509。[12] Guy E. Blelloch 和 Andrew C. Brady。2025。每次更新仅需常数工作量的并行批量动态极大匹配。arXiv:2503.09908 [cs.DS][13] Guy E. Blelloch、Jeremy T. Fineman、Yan Gu 和 Yihan Sun。2020。二叉分叉模型中的最优并行算法。载于 ACM算法与体系结构并行性研讨会（SPAA）。89–102。[14] Robert D. Blumofe 和 Charles E. Leiserson。1998。多线程计算的空间高效调度。《SIAM计算杂志》27，1（1998）。[15] P. Boldi 和 S. Vigna。2004。webgraph框架 I：压缩技术。载于第13届万维网国际会议论文集（美国纽约州纽约市）（WWW ’04）。计算机协会，纽约州纽约市，美国，595–602。[16] Siu-Wing Cheng 和 Moon-Pun Ng。1996。动态树中的同构测试与对称性显示。《SODA》，第96卷。202–211。[17] Thomas H. Cormen、Charles E. Leiserson、Ronald L. Rivest 和 Clif- ford Stein。2009。《算法导论》（第3版）。麻省理工学院出版社。[18] Quinten De Man、Laxman Dhulipala 和 Kishen N. Gowda。2025。《用于单链接聚类的全动态并行算法》。载于第37届ACM算法与体系结构并行性研讨会论文集（美国俄勒冈州波特兰）（SPAA ’25）。计算机协会，美国纽约州纽约市，397–412。doi:10.1145/ 3694906.3743315[19] Laxman Dhulipala、Xiaojun Dong、Kishen N. Gowda 和 Yan Gu。2024。《用于树状图计算和单链接聚类的最优并行算法》。载于第36届ACM算法与体系结构并行性研讨会论文集（法国南特）（SPAA ’24）。计算机协会，美国纽约州纽约市，233–245。doi:10.1145/3626183.3659973[20] Laxman Dhulipala、David Durfee、Janardhan Kulkarni、Richard Peng、Saurabh Sawlani 和 Xiaorui Sun。2020。《并行批处理动态图：算法与下界》。载于ACM-SIAM离散算法研讨会（SODA）。[21] Greg N. Frederickson。1985。《用于在线更新最小生成树的数据结构及其应用》。SIAM计算杂志14（1985），781–798。[22] Greg N. Frederickson。1997。《用于动态双边连通性和k棵最小生成树的双向数据结构》。SIAM计算杂志26，2（1997），484–538。[23] Greg N. Frederickson。1997。《一种用于动态维护有根树的数据结构》。算法杂志24，1（1997），37–65。[24] Matteo Frigo、Charles E Leiserson 和 Keith H Randall。1998。《Cilk-5多线程语言的实现》。载于ACM SIGPLAN 1998年编程语言设计与实现会议论文集。212–223。[25] Mohsen Ghaffari 和 Jaehyun Koo。2025。《具有最坏情况保证的并行批处理动态核分解》。载于ACM算法与体系结构并行性研讨会（SPAA）。[26] Mohsen Ghaffari 和 Anton Trygub。2024。《并行动态极大匹配》。载于 ACM算法与体系结构并行性研讨会（SPAA）。[27] Joseph Gil、Yossi Matias 和 Uzi Vishkin。1991。《迈向近似常数时间并行算法理论》。载于 *[1991]* 第32届计算机科学基础年度研讨会论文集。IEEE计算机学会，698–710。[28] Yan Gu、Julian Shun、Yihan Sun 和 Guy E Blelloch。2015。《一种自顶向下的并行半排序算法》。载于第27届ACM算法与体系结构并行性研讨会论文集。24–34。[29] Monika Rauch Henzinger 和 Valerie King。1995。《每次操作耗时为多对数时间的随机化动态图算法》。载于ACM计算理论研讨会（STOC）。519–527。[30] Jacob Holm、Kristian De Lichtenberg 和 Mikkel Thorup。2001。《用于连通性、最小生成树、双边连通性和双连通性的多对数时间确定性全动态算法》。ACM学报48，4（2001），723–760。[31] Jacob Holm、Eva Rotenberg 和 Alice Ryhl。2023。《伸展顶树》。载于2023年算法简洁性研讨会（SOSA）。305–331。doi:10.1137/1.9781611977585.ch28[32] Humza Ikram、Andrew Brady、Daniel Anderson 和 Guy E Blelloch。2025。《动态树上的并行批量查询：算法与实验》。载于第37届ACM算法与体系结构并行性研讨会论文集。525–539。[33] Bruce M Kapron、Valerie King 和 Ben Mountjoy。2013。《最坏情况下以多对数时间实现的动态图连通性》。载于第24届ACM-SIAM离散算法年度研讨会论文集。SIAM，1131–1142。[34] Haewoon Kwak、Changhyun Lee、Hosung Park 和 Sue Moon。2010。《Twitter是什么：社交网络还是新闻媒体？》。载于第19届万维网国际会议论文集（美国北卡罗来纳州罗利）（WWW ’10）。计算机协会，美国纽约州纽约市，591–600。

美国卡罗莱纳州）（WWW '10）。计算机协会，美国纽约州纽约市，591–600。[35] Daan Leijen、Benjamin Zorn 和 Leonardo De Moura。2019。《Mimalloc：空闲列表分片的实际应用》。载于亚洲编程语言与系统研讨会。Springer，244–265。[36] Gary L Miller 和 John H Reif。1989。《并行树收缩第1部分：基础》。计算研究进展5（1989），47–72。[37] OpenStreetMap贡献者。2017。获取的行星数据转储：https://planet.osm.org 。https://www.openstreetmap.org。[38] Ashwin Paranjape、Austin R. Benson 和 Jure Leskovec。2017。《时序网络中的模体》。载于第十届ACM网络搜索与数据挖掘国际会议论文集（英国剑桥）（WSDM '17）。计算机协会，美国纽约州纽约市，601–610。[39] Tomasz Radzik。1998。《支持子树内操作的动态树实现》。实验算法学杂志（JEA）3（1998），9–es。[40] Daniel D. Sleator 和 Robert Endre Tarjan。1983。《一种动态树数据结构》。计算机与系统科学杂志26，3（1983），362–391。[41] Thomas Tseng、Laxman Dhulipala 和 Guy Blelloch。2019。《批处理并行欧拉巡游树》。载于算法工程与实验（ALENEX）。92–106。[42] Tom Tseng、Laxman Dhulipala 和 Julian Shun。2022。《并行批处理动态最小生成森林与动态凝聚图聚类的效率》。载于第34届ACM算法与体系结构并行性研讨会论文集（美国宾夕法尼亚州费城）（SPAA '22）。计算机协会，美国纽约州纽约市，233–245。doi:10.1145/3490148.3538584[43] Yiqiu Wang、Shangdi Yu、Yan Gu 和 Julian Shun。2020。《用于最近点对问题的并行批处理动态数据结构》。CoRR（2020 2010 02379 2010 02379）。arXiv: 。[44] 2013 https://arxiv.org/abs/ 。Christian Wulff-Nilsen。。《更快的确定性全动态图连通性》。载于第24届ACM-SIAM离散算法年度研讨会论文集。SIAM，1757–1769。[45] Rahul Yesantharao、Yiqiu Wang、Laxman Dhulipala 和 Julian Shun。2021。《并行批处理动态 k-d树》。arXiv预印本arXiv:2112.06188（2021）。

A 其他相关工作

A.1 三元化

由于拓扑树和耙压缩树仅针对常度数树定义，因此三元化必须先应用于输入树，然后才能在其上构建动态树。三元化是一种标准技术，可将任意度数的树转换为最大度数为≤ 3的树。示例如图10所示。

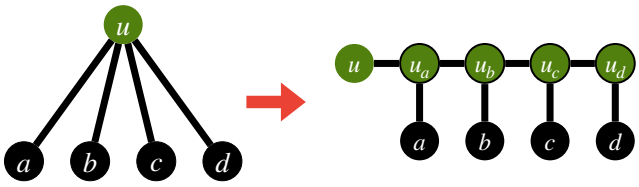

图10. 树中的某个顶点经过三元化的示意图。

对输入树 T应用三元化时，每个度为 $d(v) > 3$的顶点 v都会被替换为一条由 $d(v)$ 个顶点组成的路径，其中每个顶点的度为 ≤ 3，并与原顶点相邻的顶点相邻。这条路径称为 v 的三元化路径。对于一棵有 n 个顶点和 $n-1$ 条边的树，度为 d的顶点至多有 n/d 个。由于三元化会将这些顶点分别转换为一个度为 1 的顶点，并使其与一条长度为 d 的路径相邻（另增加d 个顶点），因此树中至多会增加 $d \times \frac{n}{d} = n$ 个顶点，最终得到一棵至多有 $2n$ 个顶点和 $2n-1$ 条边的树。这一论证可应用于森林中的每棵树，从而得出三元化后的树的规模具有线性界。

对于拓扑树，更新和查询的处理方式如下：通过连接操作添加一条边，只需在将该顶点三元化后生成的路径上再添加一个顶点即可。通过切断操作删除一条边，可以按以下方式进行。假设在顶点 u的三元化路径中，顶点u_1、 u_2、和 u_3 彼此之间通过 1 条边连接，同时通过虚边相连，并通过实边与 u 的实际邻居相连（需要注意的是， u 可能与某个同样经过三元化的顶点 v 相连，此时 u_i 会与两个顶点各自的三元化路径上的某个顶点相连）。如果我们切断与 u_2关联的实边，就将其从路径中移除：先断开它与 u_1 和 u_3的连接，再连接 u_1 和 u_3。

查询的处理方式取决于具体的查询类型。例如，连通性查询无需进行额外修改。然而，路径权重和或最大权重查询等查询要求为三元化路径上的边赋予适当的 $\perp$ 元素作为权重。构造拓扑树时，用户必须提供这些设定。

我们注意到，在批动态设置中也可以高效地维护此映射表 [32]。然而，三元化不会影响数据结构的渐近开销，但在实践中可能会大幅增加空间和时间开销。

B 详细分析

B.1 链接切割树的新分析

在此我们证明定理B.1，该定理指出，动态树操作的开销上界为输入树直径的平方。我们针对原始动态树数据结构证明这一点 [40], , 该结构使用伸展树，且此前已证明其具有 $O(\log n)$摊销开销，每次操作。我们的证明也应适用于其他链剖分树变体，例如每次操作的最坏情况开销为 $O(\log n)$ 的变体 [40]。我们假设读者熟悉这种数据结构的细节，否则请参阅原始论文。将这一新结果与上述已有结果结合，可得出链剖分树操作的开销上界为 $O(\min\{\log n,\ D^2\})$，分别适用于摊销情况或最坏情况。

定理 B.1. 链接切割树中的操作需要 $O(D^2)$ 最坏情况时间，其中 D 为输入树的直径。

证明. 链接切割树中所有操作的开销与暴露操作的开销成正比，此外还需加上拼接操作或分割操作的开销。每个拼接操作或分割操作的最坏情况时间为 $O(D)$，因为任一路径中至多有 D 个元素，因此伸展树操作的最坏情况时间为 $O(D)$。出于同样的原因，拼接操作的最坏情况时间为 $O(D)$。一次暴露操作最多调用拼接操作、拼接操作或分割操作 D 次，因为被暴露路径的长度至多为 D。因此，暴露操作的最坏情况时间为 $O(D^2)$。□

B.2 定理 3.1 的证明

引理 B.2。 在拓扑树的任意一层 $l > 0$，簇的数量至多为该层$l-1$簇数量的5/6 倍。

证明。 考虑由 N层的 $l-1$ 簇构成的树。我们将证明至少有 $N/6$ 个组合，因此第 l层的簇数量至多为 $5/6 \cdot N$。

为简化分析，我们通过折叠度为 1 和度为 2 的顶点组成的链来构造一棵新树。具有两个端点的链会被一条边取代，而具有一个端点的链则会被一个叶节点取代。由此得到的树仅包含度为 3 的节点和叶节点。在这样的树中，叶节点的数量至多比度为 3 的节点多 2。因此，如果这棵新树有 N' 个节点，那么至少有 $N'/2+1$ 的节点是叶节点。

现在，假设每个叶节点都对应原树中的一个度为 1 的顶点。由于每个叶节点的邻居都必须是度为 3 的顶点，并且假设我们不处于有 4 个节点的星形图这一退化情形（在这种情形下结论显然成立），因此至多两个叶节点可以共享同一个邻居。这意味着至少一半的叶节点可以与其邻居组合。由于至少有 $N'/2+1$ 个叶节点，因此至少有 $N'/4$ 种组合。

接下来，我们重新引入被折叠的度为 2 的顶点（原始链），以完成论证。首先，考虑那些原本是单个度为 2 的顶点、后来被一条边替代的链。请注意，这些顶点不能参与组合。由于每个这样的顶点都必须有两个度为 3 的邻点，这类顶点的数量至多比度为 3 的顶点少一个。而且，由于度为 3 的顶点数量少于度为 1 的顶点数量，这些度为 2 的顶点至多占新树中顶点的 1/3。因此，由于在其余至少 2/3 的顶点中至少出现 $N'/4$ 种组合，新树中至少出现 $N/6$ 种组合。

接下来，考虑那些被一条边替代、但包含两个或更多度为 2 的顶点的较长链。重新引入这些链不会消除已经计数的任何组合。此外，由于极大性条件，每条此类链中至少有 1/3 的顶点仍将参与一个组合。因此，组合数与顶点数的总体比率仍至少为 1/6。

现在考虑被单个叶节点替代的链。这些链由零个或多个度为 2 的顶点后接一个度为 1 的顶点组成。对于那些不与另一个叶节点共享邻居的叶节点，重新引入相应的链后，其组合数至少为顶点数的 1/3，因此不会使组合数与顶点数的比率降至 1/6 以下。对于共享同一邻居的叶节点对，重新引入两条链可能会消除一个此前计数的组合。然而，由于每条新链中至少有一个组合，我们可以重新分配较短链中的一个组合，使其计入被移除的组合。较长链相对于其规模仍至少包含 1/3 的组合，而这至少是两条链中顶点总数的 1/6。因此，组合数与顶点数的总体比率仍至少为 1/6，从而完成证明。□

前一个引理立即给出了拓扑树高度和空间的上界。

引理 B.3。 对于具有 n 个顶点的树，其拓扑树的层级数为 O（对数 n）。
证明。 由于引理 B.2 证明了每一层至多包含前一层簇数的5/6，因此层级总数至多为 对数$_{6/5}\, n$。 □

引理 B.4。 对于具有 n 个顶点的输入树，拓扑树的空间使用量为 $O(n)$。
证明。 根据引理B.2，我们知道每一层有 $(1/6)$ 的簇发生收缩。因此，簇的总数的上界为几何级数之和 $\sum_{i=0}^{\infty} n \cdot (5/6)^{\,i} \le 6n$。由于每个簇至多使用 $O(1)$ 空间，结论得证。 □

引理 B.3 和 B.4共同完成了定理 3.1的证明。

B.3 顺序拓扑树的更新效率证明

引理 B.5。 在拓扑树中进行边插入或删除期间，每一层都有 $O(1)$ 个根簇。

引理 B.6。在拓扑树中插入或删除一条边时，每一层会有 $O(1)$ 个簇被删除。

现在我们给出引理 B.5 和 B.6的证明。为了证明更新所需时间为对数时间，我们需要证明在一次更新过程中，特定层上的根簇数量和已删除簇数量都是常数个。我们的证明深受 Anderson [5]对耙压缩树中更新的分析启发，也与之有些类似。为方便起见，我们在此重述这些引理。

证明采用逐层方法，并表明每一层至多添加常数个根簇，同时其中常数比例的根簇会被收缩。这一关键事实意味着下一层有 $O(1)$ 个根簇。

为便于表述，我们将根簇受影响 簇。我们选择使用 [5], 中采用的相同术语，因为“根簇”和“受影响簇”这两个概念似乎分别在拓扑树和耙压缩树中发挥相同的作用。我们还将更新之前通过收缩形成的拓扑树所诱导的森林称为 T，将更新之后诱导的森林称为 T'。要指称收缩/更新过程中特定轮次 i 的森林，我们将使用记号 T_i 和 T'_i。

现在，我们为拓扑树定义受影响状态和受影响状态传播的概念。

定义 1（受影响簇）。 T' 中的拓扑树簇若满足以下任一条件，则称为受影响簇：
1. 该簇至少有一个子受影响簇。2. 在上一轮收缩中与该簇合并的簇不存在。3. 该簇在本轮未收缩，且与一个可以和它一起收缩的受影响簇相邻。4. 该簇是链接/切割操作指定的两个初始簇之一。

此定义的目的是为更新算法执行期间根簇的数量提供一个上界。我们将证明，更新期间的根簇集合是受影响簇集合的子集，因此，只要限定受影响簇的数量，就能同时限定根簇的数量。

在定义 1中，第一种情况同时涵盖了层 i 由层 $i-1$ 的根簇与层 i的根簇组合而成的根簇，$i-1$ 这些根簇是层 $i-1$ 的根簇的扩展，且未与任何对象合并。第二种情况涵盖了通过移除祖先步骤创建的根簇。请注意，对于这些簇，移除

祖先步骤会删除它们所簇集的簇。定义中的第三种情况涵盖了非根簇被迫与根簇合并以保持极大性的步骤。最后，最后一种情况涵盖了受到影响的最初 2 个簇。

现在，我们将定义受影响状态如何从一个受影响簇沿拓扑树“传播”到一个未受影响的邻居。其背后的想法是，尽管受影响簇的定义是静态的（即任何符合上述定义的簇都会变为受影响簇），但我们可以将层级 $i+1$ 上新出现的受影响簇（它们并非因定义1的规则（1）而受影响）与层级 $i+1$上通过规则(1) 产生的受影响簇联系起来（这些是由层级 i上的根簇收缩而产生的根簇）。在某种意义上，我们现在可以认为，这些“新”根簇是由于层级 i上的受影响簇而产生的。归根结底，这有助于说明在相邻2 个层级之间，有多少个簇会变成“新”根簇，并且该数量有一个常数上界。

类似的想法也曾用于文献 [5], 中 RC 树的相应证明，但我们的定义必须考虑到这样一个事实：与 RC 树不同，拓扑树中一个簇的邻居在某一层发生收缩后不一定保持不变。因此，我们将第 i 轮定义为从第 i 层的收缩开始，到第 $i+1$层的收缩开始之前所发生的所有操作。相应地，影响传播发生在第 i 层的收缩完成、且第 $i+1$ 层拓扑树中的簇/顶点确定之后（见图11）。现在，我们通过下面的引理来定义影响如何传播，该引理阐述了受影响簇向未受影响簇传播影响的性质。

为澄清起见，在下面的引理中，我们称簇n在 T_{i}中不存在，是指它是 T_i'中的受影响簇。

引理 B.7. 如果一个未受影响的簇 c 在第 i轮变为受影响（即在第 $i+1$轮受影响），且 c并非通过链接/切割操作变为受影响，那么在第i 轮，它有一个受影响邻居 n，并且以下条件之一成立：

1. n 在 T_i 中不存在，并且与 c 进行收缩的簇在 T_i 中被删除。*2.* n和 c 可以合并，且 n 在 T_i中没有进行收缩。

证明。 考虑一个在第 i 轮“变为受影响”的簇 c （即它在第 $i+1$轮是受影响簇/根簇）。显然，在第

i；否则，它就会完成操作并保持不受影响。

由于 c 没有任何受影响的后代（我们不考虑这种情况，因为此过程不会向层 $i+1$添加受影响簇），且它在第 i轮变为受影响，因此它必定是通过定义1中的规则（2）或规则（3）变为受影响的。规则（3）要求相关簇与一个受影响簇相邻。因此，如果 c 是通过规则（3）变为受影响的，那么影响传播的情况（2）（本引理）成立。

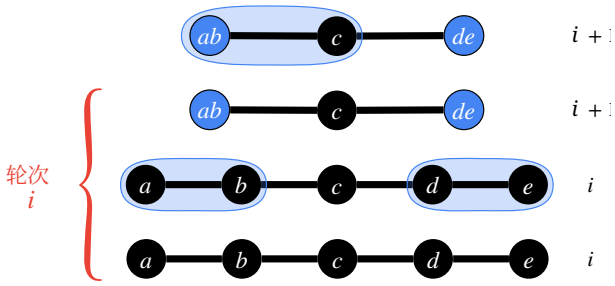

图 11. 构成一轮 i 收缩操作的步骤。

反设 c 是通过规则（2）变为受影响的，即 c 由于它此前合并的簇不再存在而变为受影响，且它不与任何受影响簇相邻。由于c 在第 i轮之前尚未受影响，因此它必定是通过与 T 中相同的收缩形成的，因此在 T_i和 T_{i}中必定具有相同的邻居和度。由于它具有相同的邻居，且它收缩时合并的邻居在 T_i'中已不存在，因此它必定与一个受影响邻居相邻，而该邻居取代了它在 T_i中收缩时合并的簇。这个邻居必定是受影响的，因为任何不在 T_i 中的簇都受影响，这就导出了矛盾。因此，本引理中的（1）或（2）成立。□

引理 B.7 说明层 i 的受影响簇如何生成层 $i+1$的受影响簇。现在我们将证明，每轮中新增的受影响簇至多为常数个。令 A^i 表示第 i轮的受影响簇集合。

引理 B.8. 在拓扑树的层 0，一次更新至多引入10 个受影响簇，即 $A^0 \le 10$。

证明。 通过规则（4），link/cut 操作指定的初始 2 个簇会受到影响。在 T_0中，它们各自与至多1 个簇收缩；根据引理 B.7 的规则（2），这些簇也会受到影响。随后，每个通过规则（2）受到影响的簇都可根据引理 B.7的规则（3）使至多另外两个簇受到影响；而最初因 link/cut 操作受到影响的簇，各自至多能使1 个其他簇受到影响（1 个邻居通过规则 4受到影响，另一个则通过规则2受到影响）。通过规则（3）新增的簇不会继续传播受影响状态，因为它们没有与其他簇合并，也不与任何可能在 T_i中与它们合并的簇相邻。因此，$|A^0| \le 2+2+2+4$，即 $|A^0| \le 10$。□

A^0 中的这 10 个受影响簇都将被视为受影响状态的起始簇 。基于此，我们将受影响森林划分为互不相交的子组件，每个子组件存储从一个起始簇扩展出的受影响簇。由于此类簇最多有 10 个，因此任何受影响森林中最多有 10 个子组件。我们将从 j^{th} 起始簇出发的受影响簇集合表示为 A^i_j。

现在，我们定义前沿簇的概念。前沿簇是指受影响森林中能够将影响传播至未受影响簇的簇，定义如下：

定义 2（前沿簇）。 前沿簇是 T_i' 中任何与未受影响簇相邻的受影响簇。

请考虑为什么这些组件必须彼此不相交：当每个起始簇传播影响时，它会朝着未受影响簇的方向传播。因此，如果两个起始簇相连，它们实际上会相互阻止对方在各自的子组件中传播影响。

现在我们证明，在任意子组件中新增的受影响簇数量是一个常数。下面，我们首先证明引理 B.9，这是一个中间结果，有助于在证明引理 B.10时进行引用。引理 B.10 关键性地证明了前沿簇的数量是一个常数，因此在任意子组件中可能变为受影响状态的簇数量也是一个常数（我们将在引理 B.11中对其进行形式化）。

引理 B.9。 任何度为 3 的受影响簇至少有1 个受影响邻居。

证明。 由于度为 3 的簇无法通过切断操作形成（因为我们的度上界为3），因此它必然是通过定义1中的规则（4）受到影响，或是被相邻簇影响（引理 B.7）。无论是哪种情况，根据影响状态的定义，它都有 ≥ 1 个受影响邻居。 □

引理 B.10。 在第 i轮，受影响森林的每个连通分量至多有 2 个前沿簇。

证明。 我们采用归纳法。在 A_j^0 时，这显然成立，因为每个子组件只包含 1 个受影响簇。

考虑第 i轮的一个子组件。假设该子组件只有 1 个前沿簇。如果该前沿簇的度为 1 ，它至多可以扩展到 1 个未受影响的邻居；扩展后，它便不再是前沿，因此前沿的数量保持不变。如果它的度为 2 或 3，则它只有 2 个未受影响的邻居（度为 B.9 的情形见引理3 ），并且至多可以扩展到这 2 个未受影响的邻居。如果它扩展到 1 个邻居，那么我们新增了 1 个前沿，而前沿的数量为 ≤ 2。如果它扩展到两个相邻的未受影响邻居，那么原来的前沿簇就不再是前沿，前沿簇的数量仍为 ≤ 2。

现在考虑一个包含 2 个前沿簇的子组件， c_1和 c_2。由于划分方案保留了受影响子树的底层连通性，我们注意到， c_1 和 c_2 都至少与 1 个受影响簇相邻（即将其连接到受影响组件子树的簇）。

假设某个前沿簇的度为 2 （在这种情况下，前沿簇的度不可能为 1 ，因为它只有 1 个受影响邻居，因此不能成为前沿簇）。这个度为 2的簇有 1 个受影响邻居，最多会扩散到1 个新的未受影响簇（根据引理 B.7）。当它扩散到相邻簇时，它的所有邻居都会变为受影响

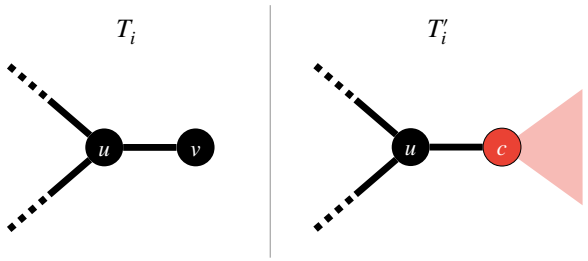

图 12. 图示了影响从前沿簇 c传播到度为 3 的簇 u 的情形。若发生这种情况，那么 c 必定是唯一的前沿簇。

，因此它不再是前沿簇。如果 c_1 和 c_2都这样做，那么前沿簇的数量仍至多为 2。

现在假设我们有一个度为 3的前沿簇。按照 B.7中所述的规则传播影响，显然存在一个问题：如果 c_1 和 c_2 都是度为 3 的前沿簇，那么如果它们各自都向两个邻居传播影响，前沿总数就会增加 2。我们将证明这是不可能的。

考虑度为 3 的簇首次受到影响的情况。它与之聚合的度为 1 的簇，要么现在被某个受影响簇取代（引理 B.7 的规则（1）），要么它被相邻的受影响度为 1 的簇传播了受影响状态（引理 B.7 的规则（2））。后一种情况与我们无关，因为我们最初只有 1 个前沿簇。同样，我们也不必考虑它在 T_i 中通过链接/切割而与之收缩的簇受到影响的情况，因为我们之前已经证明，起始簇传播受影响状态的方式彼此不相交。现在假设某个度为 3 的簇 u 受到影响，是因为一个前沿簇（不失一般性，设为 c_1）取代了某个度为 1 的簇v，而 u 此前在 T_i 中曾与该簇聚合（见图 12）。我们知道，收缩操作会保持底层子树的连通性，并且该子组件中所有能通过 v 或 v 在 T_i 中的祖先到达 u 的簇，都必定已收缩到被 c_1 取代的度为 1 的簇中。因此，该子组件中所有能通过 c_1 在 T_i'中到达 u 的簇都必定受到影响。下面进一步解释这一理由：由于这些簇都曾彼此收缩，并与起始簇一起在 T_i中形成一个度为 1 的簇，如果它们再次以相同方式彼此收缩，那么它们至少与一个受影响簇发生了收缩，因此 c_1 现在是一个度为 1 的前沿簇，而这种情况我们不予考虑。如果发生的是另一系列收缩，那么由于这些簇都曾彼此收缩，它们所收缩到的簇必然已被取代，因此现在都受到了影响。因此，当这个度为 3 的簇成为前沿簇时，它必定是其子组件中唯一的前沿簇，所以我们又回到了只有 1 个前沿簇的情况。

这并不是说我们不能有一个包含 2 个前沿簇的子组件，其中 1 个是度为 3 的簇。度为 3 的簇可以成为前沿簇，并在一轮中将受影响状态传播给它的一个邻居，只在该分支中传播受影响状态，同时仍保持为前沿簇。然而，如上所示，度为 2 的顶点无法传播受影响状态，从而使前沿簇的数量变为 > 2；而且，如果不先使子组件中的所有其他顶点都受到影响，我们就无法加入一个度为 3 的受影响簇。此外，当度为 3 的簇将受影响状态传播到它的另一个分支时，它就不再是前沿簇，因此前沿的数量仍为 ≤ 2。□

引理 B.11。 对于任意子组件 A_j^i, $|A_j^{i+1}| - |A_j^i| \leq 8$。

证明。 根据引理 B.10，至多有 2 个前沿簇。根据引理 B.9，我们知道这些簇中的每一个至多有 2 个未受影响的邻居。受影响状态最多会传播到这些未受影响的邻居。由于其中一个是通过规则（1）传播的，另一个是通过规则（2）传播的，其中一个可能通过 2 B.7 的规则（2）额外增加 受影响邻居 （这些邻居在这一层没有与任何对象合并，因此无法再增加其他邻居）。因此，每轮在一个子组件中新增的受影响顶点总数至多为 $2 \times (2 + 2) = 8$ □

引理 B.12。 在任意层 i和任意子组件 j 中，受影响簇的总数至多为 48，即对于 $i \geq 0, A_j^i \leq 48$

证明。 由于任何受影响的子组件仅由 2 个前沿簇组成，因此可以将受影响子组件的其余部分视为其自身的连通子树。因此，由于必须在该子树中进行最大数量的收缩，至少有 $1/6 - 2$ 的簇必须被收缩掉。在任意 2 个层级之间，我们至多添加 8 个新的根簇。这两个事实结合起来得到以下不等式：

$$A_j^{i+1} \leq 8 + \left(\frac{5}{6}\right)(A_j^i - 2) + 2.$$

因此，由于 $A_j^0 = 1$,

$$A_j^{i+1} \leq \frac{50}{6} \cdot \sum_{r=0}^{\infty} \left(\frac{5}{6}\right)^r \leq 50.$$

□

现在我们可以完成本分析开头所述引理的证明：B.5和B.6。

引理 B.5。 在拓扑树中进行边插入或删除期间，任意层级上有 $O(1)$ 个根簇。

证明。 最多有 10个子组件，且引理B.12证明了每个子组件至多可以有 $A_j^i \leq 50$个。因此，每层受影响顶点的总数至多为 $10 \cdot 50 \leq 500$。 □

引理 B.6. 在拓扑树中插入或删除一条边时，每一层都会有 $O(1)$ 个簇被删除。

证明。我们采用简单的逐层方法来证明此引理。考虑 T' 中的第 0 层。一次连接 / 切割操作最多创建 10 个根簇，其中每个根簇都会调用移除祖先，并删除其对应的父节点（位于 T_1中）。因此，根据引理 B.2，其中有 1/6 个此前必定已经收缩，所以这些簇中至多有 5/6（即它们在 T_1 中的父节点）被删除。现在考虑 T_1处的情况。根据引理 B.11，每个子组件会新增8个簇，其中至多6个簇是通过定义1的规则（3）新增的，因此会调用移除祖先。由于有10个子组件，以这种方式新增了 ≤ 60个簇。因此，第2层被删除的父节点数量至多为前一层被删除的簇数量加60之和的 5/6。这个规律在之后的每一层都会延续，因此每层的删除数量都有常数上界：

$$\#\,\text{Deletions} \leq 60 \cdot \sum_{r=0}^{\infty} \left(\frac{5}{6}\right)^r \leq 360.$$

□

引理 B.5 和B.6 的证明现在使我们能够完成拓扑树更新算法总开销的证明：

引理 B.13. 含有 n 个顶点的拓扑树中，边插入和删除的开销为 $O(\log n)$。

证明。 在任意层级，每个根簇都可以检查其（至多 3 个）邻居并创建一个新簇，同时可能会删除一些簇。根据引理 B.5，每层的根簇数量为常数；根据引理 B.6，每层已删除簇的数量也为常数。由于拓扑树中共有$O(\log n)$层级（引理 B.3 ），一次更新的总体成本为 $O(\log n)$。 □

最后，为结束本节，我们给出一个证明，说明更新后得到的拓扑树是有效拓扑树。

引理 B.14. 在拓扑树中执行边插入或删除的算法，在更新后会得到一棵有效拓扑树。

证明。 我们将有效拓扑树定义为满足第 3.1节中所给定义的树。

首先请注意，在每次移除祖先调用之后，根簇只能与其他未被删除的簇，或因先前某一层向其邻接表插入元素而与之相邻的簇进行组合。因此，在重新聚类期间，簇只能与同一层上与其相邻的其他簇组合。此外，当且仅当某个簇与另一簇的子簇相邻，或反之时，它才会被插入另一簇的邻接表。因此，任何簇的度都不会增长到 > 3。其次，只有按照第 3.1 节列出的方式，才允许进行组合。因此，在任何一层，组合都发生在相邻簇之间，并且只能以符合定义的方式进行

拓扑树。最后，为了证明每一层的聚类都是极大的，我们将证明每个簇都满足极大性，即它要么与另一个簇组合，要么它的所有邻居都与另一个簇组合。这里，我们用 新簇 表示新拓扑树中曾在更新算法运行期间作为根簇的任何簇。新树中的其他簇称为旧簇。可以看出，任何新簇都满足极大性，因为更新算法会检查它的所有邻居，并尝试与它们组合。

对于仅与其他旧簇相邻的旧簇，它们必须满足极大性，因为它们的所有邻居与旧拓扑树（即更新前的树）中的完全相同，而且旧簇的父节点绝不会被删除。对于与一个或多个新簇相邻的旧簇 X，要么 X 与另一个旧簇组合，并且仍然如此；要么它无法与任何旧簇收缩。对于后一种情况，如果某个相邻的新簇没有与其他任何簇组合，且能够与 X组合，那么它就会组合。因此， X 满足极大性。□

B.4 定理 4.1和4.2的证明

引理 B.15。 在 *UFO* 树的任意一层 $l > 0$ ，簇的数量至多为该层$l - 1$簇数量的 5/6 倍。

证明。 该命题的证明与引理 B.2类似。想象一下，按照引理B.2中的相同方式，将树中的每条链替换为一条边或一个顶点。由此得到的树包含 N个节点（其中不含度为 2 的节点），其叶节点数量至少比其他节点多 2 个，因此至少有 $N/2 + 1$ 的节点是叶节点。在 UFO 树中，所有这些叶节点都会与其邻居组合（除非树仅有两个节点，此时引理显然成立），因为每个叶节点的邻居都是高阶节点。因此，至少会发生 $N/4$次组合。重新引入链时，适用与引理 B.2相同的逻辑。这证明至少会发生 $N/6$次组合。 □

前面的引理立即推出了 UFO 树的高度和空间界。

引理 B.16. 对于含有 n 个顶点的树，其 *UFO* 树的层数为 $O(\log n)$。
证明. 由于引理 B.15 证明了每一层的簇数至多为上一层的5/6 ，因此层数总计至多为 $\log_{6/5} n$。 □

引理 B.17. 对于含有 n 个顶点的输入树，其 *UFO* 树的空间使用量为 $O(n)$。
证明。在 UFO 树的每一层中，有 x 个簇，所需空间为 $O(x)$ 因为诱导子图是一棵树，每条边都会存储两次，而每个簇还会使用 $O(1)$ 的额外空间。结合引理 B.15可得，总空间上界为 $\sum_{i=0}^{\infty} O(n) \cdot (5/6)^i \leq 6 \cdot O(n)$。 □

引理 B.15、B.16 和B.17 完成了定理 4.1的证明。

定理 *4.2* 的证明。 考虑树中任意一条长度为 d 的极大路径（即无法再添加顶点来延长该路径）。这条路径的端点必为叶节点。考察其中一个端点 x，设其在路径上的邻居为 y。如果 y 的度为高阶或度为 1，那么 x 会与y 合并，使路径长度至少减少一。若 y 的度为 2，那么 x 会与 y 合并，或者 y 会与它的另一个邻居 z合并，而该邻居也位于路径上。这两种情况下，路径长度都会至少减少一。对路径的另一个端点也可对称地应用同样的论证。因此，在 UFO 树的每一层中，路径至少缩短两个顶点。由此，完全收缩该路径所需的层数至多为 $\lceil d/2 \rceil$。由于输入树中的最长路径长度为 D，结论得证。

□

B.5 顺序 UFO 树证明

由于 UFO 树与拓扑树的性质存在重叠，许多引理都可以用与拓扑树证明相同的推理来表述，只需做少量调整。本节中的引理B.18-B.23证明了更新算法的效率，而引理 B.24 则证明了更新算法的正确性。

为清晰起见，在下文的引理中，我们将更新前创建的 UFO 树称为 T，将更新后创建的 UFO 树称为 T'。类似地，每一轮所诱导的树分别用 T_i和 T_i'表示。

引理 B.18。 在 *UFO* 树中插入或删除一条边时，任意层上都有 $O(1)$ 个根簇。

借鉴拓扑树证明，我们采用逐层方法进行证明。

然而，在更新期间，我们不再将受影响状态视为簇的一种静态属性，而是将其视为在链接/切割过程中由初始受影响簇传播的状态。实际上，我们证明了某些根簇的存在意味着该层上其他根簇也存在，并且这涵盖了更新算法可能创建的所有根簇。我们采用不同方法，是因为 UFO 树的高阶和高扇出情形由算法处理，因此难以为 UFO 树定义受影响状态的静态定义。

在证明的其余部分，我们将略微改变更新算法中**低度**和**高阶** 这两个术语的含义——低度现在指度为≤ 3 的簇，而高阶指度为 ≥ 4的簇。

定义 3. 如果某个 UFO 树簇 c 在第 i轮开始时未受影响，在第$i + 1$ 轮开始时受到影响（即在第 i轮期间受到影响），且不属于

初始链接/切断操作，那么它必定有一个受影响邻居n，满足以下条件之一。
1. n 是低度节点，在 T_i 中不存在，并且与之合并的簇 c 已被删除。2. n 是低度节点，而 c 在 T_i 中未合并，但现在可以与 n合并。

此外，与拓扑树类似，表示更新操作端点的簇会受到影响，任何至少有一个受影响子节点的簇也会受到影响。

基于上述引理，证明余下部分的思路是，我们可以有效利用先前受影响的簇，找出哪些未受影响的簇可能变为受影响簇。这称为影响传播。它将作为根簇数量的上界，使我们能够将其限制在一个常数范围内。请注意，高阶（度为 ≥ 4）簇可能会受到影响，但它们无法将影响传播给未受影响的邻居。这是由我们的算法推得的，因为这类簇的扇出较高，或者其父节点的度为 ≥ 3，而该父节点不会被更新算法删除。影响传播的所有其他情况都与拓扑树类似。

与我们证明引理 B.5 时类似，现在我们将第 i 轮受影响顶点的集合记为 A^i。

引理 B.19. 执行初始链接/切断操作后，第 0 轮最多有10个受影响顶点，即 $A^0 \leq 10$。
证明。 对于拓扑树，该引理可由引理 B.8 推出，因为度为 ≥ 4 的顶点无法传播受影响状态，而在所有其他情况下，传播步骤都等同于拓扑树中的步骤。 □

与拓扑树一样，我们再次将 A^i 划分为互不相交的子组件，每个子组件有 10 个影响源簇。之后，我们将因受影响状态从 j^{th} 起始簇传播而变为受影响的顶点称为 A^i_j。

现在我们将再次使用前沿簇（参见定义 2 及其后续文字），说明现有受影响簇如何导致同一层的其他未受影响簇受到影响。

引理 B.20. 在受影响森林 A^i_j 的任意层 i 子组件中，前沿簇的数量为 ≤ 2。
证明。 我们注意到，在基准层，这一点已经成立，因为每个子组件只有 1 个受影响簇，因此也只有 1 个前沿。

在任何后续层 i。假设某个子组件A^i_j只有 1 个前沿簇。前沿的度为 1、或 2 的情况，可以像处理拓扑树那样处理。对于 UFO 树，度为 3 的情况稍微棘手一些。如果度为 3 的前沿有 1 个受影响邻居，可以采用类似拓扑树的方式处理。如果它没有受影响邻居（例如，它是在一个度为 4 的簇切断 1 个邻居后形成的），那么由于它之前是高阶簇，因此它只能与相邻的度为 1 的邻居合并，也就只能将受影响状态传播给这些邻居。由于这些度为 1 的簇不可能成为前沿，无论这个度为 3 的前沿如何传播受影响状态，它都会是唯一的前沿，直至它被收缩掉。度为 ≥ 4 的簇无法传播受影响状态，任何此类前沿都会保持原状，直到它合并进能够传播受影响状态的簇中。这些情况按本引理前文所示处理。

现在考虑具有2 个前沿的情况。由于 UFO 树的聚类操作保持树的底层连通性，因此这 2 个前沿 f_1 和 f_2 必须至少有 1 个受影响邻居。度为 1和 2 的情况很容易看出可按与拓扑树相同的方式处理。由于度为 3的簇仍然只与度为 1的簇合并，因此，如果一个度为 3 的簇传播受影响状态，它就是其连通分量中唯一的前沿；因此，这种情况也可按与引理 B.10类似的方式处理。最后，高阶簇无法传播受影响状态，因此，如果 f_1 或f_2 是这样的前沿，那么在其聚类为低阶簇之前，不会增加任何前沿。上述情况同样表明可以对其进行处理。由以上分类讨论可知，每个子组件都有 ≤ 2 个前沿簇。 □

引理 B.21. 对于任意轮次 i 和子组件 j，在任意两个2 轮次之间新增的受影响簇数量为 ≤ 8，即 $|A^{i+1}_j| - |A^i_j| \leq 8$。

证明。 我们自然只需考虑每个前沿都有 2 个未受影响邻居、且可以向其传播受影响状态的情况，因为相比只有一个这样的未受影响邻居，这会新增更多受影响簇。考虑一个簇可以向其相邻的两个未受影响邻居传播受影响状态的情况。它要么是一个孤立的度为 2 的前沿，要么是一个带有受影响邻居的度为 3 的前沿。传播方式可能与拓扑树不同的唯一情形是：度为 3 的簇根据引理 1 的规则，将受影响状态传播给它的两个邻居。该规则见引理 3。然而，这最多只能新增 2个受影响簇，少于其他方法可能新增的 4 个。引理的其余部分由B.11推出。 □

引理 B.22。 任意层 i以及任意子组件中 j 受影响簇的总数最多为 50 即，对于 $i \geq 0$，$A^i_j \leq 50$。证明。 证明该引理所用的数学方法与引理B.12完全相同。 □

因此，由于每个子组件最多有 48 个根簇，且共有 10 个子组件，根簇总数为 ≤ 500。

引理 B.23。 在 *UFO* 树中插入或删除一条边时，任意层被删除的簇数量为 $O(1)$。
证明。 我们采用简单的逐层方法来证明这个引理。考虑 T' 中的第0层。一次链接/切割操作最多会创建 10 个根簇，每个根簇都会调用一次删除祖先的操作，并删除其在 T_1 中相应的父节点。因此，

引理 B.15，1/6 这些簇必定此前已经收缩。因此，在最坏情况下，叶到根路径上的父节点都不是高阶节点，而这些簇中的 5/6 ，即它们在 T_1 中的父节点，都被删除了。

现在来考虑，在 T_1时会发生什么。根据引理 B.11，每个子组件会新增 8 个簇，其中至多6个簇是通过定义 (3) 的规则 1 添加的，因此称为移除祖先。由于共有10 个子组件，≤ 60 以这种方式添加了簇。因此，层 2 删除的父节点数量至多为前一层删除的簇数量总和的5/6 ，再加上 60。这一模式在向上的每一层都会继续，因此每层的删除次数都有常数上界：

$$\#\,\text{Deletions} \le 60 \cdot \sum_{r=0}^{\infty} \left(\frac{5}{6}\right)^r \le 360.$$

□

最后，我们证明更新后生成的 UFO 树是有效的 UFO 树。

引理 B.24. 在 *UFO* 树中插入或删除边的算法会生成一棵有效的新 *UFO* 树。也就是说，新树中的所有组合都是允许的组合，并且每一层的组合都是极大的。

证明。 删除若干条祖先路径后，UFO 树的各连通分量仅包含允许的簇组合。此外，更新算法的重新聚类阶段产生的每个新组合也必定是允许的组合。因此，新 UFO 树中的所有组合都是允许的。

不涉及高阶簇的组合，依据与拓扑树相同的推理，必然是极大的。涉及高阶簇的组合也同样是极大的，因为只要存在度为 1 的根簇，它就必然会与相邻的高阶簇组合。对于任何与此前仍为高阶的簇 Y X 和 Y 相邻的非根度为 1 的簇 X，它们必须组合，因为之前的树在每一层都有一个极大的组合集合。对于任何与刚刚变为高阶的簇 Y相邻的非根度为 1 的簇 X， Y 必须恰好为度为 3。算法将检查 Y 的所有邻居，并将度为 1 的邻居与其组合，因此 X 将被找到，并与 Y组合。□

B.6 批量更新算法证明

并行批动态拓扑树。现在我们证明拓扑树批量更新算法的正确性。

引理 B.25。 对拓扑树执行批量更新的算法会在更新后得到一个有效拓扑树。

证明。 注意，惰性删除簇不会影响算法的正确性，因为在任意层所检查的簇保持不变。批量更新的其余部分算法与顺序情形相对应，只是对某些操作做了并行化修改。因此，正确性论证直接源自引理 B.14 的证明。□

接下来，我们将证明引理 B.26，并进一步用它证明 5.1，即分析拓扑树中批量更新开销的主定理。

引理 B.26。 在批次大小为 k 的拓扑树批量更新过程中，每一层都有 $O(k)$ 个根簇。此外，每一层都有 $O(k)$ 个已删除簇。

证明。 批量算法生成的根簇和已删除簇集合，是对批次中任意更新逐一执行时所生成的所有簇的子集。由引理 B.5 和 B.6 可知，每次单独更新在每一层至多产生常数个根簇和已删除簇，因此每一层总共至多有 $O(k)$ 个。□

定理 5.1。 拓扑树中的批量并行更新需要 $O(k\log(1 + n/k))$ 期望工作量，且深度为 $O(\log n \log k)$ 以高概率，其中 n 为顶点数， k 为批次大小。

证明。我们将证明，拓扑树每一层所执行的工作量为 $O(x)$，其中 x 是该层的根簇和已删除簇的数量。利用这一事实和引理 B.26，工作量可立即界定为 $O(k\log n)$。我们得到更好的界 $O(k\log(1 + n/k))$，因为在某个阈值层 $l = O(\log(1 + n/k))$之上，节点总数为 $O(k)$；而在该阈值以下，各层的工作量总和为 $O(k\log(1 + n/k))$。

在算法 3 中，映射到父节点显然可以用 $O(x)$ 的工作量完成。更新邻接表时，由于每个簇的度至多为 3，因此需要插入和删除 $O(x)$ 条边，这很容易在 $O(x)$ 时间内完成。批量插入邻居集合的总工作量为 $O(x)$。同理，出于相同原因，从子节点集合中删除元素的总工作量为 $O(x)$。已知对链进行并行最大匹配所需的工作量为 $O(x)$。最后，释放 $O(x)$ 个常数大小的簇并分配 $O(x)$ 个簇，需要 $O(x)$ 的工作量。

在二叉分叉-连接模型中，所有这些步骤都可以在 $O(\log x)$ 深度内完成。因此，总深度为 $O(\log n \log k)$。□

并行批量动态 UFO 树。与引理 B.25 的证明类似，UFO 树批量更新算法的正确性可以通过与 UFO 树顺序更新算法（引理 B.24）相同的推理得出。所有未被删除的簇仍表示有效的合并，而最大匹配步骤确保我们对树重新聚簇，从而在每一层创建最大收缩。

现在我们将证明引理 B.27，这将使我们能够证明定理 5.2，该定理是分析 UFO 树中批量更新开销的主要定理。

引理 B.27. 在批次大小为 k 的 *UFO* 树中进行批量更新时，任意层上都有 $O(k)$ 个根簇，其度数之和为 $O(k)$。此外，还有 $O(k)$

任意层上已删除簇的数量为$O(k)$，其度数之和为$O(k)$。

证明。采用与引理B.26证明类似的论证，这次应用引理B.18 和B.23可知，UFO 树批量更新算法在任意层的根簇和已删除簇的数量为 $O(k)$。

当父节点被删除时，会形成根簇。由于算法只删除扇出低且度低的簇，这些簇的任何子节点都必须是或曾经是低度的（对于插入批次，在插入新边之前它们必须是低度的；对于删除批次，在删除边之后它们必须是低度的）。只有当批次中的某些边插入到根簇上时，根簇的度才会变高。这最多使总度增加 $2k$。由于所有簇的度均为常数，且总度最多增加 $2k$，根簇的总度为 $O(k)$。

对于插入批次，将被删除的簇不会用新边进行更新。因此，由于我们只删除低度簇，这些簇的度都是常数，其总度为 $O(k)$。对于删除批次，先从树的所有层中删除k 边（耗费 $O(k\log(1+n/k))$ 的工作量）。随后，所有被删除的簇都是低度簇，因此每层的总度为 $O(k)$。□

定理 5.2。*UFO* 树中的批量并行更新需要$O(\min\{k\log(1+n/k),\ kD\})$期望工作量，并具有 $O(\log n\log k)$ 深度，且以高概率成立；其中 n 表示顶点数量，D 表示输入森林的直径，k 表示批次大小。

证明。我们将再次证明，每一层的工作量为 $O(x)$，其中 x 表示该层的根簇和已删除簇的数量。根据与定理5.1证明中相同的推理，可得工作量上界为 $O(k\log(1+n/k))$。工作量上界 $O(kD)$ 也由此得出，因为高度有上界 $O(D)$。

与拓扑树批量更新算法共用的所有步骤需要$O(x)$工作量。根据引理 B.27，按端点对边进行分组的半排序需要 $O(x)$工作量，向邻居集合哈希表中添加边也需要相同的工作量。更新子节点集合哈希表需要 $O(x)$，因为有 $O(x)$ 个根簇和已删除簇。

在深度方面，与并行拓扑树不同的唯一操作，是在邻接表和子节点集合中插入和删除元素。向并行集合中批量插入或删除元素的深度，与集合大小加上待插入或删除的批量元素数成正比。具体而言，深度为 $O(\log y)$，其中 y 是集合当前的大小加上待插入或删除的元素数。这意味着，如果需要更新一个规模远大于 k 个元素的大型邻居集合或子节点集合，该步骤的深度会高于 $O(\log k)$，甚至可能达到 $O(\log n)$。简单地执行这些操作会导致深度为 $O(\log^2 n)$。

为了将深度降至 $O(\log n\log k)$，我们可以将部分批量更新推迟到批量更新结束时执行（或者在算法继续处理更高层级的同时，在后台运行这些更新）。请注意，只有当簇已经是高阶时，我们才会向大型邻居集合中插入元素。由于我们正在插入元素，该簇会继续保持高阶，其父节点也不会被删除，因此它不会变成根簇。在这种情况下，延迟更新邻居集合没有问题，因为该簇不会变成根簇，所以我们不会遍历其邻居。如果要从当前大小为 $\Omega(k)$ 的邻居集合中删除元素，由于最多只有 $O(k)$ 次删除，该簇不会变成低度，其父节点也不会被删除，因此它不会变成根簇。基于同样的原因，我们可以延迟这项更新。如果邻居集合的大小为 $O(k)$，那么深度为 $O(\log k)$，我们可以提前执行更新。对于子节点集合，我们可以将删除和插入推迟到整个重新聚类过程结束时，因为可以通过遍历父簇某个子节点的邻居来确定该父簇的子节点。□

C 查询详情

C.1 拓扑树查询

在原始论文中，Frederickson 描述了如何在拓扑树中聚合边信息，以便在最小生成树 [21]中寻找替换边。后来，Frederickson 展示了如何修改拓扑树，以便在有根输入树 [23] 中高效回答从任意给定顶点到根的路径上的最小边权查询。这里，我们描述如何在无根输入树上构建拓扑树中的路径查询 和子树查询。我们还指出，通过查找并比较包含 x 和 y的拓扑树的根，可以在 $O(\log n)$时间内回答连通性查询，该查询用于判断两个顶点 x和 y之间是否存在路径。此外，我们展示了拓扑树还可以支持此前针对耙压缩树 [4]展示的其他几种查询。

路径查询。在输入树 $T=(V,\ E)$上进行路径查询时，其中 E 中的每条边都被赋予域 D 中的一个值，查询将对 V中两个相连顶点 u和 v之间的唯一路径上的值应用一个满足结合律和交换律的函数 $f\colon \mathrm{D}^2\to\mathrm{D}$，并返回所得结果。我们假设 f可以在 $O(1)$时间内计算得出。路径查询也可以定义在顶点值上，而不是边值上；我们专注于基于边的查询，但基于顶点的查询可以采用类似方式实现。我们的算法受到 Acar 等人针对耙压缩树 [4],中的路径查询所提出方法的启发，后来由 Anderson [5]进一步阐明。

我们将拓扑树中的一元簇 定义为这样一种簇：其中恰有一条边 E 的一个端点位于该簇中。二元簇 恰有两条这样的边。簇的边界顶点 定义为簇中与这些跨越边相接的顶点。一元簇

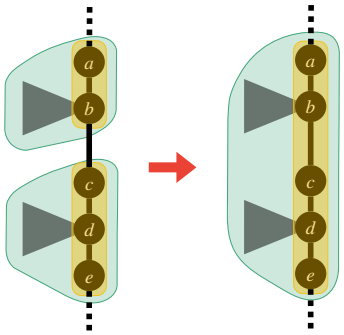

图 13。 拓扑树中二元簇的簇路径示意图，以及其簇路径如何由两个子节点的簇路径构成。

始终只有一个边界顶点，而二元簇可能有一个或两个边界顶点。簇的簇路径 定义为簇的边界顶点之间的唯一路径（见图 13）。请注意，拓扑树中的簇表示输入树的一个连通子图 T。因此，簇路径完全位于该簇内。对于一元簇或基数为1的簇，簇路径为空，只包含一个顶点，不含任何边。只有二元簇才可能有非空的簇路径。

为了支持高效的路径查询，每个二元簇都会维护其簇路径中所有边上应用 f 后的值。如果一个簇不是二元簇，或者其簇路径为空，则存储 f 的单位元。二元簇始终由两个度为 2 的簇合并而成，即 X 和 Y 其簇路径是 X 和 Y的簇路径的并集，再加上它们之间的边（见图 13）。在更新的重新聚类步骤中，新二元簇中存储的值可以通过对 $O(1)$ 时间内存储在 X、 Y中的值以及连接它们的边应用 f来计算。

为了回答顶点 u 和 v之间的路径查询，我们考虑拓扑树中包含叶簇 u 和v 的最近公共祖先 P。设包含 u 和 v的 U和 V 分别是 P的直接子节点。于是，在 U和 V 的边界顶点之间存在一条边 $e = (b_u, b_v)$：即导致 U和 V 合并并形成 P的那条边。原树中 u 和 v 之间的路径，是从 u到 b_u的路径、边 e以及从 v到 b_v的路径的串接。因此，如果我们知道从 u 和 v 分别到 U和 V的边界顶点的路径值，就可以回答该路径查询。

我们将簇 X 关于顶点 $x \in X$ 的代表路径定义为从 x 到 X的边界顶点的唯一路径。一个一元簇只有一条代表路径，而一个二元簇有两条；即使它只有一个边界顶点，我们也将其视为有两条相同的代表路径。接下来，我们将展示如何归纳地构造簇X 关于顶点 x的代表路径。从叶簇开始，x，我们归纳地计算每个父簇的代表路径，直到簇 X。如果簇 P只有一个子节点C，则代表路径保持不变。否则，我们考虑四种归纳情形，如图14所示，涉及与 C 合并形成 P的兄弟簇 S，以及连接 C和 S的边 e。

(a) 第一种情况是 C 和 P 都是一元簇。这意味着 C 与一个二元簇 S 合并。 P 的代表路径是 C 的代表路径、边 e 和 S 的簇路径的并集。

(b) 第二种情况是 C 是一元簇，而 P 是二元簇。这只有在 S 是基数为 1 的度为 3 的簇时才可能。因此， P 的两条代表路径都是 C 的代表路径与边 e 的并集。

(c) 第三种情况是 C 是二元簇，而 P 是一元簇。在这种情况下，S 必定是一个一元簇。 P 的代表路径就是 C 中不包含 e 的某个端点的那条代表路径（如果 C 只有一个边界顶点，则两条路径均可）。

(d) 第四种情况是 C 和 P 都是二元簇。这意味着 S 也必定是二元簇。 P 的一条代表路径是 C 中不包含 e 的某个端点的那条代表路径（如果 C 只有一个边界顶点，则两条路径均可）。另一条代表路径是包含 e 的某个端点的 C 的代表路径、边 e 和 S 的簇路径的并集。最后还有一种额外情况：从一个度为 3 的叶簇开始时，其父节点是一个二元簇，且两条代表路径都为空。

利用这种归纳方法，以及存储的、在每个二元簇的簇路径上应用 f 所得到的值，我们可以计算 f 在相对于 u 的 U 代表路径上的值，以及 V 在相对于 v 的代表路径上的值，从而回答路径查询。由于每个归纳步骤都能在 O(1) 时间内完成，且树的高度为 $O(\log n)$，因此我们有以下引理：

引理 C.1。 在具有 n 个顶点的输入树上，拓扑树中的路径查询需要 $O(\log n)$ 时间。

子树查询。 子树查询定义为一种操作：给定输入树 T，其中每个顶点都被赋予域 D 中的一个值，返回某个结合且交换函数 $f\colon \mathrm{D}^2 \to \mathrm{D}$ 在以相对于父顶点 p 的 v 为根的子树中所有顶点的值上应用后的结果。或者，也可以用子树中的边来定义子树查询。我们着重讨论前一种定义，但要指出，后一种定义也可以用类似的方法实现。再一次，我们的算法借鉴了 Acar 等人针对耙压缩树中的子树查询所提出的算法 [4]，该算法后来由 Anderson [5] 作了进一步阐释。

为高效支持子树查询，拓扑树中的每个簇都会维护将 f 应用于簇中所有顶点后得到的值。树中的每个叶节点都包含 f 的单位元。在更新期间重新聚类时，

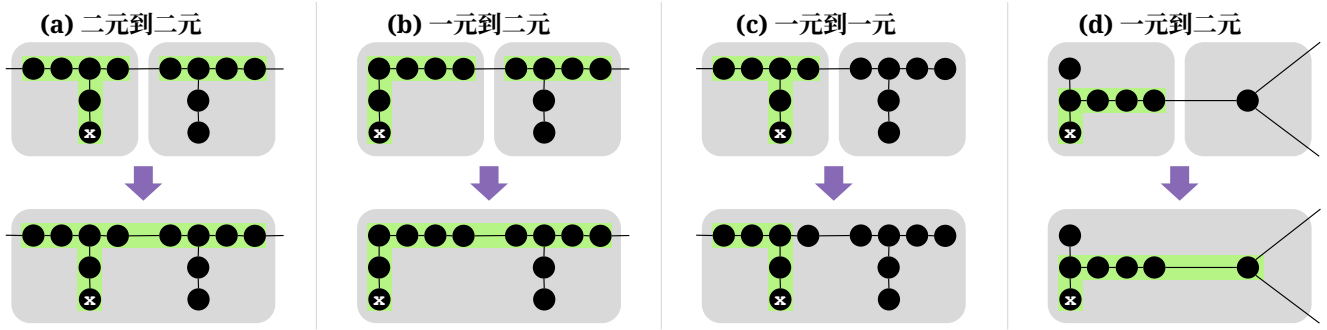

图14. 展示了在拓扑树中构造路径查询解的四种归纳情形。这四种情形分别是：(a) 二元簇及其二元父节点，(b) 一元簇及其一元父节点，(c) 二元簇及其一元父节点，以及 (d) 一元簇及其二元父节点。

任何新形成或经过修改的簇中的值，都将设为将 f 应用于该簇各子节点的值后得到的结果；此结果可在常数时间内计算。

要回答子树查询，我们将从底向上沿着从 v 和 p 到根的路径遍历拓扑树。事实上，我们将从 v 和 p的最近公共祖先开始。注意，v 和 p 的最近公共祖先是沿着边 $(v,\ p)$合并两个簇V 和 P 所得的结果。簇 V 表示输入树的一个连通子图，其中包含 v 但不包含 p。因此，V 中的每个顶点都位于以 p为根的 v 的子树中。我们的查询将从 V存储的值开始。

令 X 为最近公共祖先簇，X 在 v 一侧的边界顶点为 b_v（如果不存在这样的边界顶点，则查询完成）。此时，我们得到以 v 为根的子树减去以 b_v 为根的子树。我们将其定义为簇 X 相对于顶点 v 和父节点 p 的 代表子树 。当我们沿着路径的其余部分向上遍历至根时，可以归纳地计算父簇的代表子树。考虑父簇 P，它由 X 和另一个簇 C 组合而成。如果 C 和 X 通过 b_v 组合，那么 P 的代表子树就是 X 的代表子树与 C 中所包含顶点集合的并集，而 b_v 的值是 C 的另一个边界顶点（如果存在）。否则，如果 C 和 X 并非通过 b_v 组合，那么 P 的代表子树就只是 X 的代表子树，且 b_v 的值与之前相同。如果 b_v 不再存在，则查询完成。

由于每个归纳步骤都可以在 $O(1)$ 时间内计算完成，且树的高度为 $O(\log n)$，因此有以下引理。

引理 C.2。 在具有 n 个顶点的输入树上，拓扑树中的子树查询需要 $O(\log n)$ 时间。

其他查询。 拓扑树还支持直径、最近公共祖先、最近标记顶点以及中心/中位数查询。这些算法本质上与耙压缩树的算法相同 [4]。下面简要总结这些算法，以说明稍作修改后它们也适用于拓扑树。

直径查询 返回输入树中最长路径的长度。为支持此操作，每个一元簇维护簇内的直径，以及从边界顶点开始的簇内最长路径的长度。每个二元簇维护其直径、从两个边界顶点开始的最长路径长度，以及其簇路径的长度。这样，就可以从拓扑树的根节点读取直径查询的答案。

最近标记顶点查询 接受一个查询顶点 v，并返回预先确定的标记顶点集合中距离最近的顶点。下面介绍如何查询到最近标记顶点的距离。每个簇都维护每个边界顶点到簇内最近标记顶点的距离。此外，二元簇还存储其簇路径的长度。若要返回最近标记顶点本身，可以在每个簇中额外维护该顶点本身以及相应距离。随后，可以通过从顶点 v对应的叶节点向上遍历来回答最近标记顶点查询。对于通向根节点的路径上的每个簇，我们计算从 v 到该簇中某个标记顶点的最小距离，以及从 v 到每个边界顶点的距离。

中心查询返回树中到其他任意顶点的最大距离最小的顶点。中位数查询返回到所有其他顶点的距离加权和（权重由顶点权重决定）最小的顶点。下面介绍如何支持中心查询；中位数查询也可以用类似的方法支持。每个簇都维护从每个边界顶点到簇中任意其他顶点的最大距离。此外，二元簇还会存储其簇路径的长度。

然后，可以通过从树的根向下遍历来回答查询。对于通过边（$c_1,\ c_2$）组合簇 C_1 和 C_2 形成的每个簇，算法都会向下遍历到这样一个簇：从边界顶点 c_1 或 c_2 到树其余部分的最大距离更大。要获取从 c_1 到

树其余部分的距离，在向下遍历过程中，会为每个簇维护从 C_1 的另一个边界顶点到树其余部分的最大距离。然后，从 c_1 到树其余部分的最大距离，是从 c_1 到 C_1 中的顶点的最大距离，以及 C_1 的簇路径长度与从 C_1的另一个边界顶点出发的最大距离之和，两者中的较大值。

最低公共祖先（***LCA***）查询接受两个查询顶点 u 和 v 以及一个根 r，并返回相对于 r 的 u 和 v 的最近公共祖先。注意，u、v 和 r 完全可以互换（例如，相对于 r 的 u 和 v 的最近公共祖先，与相对于 v 的 u 和 r 的最近公共祖先相同），且最近公共祖先必须是一个 3 度顶点。该查询通过从 u、v 和 r 的叶节点向上遍历来回答。不失一般性，假设 u 和 v 的路径首先相交。令 X 为第一个包含 u 和 v 的簇。如果 X 是一个 3 度簇和一个度为 1 的簇的组合，则最近公共祖先就是 3 度簇中的唯一顶点。否则，如果 X 是一个度为 2 的簇和一个度为 1 的簇的组合，则我们沿着度为 2 的簇的路径向下遍历。最近公共祖先是第一个与该路径下方的某个簇组合的 3 度簇。否则，X 是两个度为 2 的簇的组合。然后，我们沿着 r 一侧的度为 2 的簇的路径向下遍历（可以通过向上遍历，直到 r 的路径与 u 和 v 的路径相交来确定这一侧）。最近公共祖先是第一个与该路径下方的某个簇组合的 3 度簇。如果 X 是两个度为 1 的簇的组合，则 u、v 和 r 不连通。

C.2 UFOTree 查询

路径查询。这里我们采用附录C.1中对路径查询、一元簇、二元簇、边界顶点和簇路径的相同定义。为了考虑既非一元也非二元的高阶簇，我们将 UFO 树中的超一元簇 定义为具有一个边界顶点，但其内部边数严格大于一的簇 E，且恰有一个端点位于该簇内。超一元簇可以只包含一个度为 1 的 $d \geq 2$ 顶点，也可以是将一个高阶簇与一个或多个相邻的度为 1 的簇组合而成。请注意，只包含一个度为 2 的顶点的簇可以视为超一元簇或二元簇，这对分析并无影响。根据定义，任何超一元簇的簇路径均为空。

每个二元簇都会维护其簇路径中所有 f边上应用的值。一元簇和超一元簇的簇路径为空，因此不存储任何值。只有在重新聚类过程中两个二元簇组合时，才会重新计算这些值。该过程与拓扑树相同（见图13）。

簇 X 相对于簇中包含的顶点 x 的代表路径，仍定义为从 x到 X的边界顶点的唯一路径。

一元簇和超一元簇各有一条代表路径，二元簇则有两条。为回答路径查询，我们从代表路径查询两个端点的叶节点开始，逐步归纳计算代表路径，直到到达它们的最近公共祖先。这里再次使用拓扑树中针对一元簇和二元簇的四种归纳情形（见图14）。我们还为超一元簇引入了三种额外的归纳情形（见图15）。

假设我们已有子簇 C 的代表路径，现在要计算父簇 P的代表路径。如果 C 是 P的唯一子节点，代表路径保持不变。UFO 树还有以下三个额外的归纳情形：(a) 第一种情形是 C 和 P 都是超一元簇。这意味着 C 与一个或多个一元簇组合而成。P的代表路径与 C的代表路径相同。(b) 第二种情形是 C 是超一元簇，而 P是一元簇。这意味着 C的度为 d，并且恰好与 $d-1$ 个其他一元簇组合。此时，P 的代表路径仍与 C的代表路径相同。(c) 第三种情形是C 是一元簇，而 P 是超一元簇。在这种情况下，簇 S必须是超一元簇，它通过边 e 与 C 组合形成 P。P的边界顶点就是 S的边界顶点。因此，P 的代表路径是 C的代表路径与边 e的并集。请注意，这三个额外情形已足够，因为按照 UFO 树的组合规则，无法从二元簇形成超一元簇，反之亦然。

利用这种归纳方法，以及针对每个二元簇的簇路径计算并存储的 f 的值，我们可以计算 U 的代表路径上关于 u 的 f 的值，以及 v的代表路径上关于 v的 V 的值，从而回答路径查询。由于每个归纳步骤可在 $O(1)$ 时间内完成，且树的高度为 $O(\min\{D, \log n\})$，因此我们有以下引理：

引理 C.3。对于具有 n 个顶点且直径为 d 的输入树，*UFO* 树上的路径查询需要 $O(\min\{D, \log n\})$ 时间。

使用可逆函数进行子树查询。这里我们采用与附录C.1相同的子树查询、边界顶点和代表子树定义。为高效支持子树查询，UFO 树中的每个簇都会维护在簇内所有顶点上应用 f所得的值。树中的每个叶节点都将包含 f的单位元。

更新过程中重新聚类时，任何新形成或被修改的簇，其值都必须重新计算为对该簇所有子节点的值应用 f 的结果。在拓扑树中，这总能在常数时间内完成，因为子节点最多只有两个。对于 UFO 树中的高扇出簇，如果更新将一个簇加入组合，新的值只需将 f 应用于旧值和新增子簇的值即可。难点在于从组合中移除一个簇

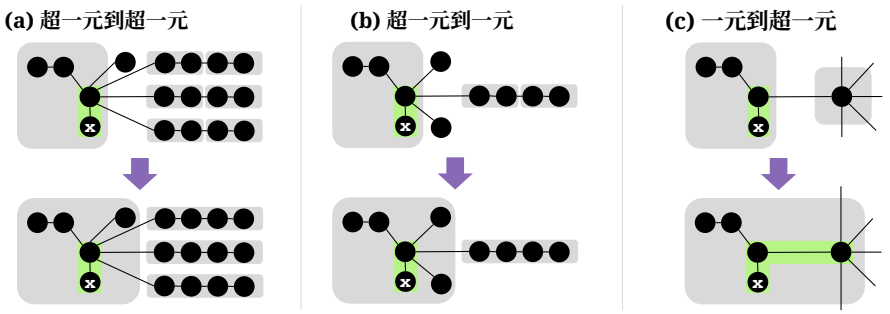

图 15. 展示在 UFO 树中构造路径查询解时另外三种归纳情况的示意图。这三种情况分别是：(a) 超一元簇的父节点是超一元簇；(b) 超一元簇的父节点是一元簇；(c) 一元簇的父节点是超一元簇。

时的情况。如果 f 是可逆函数，则可以在常数时间内计算出新值：将 f 应用于旧值和被移除子簇的值，再对结果应用逆函数即可。

要回答子树查询，我们将沿着从 v 和 p 到根的路径，自底向上遍历 UFO 树。从包含 v的 v 和 p 的最近公共祖先的子节点开始，继续向上，并归纳地计算相对于 v 和 p，将 f 应用于每个簇的代表子树所得的值。

一元簇和二元簇的归纳情况与拓扑树中的情况相同。额外的归纳情况是当前簇 X 的父节点 P是超一元簇。如果 X 是 P的高阶子节点，那么 b_v 将保持不变。否则，b_v 将变为 P的高阶子节点的边界顶点。无论 X 是否为 P 的高阶子节点，P 的代表子树都是 X 的代表子树与 P所有其他子节点所含顶点集合的并集。换言之，它是 $P \setminus (X \setminus R_X)$，其中 R_X 表示 X的代表子树。因此，我们可以通过两次调用 f的反函数，计算将 f 应用于这个顶点集合所得的值。

由于每个归纳步骤都可以在 $O(1)$ 时间内计算，且树的高度为 $O(\min\{D, \log n\})$，我们得到以下引理：

引理 C.4. 对于具有 n 个顶点、直径为 d 的输入树，*UFO* 树上的子树查询需要 $O(\min\{D, \log n\})$ 时间。

最近公共祖先查询。最近公共祖先查询可以用与拓扑树类似的方法回答。我们再次利用这样一个观察：u、v 和 r 完全可以互换。在 UFO 树中，最近公共祖先必须是高阶顶点（度不能为 1 或 2）。查询通过从 u、v 和 r 的叶节点向上遍历来回答。不失一般性，假设 u 和 v 的路径最先相交。令 X 为第一个同时包含 u 和 v 的簇。如果 X 是高阶簇与一个或多个度为 1 的簇的组合，那么最近公共祖先就是高阶簇中的中心顶点，也就是每次沿着高阶方向向下遍历树时找到的顶点子节点。否则，如果 X 是一个度为 2 的簇与一个度为 1 的簇的组合，我们就沿着度为 2 的簇的路径向下遍历。最近公共祖先是沿此路径向下与某个簇组合的第一个高阶簇的中心顶点。否则，X 是两个度为 2 的簇的组合。接着，我们沿着 r 所在一侧的度为 2 的簇的路径向下遍历（这可以通过向上遍历，直到来自 r 的路径与来自 u 和 v 的路径相交来确定）。最近公共祖先是沿此路径向下与某个簇组合的第一个高阶簇的中心顶点。如果 X 是两个度为 1 的簇的组合，那么 u、v 和 r 不连通。

其他查询。有些查询需要维护高扇出簇的子节点上某个不可逆函数的值。这些查询包括使用不可逆函数的子树查询、直径查询、最近标记顶点查询、中心查询和中位数查询。由于该函数不可逆，当删除高扇出簇的一个子节点时，朴素地重新计算这一聚合值 需要线性工作量。这里我们介绍一种技术，可以高效维护聚合值，并在删除或插入元素时将更新时间控制在对数级。采用这一技术后，UFO 树操作的开销为 $O(\log n)$，不再满足 $O(D)$ 界。

该技术将一个簇的子节点集合表示为二叉搜索树数据结构。已知可以高效地维护带有增强值的二叉搜索树。具体来说，我们使用秩树 [44] 存储父簇的子簇集合 p。每个子簇c在秩树中的权重为 $n(c)$，即其簇中的顶点数。秩树保证子节点 c的深度至多为$\log(n(p)/n(c))$。此外，c 可以在 $O(\log(n(p)/n(c))$ 时间内从秩树中插入或删除。

采用这种嵌套秩树结构，UFO 树中任意叶节点的总深度仍为 $O(\log n)$，这是由望远镜求和论证得出的。因此，查询复杂度与树的高度成正比，为 $O(\log n)$。引理 C.5 证明，采用这种表示方式，更新的开销也是 $O(\log n)$。

引理 C.5。 *UFO* 树使用秩树表示簇的子节点集合时，可以在 $O(\log n)$ 时间内执行更新，其中 n 是输入树中的顶点数。

证明。 首先注意到，由于每一层都有 $O(1)$ 个根簇，我们可以将更新期间的所有变化映射到 $O(1)$ 条路径上。我们将证明，每条路径在秩树插入或删除操作中所做的总工作量最多为 $O(\log n)$。

考虑最坏情况：路径上的每个节点都需要执行一次秩树插入或删除。由于望远镜求和论证，所有这些操作的开销至多为 $O(\log n)$。例如，设路径为 c_0、c_1、c_2、……、c_{h-1}、c_h。秩树操作的开销为：

$$\begin{aligned} &O\left(\log \frac{n(c_h)}{n(c_{h-1})} + \cdots + \log \frac{n(c_2)}{n(c_1)} + \log \frac{n(c_1)}{n(c_0)}\right) \\ &\quad = O\left(\log\left(\frac{n(c_h)}{n(c_{h-1})} \times \cdots \times \frac{n(c_2)}{n(c_1)} \times \frac{n(c_1)}{n(c_0)}\right)\right) \\ &\quad = O\left(\log \frac{n(c_h)}{n(c_0)}\right) = O(\log n). \end{aligned}$$

□

C.3 使用不可逆函数进行子树查询的下界

我们现在证明，即使输入树的直径为常数，支持子树最小值或最大值查询仍需要每次操作花费 $\Omega(\log n)$ 时间。

引理 C.6。 任何支持子树最小值或最大值查询的动态树数据结构，每次操作都必须花费 $\Omega(\log n)$时间，无论是摊还时间还是最坏情况时间，即使将其限制为直径为常数的树。否则，就会违反基于比较的排序的 $\Omega(n \log n)$ 下界。证明。我们将 n 对若干数字进行排序，归约为对表示星形图的动态树执行一系列操作。

通过一系列 n 次边插入，在 $n+1$ 个顶点上构造一棵星形树，即包含一个中心节点和 n 个叶节点，并使边权重对应输入数字。然后，迭代执行以下操作：查询中心节点处子树的最小值（或最大值），并删除对应的边。此过程实质上会找出并移除剩余元素中的最小值（或最大值）。经过 n 次迭代后，即可构造出这些数字的排序后的序列。注意，在整个过程中，树始终是一棵直径为 2 的星形树。

总成本由 n 次边插入、n 次边删除以及 n 次子树最小值（或最大值）查询的开销构成。因此，如果每项操作都能在 $o(\log n)$ 时间内完成（摊销或最坏情况），就能得到一个运行时间为 $o(n \log n)$ 的排序算法，从而违背基于比较的排序下界。□

这一下界解释了为什么 UFO 树无法针对一般的不可逆子树查询达到更强的 $O(D)$ 开销下界。我们认为，类似的下界技术可以用来证明，不可能达到对于其他查询类型，每次查询需要 $o(\log n)$ 时间，而 UFO 树无法达到 $O(D)$ 的开销下界。

D 实验设置与补充结果 D.1 顺序实现细节

现有基线。目前已有许多动态树数据结构的实现。下面列出几种我们在实验中评估的实现。我们采用 Tseng 等人的欧拉游览树（ETT）顺序实现 [41]，该实现使用跳表作为底层序列数据结构。我们还采用了伸展顶树（STT）的实现，并将其从 C 转换为现代 C++。ETT 和 STT 的实现都需要使用映射表来查找与给定边对应的叶簇。我们使用 Abseil 的扁平哈希映射来实现这一功能 [9]。

NewTopology 树实现。每个拓扑簇结构体都有一个指向父簇的指针，以及一个包含 3 个指向相邻簇的指针的数组。为避免增加内存使用量，我们的实现不会显式存储子节点。这没有问题，因为在我们的算法中，我们总能找到与某个簇具有相同父簇的兄弟簇，而无需直接查看该父簇的子节点。

全新 UFO 树实现。每个 UFO 簇结构体都包含一个指向父簇的指针、一个固定大小的指针数组（用于存储相邻簇），以及一个指向 Abseil 哈希集合的指针（用于存储其他邻居）。与拓扑树类似，我们的 UFO 树实现不会显式存储子节点，以避免增加内存使用量。这没有问题，因为在我们的所有算法中，我们都可以考虑共享同一父节点的簇的兄弟节点，而不必直接查看该父节点的子节点。

我们使用一个固定大小的小型邻居数组，这样结构体加载到内存时，也会一并载入少量邻居。如果只需访问少量邻居，这可以避免从邻居集合中读取数据时产生额外的缓存未命中。这一点尤其有用，因为在任何树中，至少有一半的顶点的度为 1 或度为 2。我们只存储指向 Abseil 哈希集合的指针，这样能够在数组中存储所有邻居的簇就不必承担未使用的空哈希集合所带来的空间开销，只需为每个簇额外占用一个 8 字节的指针。如果某个簇增加了许多邻居，就会在堆上分配哈希集合；如果它失去了许多邻居，则会释放该集合。这项优化显著降低了内存使用量，也略微加快了更新和查询的速度。

RC 树实现。 我们开发了一个顺序实现，用于实现确定性直接版的 rake-compress 树；该实现不使用自调整计算框架 [6]，也不是批动态的 [32]。我们开发自己的实现，是因为尽管自调整计算框架具有很强的实用性，能够解决许多问题，但它会占用大量内存，而且针对单个问题调优实现也很有益处。

同样，我们发现，在使用 1 个线程且批次大小为 $= 1$ 时，我们的 RC 树实现比批量动态版本快得多。我们进一步优化了 RC 树实现，以确保与经过性能优化的拓扑树和 UFO 树实现进行公平比较，并确认观察到的任何加速或减速都源于数据结构本身的特性。为此，我们决定使用向量来模拟受影响顶点集合的数据结构，因为在顺序情形下，该集合的大小可证明小于 312 个元素，而实际情况中更小。此外，我们还尝试尽量减少指针跳转，以改善局部性，并据此设计了数据结构。这些改动包括：在适用时使用定长数组代替向量，以及尽量减少对 RC 树簇数组的查找（优先以有利于局部性的方式存储这些引用）。我们认为，这些改动也优化了数据结构的空间使用量，并使其能够与其他动态树进行准确比较。

三元化实现。为了评估三元化在实际顺序执行中的影响，我们实现了自己的抽象动态三元化接口。该接口将任意输入树映射为度为 ≤ 3 的底层树，并将连接操作和切断操作映射为底层树上所需的连接和切断操作。

Link-Cut 树实现。我们采用 Tseng 等人提出的实现方式 [41] 来实现 Link-Cut 树（LCT）。该实现采用基于伸展树的 LCT 摊还版本。每个顶点对应一个伸展树节点，每个节点包含一个父指针、两个子指针和一个翻转位。我们修改了该实现，使其也支持整数加权边的路径最大值查询。请注意，顶点加权路径查询很容易实现，因为伸展树节点与输入树的顶点一一对应，所以我们只需扩展每棵伸展树，使其存储子树中所有节点的聚合值。然后，路径查询只需暴露目标路径并读取根节点的聚合值即可。

Sleator 和 Tarjan 在原始论文 [40] 中描述了如何支持边加权路径查询，以实现最大流算法。虽然在理论上高效，但其实现相当复杂，而且每个树节点都需要许多额外字段。我们的实现中，每个树节点只使用两个额外的整数型字段，用于存储与一个顶点相连的优先边的“向上”和“向下”权重。随后，每个伸展树节点都会维护其子树中所有节点的这两种优选边权重的聚合值。之所以需要同时存储向上边权重和向下边权重（因此每条优先边的权重都存储了两次），而不是只存储向上权重，是因为当根优选路径反转时，所有向上权重都会变为向下权重，反之亦然。向上权重和向下权重之间的转换，采用与反转伸展树节点的左右子节点时相同的技术处理（为每个节点保留一个翻转位，并在向下遍历时将翻转延迟传递给子节点）。

向上权重字段还用于存储每条路径的头节点伸出的非优选边的权重。该值不计入伸展树中的聚合值。幸运的是，头节点伸出非优选边的路径从不会被反转（因为只有根路径会被反转），因此，当路径被反转时（原来的尾节点成为新的头节点），我们无需确定新头节点伸出的非优选边是什么。当两条路径连接时，我们会更新上方路径尾节点的向下优选边权重。同样，当我们拆分一条路径时，会清除拆分边上方节点中的向下权重。每当一个节点获得或失去优选边权重时，都会对其执行伸展操作，以重新计算聚合值。

D.2 并行批量动态树实现

我们的并行实现使用 ParlayLib，这是一个用于共享内存并行编程的 C++ 库。我们将微软的 mimalloc 与 ParlayLib 中的并行分配器结合使用，以进行并行内存分配。

并行拓扑树实现。 对于我们的并行拓扑树实现，我们采用一种简单的方法来并行更新邻接表：遍历最多 3 个邻居，并对每个索引使用原子比较并交换操作。

为了在高阶输入上测试拓扑树，我们使用了 [32] 中的批量动态三元化实现。

并行 UFO 树实现。开发高性能的并行 UFO 树实现面临诸多挑战。首个也许最明显的挑战，是以较低深度维护树邻居集合。我们使用并行增强映射表（PAM）来表示 UFO 树中每个簇的邻居集合。

由于算法接收到的更新批次没有特定顺序，我们必须做的一件事是将此批次转换为按第0层各顶点分组的更新。使用 ParlayLib 中的“group_by_key”函数开销很大，因此我们开发了自己的更快方法“integer_group_by_key_inplace”。顾名思义，该方法利用了边的端点可解释为整数这一事实。此外，它不会为每个组分配新的内存，而是直接在原处完成分组。该方法首先创建一个包含两个方向边的序列，然后按第一个端点对其进行整数排序，最后返回排序范围中各组起始和结束索引的指针。我们发现，并行样本排序的性能优于并行整数排序。

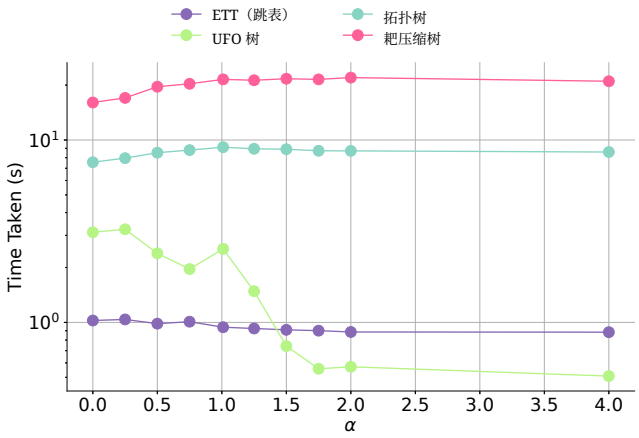

图16。在所有批动态树上运行我们的直径扫描实验 $n = 10^7$ 的结果。

在我们的理论批并行算法中，我们为方便起见，假设每个簇都维护其邻居集合和子节点集合。我们的实现不存储子节点集合，以避免额外的空间和更新开销。这给正确实现带来了许多挑战，但也换来了显著更好的性能。例如，如何判断一个簇是否为低扇出（子节点少于 3 个），从而应该将其删除？此外，当你决定删除层 $i+1$ 簇时，如何确定哪些层 i 子节点会成为根簇？

我们没有仅仅采用一个适用于一般图的知名最大独立集算法，而是专门为链输入设计并实现了一种快速的异步极大独立集算法。我们的重新聚类步骤利用该算法来确定度为 2 的簇之间，以及与度为 2 的簇相邻的度为 1 的簇之间的合并。对于其余簇，我们只需始终将度为 1 的簇与其邻居合并，或将高阶簇与其所有度为 1 的邻居合并。

D.3 附加实验结果

并行直径扫描结果。图16 展示了我们针对并行实现的更新速度所进行的直径扫描实验结果。结果表明，对于批量更新，随着输入直径减小，并行 UFO 树的速度会提升。其他所有实现的运行时间则随着树直径减小而基本保持稳定。当直径足够小时，并行 UFO 树始终快于并行 ETT，并且是速度最快的实现。

与顺序执行的情况类似，我们观察到，在直径较小的输入上，并行拓扑树和并行 RC 树实现处理批量更新的速度会变慢。这是因为，对于包含许多高阶节点的输入（低直径所必需的），三元化开销会变得更大。