arXiv:2603.11763v1 [cs.DS] 12 Mar 2026

以最优时间枚举所有有向生成树

Paweł Gawrychowski*[1] 和 Marcin Knapik[1]

[1]波兰弗罗茨瓦夫大学

摘要

我们研究这样一个问题：给定有向图 $G = (V, E)$ 和一个节点 $r \in V$，枚举所有以 r为根节点的有向生成树。对于无向图，枚举所有生成树这一对应问题已受到广泛关注，最终产生了 Kapoor and Ramesh 的算法 [SIAM计算机学报 1995]，其运行时间为 $\mathcal{O}(n+m+N)$，其中 N, n, m 分别表示 G的生成树数量、顶点数量和边数。在枚举算法领域，这被称为常数摊销时间，简称 CAT。为了使每棵生成树的输出时间仅为常数，该算法输出相邻生成树之间的相对变化，而非生成树本身的完整内容。

枚举所有有向生成树这一自然推广问题，Gabow and Myers [SIAM计算机学报 1978], 已对此进行研究，并提出了一种运行时间为 $\mathcal{O}(n + m + Nm)$ 的算法。此后，这一时间复杂度又经过数次改进。1998 年，Uno 引入了修剪与平衡框架，使其得以为该问题设计出运行时间为 $\mathcal{O}(n + m\log\ n + N\log^2 n)$ 的算法。结合后续成果，可以直接将时间复杂度改进至$\mathcal{O}(n + m + N \log n)$，但在这一框架下似乎难以达到最优界 $\mathcal{O}(n + m + N)$。

本文展示了如何在 $\mathcal{O}(n + m + N)$ 时间和$\mathcal{O}(n + m)$ 空间内枚举所有有向生成树，达到与无向图相同的时间界。我们通过设计一种纯粹的图论刻画，描述具有极少有向生成树的图，并利用这类图的结构加速算法，从而实现了这一改进。

1 引言

枚举算法领域的目标，是列出给定算法问题的所有解。例如，对于寻找给定图的生成树这一问题，我们要列出所有这样的生成树。这一目标源于（实际中）在不知道目标函数的情况下选择最佳解的需求：只要能够列出所有解，我们就可以将整个流程串联起来，并通过一次黑箱调用目标函数来评估每个解。解的数量甚至可能随输入规模呈指数级增长，因此，我们不应以设计一个关于输入规模、运行时间为多项式的算法为目标。相反，我们的目标是设计一个关于解的数量、运行时间为多项式（理想情况下为线性）的算法。Knuth 著名著作的第 4A 卷 [10] 介绍了基本组合对象的列举方法。若想查阅较为及时的枚举算法目录，请参见 Wasa 的列表 [21]。

本文的重点是枚举给定图的所有有向生成树。在这个问题中，我们给定一个有向图 $G = \langle V,\ E\rangle$ 其中有一个指定的根节点 $r \in V$。图 G 的一棵有向生成树（或树形图）是 $n - 1$ 边 $E' \subseteq E$ 的一个子集，满足仅使用 E'中的边即可从 r 到达任意节点 $u \in V$。

Shinoda [16] 设计了一种用于枚举所有树形图的指数时间算法。Gabow and Myers [3]显著改进了其时间复杂度：对于包含 n个节点和 m条边的图，他们的算法可在 $\mathcal{O}(n + m + Nm)$时间内枚举所有 N 个解。对于无向图（以及无向生成树的枚举），他们的方法实际上可在 $\mathcal{O}(n+m+Nn)$时间内运行，改进了由 Minty [14] 首先提出的回溯方法所达到的$\mathcal{O}(n + m + Nm)$时间复杂度，并

*本研究部分由波兰国家科学中心资助，资助编号为 2023/51/B/ST6/01505。

随后由 Read 和 Tarjan [15]将其形式化（并显著改进了 McIlroy [13]提出的指数时间算法）。

乍看之下， $\mathcal{O}(n+m+Nn)$ 似乎是输出所有 N 生成树（无向或有向）的渐近最优时间复杂度，因为每棵树都由 $n-1$ 条边组成。为了突破这一限制，枚举问题中的标准做法是只输出前一个解与下一个解之间的差异。也就是说，我们只显式输出第一棵生成树 T_1，然后通过指定应从 T_i 中移除哪些边、再添加哪些边来生成后续各棵生成树，从而得到 T_{i+1}。Kapoor and Ramesh [8]设计了一种枚举无向生成树的算法，其运行时间为 $\mathcal{O}(n+m+N)$，空间复杂度为 $\mathcal{O}(nm)$；Shioura、Tamura 和 Uno [17] 则将空间复杂度改进为 $\mathcal{O}(n+m)$。事实上，这两种算法都是通过交换边，从 T_i 生成 T_{i+1}：反复添加一条新边，并移除其基本回路上的一条边。正如 Matsui [12], 所观察到的，这可以表述为遍历生成树多面体骨架图上的一棵有根生成树。我们将只输出每对相邻解之间差异的枚举算法称为隐式枚举算法（相对于显式枚举算法）。有时，将能在 $\mathcal{O}(N)$时间内生成所有N 解的隐式枚举算法称为常数摊销时间（CAT）算法（严格来说，这并非通常在算法分析中定义的、每个解具有常数摊销时间的算法，因为我们只衡量输出所有解所需的总时间，因此尤其不能保证对于每个 $t \leq N$，该算法都能在 $\mathcal{O}(t)$时间内生成前 t 个解）。因此，目标是设计一种 CAT 算法。

对于树形图，Kapoor and Ramesh [9] 设计了一种隐式枚举算法，可在$\mathcal{O}(n^3+Nn)$时间内运行（隐含假设图中不存在重复边），该算法可以很容易地转换为具有相同时间复杂度的显式枚举算法。即使忽略需要花费 $\mathcal{O}(n^3)$时间的额外预处理，这种方法也并未真正受益于只需输出前一个解与下一个解之间差异这一条件，并且使得设计 CAT 算法仍然成为一个开放问题。Uno [18] 设计了一种用于枚举树形图的算法，该算法利用了一种维护最小权生成树的数据结构。代入 Holm、de Lichtenberg 和 Thorup [7], 后来提出的数据结构后，时间复杂度变为 $\mathcal{O}(n+(m+N)\log^4 n)$。在后续论文中， [19], 他设计了另一种避免使用任何此类黑盒结构的技术，并将时间复杂度提升至 $\mathcal{O}(n+m\log\ n+N\log^2 n)$。接下来，我们将简要介绍这些先前的方法。

前述算法概述。生成所有树形图最自然的方法，或许是选择一条边 e，然后递归地报告所有包含和不包含 e 的树形图。具体来说，在前一种递归调用中，我们收缩 e；在后一种调用中，则直接将其从图中移除。此外，还可以进行两项优化：如果 e 属于每个树形图，就无需进行后一种调用；同样，如果 e 不属于任何树形图，就无需进行前一种调用。这种方法（或者说其针对无向生成树的版本）可以追溯到 Minty [14], ；Read 和 Tarjan [15] 则给出了更精确的表述和运行时间上界。Gabow and Myers [3] 展示了如何利用深度优先搜索组织这种递归，使得轻松检测某条边是否被所有树形图使用成为可能，并由此得到了一个枚举树形图的算法，其时间复杂度为 $\mathcal{O}(n+m+Nm)$。不过，这仍未达到 $\mathcal{O}(n+m+Nn)$ 的时间复杂度；如果要显式输出每个树形图，这一复杂度才是最优的。

Kapoor and Ramesh [9] 设计了一种算法，可在 $\mathcal{O}(n^3+Nn)$时间内显式枚举所有树形图（针对简单图）。该算法首先通过深度优先搜索获得一个树形图 T，并维护其回边集合（其中头端是尾端的祖先）以及非回边集合（所有其余的非树边）。在所有非回边中，它选择头端先序遍历序最小的 f （若相同则任意打破平局），并考虑将其加入树形图，即与 e交换（这要求移除所有其他头端相同的边），或将其丢弃。交换后，一些回边可能会变为非回边。令 $h(e)$和 $t(e)$分别表示一条边 e的头端和尾端。为正确检测这些边，我们遍历 $h(f)$的子树中的所有节点 w，并考虑尾端位于 w中的所有回边。随后，头端位于 $h(f)$与 $h(f)$和 $t(f)$的最近公共祖先之间的那些边会变为非回边，我们需要将它们列出。为此，对于每个 w，我们存储一个额外的数据结构，用于查找尾端位于 w 且头端位于某个节点 p 之上的第一条回边。这需要 $\mathcal{O}(n^2)$空间，并需要 $\mathcal{O}(n^3)$时间进行预处理。得益于深度优先搜索的性质，该结构在整个算法过程中实际上不会发生变化，因此我们能够在 d 中列出所需的边

$\mathcal{O}(n+d)$时间。移动这些 d 边中的每一条同样需要 $\mathcal{O}(n)$时间，但我们注意到，每条这样的边都对应于一个新的树形图，因此总复杂度为 $\mathcal{O}(n^3+Nn)$时间。

Uno [18]提出了一种使用所谓反向搜索技术的更快隐式枚举算法。首先，他为一些树形图引入了父子关系。设 T_0 是通过深度优先搜索得到的树形图，则我们按照各边头端在 T_0 中的先序遍历次序为所有边编索引。现在，对于任意树形图 T_c，我们称其父树形图 T_p 是通过选取索引最低的一条边 $f \in T_c \setminus T_0$（记作 $v^*(T_c)$），并将其替换为 $e \in T_0$ 得到的，其中 $h(e) = h(f)$。然后，算法遍历当前 T_p 的所有子树形图并递归执行。为此，它会考虑所有索引小于 $v^*(T_p)$ 的非回边 T_0 f；如果将其替换为相应的树边不会产生环，就将其视为有效的子树形图（f 是一条非回边）。T 中此类边的集合按头端索引顺序存储，记作 $S(T)$。选取 f 后，为了维护这个集合，我们注意到 $S(T_c)$ 是 $S(T_p)$ 的某个前缀，再加上 T_p 的回边：这些回边的头端位于 $v^*(T_c)$ 的子树中，尾端位于从 $v^*(T_c)$ 到其与 $t(f)$ 的最低公共祖先的路径内部部分。后一部分较难计算，但可以利用任意一种用于维护无向图最小生成树的数据结构，其中 T_p 的边权为 0，而回弧的边权为 n 减去其索引。最终，我们得到 $\mathcal{O}(n+m+N\cdot D(n, m))$ 和 $\mathcal{O}(n+m+DS(n,m))$，其中 $D(n,m)$ 和 $DS(n,m)$ 分别是这种数据结构的时间和空间复杂度。

随后，Uno [19] 描述了一种更快的隐式枚举算法，该算法通过交替进行修剪与平衡实现（这也是适用于其他问题的一般框架 [20]）。修剪可确保每条边都属于至少一个、但并非所有树形图。在原论文中，这一过程可在 $\mathcal{O}(m\log\ n)$ 时间内完成，但可以利用关于在线性时间内构造支配树的已知结果，将其改进为 $\mathcal{O}(m)$。接着可以看出，含有 m条边的修剪图至少包含 $\Omega(m)$ 个树形图。接下来，主要的技术贡献是证明这样的图中存在一条边e，使得至少有 $\Omega(m)$ 个树形图包含该边，且至少有 $\Omega(m)$ 个树形图不包含该边 e。通过对这样的边递归处理，可以保证总时间为 $\mathcal{O}(n+m+N\log\ n)$；粗略地说，这是因为递归树不可能非常不平衡。

我们的结果。我们的主要结果是一种算法，可在 $\mathcal{O}(n+m+N)$ 时间内枚举含有 n个节点和 m 条边的图的全部 N 个树形图。第一个有向生成树 A_1 会被显式输出；对于之后的每个树形图 A_{i+1}，我们会指定应从前一个树形图中移除哪些边A_i，以及应添加哪些新边。此外，我们的算法的空间复杂度（不计用于写明应移除和添加哪些边的只追加输出带所占用的空间）仅为 $\mathcal{O}(m)$，而获得两个连续树形图之间的延迟也为 $\mathcal{O}(m)$。换句话说，延迟始终受 $\mathcal{O}(m)$ 限制，但平均而言仅为常数。

定理 1. 有向图的所有 N 树形图均可在 $\mathcal{O}(n$ 内报告，其中该图有 n 个节点和 m 条边 $+m+N)$ 时间和 $\mathcal{O}(n+m)$ 空间，而任意两个已报告解之间的延迟为 $\mathcal{O}(m)$。

新算法概述。我们的算法是递归的。每次递归调用的输入都是一个图，其中每条边都是非平凡的：它属于某个树形图，但不属于所有树形图。我们称这样的图为已修剪图。为方便起见，我们假设图中不含平行边；在不失一般性的前提下，可以确保这一点（但需要一些额外的技术细节，本描述中将略去）。

递归过程的总体思路是，首先选择一个树形图 A，并立即将其报告为下一个解。接着，根据其共享的 A 前缀，将剩余的树形图划分开来。更正式地说，设 $A=\{a_1, a_2, \ldots, a_{n-1}\}$，其中边 a_i 按其头端的先序编号排序。然后，我们首先递归报告不使用 a_1 的树形图。接着，递归报告使用 a_1 但不使用 a_2 的树形图，依此类推。在一般的第 i步中，我们要报告使用边 a_1、a_2、.....、a_{i-1} 但不使用边 a_i 的树形图。具体做法是创建一个新图 G_i，在其中移除 a_i 并收缩 a_1、a_2、.....、a_{i-1}。借助标准工具（即支配树），可以在 $\mathcal{O}(m)$ 时间内构造新图 G_i，并使其成为已修剪图。这需要识别哪些边不会出现在任何树形图中并将其移除，然后识别哪些边会出现在每个树形图中并将其收缩。

目前还不清楚，我们是否能为每个 i 花费 $\mathcal{O}(m)$ 时间。我们的第一个观察是，如果当前图 G 中有许多树形图，即至少有 m^4 个，那么实际上这样的时间就足够了。证明方法是将这段时间分摊到后续递归调用中报告的所有树形图上。

剩下的情况是， G 中的树形图少于 m^4 个。这时，我们似乎无法为每个 i 花费那么多时间，因此需要设计一种单独的方法来处理这类图。另一个技术难点是，我们需要有效地区分树形图较少的图和树形图较多的图，这可能需要估算它们的数量。

我们的主要概念性贡献，是对具有少于m^4 个树形图的图给出纯组合刻画。首先，我们证明这类图是稀疏的： $m = \mathcal{O}(n)$。接下来，我们证明其节点可以划分为少量（事实上为 $\tilde{\mathcal{O}}(1)$[1]）条链。一条链是节点序列 $u_1, u_2, \ldots, u_k$，满足对于每个 $i = 1, 2, \ldots, k-1$，都有边（u_i, u_{i+1}）和（u_{i+1}, u_i），但没有其他边进入 $u_2, u_3, \ldots, u_{k-1}$。直观地说，这使我们能够将 G“压缩”为一个更小的图 H，其中每条边都对应于原图边的一个子集。我们将 H 称为 G的仿真图。

现在，我们的方法如下。我们尝试应用上述刻画，并获得这样的 H。如果未能成功，那么我们知道 G 至少有 m^4 个树形图，因而可以或多或少地采用朴素方法继续处理。否则，我们得到 H。另一个复杂之处在于，在第 i 次递归调用中，我们需要处理 G_i 而不是 G。不过，我们证明可以有效地更新 H，从而得到 G_i的仿真图 H_i。 H_i 的主要性质是，它允许我们在与其非平凡边数量成正比的时间内提取 G_i的非平凡边。我们证明，这实际上已经足够快。另一个困难是，这样只能得到每个 G_i的非平凡边，但可能还存在多条其他边，因为它们出现在 G_i的每个树形图中，所以之后会被收缩。我们证明，当考察$i = 1, 2, \ldots, n-1$时，这个收缩边集合发生变化的总次数总体上只有 $\mathcal{O}(n)$。下一个困难是，收缩会改变 G_i的非平凡边的端点。我们设计了一种高效的模拟方法，其运行时间与每个G_i的非平凡边数量成正比。这要求将所有 i 一起处理，因为我们需要在一棵树上使用某种数据结构，而该数据结构通常不允许常数时间实现（但在离线变体中可以实现常数时间）。

如果我们想将空间复杂度保持为 $\mathcal{O}(m)$，还会出现一些额外的复杂情况。具体来说，我们不能再一次处理所有 i，而是要将它们分批，每批包含若干实例，且总规模约为 $\mathcal{O}(n)$。此外，在某些情况下，我们无法在后续递归调用期间保留算法的完整内部状态，因此需要在递归返回后重新计算。

计算模型。 我们的算法采用标准的 Word RAM 模型进行描述，例如参见 [6]。我们假设 m和 n都能装入一个机器字，并且对能装入常数个机器字的数进行基本算术运算只需常数时间。具体来说，我们可以进行常数时间的间接寻址，从而实现具有常数时间查找的数组。需要强调的是，有向生成树的总数 N（甚至可能随 m呈指数增长）不一定能装入一个机器字。

2 预备知识

有根图。我们考虑带有指定根节点 r的有向图，通常记作$G = \langle V, E, r\rangle$，其中 $V = \{1, 2, \ldots, n\}$ 是节点集，而 $E = \{(u_1, v_1), (u_2, v_2) \ldots, (u_m, v_m)\}$ 是边的多重集，且对于 $i = 1$，有 $u_i, v_i \in V$，其中 $i = 1$取值为$2, \ldots, m$。我们明确允许平行边，但图中不存在自环。边 $e = (u, v)$是一个原子对象，可访问其尾端 u 和头端v，分别记作 $t(e)$和 $h(e)$。我们假设图采用如下表示方式：有两个数组$\mathsf{in}[]$ 和$\mathsf{out}[]$，； $\mathsf{out}[u]$是从 u出发的边的列表，而$\mathsf{in}[v]$ 是进入v的边的列表。算法执行期间，我们可以通过收缩边 $e = (u, v)$来修改图。我们认为 v 会从图中移除，并且每条从 v 出发的边 e' 的尾端都应改为 u （进入 v 的边将始终被移除）。我们将此视为修改 e'，而非创建一条新边。对于图 G，我们分别用 $n(G)$和 $m(G)$表示其节点数和边数。G始终表示原始输入图，我们通常用 n 和m 分别表示 $n(\mathsf{G})$和 $m(\mathsf{G})$。

支配点。 对于一个有根图 $G = \langle V, E, r\rangle$，如果从 r 到 v 的每条路径都必须经过 u，我们就说节点 $u \in V$支配节点 $v \in V$。众所周知，可以在 的节点上定义一棵树 $T(G)$，称为支配树，使得当且仅当 是 E中的 的祖先时，支配 $G = \langle V$，参见例如 。特别地，$r\rangle$($u \in V$)以 $v \in V$为根节点。此外，可以在线性时间内找到这样一棵树。

[1]这里， $\tilde{\mathcal{O}}(.)$省略了所考虑的图的大小中的多对数因子；在本例中，这里是 $m(G) = \mathcal{O}(n($ $G))$.

的节点上定义一棵树，称为支配树，使得当且仅当 u 是 T中的 v 的祖先时， u 支配 v ，参见例如 [11]。特别地， $T(G)$以 r为根节点。此外，可以在线性时间内找到这样一棵树。

引理 1 ([1])。 给定一个图 G，我们可以在线性时间内构造 $T(G)$ 。

有向生成树。 $G = \langle V, E, r\rangle$的有向生成树，简称树形图，是由 $n-1$ 条边 $F \subseteq E$ 组成的集合，满足 G 中的任意节点都可以仅通过 F中的边从 r 到达。我们有时需要用到部分树形图的概念，它是一个由 $k-1$ 条边$F \subseteq E$组成的集合，满足 V中的 k 个节点可以仅通过 F中的边从 r 到达。我们始终假设 G中的每个节点都可以从 r到达，因此至少存在一个树形图。

枚举树形图。 我们用 $\mathcal{A}(G)$ 表示 $G = \langle V, E, r\rangle$的所有树形图组成的集合，并定义 $N = |\mathcal{A}(G)|$。我们的目标是列出 $\mathcal{A}(G)$ 中的所有元素。如果要将每个树形图都作为边集显式输出，那么输出的大小必然是 $\Theta(Nn)$。绕过这一限制的标准方法是只输出每对连续树形图之间的差异。更正式地说，设这些树形图为 $A_1, A_2, \ldots A_N$。然后我们显式报告 A_1，接着对于每个 $i = 1, 2, \ldots N-1$，只报告 A_i 中不属于 A_{i+1}的边，以及 A_{i+1} 中不属于 A_i的边。为此，我们使用以下底层接口。设 A 表示当前的边集，它最终将构成下一个树形图，初始为空。将 A 视为全局变量会很方便。算法会反复调用以下原子操作：ADD(e)（将 e 添加到 A 中）、REMOVE(e)（从 A 中移除 e ）和 REPORT()（声明当前的 A 是下一个树形图）。我们对 ADD() 和 REMOVE() 进行重载，使其可以接受边集作为参数，从而依次对集合中的每个元素执行相应的操作。

基数排序。 在字长RAM模型中，我们可以在 $\mathcal{O}(n+s)$ 时间内，对 s 对数字（x_i, y_i）按字典序进行排序，其中 $x_i, y_i \in \{1, 2, \ldots, n\}$。

3 工具和算法概述

我们的枚举算法总体上是递归的。每次递归调用的输入都是一个图 G ，它是通过从 G 中移除并收缩某些边得到的。对于已移除的边，我们已经确定不将其包含在树形图中。对于已收缩的边，理想情况下，我们希望已经确定将其包含在树形图中，并将其添加到 A 中。然而，由于存在平行边，实现这一点会更加复杂，具体细节将在本节稍后说明。

名为 Recurse() 的主要递归过程要求输入图 G 具有另外两个性质。首先，对于每条边 $e \in E$，存在一个树形图 $A \in \mathcal{A}(G)$，使得 $e \in A$；同时还存在一个树形图 $A' \in \mathcal{A}(G)$，使得 $e \notin A'$。我们称这样的图为已修剪图。其次，理想情况下，图中不应存在平行边。然而，在整个算法中维持这一条件事实证明颇为困难，因此我们退而求其次，采用一个较弱的条件：每条边至多有常数个副本（具体而言：至多两个）。我们称这样的图为扁平图。我们首先说明如何将一般情况归约为已修剪且扁平的图。

修剪。 考虑一条边 $e \in G$。有三种可能：不存在满足 $e \in A$ 的 $A \in \mathcal{A}(G)$（我们称这种边为 无用）；对于每个 $A \in \mathcal{A}(G)$，都有 $e \in A$ （我们称这种边为 强制）；最后，存在一棵树形图 $A \in \mathcal{A}(G)$ 满足 $e \in A$，同时也存在一棵树形图$A' \in \mathcal{A}(G)$ 满足 $e \notin A'$ （我们称这种边为 非平凡）。我们分别用 $U(G)$、 $F(G)$ 和 $NT(G)$ 表示相应的边集合。注意，由于我们只考虑至少包含一棵树形图的图，这些集合两两不相交，并构成 E的一个划分。当$E = NT(G)$时，我们称 G 为已修剪。正如先前研究所指出的，无用边和非平凡边可按如下方式刻画。

命题1 ([19])。 **</style}边 (**u, v) 是非无用的，当且仅当存在一条从r 到 u 且不包含 v的简单路径。此外，若存在另一条头端相同的非无用边，则该边是非平凡的。

引理2。 给定一个图 G，我们可以在线性时间内确定 $U(G)$、 $F(G)$和 $NT(G)$。

证明。 对于每条边 $e = (u, v)$，我们需要检查是否存在一条从 r 到 u 且不包含v的简单路径。掌握这些信息后，我们可以应用命题1的第一部分来确定 $U(G)$，然后通过在线性时间内扫描整个图，应用命题1的第二部分来确定 $NT(G)$。每条既不属于 $U(G)$也不属于 $NT(G)$ 的边，都属于 $F(G)$。

我们应用引理 1 来构造支配树 $T(G)$。在 $T(G)$ 中，节点 v是节点 u 的祖先，当且仅当从 r 到 u 的每条路径都经过 v。我们观察到，从 r 到 u存在一条不经过 v 的路径，当且仅当从 r 到 u 存在一条不经过 v的简单路径。因此，对于边 $e = (u, v)$，存在一条从 r 到 u 且不包含 v 的简单路径，当且仅当 v 不是 $T(G)$ 中 u 的祖先。在线性时间内为 $T(G)$ 的节点赋予前序和后序编号后，可以在常数时间内检查每条边是否满足此条件，从而使我们能够在线性总时间内确定所有三个集合。

□

为了将一般情形归约为修剪图 G 的情形，我们按如下步骤进行。首先，我们应用引理2，在线性时间内确定 $U(G)$、 $F(G)$ 和 $NT(G)$。只需从图中移除无用边即可。对于每条边 $e \in F(G)$，我们知道，每个应当报告的树形图都包含 e。我们将每条这样的边 e 添加到 A，并按如下方式修改 G 。我们收缩 $e = (u, v)$：对于每条边 $e' = (v, w)$，我们将 e 的尾端更新为 u，然后移除 v 及其所有入边（在这种情况下，实际上只有 e，因为我们此前已移除所有无用边）。所有强制边都可以在线性时间内同时收缩：首先考虑由强制边构成的连通分量，并注意到每个连通分量都是一棵有根树。通过一次简单遍历，我们可以在线性总时间内确定此类连通分量中每个节点所属树的根节点。这样，我们便获得了足够的信息，只需再次扫描图，就能更新每条剩余边的尾端。最后，由于我们移除了图中的一些节点，因此需要重新标记剩余节点，以确保其标识符为 $\{1, 2, \ldots, n'\}$，其中 n'是节点总数。这也可以在线性时间内轻松完成，由此得到修剪图 $G' = \text{Trim}(G)$，使得每个树形图 $A \in \mathcal{A}(G)$ 都对应一个树形图$A' \in \mathcal{A}(G')$，且满足 $A = A' \cup F(G)$。因此，由于我们已将 $F(G)$ 中的所有边添加到 A，枚举 $\mathcal{A}(G)$ 就归约为枚举 $\mathcal{A}(G')$。完成后，我们必须从 A 中移除 $F(G)$ 中的边。我们将此总结为以下引理。

引理 3. 给定一个图 G，我们可以在线性时间内将枚举 $\mathcal{A}(G)$ 归约为枚举 $\mathcal{A}(G')$，其中 G' 是一个已修剪的图，仅包含 G的非平凡边。

扁平化。 我们允许原始输入图 G 中存在平行边。不过，假设图是简单图会更方便（例如，这一假设对 Uno 的方法 [19] 至关重要）。考虑一组连接 u和 v 的平行边，记作 e_1、e_2、.....、e_k。自然的做法是将它们替换为一条新的边 e，并在其中存储原始 k 边的列表。这样一来，每当枚举算法尝试调用 ADD(e) 时，我们实际上就需要遍历 k 种可能性。不过，这种做法不太便于分析，因此我们将采用另一种方式。我们仍然会将平行边合并为一条边，并在其中维护它们的列表。这个过程可以递归地重复，因此一条边 e 的物理表示是一棵树，其中每个叶节点都对应 G 的一条边 e。我们将这样的 e 视为 G 的边集，并用 $|e|$ 表示其数量。每当枚举算法尝试调用 ADD(e) 时，我们都会检查 $|e| = 1$。若是如此，我们只需将 e 添加到 A 中。否则，我们将 e 添加到另一个集合 A′ 中。最后，当枚举算法尝试调用 REPORT() 时，我们会递归遍历所有可能的选择：从每个 $e \in$ A′ 中恰好选取一条边 e $\in e$ 。具体通过调用以下方式实现的 Choose() 来完成。

算法 1： CHOOSE()

```
如果 A′ 为空 则
    报告();
否则
    e ← A′.top();
    A′.pop();
    对每个e ∈ e 执行
        添加(e);
        Choose();
        移除(e);
    end
    A′.push(e);
结束
返回
```

由于只有在 $|e| \geq 2$ 时我们才会将 e 纳入 A′，因此不难验证，在上述递归中，每报告一棵树形图所花费的总时间仅为常数。

自然的做法是将每个极大平行边组并为一条边。然而可以看出，即使从修剪图出发，这样做也可能产生强制边。这并不可取，因为我们希望确保图既已修剪又是平坦的。因此，对于每个由边 $e_1, e_2, \ldots, e_k$ 组成且满足 $k \geq 2$ 的此类组，我们只合并 $e_2, e_3, \ldots, e_k$。我们声称，给定一个修剪图 G，这使我们能够在线性时间内得到一个既已修剪又平坦的新图 G'。为证明新图不含任何强制边，考虑 G 中的一条简单路径 p，它以边 e 结束，并证明边 f（满足 $h(e) = h(f)$）不是强制边。除 e 外，p 中的每条边都可以假定是其所属组中的第一条边，因此它仍存在于 G' 中。此外，如果 e 和 f 不平行，那么也可以假定 e 是其所属组中的第一条边；否则，可以假定它是其所属组中的第一条或第二条边。无论哪种情况，G' 仍包含 p，从而证明 f（如果它在 G' 中仍然存在）不是强制边。我们将此总结为以下引理。

引理 4. 给定一个已修剪图 G，我们可以在线性时间内将枚举 $\mathcal{A}(G)$ 化简为枚举$\mathcal{A}(G')$，其中 G' 既是已修剪的，也是平坦的。

算法概述。 算法首先应用引理 3，将一般情形化简为已修剪图的情形。随后，它调用主递归过程 Recurse(G)。我们始终确保传递给 Recurse() 的参数是已修剪的。Recurse() 的第一步是应用引理 4，确保 G 既是已修剪的，也是平坦的。随后，该方法的高层思路如下。

我们先寻找并报告一棵树形图 $A = \{a_1, a_2, \ldots a_{n(G)-1}\}$。从根节点 r开始遍历 A，为 G 中的节点赋予先序编号，并假设边a_i 按照 h的先序编号递增顺序排列（a_i）。然后，可以根据其不使用的第一条边 a_i，对其余的树形图，即 $\mathcal{A}$（G）$\setminus \{A\}$，进行分组。我们将此形式化如下：令 G_i 为通过收缩 A 中由 $\{a_1, a_2, \ldots, a_{i-1}\}$ 构成的前缀并移除 a_i后得到的图。那么，对于 $i = 1$、2、.....、n（G）-1，我们确保与当前调用开始时的状态相比，A 还包含在 G_i中已收缩的 G的边，然后对 G_i进行递归调用。我们不能直接调用 Recurse(G_i)，因为 Recurse() 的输入必须是已修剪的。为确保这一点，我们实际上会采用两种不同的方法。简单的情况是，我们能够保证 G有足够多的树形图（具体而言，至少有（m（G））[4]）。在这种情况下，我们实际上可以在线性时间内调用引理 3。较复杂的情况是，G 的树形图较少。我们将证明这类图具有某种结构，特别是 m（G）$= \mathcal{O}$（n（G））。这将使我们能够设计一种机制来提取 NT（G_i），并在 $\mathcal{O}$（$|NT$（G_i）$|$）时间内获得 Trim(G_i)，前提是进行一些额外的记账，这需要总计 $\mathcal{O}$（m（G））$= \mathcal{O}$（n（G））时间。另一个难点是，在对 Trim(G_i) 进行递归调用之前，我们需要将 F（G_i）添加到 A 中，而 F（G_i）可能远大于NT（G_i）。为解决这一问题，我们将用总计 $\mathcal{O}$（n（G））时间维护一个边集 B（事实上，起初它是一棵与 A边不相交的树形图），其性质是包含 F（G_i）中的所有边，以及 NT（G_i）中的一些边。然后，我们维护一个不变式：当前 B中的每条边都属于 A；在递归调用之前，暂时从 A 中移除 $B \cap NT$（G_i）中的每条边，这只需要 $\mathcal{O}$（NT（G_i））次操作。最后，将 A 恢复到当前调用开始时的状态。假设我们

能够以这种方式实现每次递归调用，我们就能达到所承诺的整体线性时间复杂度。

引理 *5.* 假设每次调用 *Recurse(G)* 时，都会先将 G扁平化，然后报告 A 并对 *Trim(*G_i*)* 进行递归，依次处理 $i = 1$、*2*、*.....*、$n(G) - 1$，耗时为 $\mathcal{O}(n(G)m(G))$（在这种情况下，我们可以保证 $|\mathcal{A}(G)| \geq (m(G))^4$），或者耗时为 $\mathcal{O}(m(G) + \sum_i |NT(G_i)|)$（在这种情况下，我们可以保证$m(G) = \mathcal{O}(n(G))$）。那么，总时间复杂度为 $\mathcal{O}(m + N)$。

证明。我们首先将一般情形归约为如下情形：对于每次调用 Recurse(G)，输入图 G都已是平坦的。考虑这样一次调用，其中图 G最初在 $\mathcal{O}(m(G))$ 时间内被化为平坦图 G'。我们将扁平化的开销分为 $\mathcal{O}(m(G) - m(G'))$ 和 $\mathcal{O}(m(G'))$。前一部分可由以下论证说明。当前图 G 或集合 A′ 中的每条边 e 都带有 $|e| - 1$ 个单位的债务。在递归调用中，当前图 G 中的每条边 e都是非平凡的，因此要么属于第一棵树形图 A，要么属于某次后续递归调用所报告的树形图（要么存在于 Trim(G_i) 中，要么在调用前被添加到 A 或 A′ 中）。无论哪种情况，债务都不会减少，而且在 Choose() 报告新的树形图时，每条边最终至少会有一次出现在 A 或 A′ 中。随后，Choose() 会生成 $\prod_{e \in A'} |e| \geq \sum_{e \in A'} |e| - 1$ 棵树形图，足以偿还这些债务。接下来，将平行边 e_1、e_2、.....、e_k 合并为单条边e 时，我们可以将债务增加 $|e| - 1 - \sum_i (|e_i| - 1) = \sum_i |e_i| - 1 - \sum_i (|e_i| - 1) = k - 1$，将所有这类边组的增加量相加后，恰好为 $m(G) - m(G')$，符合要求。因此，在证明的剩余部分中，我们只需计算剩余开销 $\mathcal{O}(m(G'))$（以及扁平化后在 Recurse(G) 内花费的时间）；这实际上允许我们假设输入图 G已经是平坦的。

我们有两种类型的 Recurse(G) 调用，将分别进行分析。

1. 我们可能会花费 $\mathcal{O}(n(G)m(G)) = \mathcal{O}((m(G))^2)$ 的时间，但此时可以保证 $|\mathcal{A}(G)| \geq (m(G))^4$。我们将这段时间分摊给后续递归调用中报告的至少 $(m(G))^4$ 个树形图，每个树形图分配 $\mathcal{O}(1/(m(G))^2)$ 的时间。接下来需要分析分配给单个树形图的总时间。我们注意到，在每次后续的 Recurse() 递归调用中，当前图的边数都会严格减少，因为我们会在 Trim(G_i) 中至少移除一条边 a_i。因此，对于算法报告的单个树形图，分配给它的总时间是所有祖先递归调用中 $\mathcal{O}(1/(m(G))^2)$ 的总和。根据 $m(G)$ 的严格单调性，该总和至多为 $\mathcal{O}(\sum_i 1/i^2) = \mathcal{O}(1)$。

2. 我们可能会花费 $\mathcal{O}(m(G) + \sum_i |NT(G_i)|)$ 时间，但这样就能保证 $m(G) = \mathcal{O}(n(G))$。回想一下，对于 $i = 1$、2、.....、$n(G) - 1$，我们会对 $G'_i =$ Trim(G_i) 递归，并注意到 $m(G'_i) \leq |NT(G_i)|$。因此，所有第二类调用中的 $\mathcal{O}(\sum_i |NT(G_i)|)$ 之和，上界为所有调用中的 $\mathcal{O}(m(G))$ 之和。于是，只需给出所有调用中的 $\mathcal{O}(m(G))$ 之和的上界。此前已经证明，第一类调用中的 $\mathcal{O}(m(G))$ 之和为 $|\mathcal{A}(G)|$，因为$\mathcal{O}(m(G)) = \mathcal{O}(n(G)m(G))$。在假设 $m(G) = \mathcal{O}(n(G))$ 的条件下，所有第二类调用中的 $\mathcal{O}(m(G))$ 之和，上界为所有调用中的 $\mathcal{O}(n(G))$ 之和。每次调用中，我们先报告一个树形图，然后分出 $n(G) - 1$ 次后续递归调用。不难验证，这使得所有调用中的 $\mathcal{O}(n(G))$ 之和不超过报告的树形图总数，即 $\mathcal{O}(|\mathcal{A}(G)|)$。

我们已证明，所有递归调用中 Recurse() 内部耗时的总和为 $\mathcal{O}(N)$。

Recurse(G) 假设 G 已修剪，因此，整个算法 TrimAndRecurse(G) 会先使用引理 3 将 G 转换为修剪图，耗时为 $\mathcal{O}(n + m)$。因此，总体时间复杂度如所述。

算法 2： TrimAndRecurse()

1 确定 $U(G)$、$F(G)$、$NT(G)$；2 添加($F(G)$)；3 从 G 构造 Trim(G)，方法是移除 $U(G)$ 中的边并收缩 $F(G)$ 中的边；4 Recurse(Trim(G))；5 移除($F(G)$)；6 返回

□

4 算法细节

本节将介绍如何实现 Recurse()，以满足引理 5 的要求。该过程首先对输入图 G 进行扁平化；从现在起，我们假设每条边最多有两个副本。此外，每条边都是非平凡的：它属于某个树形图，但不属于另一个树形图。

该过程首先要找到一棵树形图 $A = \{a_1, a_2, \ldots, n(G) - 1\}$。如果当前是第一类调用（因此 G 至少包含 $(m(G))^4$ 棵树形图），那么 A 无须具备其他性质。然而，在第二类调用中，我们实际上拥有两棵边不相交树形图 A 和 B 至关重要：这使我们能够有效地维护一个边集 S，初始设为 B，使得对于每个 $i = 1, 2, \ldots, n(G) - 1$，都有 $F(G_i) \subseteq S$ 成立。Edmonds 的一项早期结果 [2] 表明，G 已修剪意味着确实存在这样的边不相交树形图。尽管原始证明没有给出高效的构造算法，但后续结果表明存在一个线性时间算法。此外，可以假设找到的边不相交树形图具有额外的结构；在后续分析含有少量树形图的图的结构时，这种结构将发挥作用。

引理 6. 给定一个已修剪图 $G = (V, E)$，我们可以找到两棵边不相交的树形图 $A, B \subseteq E$，使得对于每条边 $(u, v) \in E$，v 都不是 A 或 B中的 u 的祖先。

证明。 首先回顾一下，当从 r 到 v 的每条路径都经过 u时，节点 u 是 v 的支配点。Georgiadis 和 Tarjan [5] 提出了发散生成树的概念：用我们的记号表示，它们是两棵树形图 A 和 B（不一定边不相交），对于每个节点 v，v 在 A和 B 中的共同祖先恰好是它的支配点。此外，他们证明了可以在线性时间内找到这样的树形图。事实上，他们的算法保证了一个更强的性质：这些树是强发散树，并且除桥（用我们的记号表示即强制边）之外彼此边不相交。强发散树这一概念对我们的应用并不重要，重要的只是强发散树也是发散树。

我们声称，运行他们的算法会找到满足所需性质的两棵边不相交树形图。首先，考虑一条边 (u, v) 同时出现在 A 和 B 中的情况。此时它是一座桥，这与该图已修剪相矛盾。因此，这两棵树形图确实边不相交。接下来，取一条边 $(u, v) \in E$ 并假设 v 在 A 和 B中都是 u 的祖先。于是，v 支配u，这使得 (u, v) 成为无用边。这再次与该图已修剪相矛盾。 □

我们应用引理 6 中的过程，将 E 划分为 A、B和 $C := E \setminus (A \cup B)$。接下来，我们报告 A 作为下一棵树形图。然后，我们尝试检查 G 是否至少有 $(m(G))^4$ 棵树形图。不过，这有些棘手，因此我们需要采取一条更间接的途径。具体而言，我们将在第 5 节证明，少于 $(m(G))^4$, 棵树形图的修剪图称为稀疏图（与厚图相对），并具有某种简单结构。具体而言，我们将证明 $m(G) = \mathcal{O}(n(G))$，并且 G 的节点可以划分为少数几条链。一条链 ζ是节点 $\{u_1, u_2, \ldots, u_k\}$ 的集合，使得对于每个 $i = 1$、2、.....、$k - 1$，都有 $(u_i, u_{i+1}) \in A$和 $(u_{i+1}, u_i) \in B$。此外，记 $\zeta^\circ := \zeta \setminus \{u_1, u_k\}$，则没有其他边进入属于 ζ° 的任何节点。图 1 展示了一个示例。此外，如果存在 $\Theta(|C|)$ 条形如 (u, v) 的边，其中 u 在 A的先序编号中早于 v，我们就称节点 v 为粗壮节点。

引理 7。 一个薄图 G 具有 $m(G) = \mathcal{O}(n(G))$，并且可以分解为 $\mathcal{O}(1)$ 条链，使得要么 $|C| = \mathcal{O}(\sqrt{n(G)})$，要么存在一个胖节点。此外，存在一个线性时间过程，该过程要么找到这样的分解（在这种情况下，图可能是薄图，也可能不是薄图），要么确定该图是厚图。

我们应用引理 7 中的过程。如果该过程确定图是厚图，那么我们只需依次遍历 $i = 1$、2、.....、$n(G) - 1$，并分别为其中每个对象利用引理 3 构造 Trim(G_i)，然后调用 Recurse(Trim(G_i))。在描述的其余部分中，我们假设已经找到一个分解为 $\tilde{\mathcal{O}}(1)$ 条链的分解。此外，我们假设要么 $|C| = \mathcal{O}(\sqrt{n(G)})$，要么存在一个节点 v，使得存在 $\Theta(|C|)$ 条形如 (u, v) 的边，其中 u 在先序编号中早于 v。

我们仍然遍历 $i = 1,2, \ldots, n(G) - 1$。不过，现在我们将不再调用引理 3 来构建 Trim(G_i)，而是维护一种单独的机制，以在 $\mathcal{O}(|NT(G_i)|)$ 时间内提取 $NT(G_i)$。该机制将利用链分解，在 $\mathcal{O}(m(G))$ 时间内为 G 构建一个模拟图 H。

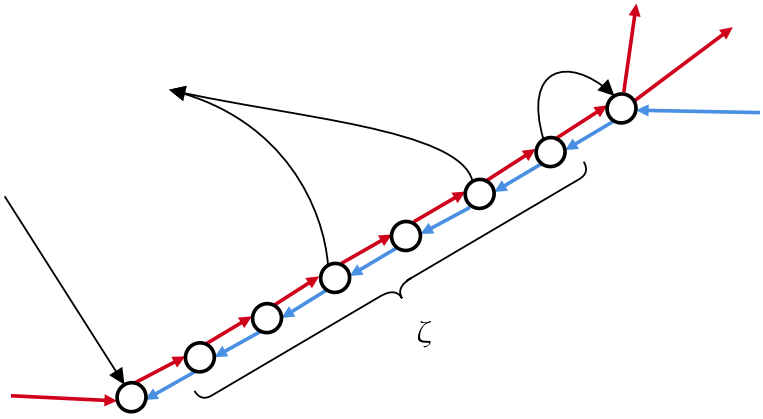

图 1：链 ζ 的一个示例。 A 和 B 的边分别绘制为红色和蓝色。第一个和最后一个节点是单节点链：其中一个的入边来自 C，另一个则有两条来自 A 的出边。

˜时间。得到 H后，对于任意 i，我们都能够在 $\mathcal{O}(|C|+\mathcal{O}(1)+|NT(G_i)|)$ 时间内提取 $NT(G_i)$。若对每个 $i=1,2,\ldots,n(G)-1$分别应用，这会太慢。我们称 i有趣，如果 $h(a_i)$ 是某条链的第一个或最后一个节点，或者存在一条从 $h(a_i)$ 发出的边 C。因此，这类 i 的数量为 $\mathcal{O}(|C|)+\mathcal{O}(1)$。我们还将证明，如果 i 不有趣，那么 $NT(G_i)=NT(G_{i+1})$；因此，实际上只需在 i有趣时，才使用 H提取 $NT(G_i)$。

˜**引理 8.** 给定一个包含 $\mathcal{O}(1)$ 条链的链分解，我们可以在 $\mathcal{O}(m(G))$ 时间内构建一个模拟˜图 H，使其能够在 $\mathcal{O}(|C|+\mathcal{O}(1)+|NT(G_i)|)$ 时间内对任意 i 支持提取 $NT(G_i)$。此外，如果 i 不重要，则 $NT(G_i)=NT(G_{i+1})$。

我们使用引理 8 在 $\mathcal{O}(m(G))$ 时间内构造 H。然后，当 $i=1$ 或 $i-1$ 是重要的时，我们使用 H 在 $\mathcal{O}(|C|+\mathcal{O}(1)+|NT(G_i)|)$ 时间内提取 $NT(G_i)$；对于其余的 i，我们只需重用$NT(G_{i-1})$。因此，对于 $i=1$、2、.....、$n(G)-1$，获得 $NT(G_i)$ 所需的总时间为 $\mathcal{O}(m(G)+(|C|+\tilde{\mathcal{O}}(1))^2+\sum_i|NT(G_i)|)=\mathcal{O}(n(G)+|C|^2+\sum_i|NT(G_i)|)$。我们需要将 $|C|^2$ 上界限制为$\mathcal{O}(n(G)+\sum_i|NT(G_i)|)$。如果 $|C|=\mathcal{O}(\sqrt{n(G)})$，那么这就是 $|C|^2=\mathcal{O}(n(G))$，符合要求。否则，根据引理 7 的假设，存在一个节点 v，使得有 $\Theta(|C|)$ 条形如 (u,v) 的边，其中u 在先序编号中早于 v。将这些边记为 $\{e_1,e_2,\ldots,e_d\}$，并按其尾节点的先序编号排序。此外，令 $a_{k(i)}$ 为 A 中满足 $h(a_{k(i)})=t(e_i)$ 的边（如果 $t(e_i)=r$，则令 $k(i)=0$），并令 a_ℓ 为 A 中满足 $h(a_\ell)=v$ 的边（我们可以假设 $v\neq r$，否则$n=\mathcal{O}(1)$）。接下来，我们声称，对于每个 $j=k(i)+1,k(i)+2,\ldots,\ell$，$e_i$ 都存在于 $NT(G_j)$ 中。这是因为，对于每个这样的 j，$t(e_i)$ 已经收缩到 r中，但 v 尚未收缩。同时，我们可以构造一个由 G_i 构成的树形图 $B'\subseteq B$。根据定义，$e_i\notin B'$，并且我们可以通过先移除进入 $t(e_i)$的入边，修改 B' 以纳入 e_i。因此，e_i 在 G_i 中确实是非平凡的。总之，我们得到 $\sum_i|NT(G_i)|=\Omega(|C|^2)$。因此，无论哪种情况，$|C|^2=\mathcal{O}(n(G)+\sum_i|NT(G_i)|)$。

此外，我们维护一个集合 S，使得 $F(G_i)\subseteq S$。为此，我们使用以下观察结果。设 b_i 是 B 中满足 $h(a_i)=h(b_i)$ 的唯一一条边。

引理 9. $F(G_i)\subseteq\{b_i,b_{i+1},\ldots,b_{n(G)-1}\}$

证明。回想一下，G_i是通过收缩 A的前缀（由 $\{a_1,a_2,\ldots,a_{i-1}\}$组成）并移除 a_i得到的。我们声称，$\{b_i,b_{i+1},\ldots,b_{n(G)-1}\}$包含 G_i的一棵树形图。对于整个 B，这显然成立。考虑 G_i的一个节点 u，令 p 为一条路径，它仅由 B 中的边组成，并在 G中连接r和 u。从 p上属于 $\{r,h$的最后一个节点开始的 p的一个后缀 p'（a_1） h（a_2）.....，h（a_{i-1}）$\}$是在 G_i中连接 r和 u的一条路径，因为在 G_i中，我们已将所有边 a_1，a_2，.....，a_{i-1}收缩为 r。因此，对于 G_i的每个节点 u，都存在一条仅由 $\{b_i,b_{i+1},\ldots,b_{n(G)-1}\}$中的边组成、连接 r

和 u 的路径，该路径位于 G_i中，因此 $\{b_i, b_{i+1}, \ldots, b_{n(G)-1}\}$ 包含 G_i 的一棵树形图（事实上，这恰好就是这样一棵树形图）。因此， G_i 的每条强制边都必须属于 $\{b_i, b_{i+1}, \ldots, b_{n(G)-1}\}$。 □

根据引理 9，对于 $S := \{b_i, b_{i+1}, \ldots, b_{n(G)-1}\}$，我们确实有 $F(G_i) \subseteq S$。因此，当我们遍历 $i = 1$、2、.....、$n(G) - 1$ 时，可以确保与调用开始时的内容相比，A 还包含来自 $\{a_1$、a_2、.....、$a_{i-1}\}$ 的所有边，以及来自 S的所有边。总共只需 $\mathcal{O}(n)$ 时间：首先，我们将整个 B 添加到 A 中，然后 for 循环的每次迭代都需要添加 a_i 并移除 b_i。

现在，对于每个 $i = 1$、2、.....、$n(G) - 1$，我们都可以访问 $NT(G_i)$。此外，A 包含边a_1、a_2、.....、a_{i-1}，它们已被收缩以得到 G_i。另外，A 还包含来自 S的所有边，其中 $F(G_i) \subseteq S$。因此，要对 Trim(G_i) 进行递归，我们只需构建 Trim(G_i)，并（暂时）从 A 中移除 $NT(G_i) \cap S$中的所有边。后者可以轻松地在 $\mathcal{O}(|NT(G_i)|)$ 时间内实现。前者则更复杂：对于每条边 $e \in NT(G_i)$，我们需要确定其新的 $t(e)$；此外，还需要确保 Trim(G_i) 的节点是 $\{1, 2, \ldots, n(\text{Trim}(G_i))\}$。后者需要对 $s = \mathcal{O}(|NT(G_i)|)$ 个元素进行排序；原则上，在比较模型中需要 $\mathcal{O}(s \log s)$ 时间。然而，如果 $|NT(G_i)| = \Omega(n)$，那么我们可以在 $\mathcal{O}(n + |NT(G_i)|) = \mathcal{O}(|NT(G_i)|)$ 时间内应用基数排序，从而避免额外付出 $\log s$ 因子，因为这些元素是 $\{1, 2, \ldots, n\}$。我们将把针对 $i = b$、$b+1$、.....、e 且满足 $\sum_{i=b}^{e} |NT(G_i)| = \mathcal{O}(n)$ 的实例批次化；对于其他每个这样的批次 $\sum_{i=b}^{e} |NT(G_i)| = \Omega(n)$。这样，我们就能对每个这样的组只应用一次基数排序，并将额外的 $\mathcal{O}(n)$ 时间摊销掉。前者可归结为一个关于对应于 B 的树的数据结构问题；类似地，对于批次中的每个 i，都可以在 $\mathcal{O}(|NT(G_i)|)$ 时间内解决。以上内容由以下引理概括，证明推迟到第 7 节。

引理 10. *对于每个 $i = b, b+1, \ldots, e$，我们可以在 $NT(G_i)$ 时间内构建并存储 Trim(G_i)，空间为 B，其中 $\mathcal{O}(n + \sum_{i=b}^{e} |NT(G_i)|)$。*

我们指出，从达到所声称的总体时间复杂度这一角度来看，可以应用引理 10，其中 $b = 1$ 和 $e = n(G) - 1$。但是，这会使空间复杂度增加到 $\Omega((n(G))^2)$，而我们的目标是将空间复杂度保持为 $\mathcal{O}(m)$，如第 8 节所述。相反，我们将 $1, 2, \ldots, n(G) - 1$ 划分为若干片段 $b, b+1, \ldots, e$。对于每个这样的片段，要么 $b = e$ 和$|NT(G_i)| > c \cdot n$，要么 $\sum_{i=b}^{e} |NT(G_i)| \leq c \cdot n$ 但 $\sum_{i=b}^{e+1} |NT(G_i)| > c \cdot n$ 或 $e = n(G) - 1$，其中 c是一个足够小的常数，稍后讨论空间复杂度时再进行调整。对于满足$b = e$的片段，我们通过对 $B \cup NT(G_i)$ 应用引理 3，在 $\mathcal{O}(n)$ 时间内得到 Trim(G_i)。根据假设$|NT(G_i)| > c \cdot n$，这实际上是 $\mathcal{O}(|NT(G_i)|)$。对于其余的每个片段，我们应用引理 10。可以直接验证，总时间复杂度为 $\mathcal{O}(n + \sum_{i=1}^{n(G)-1} |NT(G_i)|)$。因此，总时间复杂度满足应用引理 5 的要求。

最后，我们给出整个过程的伪代码，并通过一个定理总结所得的时间复杂度（具体细节将在后续章节中介绍）。需要指出的是，时间复杂度中的额外$\mathcal{O}(n)$仅仅是因为输入图可能连一棵树形图都不存在。

定理 2. *有向图中，具有 n 个节点和 m 条边的所有 N 树形图均可在总计 $\mathcal{O}(n + m + N)$ 时间内报告。*

算法 3： RECURSE(G)

1 平坦化 G**； 2 查找两个边不相交树形图** A **和** B**； 3 添加(**A**)、Choose() 和移除(**A**)； 4 如果** G**的链分解失败，则 5 对于** $i = 1$**，2，. . . ，** n**(**G**) −** 1 **执行 6 TrimAndRecurse(**G_i**)； 7 结束 8 否则** 9 $S \leftarrow B$ **； 10 添加 (**B**)； 11 对于** $i = 1$**，2，. . . ，** n**(**G**) −** 1 **执行 12 维护一个模拟图** H_i**，以便快速检索** NT**(**G_i**)； 13 移除 (**NT**(**G_i**) ∩** S**)； 14 使用** NT**(**G_i**) 和** B **构建 Trim(**G_i**)；** ▷ **对** $i = i_j$**、** $i_j + 1$**、. . . 、** $i_{j+1} - 1$**进行批处理 15 Recurse(Trim(**G_i**))； 16 添加(**NT**(**G_i**) ∩** S**);** 17 $S \leftarrow S \setminus \{b_i\}$**； 18 添加(**$a_i$**)并移除(**$b_i$**)； 19 结束 20 移除(**$A$**)； 21 结束**

5 薄图的链分解

本节的目标是分析修剪图 $G = \langle V, E, r\rangle$ 的结构，该图有 n 个节点和 m 条边，且所含树形图少于 m^4 个。我们首先应用引理 6，在线性时间内找到两个边不相交树形图 A 和 B，使得对每条边 $(u, v) \in E$，v 在 A 中或在 B 中都不是 u 的祖先。这将 E 划分为 A、B 和 $C := E \setminus (A \cup B)$。我们的目标是证明：假设该图所含树形图少于 m^4 个，则有 $|C| = \mathcal{O}(n)$，并且 G 的节点可以被˜划分为 $\mathcal{O}(1)$ 条链。回顾一下，链 ζ 是节点集合 $\{u_1, u_2, \ldots, u_k\}$，使得对每个 $i = 1$、2、.....、$k-1$，都有 $(u_i, u_{i+1}) \in A$ 以及 $(u_{i+1}, u_i) \in B$。此外，记 $\zeta^\circ := \zeta \setminus \{u_1, u_k\}$，则不存在其他边进入属于 ζ° 的任何节点。本节的目标是证明如下结论。

˜**引理 7**。稀疏图 G 有 $m(G) = \mathcal{O}(n(G))$，并且存在由 $\mathcal{O}(1)$ 条链组成的分解，使得满足以下条件之一：$|C| = \mathcal{O}(\sqrt{n(G)})$，或者存在一个胖节点。此外，存在一个线性时间过程，它要么找到这样的分解（此时图可能是也可能不是稀疏图），要么判定该图是厚图。

我们首先分析这类图中的边数。

引理 11. $|C| = \mathcal{O}(m)$

证明。回忆一下，对于每条边 $(u, v) \in C$，v 在 A 中或 B中都不是 u 的祖先。假设$|C| \geq 64n$。要么至少有 $|C|/2$ 条边 $(u, v) \in C$ 满足：v 在 A中不是 u 的祖先；要么至少有 $|C|/2$ 条边 $(u, v) \in C$ 满足：v 在 B中不是 u 的祖先。假设前一种情况成立（另一种情况对称），并将相应的边集记为 $C' \subseteq C$，其中 $|C'| \geq 32n$。我们为 A中的每个节点赋予先序编号。要么至少有 $|C'|/2$ 条边 $(u, v) \in C'$ 满足：u 在先序编号中位于 v 之前；要么至少有 $|C'|/2$ 条边 $(u, v) \in C'$ 满足：u 在先序编号中位于 v 之后。假设前一种情况成立（另一种情况对称，因为 v 不是 u的祖先），并将相应的边集记为 $C'' \subseteq C$，其中 $|C''| \geq 16n$。总而言之，对于每条边 $(u, v) \in C''$，u 在先序编号中位于 v 之前。

对于节点 $v \in V$，我们将 $d(v)$ 定义为边数（u，v）$\in C''$。由于该图是已修剪的，我们有 $d(v) \leq 2\,n - 1$。我们声称可以构造 $\prod_{v \neq r}(1 + d(v))$ 个互不相同的树形图。为此，考虑节点 $v \in V$，并按照其先序编号递增的顺序排列。对于每个节点 $v \neq r$,

可以从 C'' 中选择其 $d(v)$ 条入边之一，或者选择其父节点在 A 中的边。根据 C'' 的定义，任何此类选择都会产生一个树形图，总共有 $\prod_{v\neq r}(1+d(v))$ 种可能性。

现在，为了给树形图的数量建立下界，我们只需在约束 $\sum_{v\neq r} d(v) \geq 16n$ 和 $d(v) \leq 2(n-1)$ 下，给 $\prod_{v\neq r}(1+d(v))$ 建立下界。通过分析函数 $f(x) = x(s-x)$，我们可以看出，为了使乘积最小，应尽可能多地将节点 v 的 $d(v)$ 设为 $= 2(n-1)$。但是，由于 $\sum_{v\neq r} d(v) \geq 16n$，我们随后必须将至少 8 个节点 v 的 $d(v)$ 设为 $= 2(n-1)$，这会使乘积至少为 $n^8 \geq m^4$，与假设矛盾。 □

接下来，我们分析具有来自 C 的入边的节点数量。

引理 12. 存在 $\mathcal{O}(\text{对数 } n)$ 个节点 v，使得存在一条边 $(u,v) \in C$。

证明。 反设至少存在 $32\log n$ 个这样的节点。我们按照引理 11 的证明，对 C 中的边进行类似划分，但不再考虑边的数量，而是跟踪具有入边的节点数量。由此得到边集 $C'' \subseteq C$，使得至少有 $8\log n$ 个节点 v 满足存在一条边 $(u,\ v) \in C''$。但这使我们能够构造至少$2^{8\log n} = n^8 \geq m^4$ 个树形图，产生矛盾。 □

最后，我们分析 A 和 B 中叶节点的数量。

引理 13。 $\mathcal{O}(\log n)$ 叶节点位于 A 和 B 中。

证明。 考虑 A 中的叶节点（分析 B 中叶节点数量的方法相同），并用 $L = \{v_1, v_2, \ldots, v_\ell\}$ 表示它们。假设 $\ell \geq 8\log n$。对于每个子集 $L' \subseteq L$，我们按如下方式得到一个树形图：取出 A 中除通向v中叶节点的边以外的所有边。这将生成一个部分树形图 A'。然后，对于每个 $v \in L'$，我们将一条边从 v 的父节点添加到 B 中的 v。这样可以保证每个节点 $v \neq r$ 恰好有一条入边。为说明添加的边不会形成环，注意，我们添加到 A' 中的、来自 B 的边构成若干条路径；每条路径都从某个 $u \in A'$ 开始，然后经过 L'中的一些叶节点，并且任意两条路径都不会经过同一个 $v \in L'$。得到的树形图数量为 $2^{8\log n} = n^8 \geq m^4$，产生矛盾。 □

现在我们可以定义链了。这将分两步完成：首先，我们将节点划分为 $\mathcal{O}(\log n)$ 条近似链，作为初始划分；然后再将每条近似链进一步细分为$\mathcal{O}(\log n)$ 条链，从而总共得到 $\mathcal{O}(\log^2 n)$ 条链。

近似链的定义如下。首先，我们指定根节点以及 A 中每个有多个子树形图的节点各自构成一条近似链。同样，我们指定每个有来自C 的入边的节点各自构成一条近似链。根据引理 12 和引理 13，这样会产生 $\mathcal{O}(\log n)$ 条单节点近似链。接着，我们注意到，剩余节点可以在 A中划分为极大路径；每条这样的路径都从 A 中某个有多个子树形图的节点的子树形图开始，或者从某个有来自C的入边的节点开始，然后沿着 A向下延伸，直到另一个这样的节点之前终止（或者到达叶节点后终止）。此外，这类极大路径的数量至多为：对 A 中所有有多个子树形图的节点，将它们在 A 中的度相加，再加上来自 C 的入边所对应的节点数量。根据引理 13 和引理 12，这分别为 $\mathcal{O}(\log n)$。

现在，考虑第 i 条近似链，并设其节点为 $\{u_1^i, u_2^i, \ldots, u_{k_i}^i\}$，其中对于每个 $j = 1$、2、.....、$k_i - 1$，都有 $(u_j^i, u_{j+1}^i) \in A$。我们为 B 中的每个节点赋予先序编号。我们注意到，对所有 i 求和，满足以下条件的索引 j 的数量有界于 $8\log n$：u_j^i 是 B 中 u_{j+1}^i 的祖先；否则，我们就可以（按照引理 12 的证明中的方法）构造至少 m^4 个树形图。类似地，对所有 i 求和，满足以下条件的索引 j 的数量有界于 $8\log n$：u_j^i 的先序编号小于 u_{j+1}^i 的先序编号，且两节点互不为对方的祖先。对称地，对所有 i 求和，满足以下条件的索引 j 的数量也有界于 $8\log n$：u_j^i 的先序编号大于 u_{j+1}^i 的先序编号，且两节点互不为对方的祖先。对于每个这样的索引 i，我们通过删除边 (u_i, u_{i+1})，将对应的近似链拆分为两条。这样会使近似链的数量增加 $\mathcal{O}(\log n)$。因此，现在我们有 $\mathcal{O}(\log n)$ 条近似链。将其中第 i 条近似链的节点记为 $\{u_1^i, u_2^i, \ldots, u_{k_i}^i\}$，对于每个 $j = 1$、2、.....、$k_i - 1$，仍有 $(u_j^i, u_{j+1}^i) \in A$；此外，对于每个 $j = 1$、2、.....、$k_i - 1$，u_{j+1}^i 是 B 中 u_j^i 的祖先。

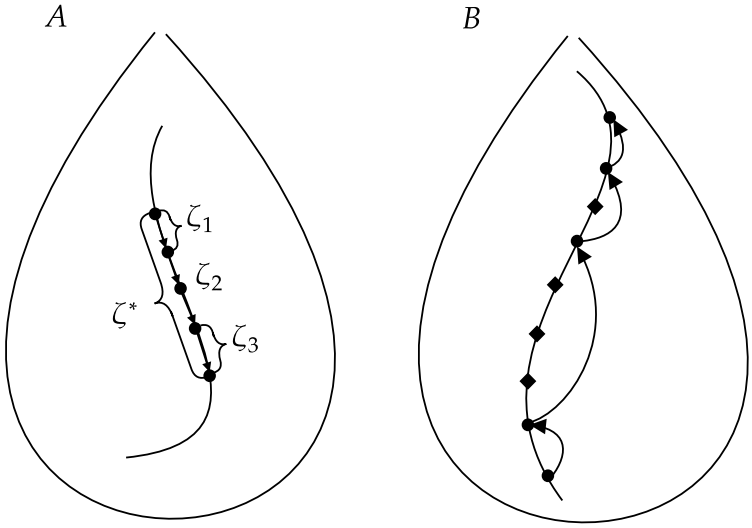

图2： ζ^* 是一个被划分为三条链的近似链。

最后一步是考虑一条近似链，设其节点为 $\{u_1, u_2, \ldots, u_k\}$，并对满足 $(u_{i+1},\ u_i) \notin B$ 的索引 i 的数量进行界定，换言之，u_{i+1} 不是 B 中 u_i 的父节点。对于每个这样的 i，都存在一条从 u_i 到 u_{i+1} 的路径，该路径由 B 中至少两条边组成。此外，这些路径节点不相交，但每条路径的末节点可能与后续路径的首节点相同。对于这样的每条路径，我们声称可以找到一个内部节点 v，使得对于它在路径上的后继 v'，有 $(v,\ v') \in B$，且 v' 不是 A 中 v 的祖先。为证明这一点，反设不然。那么，路径上的第一条边从 u_{i+1} 指向 u_i 的一个真祖先（它不可能是 u_i 本身，因为该路径至少由两条边组成），随后路径便沿着 A 向上延伸。因此，它不可能返回到 u_i，这就产生了矛盾。所识别出的节点 v' 两两不同，因为去掉首节点后的路径节点不相交。于是，如果此类问题索引 i 至少有 $32 \log n$ 个，那么按照引理 12 证明中的论证就会得到矛盾。因此，这类索引有 $\mathcal{O}(\log n)$ 个；对其中每个索引，我们都通过删除边 $(u_i,\ u_{i+1})$，将近似链拆分为两条。见图2。至此，构造完成。

我们证明了，对于一个包含 n 个节点和 m 条边、且其树形图数量少于 m^4 的修剪图，~我们可以将其节点划分为 $\mathcal{O}(1)$ 条链，并满足所需性质。此外，我们观察到，直接遵循上述过程即可在线性时间内找到这样的划分。该过程可用于任何修剪图，无论其树形图数量如何；显然，~它总能在线性时间内运行。如果该过程只生成 $\mathcal{O}(1)$ 条链，我们就称该图为稀疏图。最终，我们得到如下结论：给定一个修剪图，我们可以在线性时间内处理它，并得出其至少包含 m^4 个树形图，或者找到其节点的一个划分~，将节点划分为 $\mathcal{O}(1)$ 条满足所需性质的链。

剩下要证明的是，如果 $|C| = \Omega(\sqrt{n})$，那么就存在一个胖节点。与引理 11 的证明一样，令我们选择 $C'' \subseteq C$，使得 $|C''| \geq |C|/4$；在必要时交换 A 和 B，并反转每个节点处子节点的顺序后，对于每个 $(u,\ v) \in C''$，v 不是 A 中 u 的祖先，并且v 在 A 的先序编号中位于 u 之后。接下来，我们按头端对 C'' 中的边分组。由引理 12 可知，这样的组数仅为 $\mathcal{O}(\log n)$。因此，大小小于 $|C''|/\log^2 n$ 的组中的边总数为 $\mathcal{O}(|C''|/\log\ n)$，较大组中的边总数为 $\Omega(|C''|)$。此外，如果至少有 17 个这样的较大组，那么从每个组中选择一条边，再从 A 中添加足够的边以构成一个树形图，就能构造出至少 $(|C''|/\log^2 n)^{17} > n^8 \geq m^4$ 个树形图（当 n 足够大时），因此该图不是稀疏图，我们可以就此结束。否则，较大组少于 17 个，所以其中必有一组包含 $\Omega(|C''|) = \Theta(|C|)$ 条边，且这些边的头端都相同，对应于所需的胖节点。

这就证明了引理 7。

6 模拟图

目标是　本节旨在说明如何构造具有以下性质的模拟图 H　　s.

˜**引理 8**。给定一个含有 $\mathcal{O}(1)$ 条链的链分解，我们可以在 $\mathcal{O}(m(G))$ 时间内构建一个模拟˜图 H，该图支持在 $\mathcal{O}(|C|+\mathcal{O}(1)+|NT(G_i)|)$ 时间内提取 $NT(G_i)$，适用于任意 i。此外，如果 i 不是感兴趣节点，则 $NT(G_i) = NT(G_{i+1})$。

我们首先将每条链划分为子链，定义如下。考虑一条链 $\zeta = \{u_1, u_2, \ldots, u_k\}$，使得对于每个 $i = 1, 2, \ldots\ldots, k-1$，$(u_i, u_{i+1}) \in A$ 且 $(u_{i+1}, u_i) \in B$。那么，子链是两个感兴趣节点之间的一个极大片段 $\{u_b, u_{b+1}, \ldots, u_e\}$，即 u_b 和 u_e 是˜感兴趣的，而 $u_{b+1}, u_{b+2}, \ldots\ldots, u_{e-1}$ 不是。由于链的数量为 $\mathcal{O}(1)$，˜子链的数量为 $\mathcal{O}(|C|+\mathcal{O}(1))$。现在我们按如下方式从 G 得到 H。对于每个子链 $\{u_b, u_{b+1}, \ldots, u_e\}$，若满足 $b < e$ (我们称这样的子链为长)，则用以下构件替换它。首先，移除原有的内部节点 $\{u_{b+1}, u_{b+2}, \ldots, u_{e-1}\}$。接着，添加以下边：

(u_b, u_e)：这条边模拟边 (u_i, u_{i+1})，适用于 $i = b, b+1$、……、$e-1$。

(u_e, u_b)：这条边模拟边 (u_{i+1}, u_i)，适用于 $i = b, b+1$、……、$e-1$。

此外，每条长子链都存储两个边的双向链表：(u_b, u_{b+1})，(u_{b+1}, u_{b+2})……，以及类似的 (u_e, u_{e-1})，(u_{e-1}, u_{e-2})……。对于 G 中其余的每条边 e，它仍然存在于 H 中（并模拟自身）。因此，G 的边被划分为一个边集，其中每条边都由 H 中的一条边模拟；我们还维护一个列表，记录 H 中每条边所模拟的边。

对于 G 的一个子图 G'（通过仅保留边的一个子集 $E' \subseteq E$ 得到），我们可以自然地定义其模拟图 H'：移除 H 中模拟 $E \backslash E'$ 中某条边的每条边。此外，对于每条长子链，我们存储一个指针，指向列表 (u_b, u_{b+1})，(u_{b+1}, u_{b+2})…… 中第一条被移除的边；类似地，还存储一个指针，指向列表 (u_e, u_{e-1})，(u_{e-1}, u_{e-2})…… 中第一条被移除的边。现在我们有如下性质。

引理 14. 考虑 G 的一个子图 G'，其中 r 可以到达 G' 中的任意节点，并令 H' 为其模拟图。那么，给定 $U(H')$、$F(H')$ 和 $NT(H')$，我们可以在 $\mathcal{O}(|H'|+|NT(G')|)$ 时间内生成 $NT(G')$。

证明。G' 的每棵树形图都可以自然地映射到 H' 的一棵树形图（H' 的一棵树形图可能对应 G' 的多棵树形图）。对于 H' 中不属于为长子链创建的构件的边，我们只需检查它是否为非平凡边；如果是，就将其仿真边添加到结果中。

考虑为长子链 $\{u_b, u_{b+1}, \ldots, u_e\}$ 创建的构件。检查 $U(H')$ 后，我们立即可以推断是否存在一棵包含 (u_b, u_e) 的 H' 树形图；同理，也可以推断是否存在一棵包含 (u_e, u_b) 的 H' 树形图。检查进入 u_b 和 u_e 的其他边后，我们可以推断是否存在一棵不包含 (u_b, u_e) 的 H' 树形图；同理，也可以推断是否存在一棵不包含 (u_e, u_b) 的 H' 树形图。可以验证，当存在一棵不包含前一条边的树形图时，也存在一棵不同时包含 (u_b, u_e) 和 (u_e, u_b) 的 H' 树形图。对于所有长子链，这些信息都可以在 $\mathcal{O}(|H'|)$ 总时间内提取出来；从现在起，我们假设这些信息已可用。

现在我们可以说明如何检查长子链的每条边是否并非无用。首先，如果存在包含 (u_b, u_e) 的 H' 的树形图，则存在包含 (u_b, u_{b+1})、(u_{b+1}, u_{b+2})……(u_{e-1}, u_e) 的 G' 的树形图。类似地，如果存在包含 (u_e, u_b) 的 H' 的树形图，则存在包含 (u_e, u_{e-1})、(u_{e-2}, u_{e-3})……(u_{b+1}, u_b) 的 G' 的树形图。最后，如果存在不同时包含 (u_b, u_e) 与 (u_e, u_b) 的 H' 的树形图，则我们可以推断如下。Choose 最大的 e'，使得 (u_b, u_{b+1})、(u_{b+1}, u_{b+2})……$(u_{e'-1}, e')$ 存在于 G' 中。类似地，选择最小的 b'，使得 (u_e, u_{e-1})、(u_{e-1}, u_{e-2})……$(u_{b'+1}, b')$ 存在于 G' 中。我们观察到必须有 $e' \geq b'-1$，否则在 G' 中并非所有节点都可从 r 到达。于是，存在包含 (u_b, u_{b+1})、(u_{b+1}, u_{b+2})……$(u_{e'-1}, e')$ 的 G' 的树形图，类似地，存在包含 (u_e, u_{e-1})、(u_{e-1}, u_{e-2})……$(u_{b'+1}, b')$ 的 G' 的树形图。这完全刻画了 G' 的哪些边并非无用：(u_b, u_{b+1})、(u_{b+1}, u_{b+2})……$(u_{e'-1}, u_{e'})$，以及类似地 (u_e, u_{e-1})、(u_{e-1}, u_{e-2})……$(u_{b'+1}, u_{b'})$。

回想一下，当边 (u, v) 并非无用，且存在一条并非无用的替代边 (u', v) 时，该边是非平凡的。对于 $(u_b, u_{b+1}), (u_{b+1}, u_{b+2}) \ldots, (u_{e-2}, u_{e-1})$，替代边是

(u_e, u_{e-1})、(u_{e-1}, u_{e-2}).....$(u_{b'+1}, u_{b'})$ 之一。因此，在这些边中，非平凡边恰好是 $(u_{b'}, u_{b'+1})$、$(u_{b'+1}, u_{b'+2})$.....$(u_{\min(e'-1,e-2)}\ u_{\min(e',e-1)})$。对于 (u_e, u_{e-1})、(u_{e-1}, u_{e-2}).....(u_{b+2}, u_{b+1})，替代边是 (u_b, u_{b+1})、(u_{b+1}, u_{b+2}).....$(u_{e'-1}, u_{e'})$ 之一，从而使非平凡边恰好是 $(u_{e'}, u_{e'-1})$、(u_{e-1}, u_{e-2}).....$(u_{\max(b'+1,b+2)}\ u_{\max(b',b+1)})$。无论哪种情况，都可以先沿着指针找到相应列表上第一条被移除的边，必要时再移动到前驱，然后逐一提取这些边，从而提取出非平凡边。最后，对于 (u_{e-1}, u_e) 和 (u_b, u_{b+1})，我们可以在常数时间内直接检查相关的边。

□

回忆一下，G_i 是通过对由 $\{a_1, a_2, \ldots, a_{i-1}\}$ 构成的 A 的前缀进行收缩并移除a_i而得到的，我们的目标是得到 $NT(G_i)$。我们将 G'_i 定义为从 G 得到的图：对每一个 $j = 1$、2、.....、$i-1$，移除所有进入 $h(a_j)$的边（ a_j除外），并额外移除 a_i。可以很容易验证 $NT(G_i) = NT(G'_i)$（此处我们略微滥用记号：由于 G_i中的收缩，某些边的尾节点有所不同，但它仍然是同一个边集）。此外，我们可以在 $\mathcal{O}(|H|)$时间内从 H 得到 G'_i的模拟图，记为 H'_i： H 中每一条仿真自身的边都可以在常数时间内直接检查，并且对于 H中其余的每一条边，我们可以通过存储其尾节点的先序编号范围（在 A中），在常数时间内检查 G 的任意仿真边是否在 G'_i 中被移除。接下来，我们在 $\mathcal{O}(|H'|)$时间内对 H' 应用引理2，以在 $\mathcal{O}(|H'|)$时间内得到$U(H')$、 $F(H')$和 $NT(H')$。最后，我们使用引理 14，在 $\mathcal{O}(|H'| + |NT(G'_i)|)$时间内得到 $NT(G'_i) = NT(G_i)$。这需要获取每个长子链的列表上第一条被移除边的指针，而这些指针可以对每一个 i 在 $\mathcal{O}(m(G))$时间内预先计算。总体而言，这使我们˜能够按要求在 $\mathcal{O}(|C| + \mathcal{O}(1) + |NT(G_i)|)$时间内得到 $NT(G_i)$。

剩下需要说明的是，当 i不具有特殊性时， $NT(G_i) = NT(G_{i+1})$。如果 i不具有特殊性，那么a_i和 a_{i+1}属于长子链 $\{u_b, u_{b+1}, \ldots, u_e\}$，其中 $a_i = (u_j,\ u_{j+1})$，而 $a_{i+1} = (u_{j+1},\ u_{j+2})$。 G_i 与 G_{i+1} 的区别在于，在 G_i中，我们移除 $(u_j,\ u_{j+1})$；而在 G_{i+1} 中，我们移除 $(u_{j+1},\ u_{j+2})$ 和 $(u_{j+2},\ u_{j+1})$。接着，由于没有其他边从 u_{j+1}出发， G_i 的树形图与 G_{i+1}的树形图之间存在自然双射：前者的每个树形图中都有 $(u_{j+2},\ u_{j+1})$，而后者的每个树形图中都有 $(u_{j+1},\ u_{j+2})$。可以验证，这不会改变任何边的非平凡性。

7 Trim(G_i) 的批量构造

本节旨在说明如何以如下时间复杂度，为每个 $i = b$、$b+1$、.....、e 构造 Trim(G_i)。

引理 10. 对于每个 $i = b$、$b+1$、.....、e，我们可以在 $\mathcal{O}(n + \sum_{i=b}^{e} |NT(G_i)|)$ 时间内，使用 $NT(G_i)$ 和 B 构建并存储 *Trim(G_i)*。

我们首先注意到，Trim(G_i) 的节点恰好是 r以及所有满足 $f(h)$ 的节点，其中 $G_i(f \in NT)$。它的边对应于 $G_i(NT)$ 中的边；更确切地说，对于每条这样的边 $G_i(f \in NT)$，我们应将 $f(t)$ 替换为 B 中最近的祖先，该祖先要么是 Trim(G_i) 的节点，要么属于 A 的已收缩前缀；而 $f(h)$ 保持不变。我们需要确定如何为每个 $f \in NT(G_i)$ 更新 $f(t)$。这将分两步完成。首先，对于每个 $f \in NT(G_i)$，我们将在 B 中找到 $f(t)$ 的最近祖先，该祖先是 Trim(G_i) 的节点。其次，对于每个 $f \in NT(G_i)$，我们将找到 $t(f)$ 中属于 A 已收缩前缀的最近祖先。我们先对每个 $i = b$、$b+1$、.....、e 一起执行第一步，再对每个 $i = b$、$b+1$、.....、e 一起执行第二步，然后，对于每个 $i = b$、$b+1$、.....、e 以及每个 $f \in NT(G_i)$，将 $t(f)$ 设为第一步和第二步找到的祖先中较近的那个。最后，我们需要确保：若以 n_i 表示 Trim(G_i) 中节点的数量，则其节点的编号为 $\{1, 2, \ldots, n_i\}$（而非 $\{1, 2, \ldots, n\}$）。如果 Trim(G_i) 的节点已经排好序（例如，按它们在 B 中的先序编号排序），那么我们可以在线性时间内扫描并重新编号。不过，我们需要谨慎排序，以确保总时间为线性时间。当然，我们可以使用基数排序，在 $\mathcal{O}(n + |NT(G_i)|)$ 时间内对 Trim(G_i) 的节点排序。为了避免对每个 i 单独付出 $n(\mathcal{O})$ 的代价，我们将所有输入合并成一批，针对 $i = b$、$b+1$、.....、e 一起进行基数排序，并将它们一并排序；每个组只需付出一次 $\mathcal{O}(n)$ 的代价。

现在我们来说明如何实现第一步和第二步。

第一步。在这一步中，对于每个 $i = b, b+1, \ldots, e$ 以及每个 $f \in NT(G_i)$，我们要在 B 中找到 $t(f)$ 的最近祖先，且该祖先是 Trim(G_i)中的一个节点。我们为 B中的每个节点分配前序和后序编号，并构造两个节点集合。集合 D_i 由 r 以及对每个 $f \in NT(G_i)$而言的节点 $h(f)$组成。集合 Q_i则包含对每个 $f \in NT(G_i)$而言的节点 $t(f)$。接着，我们的目标是为每个 $u \in Q_i$找到其最近祖先 $v \in D_i$。我们按照先序编号对 Q_i 和 D_i 中的所有节点进行排序。使用基数排序，总时间可以控制在 $\mathcal{O}(n + \sum_{i=b,\ldots,e} |NT(G_i)|)$。然后，我们按先序编号递减的顺序遍历 Q_i和 D_i中的节点（如果某个 u同时属于 Q_i和 D_i，则应先将其视为 $u \in Q_i$），同时维护一个初始为空的栈。遇到节点 $u \in Q(i)$后，将其压入栈中。遇到节点 $v \in D(i)$后，我们检查栈顶节点 u 是否位于 v的子树中；借助预先计算好的前序和后序编号，可以在常数时间内完成检查。如果是，那么我们就确定了 u的最近祖先是 v，将 u 从栈中弹出，并重复检查。操作总数与 $Q(i)$和 $D(i)$的规模之和呈线性关系。

第二步。在这一步中，对于每个 $i = b, b+1, \ldots, e$ 以及每个 $f \in NT(G_i)$，我们旨在找出 B中 $t(f)$ 的最近祖先，且该祖先属于 A 已收缩的前缀。令 $M(i)$ 表示所有这类节点组成的集合。我们注意到，集合 $M(.)$ 具有单调性： $M(b) \subseteq M(b+1) \subseteq \ldots \subseteq M(e)$。事实上， $M(1) = \{r\}$，且 $M(i+1) = M(i) \cup \{h(a_i)\}$。因此，我们可以将这一步重新表述为经典的并查集问题，具体如下。起初，每个节点各自构成一个单元素集合。对于每个节点$u \notin M(e)$，我们将其所在集合与 B中其父节点所在的集合合并。每个集合都会记录其最靠近根的节点。我们依次处理 $i = e, e-1, \ldots, b$。首先，对于每个 $f \in NT(G_i)$，我们找到包含 $t(f)$ 的集合，并取出其中最靠近根的节点 u。根据构造， u 是 B 中 $t(f)$ 的最近祖先，且属于 A 已收缩的前缀。然后，我们将 $h(a_i)$ 所在的集合与 B中其父节点所在的集合合并。最后，我们注意到，这是并查集问题的一个特例：也就是说，我们预先知道这些合并操作的结构。因此，我们可以采用 Gabow 和 Tarjan 的线性时间结构 [4]，在总计 $\mathcal{O}(n + \sum_{i=b,b+1,\ldots,e} |NT(G_i)|)$ 时间内，处理大小为 n 的集合上的至多 $n + \sum_{i=b,b+1,\ldots,e} |NT(G_i)|$ 次合并与查找操作。

8 改进空间复杂度

在本节中，我们将仔细分析枚举算法的空间复杂度。调用 Recurse(G) 内部使用的空间仅为 $\mathcal{O}(m(G))$。然而，栈上最多可能有 m 次这样的递归调用，因此总体空间使用量为 $\mathcal{O}((m)^2)$。

我们的目标是将总体空间使用量降至仅 $\mathcal{O}(m)$。为此，我们首先需要避免在扁平化过程中创建 G 的新副本。我们改为修改 G，并在递归调用返回时恢复其原始状态。具体实现方式是：对于合并到 e 中的每个边组 e_1，e_2，$\ldots$，e_k，存储当前递归调用的深度。然后，在递归调用返回之前，我们扫描 G，并将当前调用开始时在扁平化过程中创建的每条边恢复为原始边。这只需额外花费 $\mathcal{O}(m(G))$ 的时间。

接下来，查找 A 和 B 并存储链分解需要 $\mathcal{O}(m(G))$ 的工作空间，而且这不会随着后续递归调用累积。然而，我们需要存储 A、 B ，以及可能获得的链分解；每次递归调用都需要 $\mathcal{O}(n(G))$ 的空间，因此我们需要调整实现。

我们注意到，如果 G是厚的，那么实际上可以为每个 $i = 1$，2、.....、 $n(G)-1$重新计算 A（而且完全不需要 B）。因此，我们只需存储 i，这只占用常数空间。接下来，需要构造 Trim(G_i)，并将其作为参数传递给后续递归调用。我们无法在后续递归调用期间保留整个 G。相反，我们只保留足够的信息，以便在递归调用返回后重建 G。回顾一下，Trim(G_i)包含 G_i的非平凡边，而这些边是 G的边的子集。因此，我们只需保留 G 中未出现在 Trim(G_i)中的边的列表，以及这些边在 G 中对应的端点（由于收缩，这些端点在 Trim(G_i)中可能不同），并记录它们在对应的出边和入边列表中的位置。对栈上所有递归调用的此类列表求和，其总大小仅为 $\mathcal{O}(m)$。此外，对于 Trim(G_i)的每条边 (u', v')，我们还需要能够恢复其在 G中的原始端点 (u, v)。我们只说明如何恢复 u，因为恢复 v 也可以用相同的方法实现。回顾一下，Trim(G_i)的构造过程是先收缩一些边，这可以看作是将多个节点合并

u_1，u_2、……、u_k 合并为 u_1，然后重新命名节点，使整个节点集合变为 $\{1, 2, \ldots, n'\}$，其中$n' = n$(Trim(G_i))。我们分两步恢复 u：先撤销重命名，再撤销合并。

重命名。设重命名之前 Trim(G_i) 的节点为 $V' = \{u_1, u_2, \ldots, u_{n'}\}$，其中 $u_1 < u_2 < \ldots u_{n'}$。于是，$u_i$ 的新名称为 i。我们存储 $V \setminus V'$；在栈上的所有递归调用中求和时，这只需 $\mathcal{O}(n)$ 空间。然后，从递归调用返回后，我们使用 $V \setminus V'$ 准备一个将 i 映射回 u_i的数组，并扫描 $V \setminus V'$ 的边，以恢复其端点的原名称。

折叠。从 u_1 出发的每条边都对应于原本从某个 u_i出发的一条边。从每个节点出发的边都存储在一个列表中，并且我们维持一个不变式：递归调用不会改变每个列表中边的顺序。因此，我们可以安排从 u_1 出发的边，使得首先是原本从 u_1出发的边，然后是从 u_2出发的边，依此类推（直到 u_k）。对于 $k = 1$，无需进行任何更改，因此不需要额外内存。如果 $k \geq 2$，那么我们将该列表划分为 k 个片段，使得第 i个片段对应于原本从 u_i出发的边，并存储一个包含 u_1、u_2、……、u_k的列表。这需要 $\mathcal{O}(k)$ 空间，并允许通过对从 u_1出发的边进行线性扫描来恢复原始端点。此外，$k-1$ 等于为获得 u_1而收缩的边数，因此，在所有收缩节点组上对 $\mathcal{O}(k)$ 求和，至多为 G 中不存在于 Trim(G_i) 的边数；在栈上的所有递归调用中求和时，这一数值为 $\mathcal{O}(m)$。

现在我们转而调整稀疏 G 的实现。回想一下，我们区分两种情况。如果 $|NT(G_i)| \geq c \cdot n(G)$，那么我们会在 $\mathcal{O}(n(G))$ 时间内处理这类 i。否则，我们考虑一个批次b，$b+1$，$\ldots$，e，使得 $\sum_{i=b}^{e} |NT(G_i)| \leq c \cdot n(G)$。对于后一种情况，通过调整 c，我们可以确保后续递归调用中的节点数至多为 $n(G)/2$，因此可以在后续递归调用期间额外使用 $\mathcal{O}(n(G))$ 空间。最后，对于 $|NT(G_i)| \geq c \cdot n(G)$，我们可以在对 Trim($G_i$) 进行递归调用前后使用 $\mathcal{O}(n(G))$ 时间，因此可以像处理厚 G 一样处理这类 i。注意，在这种情况下，我们无法保留 A、B 和 H，但可以在递归调用返回后重新构建它们。

综上，我们得到以下最终的时间和空间复杂度。

定理 3. 具有 n 个节点和 m 条边的有向图的所有 N 树形图，都可以在总计 $\mathcal{O}(n+m+N)$ 时间内报告，所需空间为 $\mathcal{O}(n+m)$。

9 改进延迟

本节分析任意两个已报告树形图之间的最坏情况延迟。虽然这看起来可能相当长，但我们将证明它仅为 $\mathcal{O}(m)$。

我们首先注意到，由于递归深度上限为 m，调用 Choose() 与报告一个树形图之间的延迟，以及在一次 Choose() 调用中报告两个树形图之间的延迟，均为 $\mathcal{O}(m)$。

接下来，我们分析主递归中的延迟。进入 Recurse(G) 后，我们只需花费线性时间进行扁平化并找到 A 和 B，然后报告 A 作为下一个树形图。因此，从进入过程到报告下一个解之间的延迟仅为 $\mathcal{O}(m(G))$。接下来，在该调用内部，无论哪种情况，每次 for 循环迭代我们都只花费 $\mathcal{O}(m(G))$ 时间，因此，从子递归调用返回到进入下一个子递归调用之间的时间也为 $\mathcal{O}(m(G))$。最后，从最后一个子递归调用返回到从 Recurse(G) 返回所花费的时间也为 $\mathcal{O}(m(G))$。

仅凭这一点还不足以将延迟上界限定为 $\mathcal{O}(m)$，因为递归深度为 m，所以在从整个递归树中深度为 m 的子树里的最后一次递归调用返回时，我们可能需要花费 $\mathcal{O}(m^2)$ 时间。不过，实际情况并非如此。

我们观察到，对于最后一个子树形图递归调用，有 $n(G_{n(G)-1}) \in \{1, 2\}$（可能包含许多平行边），因此在此处花费的时间为 $\mathcal{O}(m)$。所以，如果 $n(G) > 2$，那么当前递归调用要么是r，要么不可能是其父递归调用中的最后一个子树形图。因此，我们可以如下看待整个报告过程：对于每个满足 $n(G) > 2$ 的递归调用，我们首先花费 $\mathcal{O}(m(G))$ 时间，然后

报告一个树形图，接着对多个更小的图进行递归调用，并在这类调用之间花费 $\mathcal{O}(m(G))$，然后报告最后一个树形图（这对应于对 $G_{n(G)-1}$进行递归调用），并花费 $\mathcal{O}(m(G))$ 时间进行清理。此外，如果 $G =$G（因此我们在从调用返回后终止），或者可以保证返回到父递归调用后，下一个树形图将在 $\mathcal{O}(m)$ 时间内报告，那么情况也是如此。由此可立即验证，延迟为 $\mathcal{O}(m)$。

最后，我们给出最终定理。

定理1. 一个含有 n 个节点和 m 条边的有向图的所有 N 树形图，均可在总计 $\mathcal{O}(n+m+N)$ 时间内报告，并使用 $\mathcal{O}(n+m)$ 空间；任意两个已报告解之间的延迟为 $\mathcal{O}(m)$。

参考文献

[1] A. L. Buchsbaum、L. Georgiadis、H. Kaplan、A. Rogers、R. E. Tarjan 和 J. R. Westbrook。用于支配点及其他路径求值问题的线性时间算法。《*SIAM*计算杂志》，38(4):1533–1573，2008。[2] J. Edmonds。边不相交分支树。组合算法，第 91–96 页，1973。[3] H. N. Gabow 和 E. W. Myers。求有向图和无向图的所有生成树。《*SIAM*计算杂志》，7(3):280–287，1978。[4] H. N. Gabow 和 R. E. Tarjan。用于不相交集合并特殊情形的线性时间算法。《计算机与系统科学杂志》，30(2):209–221，1985。[5] L. Georgiadis 和 R. E. Tarjan。支配树验证与发散生成树。《*ACM*算法汇刊》，12(1):11:1–11:42，2016。[6] T. Hagerup。字长RAM上的排序与查找。载于$STACS$，收入《计算机科学讲义》第 1373 卷，第 366–398 页。Springer，1998。[7] J. Holm、K. de Lichtenberg 和 M. Thorup。用于连通性、最小生成树、2-边连通性和双连通性的多对数时间确定性全动态算法。《*ACM*杂志》，48(4):723–760，2001。[8] S. Kapoor 和 H. Ramesh。枚举无向加权图所有生成树的算法。《*SIAM*计算杂志》，24(2):247–265，1995。[9] S. Kapoor 和 H. Ramesh。枚举有向图所有生成树的算法。《算法学》，27(2):120–130，2000。[10] D. E. Knuth。《计算机程序设计艺术》第 *4A* 卷：组合算法，第 *1* 部分。艾迪生-韦斯利，2014。[11] T. Lengauer 和 R. E. Tarjan。用于在流图中查找支配点的快速算法。《*ACM*程序设计语言与系统汇刊》，1(1):121–141，1979。[12] T. Matsui。用于生成无向图所有生成树的灵活算法。《算法学》，18(4):530–543，1997。[13] M. D. McIlroy。算法354：生成树生成器 [H]。《*ACM*通讯》，12(9):511，1969。[14] G. Minty。列出图中所有树的简单算法。《*IEEE*电路理论汇刊》，12(1):120–120，1965。[15] R. C. Read 和 R. E. Tarjan。列举环、路径和生成树的回溯算法的界。《网络》，5(3):237–252，1975。[16] S. Shinoda。求有向图的所有可能有向树。《日本电子与通信》，51(7):45–46，1968。[17] A. Shioura、A. Tamura 和 T. Uno。扫描无向图所有生成树的最优算法。《*SIAM*计算杂志》，26(3):678–692，1997。

[18] T. Uno。在有向图中枚举所有有向生成树的算法。载于*ISAAC*，收入《计算机科学讲义》第 1178 卷，第 166–173 页。Springer，1996。[19] T. Uno。加速枚举算法的新方法。载于*ISAAC*，收入《计算机科学讲义》第 1533 卷，第 287–296 页。Springer，1998。[20] T. Uno。加速枚举算法的新方法及其在拟阵基上的应用。载于*COCOON*，收入《计算机科学讲义》第 1627 卷，第 349–359 页。Springer，1999。[21] K. Wasa。枚举算法的枚举。*CoRR*，abs/1605.05102，2016。