arXiv:2301.03074v1 [cs.DS] 8 Jan 2023

SeedTree：一种动态最优的局部自调整树

Arash Pourdamghani[1], Chen Avin[2],Robert Sama[3],Stefan Schmid[1,4]柏林工业大学，德国 [2]电气与计算机工程学院，内盖夫本-古里安大学，以色列[3]计算机科学学院，维也纳大学，奥地利 [4]弗劳恩霍夫*SIT*，德国

摘要—**我们研究设计自调整树这一基本问题：树如何高效且局部地适应其所服务的需求（即对树节点所存储项的访问），并在调整带来的收益（实现更快的访问）与其成本（重新配置）之间取得平衡。该问题在新兴的需求感知型可重配置数据中心网络等领域有着广泛应用，并与自调整数据结构密切相关。我们的主要贡献是*SeedTree*，一种动态最优的自调整树，支持局部（即贪心）路由，这种特性在需求高度动态变化时尤为理想。*SeedTree* 采用了一种创新方法，基于随机化项目地址定义一组独特路径，并且每个节点仅使用常数个项。我们还通过在各种合成和真实世界通信轨迹上进行评估，从实验角度展示了*SeedTree* 的优势，以补充我们的分析结果。**

索引词—**可重构数据中心、在线算法、自调整数据结构**

一、引言

本文研究自调整树设计这一基本问题：树会根据其所服务的需求进行自我调整。此类自调整树需要在调整带来的益处（提升未来性能）与调整成本（当前的重配置开销）之间实现高效权衡。该问题的研究动机在于，实际工作负载往往具有丰富的时间和空间结构，可利用自调整优化加以改善 [1], [2]。此外，此类调整正变得越来越可行，因为研究人员和从业者目前正努力提升网络化和分布式系统的灵活性，使其能够支持动态重配置，例如利用可编程性（通过软件定义网络） [3], [4],网络虚拟化 [5], 或可重构光通信技术 [6]。

具体而言，我们研究以下抽象模型（应用将在后文介绍）：考虑一棵二叉树，它负责处理在树根节点发出的访问请求，以访问存储在各节点中的项。每个节点（例如服务器）最多可存储c 个项（例如虚拟机），其中 c 是表示节点容量 的参数。我们从在线的角度进行研究，项会随时间被请求。在线算法旨在优化这棵树，以尽量降低未来访问请求的成本（定义为根节点与被访问项之间的路径长度），同时尽量减少在树中向上或向下移动的项数：即重新配置次数。我们将每次移动称为一次重新配置，并记录其成本。具体而言，在线算法并不知道未来的访问请求，其目标是与预先知道整个请求序列的最优离线算法相比具有竞争力。换言之，我们关注的是这样一种在线算法：对于任意请求序列（即使是最坏情况），其竞争比 [7] 最小。

自调整树不仅本身就是最基本的拓扑结构之一，也有许多有趣的应用。例如，这类树是构建更通用的自调整网络的重要基石：Avin 等人 [8]最近指出，多个分别针对单一根节点进行优化的树，可以组合起来构建通用通信网络，并实现较低的度和失真。因此，本文研究如何设计具有竞争力的自调整树，是迈出的重要一步。

自调整树还与自调整数据结构有着有趣的联系（详见第VI 节的详细讨论）；对于其中一些问题，设计并证明具有常数竞争比的在线算法仍是一个开放问题 [9]。有趣的是，近期一项研究表明，如果维护树中项的全局映射，那么自调整平衡二叉树存在具有常数竞争比的在线算法；研究者提议将此类映射集中存储在逻辑根节点处 [10]。本文关注的问题是：能否克服这一限制，以及能否实现具有竞争力的去中心化方案。

我们的主要贡献是一种动态最优的自调整树，*SeedTree**，它通过将最近访问过的项放置在更靠近根节点的位置，实现了常数竞争比，并保证工作集定理 [9]。我们的结果还蕴含了键无关最优性等较弱的性质 [11]（详情将在后文介绍）。*SeedTree* 还支持局部（即贪心且因此去中心化的）路由，这在动态网络中特别有吸引力。它采用一种创新而简单的路由方法，使节点能够自主做出局部转发决策：*SeedTree* 将项哈希到独立同分布的随机地址，并基于这些地址定义一组贪心路径。我们的一个主要发现是，如果

本项目获得了欧洲研究理事会（ERC）根据编号为 864228（AdjustNet）的拨款协议提供的资助，资助期限为 2020年至2025年。

*这个名称源于树中节点的额外容量，它类似于树上果实中的种子。

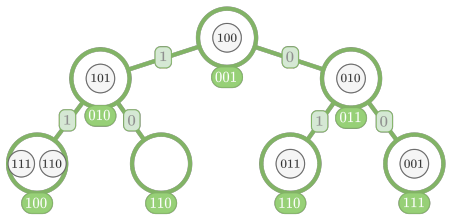


图 1：容量为 2 的 *SeedTree* 示意图。大圆表示系统的节点，小圆表示项。每个小圆内的数字是对应项的哈希值。

节点具有较小的常数容量，也就是说，允许节点存储少量项。让一个节点存储多个项而非仅一个，往往很实用，例如在服务器或对等节点上 [12]，而且在处理冲突的哈希数据结构中也很常见 [13], [14]。我们还通过实证评估了*SeedTree*，既使用具有不同时间局部性的合成轨迹，也使用源自 Facebook 数据中心网络的数据 [1]，展示了如何调整*SeedTree* 的参数，从而在各种场景下降低总成本（以及访问成本）。

本文其余部分安排如下。§II介绍我们的模型和预备知识。我们在 §III中给出并分析我们的在线算法，并在 §IV中将其转换为数据中心网络的匹配模型。在 §V讨论实证评估结果后，我们在 §VI 回顾相关工作，并在 §VII总结我们的贡献。

II. 模型与预备知识

本节介绍我们的模型，以及设计 *SeedTree* 所用的预备知识。

项和节点。 我们假设有一个项 $V = (v_1, \ldots, v_n)$集合，以及一个节点 $S = (s_1, \ldots)$† 集合，它们排列成一棵二叉树 T。我们称节点 s_1 为根节点，它在树 T中的深度为 0，而节点 s_j的深度为 $\lfloor \log j \rfloor$。

每个节点可存储 c个项，其中 c是表示节点容量 的参数。在我们的模型中，假设 c 为常数。项在节点间的分配会随时间改变。如果一个节点包含 满 有 c个项，我们称该节点为满节点；如果节点不包含任何项，则称其为 空 节点（见图 1中的示例）。

我们将时刻 t项 v 的层级 $\text{level}_t(v)$ 定义为包含 v的节点的深度。例如，如果项 v 在时刻 t位于节点 s_j，则有 $\text{level}_t(v) = \lfloor \log j \rfloor$。

请求序列与工作集。项以在线方式随时间被请求，这可建模为一个请求序列$\sigma = (\sigma_1, \ldots\ldots, \sigma_m)$，其中 $\sigma_t = v \in V$表示在时间 t请求项 v。我们有时会关注项请求的近期性，尤其是工作集的大小。形式化地，我们将 $ws_t(\sigma, v)$定义为请求序列 σ中时间 t 时项 v 的工作集。工作集 $ws_t(\sigma, v)$是一个集合，包含在时间 t之前对项 v 的上一次请求之后所请求的所有不重复项。我们将时间 t时项 v 的秩定义为$\text{rank}_t(v)$，即时间 t时项 v 的工作集大小。

†我们假设节点集合的规模可以任意大，因为节点的确切数量将根据其已用容量来确定。

成本与竞争比。我们将算法 ALG产生的成本分为两部分：查找项的成本，即访问成本；以及重新配置的成本，即重配置成本。对任意项的搜索都从根节点开始，并在包含该项的节点处结束。基于固定容量这一假设，我们认为在节点内部搜索的成本可以忽略不计。此外，假设满足局部路由性质，我们通过沿树中的单一路径遍历来查找项；因此，对于访问请求 σ_i，$C^A_{ALG}(\sigma_i)$，其访问成本等于存储该项的层级。

在我们的模型中，一次重配置包括将一个项在树中上移或下移一层，并可能需要在节点内部进行额外查找。我们用 $C^R_{ALG}(\sigma_i)$ 表示访问请求 σ_i 之后的重配置总成本。因此，每次访问请求的总成本为 $C^A_{ALG}(\sigma_i) + C^R_{ALG}(\sigma_i)$，算法处理整个请求序列的总成本为： $C_{ALG}(\sigma) = \sum_{i=1}^{m} C^A_{ALG}(\sigma_i) + C^R_{ALG}(\sigma_i)$。*SeedTree* 的目标是以尽可能低的成本运行，具体而言，尽可能接近最优离线算法 OPT 的成本。

定义 1（竞争比）。 给定一个在线算法ALG和一个最优离线算法 OPT，*(*严格*)*竞争比定义为： $\rho_{ALG} = \max_\sigma \frac{C_{ALG}(\sigma)}{C_{OPT}(\sigma)}$

此外，如果只考虑在线算法 ALG 的访问成本（不包括重配置成本），我们称该算法具有（严格）访问竞争比。

本文证明了*SeedTree* 是动态最优的。这意味着，从渐近意义上说，我们的算法成本与最优离线算法的成本相匹配。

定义 2（动态最优性）。 如果算法 ALG 具有常数竞争比，即$\rho_{ALG} = O(1)$，则该算法是动态最优的。

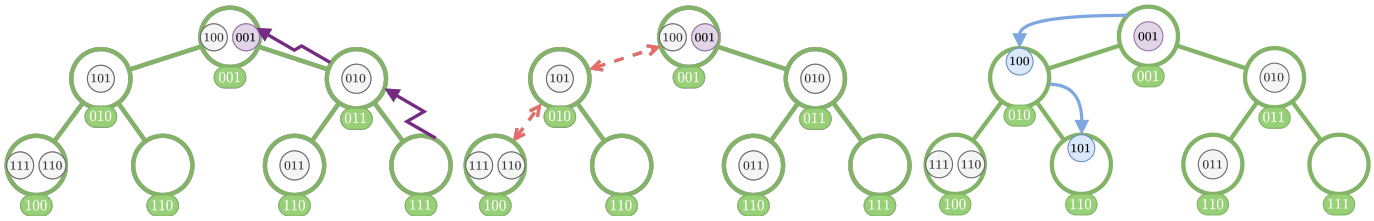
MRU **树。** 我们定义了一类特定的自调整树，即 MRU树。如果一个算法使各项保持在与其秩相近的层级上，则称该算法维护了一棵 MRU树。

定义 3（MRU树）。 如果对于树中的任意项 v 以及任意给定的时间 t，等式 层级$_t(v) = \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor$ 成立，则该算法具有 $MRU(0)$性质。

类似地，如果某个算法对树中的任意项 v 都确保满足宽松界 $MRU(\beta)$，我们就称它维持了 $\text{level}_t(v) \leq \lfloor \log \lceil \frac{\text{秩}_c^t(v)}{c} \rceil \rfloor + \beta$层级。

III. 在线 SeedTree

本节介绍*SeedTree*，这是一种期望意义下动态最优的在线算法。该算法基于均匀随机生成的地址，并支持局部路由，同时保证动态最优性。算法的具体步骤如下：算法 1 始终从根节点开始。收到对项 v的访问请求后，它会根据为节点 v生成的均匀随机二进制地址执行局部路由（见过程LocalRout-ing），该地址唯一确定了 v 在树中的路径。我们将地址的第 i位称为 v 的 $H(v, i)$。假设节点 v 的局部路由结束于层级 ℓ。



(a) 项目 001 逐个节点向上移动，直到到达根节点。(b) 第一次下推失败了，因为节点 100 已满。(c) 找到未满的节点后，项逐个节点下推。

图 2：算法1执行步骤示例，起始状态为图*1*中的SeedTree，其容量为2。在此示例中，请求是对哈希值为001 的项发出的访问请求（紫色圆圈）。子图2a展示了移至根节点阶段，子图2b 和2c 展示了下推阶段。

过程 LocalRouting(s,v)

如果 $H(v$,层级(s)) 等于 *0* **则**
    返回 s的左子节点。
否则
    返回 s的右子节点。

然后，*SeedTree* 执行以下两个阶段的重构。这两个阶段旨在确保项的层级始终与其秩处于相同范围内（详情将在后文介绍），并确保每一层的项数保持不变。

1) 移至根节点： 此阶段会将被访问项移至尽可能低层级的节点，即树的根节点。项按步骤移动，同时所有其他项都留在原来的节点中（如果路径上的某个节点已满，我们会将该项保存在临时缓冲区中）。图 2a 中的锯齿状紫色箭头展示了这一过程。

2) 下推： 在此阶段，算法从根节点开始，从该节点中均匀随机地选取一项（包括刚移至该节点的项），并将此项向下移动一层，移至 LocalRouting 过程中选定的新节点。对新节点重复同样的过程，直至到达 ℓ 层，即被访问项所在的层级。如果 ℓ 层的节点未满，则成功恢复了平衡。否则，如果此次尝试失败，算法会将之前的下推操作逆向撤回至根节点，然后重新开始，直至尝试成功。例如，图 2b 中的红色虚线边展示了此阶段的一次失败尝试，而图2c 中的弯曲蓝色箭头展示了最后一次成功的尝试。

算法1总会终止，因为被移至根节点的项始终有机会从所有候选项中被选中，而且我们知道，取出该项的节点并未满。现在我们陈述本文的主定理，该定理证明了*SeedTree*的动态最优性。

定理 1. 对于任意给定的容量 $c \geq 1$，*SeedTree* 都是动态最优的。

算法 1： 在线 *SeedTree*

输入： 被访问项 v。

将 s 设为根节点。
当 s 不包含 v **时**
    s = LocalRouting(s,v).
将 v 的当前层级称为 ℓ。
将 s 设为根节点，并将项 v 移至 s。
当平衡未固定**时**
    称当前节点为 s。
    当 s 的层级小于 ℓ **时**
        从节点 s中均匀随机地取一个项，称之为 $s \xleftarrow{v}$ LocalRouting（s，v）。
        将项 v 添加到节点 s中。
    如果 最后选中的节点已满，**则**
        将下推反向执行，直至根节点。

定理的证明 1 见本节末尾。证明的第一步是说明每一层级中的项数保持不变。这是因为在某一层级移除一个项后，算法会在下推阶段将一个项添加到同一层级。

观察 1. *SeedTree* 在每个层级上保持相同的项数。

其余分析基于这样一个假设：算法初始化时，每个层级的容量都具有固定的分数占用率$0 < f < 1$，也就是说，在层级 i中，初始树恰好有 $\lfloor c \cdot f \cdot 2^i \rfloor$ 项。在本节末尾，我们会看到，$f = \frac{1}{2}$ 最适合我们的分析。不过，我们强调，具有 $0 < f < 1$ 就足以让*SeedTree*正常运行。

第二个观察结论是观察 1 的推论。由于每个层级中的项数保持不变（根据观察 1），至多有所有节点中的 f 是满的。在最低层级，满节点的数量可能更少；因此，在下一次请求时，均匀随机选择的节点为满节点的概率至多为 f。

观察 2. 算法 *1* 确保在处理每个访问请求后，*SeedTree* 中均匀随机选取的任意节点为满节点的概率至多为 f。

根据算法 1，在一个节点内均匀随机选取项。在以下引理中，我们将证明某一层级中的节点也是均匀随机选取的，这使得证明的其余部分得以进行。

引理 1. 在下推阶段的最终路径上，层级低于 ℓ 的节点均被均匀随机选中。

证明. 记 ℓ' 个路径上的节点（即层级为 ℓ' 的节点，记为 $s_{\ell'}$）被选中的概率为 $\frac{1}{2^{\ell'}}$。我们通过归纳法证明。对于基例 $\ell' = 0$，结论成立，因为我们只有一个节点，即根节点。现在假设在下推后的最终路径上，我们要考察到达当前节点 $s_{\ell'}$ 的概率。根据归纳假设，我们知道 $s_{\ell'}$ 的父节点，即节点$s_{\ell'-1}$，已被均匀随机选中，概率为$\frac{1}{2^{\ell'-1}}$。根据算法 9 的第 1 行，从 $s_{\ell'-1}$ 中的项均匀随机选取一项；再结合生成所选项地址的哈希函数所提供的独立性保证，我们可以得出，从 $s_{\ell'-1}$ 向左或向右的决策也是均匀随机的，因此到达 $s_{\ell'}$ 的概率为 $\frac{1}{2^{\ell'-1}} \cdot \frac{1}{2} = \frac{1}{2^{\ell'}}$。请注意，上述选择与下降 $s_{\ell'-1}$ 是否完整无关。因此，该选择独立于下推阶段此前可能失败的尝试（失败可能是因为层级为 ℓ 的节点已满）；也就是说，此前的尝试不会影响选择节点 $s_{\ell'}$的概率。

□

定理1证明中的一个关键要素是，项的秩和层级彼此相关。引理2描述了这种关系的一个方面。

引理 2. 在 *SeedTree* 的执行过程中，对于时间 t 的项 v 和 u，若 秩$_t(v) >$ 秩$_t(u)$，则 E[层级$_t(v)] >E$[层级$_t(u)$]。

证明。 由于秩$_t(v) >$ 秩$_t(u)$，我们知道 u 比 v更晚被访问。考虑时间 t'，即最后一次访问 u 的时间。由于 v 的秩严格大于 u的秩，并且 u 在时间 t' 被移至根节点，因此我们知道层级$_{t'}(v) >$层级$_{t'}(u)$。

项 u 和 v 在时间 t' 之后可能到达同一层级，但并非必然如此。我们将它们首次相遇的层级视为随机变量 L_{uv}。如果 u 和 v 在时间 t' 之后从未出现在同一层级，我们记作 $L_{uv} = -1$。下面使用全期望定律，量化 u 和 v 的期望层级之差：

$$
\begin{aligned}
&E[\text{level}_t(v)] - E[\text{level}_t(u)] \\
&= \sum_{k=-1}^{\lfloor \log \lceil \frac{n}{c} \rceil \rfloor} Pr(L_{uv} = k) \cdot (E[\text{level}_t(v) | L_{uv} = k] \\
&\quad - E[\text{level}_t(u) | L_{uv} = k])
\end{aligned}
$$

对于情况 $L_{uv} = -1$，我们知道 u 和 v 从未到达同一层级，并且以下结论始终成立：

$$E[\text{level}_t(v) | L_{v,u} = -1] > E[\text{level}_t(u) | L_{v,u} = -1]$$

对于 $k \geq 0$，考虑时间 t''，此时 u 和 v 相遇并处于同一层级，即 level$_{t''}(u)$ =level$_{t''}(v)$。项 u和 v 首次相遇后，它们预期取得的进展相同。更准确地说，考虑时间 t''时包含 v 的节点所对应的当前子树，并将其记为 T'。由于项目地址是均匀随机选取的，T' 是包含 v 的节点的子树的预期次数，等于在同一层级中，T' 可能成为包含 u 的节点的子树的次数。因此，从时间 t'' 起，项 u 和 v 的层级预期增幅始终相同。 □

接下来，我们说明对于任意给定的项，在期望意义下，在更高层级访问的项数是有限的。

引理 3. 对于给定的项 v 在时间 t，自上次访问 v 以来，在期望意义下，至多有$2 \cdot$秩$_t^{ws}(v)$ 个项在更高层级被访问。

证明。 根据引理2，证明思路与 [10] 中引理 4 的证明类似。由于篇幅限制，我们省略了证明细节。

□

现在我们证明，在线*SeedTree*维护的树中的项，与满足精确工作集性质的树中对应位置相比，不会离得太远。这样一来，我们便能将在线*SeedTree* 的总成本与容量相同的最优离线算法的总成本进行比较，并对其进行近似。近似因子 $2 - \log(f)$ 很直观：每一层的容量越小（层级的分数占用率越低），我们就需要把项放得越靠下。

引理 4. *SeedTree* 的期望复杂度为 $MRU(2 - \log(f))$。

证明。 对于任意给定的项 v和时间 t，我们证明 E[层级$_t(v)] \leq \lfloor \log \lceil$秩$_c^t{}^v \rceil \rfloor + 2 - \log(f)$ 在移至根节点和下推阶段期间仍然成立。如算法 8 第 1 行所示，项 v 可能会向下移动，若 v 的当前层级低于被访问项的层级。

我们将从时间 t' 到时间t 层级的增加量记为随机变量 $D(t', t)$。我们用指示随机变量 $I(t', t, \ell)$ 来表示项 v 在 $[t', t]$ 期间是否从层级 ℓ 向下移动。我们知道：

$$D(t', t) = \sum_{\ell} I(t', t, \ell)$$

令 K 表示从更高层级访问的项数，并记作 $K = k_1 + \cdots + k_{\lceil \frac{n}{c} \rceil}$，其中 k_ℓ表示当项 k_ℓ 处于层级 v 时发生了 ℓ次此类访问。对于层级 ℓ，根据观察1 和引理1，以及每个层级包含 $f \cdot c \cdot 2^\ell$ 个项这一事实，我们可以得出 v在经过 $k_\ell - 1$次访问后被选中的概率为$(1 - \frac{1}{f \cdot c \cdot 2^\ell})^{k-1} \cdot (\frac{1}{f \cdot c \cdot 2^\ell})$。

$$I(t', t, \ell) = \min(1, \sum_{k_\ell=0}^{K} (1 - \frac{1}{f \cdot c \cdot 2^\ell})^{k_\ell - 1} \cdot (\frac{1}{f \cdot c \cdot 2^\ell}))$$

$$= \min(1, (\frac{1}{f \cdot c \cdot 2^{\ell}}) \cdot \sum_{k_\ell=0}^{K} (1 - \frac{1}{f \cdot c \cdot 2^{\ell}})^{k_\ell - 1})$$

$$\leq \min(1, (\frac{K}{f \cdot c \cdot 2^{\ell}}))$$

回到我们最初的目标：求项在时间段 $[t', t]$ 内下降了多少个层级，我们有：

$$E[D(t', t)] \leq \sum_{\ell} E[\min(1, (\frac{K}{f \cdot c \cdot 2^{\ell}}))]$$

$$= \log(\frac{E[K]}{f \cdot c}) + 1 = \log(E[K]) - \log(c) - \log(f) + 1$$

最后一个等式源于这样一个事实：对于 $\ell = \log(\frac{E[K]}{f \cdot c})$，我们有 $\frac{K}{f \cdot c \cdot 2^{\ell}} \leq 1$；而对于所有更大的 ℓ 值，该值将以 2 为因子呈指数下降。

根据引理 3，我们知道 K 小于等于 $2 \cdot$秩$_t(v)$的期望值；因此，期望增量为：

$$E[D(t', t)] \leq \log(2 \cdot \text{rank}_t(v)) - \log(c) - \log(f) + 1$$

$$= \log(\text{rank}_t(v)) - \log(c) + 2 - \log(f)$$

□

下面的引理说明了在线 *SeedTree* 的总成本与分数占用率 f 之间的关系。这种关系很自然：随着 f 变小，找到非满节点的概率会增大，因此找到非满节点所需的尝试次数会减少。

引理 5. *SeedTree* 的期望成本小于等于 $2 \cdot (\lceil \frac{1}{(1-f)} \rceil + 1)$ 倍的访问成本。

证明。 考虑层级 v 上的被访问项 ℓ。在算法的第一部分，移至根节点阶段的成本与访问成本相同，即遍历 ℓ 条边。根据观察 $1-f$，节点非满的概率为 2；又根据观察 1，节点是均匀选取的，因此在下推阶段只需经过$\lceil \frac{1}{1-f} \rceil$ 次迭代即可找到一个非满节点，每次迭代的成本为 $2 \cdot \ell$。因此，根据期望的线性性质，我们有：

$$E[C_{ALG}] = E[C_{ALG}^{\text{Access}} + C_{ALG}^{\text{Move-to-the-root}} + C_{ALG}^{\text{Push-down}}]$$

$$\leq 2 \cdot (1 + \lceil \frac{1}{1-f} \rceil) \cdot \ell = 2 \cdot (1 + \lceil \frac{1}{1-f} \rceil) \cdot C_{ALG}^{\text{Access}}$$

□

现在说明，在重新配置成本不高的情况下（引理5对此进行了证明），为什么工作集最优性足以保证动态最优性。因此，其他形式的最优性（例如键无关最优性或手指最优性）也会自动得到保证 [11]。

引理 6. 对于任意给定的 c，一个 $MRU(0)$ 算法的$(1+e)$访问竞争比为。

证明。 证明依赖于势函数论证。我们将 t 时刻的势函数描述为 ϕ_t，并证明从 t 时刻到 $t+1$ 时刻的势变化为 $\Delta\phi_{t \to t+1}$。

我们的势函数在时间 t 计算最优离线算法 OPT 的树中相对于秩位置错误的项数。（正如 $MRU(0)$ 的定义所示，这样的树中不存在逆序，因此我们只关注 OPT 中逆序对的数量。）具体而言，如果 $\text{rank}_t(v) < \text{rank}_t(u)$，但 $\text{level}_t(v) > \text{level}_t(u)$，我们就称项对 (v, u) 构成一个逆序。我们将时间t 时与项 v 构成逆序的项数记为 $\text{inv}_t(v)$，并定义 $B_t(v) = 1 + \text{inv}_t(v) \cdot c \cdot 2^{\text{level}_t(v)}$。此外，定义 $B_t = \prod_{v=1}^{n} B_t(v)$。我们将时间 t 的势函数定义为 $\phi_t = \log B_t$。我们假设在线 *SeedTree* 在最优算法进行重新配置之前，先在树中重新排列其所需的项。首先描述访问项 $\sigma_t = v$ 后在线 *SeedTree* 中重新配置所导致的势变化。这一变化产生以下影响：

1）被访问项 v 的秩已被设为 1。2）树中其他项的秩最多可能增加 1。

由于第二种影响不会改变 v 以外各项的相对秩，因此不会影响逆序对的数量，进而也不会影响势函数。因此，我们关注第一种影响。由于 OPT 的配置未发生变化，对于 OPT 中所有存储在比 v 较低层级的项 u，都会产生一个逆序对，因此有 $B_{t+1}(u) = B_t(u) + \frac{1}{c \cdot 2^{\text{level}_t(u)}}$。对于被访问项 v，由于其秩已变为 1，它的所有逆序对都会被删除。除 v 外，其他项的逆序对数量保持不变。令时刻 t 比 v 处于较低层级的项数为 $L_t(v)$，并按照我们讨论过的方式将 $\prod_{i=1}^{n} B_{t+1}(i)$ 划分为三部分（v、存储在比 v 较低层级的项，以及由集合 $O_t(v)$ 表示的其他项）：

$$\prod_{i=1}^{n} B_{t+1}(i) = B_{t+1}(v) \cdot \prod_{i \in L_t(v)} B_{t+1}(i) \cdot \prod_{i \in O_t(v)} B_{t+1}(i)$$

将 $B_{t+1}(i)$ 改写为关于 $B_t(i)$ 的形式，可得：

$$\prod_{i=1}^{n} B_{t+1}(i) = 1 \cdot \prod_{i \in L_t(v)} (B_t(i) + \frac{1}{c \cdot 2^{\text{level}_t(i)}}) \cdot \prod_{i \in O_t(v)} B_t(i)$$

现在，我们来看看时间t 到 $t+1$ 期间由 $\Delta\phi_{t \to t+1}^{1}$产生的第一种效应所对应的势，并对其进行更详细的描述：

$$\Delta\phi_{t \to t+1}^{1} = \log B_{t+1} - \log B_t = \log \frac{B_{t+1}}{B_t}$$

$$= \log \frac{\prod_{i=1}^{n} B_{t+1}(i)}{\prod_{i=1}^{n} B_t(i)} = \log(\frac{1}{B_t(v)} \cdot \frac{\prod_{L_t(v)} (B_t(i) + \frac{1}{c \cdot 2^{\text{level}_t(i)}})}{\prod_{L_t(v)} B_t(i)})$$

$$\leq \log(\frac{1}{B_t(v)} \cdot e^{|L_t(v)|})$$

其中，最后一个不等式是由以下事实得出的：$|L_t(v)| = c \cdot 2^{\text{level}_t(i)}$，以及以下不等式：

$$\prod_{i=1}^{|L_t(v)|}(B_t(i)+\frac{1}{|L_t(v)|}) \le \prod_{i=1}^{|L_t(v)|}(B_t(i)+\frac{B_t(i)}{|L_t(v)|})$$

$$= (1+\frac{1}{|L_t(v)|})^{|L_t(v)|} \cdot \prod_{i=1}^{|L_t(v)|} B_t(i) \le e^{|L_t(v)|} \cdot \prod_{i=1}^{|L_t(v)|} B_t(i)$$

现在让我们关注 $B_t(v)$，并首先假设 $\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor$ 层级 $_t(v)$。我们要找出可能与被访问项 v 发生逆序的项的最大数量。

在所有 $c \cdot 2^{\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor} - 1$ 可能比它们秩更高的项中，至多有 $c \cdot 2^{\text{level}_t(v)} - 1$ 个在树 OPT 中处于较低层级。因此我们有：

$$B_t(v) = \frac{(c \cdot 2^{\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor} - 1) - (c \cdot 2^{\text{level}_t(v)} - 1)}{c \cdot 2^{\text{level}_t(v)}}$$

$$\ge \frac{(2^{\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor} - 1)}{2^{\text{level}_t(v)}} - 1$$

$$\ge \frac{2^{\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor}}{2^{\text{level}_t(v)+1}} = 2^{\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor - \text{level}_t(v) - 1}$$

因此，第一种影响导致的势变化为：

$$\Delta\phi^1_{t\to t+1} \le \log(\frac{1}{2^{\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor - \text{level}_t(v) - 1}} \cdot e^{\text{level}_t(v)})$$

$$= \log(2^{(1+\log e)\cdot \text{level}_t(v) - \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor})$$

$$= (1+\log e)\cdot \text{level}_t(v) - \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor$$

对于 $\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor$ 层级 $_t(v)$ 的情况，根据下面第一个不等式中的事实 $B^t_v > 1$，我们使用：

$$\Delta\phi_{t\to t+1} = \log(\frac{1}{B^t_v} \cdot e^{\text{level}_t(v)})$$

$$\le \log(2^{\log e \cdot \text{level}_t(v)}) = \log e \cdot \text{level}_t(v)$$

$$= (1+\log e)\cdot \text{level}_t(v) - \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor$$

因此，无论 $\lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor$ 大于还是小于层级 $_t(v)$，我们都有 $\Delta\phi_{t\to t+1} \le (1+\log e)$ 层级 $_t(v) \log e$。

接下来，我们展示由于 OPT 的重配置而导致的势变化。受篇幅限制，我们省略计算细节，但这些变化与 ON 的算法中重排导致的势变化类似，结果是每次 OPT 的移动成本都小于 $\log e$。

汇总 ON 和OPT重配置后势的变化，假设 OPT在时间 t进行了 w_t次移动，我们得到：

$$\Delta\phi^{t\to t+1} = (1+\log e)\cdot \text{level}_t(v) - \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor + w \cdot \log e$$

因此，在线算法 $MRU(0)$ 在时间t的代价至多为：

$$C^t_{MRU(0)} = C^t_{Amortized} + \Delta\phi^t$$

$$= \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor + (1+\log e)\cdot \text{level}_t(v)$$

$$- \lfloor \log \lceil \frac{\text{rank}_t(v)}{c} \rceil \rfloor + w_t \cdot \log e \le (1+\log e)\cdot(\text{level}_t(v) + w_t)$$

然后，对整个请求序列中 $MRU(0)$ 和 OPT 的成本求和，得到：

$$C_{ON} = \sum_t C^t_{ON} \le \sum_t (1+\log e)\cdot(\text{level}_t(v) + w_t)$$

$$= (1+\log e)\cdot C_{OPT}$$

其中，最后一个等式源于这样一个事实：OPT 也需要访问该项，并且我们假设还需要额外进行 w_t 次重新配置。□

作为引理6 的第一个应用，我们证明了任何在线算法的成本下界，该下界仅取决于请求序列中被访问项的工作集大小。

定理 2。任何在线算法若在请求序列 $\sigma = \sigma_1, \ldots \sigma_m$ 上维护容量为 $c > 1$ 的自调整完全二叉树，则其访问成本至少为 $\sum_{i=1}^m \lfloor \log \lceil \frac{\text{rank}_t(\sigma_i)}{c} \rceil \rfloor (1+e)$。

证明。此证明扩展并改进了 [10] 对于任意 $c > 2$ 的证明。引理 6 的结果表明，即使是最优算法也不可能优于 $\frac{1}{(1+e)}$ $MRU(0)$，否则就会与引理 6 矛盾。由于每次访问项 σ_i 的成本在 $MRU(0)$ 中为 $\lfloor \log \lceil \frac{\text{秩}_t \sigma_i}{c} \rceil \rfloor$()，因此可以得出，任何算法的总成本都应大于 $\sum_{i=1}^m \lfloor \log \lceil \frac{\text{rank}_t(\sigma_i)}{c} \rceil \rfloor (1+e)$。□

引理 7。任何 $MRU(\beta)$ 树都是 $\beta\cdot(1+e)$-访问竞争比。

证明。引理 6 表明，$MRU(0)$ 是 $(1+e)$-访问竞争比。$MRU(0)$ 中处于层级 k 的任意项，在 $MRU(\beta)$ 中都处于层级 $k+\beta$。由于 $MRU(\beta)$ 算法会将秩为 $^c(0)$ 的项保留在 $level(0)$，并且对于任意 $k \ge 1$，都有 $k+\beta \le \beta k$，因此可得 $MRU(\beta)$ 是 $(\beta)\cdot(1+e)$-访问竞争比。□

我们通过证明在线*SeedTree*的动态最优性这一主定理，结束本节。

定理*1*的证明。结合引理 4、引理 5 和引理 7 可得，竞争比的上界为$(1+e)\cdot(2\cdot(1+\lceil \frac{1}{1-f} \rceil))\cdot(2-\log(f))$。上述公式中的分数占用率 $f = 1/2$是 f的最优值，因此我们得到43竞争比。□

需要指出的是，上述计算仅仅是竞争比的上界。正如我们将在 §V 中讨论的，最佳结果通常是在 f 稍高时取得的。我们推测，这可能是因为理论分析中高估了项的深度。

IV. 在可重构数据中心中的应用

SeedTree 提供了一种基础的自调整结构，可用于不同场景。例如，它可用于调整虚拟化环境中容器的放置位置，以降低通信成本。不过，*SeedTree* 也可应用于链路能够调整的可重构网络。接下来，我们将更详细地介绍如何在此类应用场景中使用 *SeedTree* 。具体而言，我们考虑一种可重构数据中心，其中机架之间的连接，或者更具体地说，机架顶部（ToR）交换机之间的连接，可以动态调整，例如通过光路交换机 [6]。光交换机可在机架之间建立匹配，因此文献中将这一模型称为 匹配模型 [15]。接下来，我们将说明如何从 *SeedTree* （容量为 c 、分数占用率为 $f = \frac{1}{c}$）的角度理解 $2+c$匹配，并说明如何将重配置转换为匹配模型[‡]。我们将这些匹配分为两个集合：

- **拓扑匹配**：由 2 个静态匹配组成，用于嵌入 *SeedTree* 的底层二叉树。第一个匹配表示节点与其左子节点之间的边（左子节点的 ID 是该节点 ID 的两倍）；同理，第二个匹配表示节点与其右子节点之间的边（右子节点的 ID 是其父节点 ID 的两倍加一）。图 3 中以实线边展示了一个示例。
- **成员匹配**：包含 c 个动态匹配，用于将节点连接到其中的项。如果一个节点包含多个项，则项与匹配之间的对应顺序可以任意确定。图 3 中以虚线边展示了一个示例。

完成匹配后，我们简要讨论如何实现搜索和重构操作。搜索某个项时，从 ID 为 001 的节点（即根节点）开始。接着检查该节点的成员匹配。如果这些匹配指向该项，我们就找到了包含该项的节点，搜索成功。否则，我们根据该项的哈希值确定要沿着哪条边前进，从而到达可能存放该项的新节点。我们重复检查成员匹配并沿拓扑匹配前进，直到找到该项。由于该项存储在其哈希值所确定路径上的某个节点中，因此最终一定能够找到。每次移动项，都可以在匹配模式下通过成员匹配中的一次边删除和一次边添加来实现。

V. 实验评估

我们通过在多个数据集上评估*SeedTree* ，补充了分析结果。具体而言，我们希望回答以下问题：

Q1 我们算法的访问成本与静态最优算法（基于频率优化）和需求无关算法相比如何?

‡匹配模型仅考虑完美匹配，但在实践中，可以通过交换机中的忽略规则来强制执行不完美匹配。

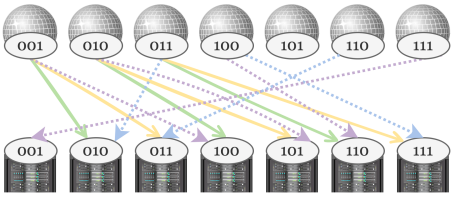


图 3：对图 *1*所示 SeedTree 示例的一种转换，其容量为 $c = 2$ ，分数占用率为 $f = \frac{1}{2}$ 。顶部的迪斯科球代表可重构交换机，下方是数据中心机架。实线边表示结构匹配，虚线边表示成员匹配。

Q2 在每个层级的分数占用率固定的情况下，额外容量如何提升在线*SeedTree*的性能？ Q3 在容量固定的情况下，在线*SeedTree*的最佳初始分数占用率是多少?

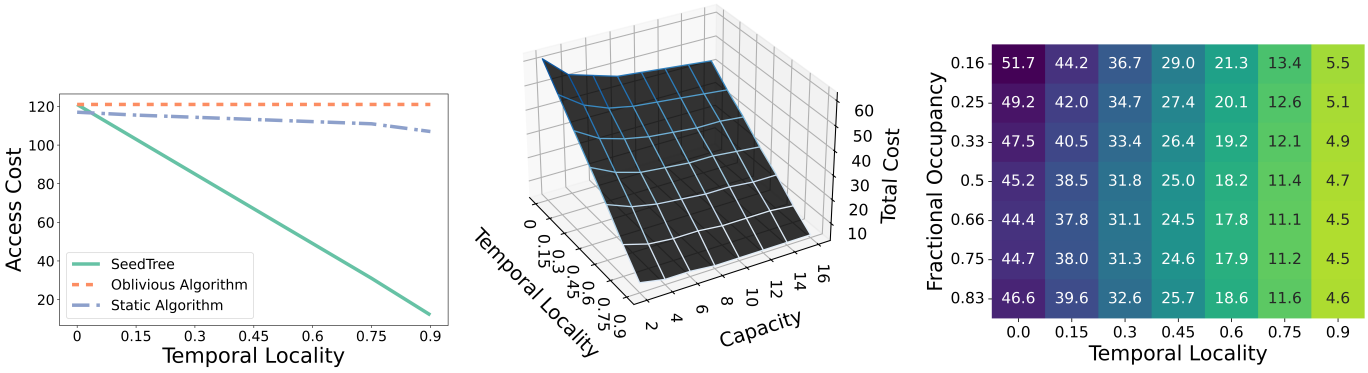
回答这些问题有助于开发人员根据自身要求和需求调整 *SeedTree* 的参数。在介绍结果之前，我们先描述所采用的设置：我们的代码使用 Python 3.6 编写，并使用 seaborn 0。11 [16]和 Matplotlib 3。5 [17] 库进行可视化。我们的程序在一台配备 2 个 Intel Xeons E5-2697V3 SR1XF 处理器的机器上运行，每个处理器主频为 2.6 GHz、拥有 14 个核心，并配有总计 128 GB 的 DDR4 内存。

A. 输入

- **真实世界数据集**：我们的真实世界数据集是三个不同 Facebook 簇内服务器之间的通信数据，取自 [1]。我们对该数据集进行了后处理，以提取单源通信。在所有可能的源中，我们选择了出现频率最高的源。
- **合成数据集**：我们使用 [1], [18] 中讨论的马尔可夫模型生成序列，该模型基于时间局部性参数，其取值范围为 0（均匀分布，无局部性）至 0.9（高时间局部性）。我们的合成输入包含 65,535 个项和 100 万次请求。生成此类数据集时，我们从项的随机样本开始。随后对该序列进行后处理，以由时间局部性参数决定的概率，将每次请求替换为前一次请求。之后，我们执行第二次后处理，以确保最终轨迹中恰好包含 65,535 个项。

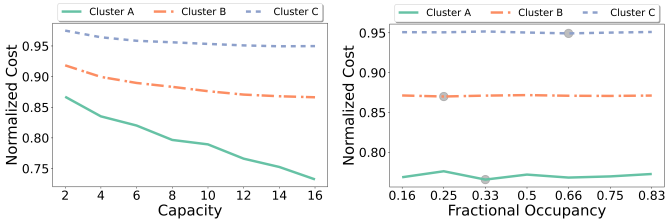
B. 算法设置

在我们的实现中，我们使用 *hashlib*-library 中的 SHA-512 [19] 作为哈希函数，以近似均匀分布来生成项的地址。为了在节点中存储项，我们使用了链表；当我们将某个项移至一个已被其他项填满的节点时，这些项会存储在临时缓冲区中。我们假设从一棵预先填充了项、且符合分数占用率参数的树开始。



(a) (b) (c)

图4：通过参数微调提升*SeedTree* 性能。图中数据基于具有不同局部性值的合成数据集生成。（4a）比较*SeedTree* 在分数占用率为 $f = \frac{1}{2}$ 时的访问成本与最佳静态算法和需求无关算法的访问成本，所有算法的容量均为 $c = 4$。访问成本除以10万。（4b）提高节点容量和输入的时间局部性对算法总成本的影响。所有容量下，分数占用率均设为 $f = \frac{1}{2}$。此图中的总成本除以100万。（4c）在一系列时间局部性下，总成本与分数占用率之间的权衡。节点容量设为12。每个单元格中的数值表示成本，均除以100万。



(a) (b)

图5：通过调整*SeedTree* 参数降低算法的归一化访问成本。结果基于真实世界数据集中三个簇内访问频率最高的源的通信情况得出。成本按需求无关算法的成本归一化。（5a）改变容量时归一化成本的变化。分数占用率设为 $f = \frac{1}{2}$。（5a）改变分数占用率时归一化成本的变化。灰点表示最小值。节点容量设为12。

在我们的实验中，我们将容量（c）的范围设为 2 到 16，将分数占用率（f）的范围设为 0.16 到 0.83。由于我们的算法和输入生成具有随机性，我们将每项实验重复最多 100 次，以确保结果一致。

C. 结果

随着时间局部性的增强，*SeedTree* 的性能显著提升，如图 4所示。此外，我们对本节开头提出的问题得出了以下经验性答案：

A1：*SeedTree*显著降低了访问成本，并增强了时间局部性，如图 4a 所示，该图比较了*SeedTree* 与静态算法和需求无关算法的访问成本。

A2：如图 4b 和 5a 所示，增大容量会降低算法的成本。不过，我们可以看到，容量超过 8 后，收益便会减缓，因此在实际应用中，这个值可视为最佳选择。

A3：正如 §III 末尾所讨论的，以及图 4c 和 5b所示，分数高于或低于1 2时都可能达到最低成本，但在大多数情况下，$f = \frac{1}{2}$ 接近最优。

六、其他相关工作

自调整列表与树作为数据结构，早已得到深入研究。开创性工作由 Sleator 和 Tarjan 完成 [20], ，他们开创了动态列表更新问题的研究，并提出了移至前端算法，启发了许多确定性 [21][22] 和随机化 [23]–[26]数据结构方法，以及该问题的其他变体 [27]。

自调整二叉搜索树也旨在让最近使用过的元素靠近根节点，这与本文的方法类似（结果概要见表I）。不过，二叉搜索树中的调整是通过旋转实现的，而不是在不同节点之间移动项。伸展树是著名的自调整二叉搜索树之一 [9],尽管目前仍不清楚这种树是否动态最优；对于 Zipper Tree 等较新的变体，这个问题也仍未解决 [31], Multi SplayTree [32]和 Chain Splay [33]，它们将伸展树的 $O(\log n)$ 竞争比提升至 $O(\log \log n)$。对于探戈树 [29]，已知一个匹配的$\Omega(\log \log n)$ 下界。我们还知道，如果允许在访问后免费进行旋转，动态

数据结构	操作	比例	搜索
伸展树 [9]	旋转	$O(\log\ n)$	Yes
贪心未来 [28]	旋转	$O(\log\ n)$	Yes
探戈树 [29]	旋转	$\theta(\log\log n)$	Yes
自适应哈夫曼 [30]	子树交换	$\theta(1)$	No
下推树 [10]	顶交换	$\theta(1)$	No
SeedTree	顶移动	$\theta(1)$	Yes

表 I：自调整树数据结构的性质比较。迄今已知的最佳竞争比，是根据相应数据结构的成本模型和最优离线算法来衡量的。值得注意的是，除我们的模型外，上述树结构都未考虑额外容量。

最优性便可实现 [34]。我们还指出，其中一些结构，尤其是 Multi Splay Tree 和 Chain Splay，也受益于额外的内存；不过，这些内存的用途有所不同，即用于为每个节点保存额外属性。另一种由 Lucas 首次提出的变体 [28] 于 1988 年提出，称为贪心未来。这种树最初作为离线二叉搜索树算法受到关注 [35], [36], , 随后有人提出，在在线场景中它具有 $O(\log n)$ 摊销时间复杂度，Fox [37]。贪心未来促使研究人员从几何视角研究在线二叉搜索树 [36], [38]。我们指出，与二叉搜索树不同，我们的局部树不要求节点左右子树中的项具有顺序。

自调整树也曾在编码领域得到研究，例如，自适应霍夫曼编码 [30][39]–[42]用于尽量降低出现频率最高的项的深度。不过，其重配置成本有所不同：在自适应霍夫曼算法中，交换两棵子树的成本可能仅为一次操作。

一些数据结构尝试通过扩展和修改二叉搜索树来获得更好的竞争比（参见表 II，其中总结了相关内容）：第一个例子是 PokeTree [43]，它在树的内部节点之间添加了额外指针，并实现了相对于最优二叉搜索树的 $O(\text{loglog } n)$ 竞争比。此外，还有基于跳表的自调整数据结构 [44], [45], , 它们作为平衡树的替代方案，以概率平衡取而代之。文献 [46]，研究了跳表的偏置版本，随后文献 [47] 给出了静态最优变体，而文献 [48]则在受限模型中提出了动态最优版本。另一个例子是 Iacono 工作集结构 [49]，它结合了一系列自调整平衡二叉搜索树和双端队列，最坏情况下的运行时间为 $O(\log n)$，但不具备动态最优性。我们尚未发现有研究通过增强这些数据结构来改善其竞争比。

我们的工作也受到新兴的自调整数据中心网络的启发。近期的光通信技术使数据中心能够快速且频繁地重新配置 [8], [18], [50]–[58], , 关于近期综述，参见 [59] 。本文提及的数据中心应用基于 [15] 提出的匹配模型。近期，[60] 引入了

数据结构	结构	比例
Iacono 结构 [49]	树与双端队列	$O(\log\ n)$
跳表 [44]	链表	$O(\log\ n)$
PokeTree [43]	树与动态链接	$O(\log\log n)$
SeedTree	Tree	$\theta(1)$

表二：与其他支持局部搜索的自调整数据结构的比较。迄今已知的最佳竞争比，是按照相应数据结构的成本模型和最优离线算法计算的。我们注意到，其他数据结构在设计中都未考虑容量。

一种基于该模型构建自调整网络的在线算法，但作者并未为其方法提供动态最优性证明。

研究表明，可以利用独立树构建具备需求感知能力的自调整数据中心网络 [61]，这些树称为自我树，广泛应用于各种网络设计 [8][50], [62], [63]，也为我们的模型提供了启发。然而，迄今为止，如何设计支持局部路由和调整的自调整常数竞争比树，仍是一个未解决的问题；而这在动态环境中是一个理想的特性。

最后但同样重要的是，我们的工作还与点对点网络有着有趣的联系 [12], [64]。已知采用预先分配且固定容量的一致性哈希可以显著改善负载均衡 [13], [14]，这具有有趣的应用，例如用于 Vimeo 的流媒体服务 [65]和 Google 的云服务 [13]。尽管这些方法受益于具备容量的数据结构，但它们并不具备需求感知能力。

七、结论与未来工作

本文提出并评估了一种自调整的局部树结构，*SeedTree*，它能够以在线且具有常数竞争比的方式适应工作负载。*SeedTree* 支持容量增强方法，同时提供局部路由，这对其他自调整结构和应用也可能有所助益。我们展示了如何将算法转换为匹配模型，以便应用于可重构数据中心，并在合成和真实世界的通信轨迹上评估了该算法。实验评估所用的代码可在github.com/inet-tub/SeedTree获取。

我们认为，这项工作为未来研究开辟了若干有趣的方向。具体而言，虽然我们迄今主要关注随机化方法，但探索*SeedTree*的确定性变体将很有意义。此外，尽管树是构建更复杂网络的基本组件（例如，它们如今已应用于数据中心），但基于*SeedTree*的网络仍有待设计和评估。

参考文献

[1] C. Avin、M. Ghobadi、C. Griner 和 S. Schmid，《流量轨迹的复杂性及其影响》，发表于*ACM SIGMETRICS*，2020年。[2] T. Benson、A. Anand、A. Akella 和 M. Zhang，《了解数据中心流量特征》，发表于*ACM SIGCOMM CCR*，2010年。[3] O. Michel、R. Bifulco、G. Retvari 和 S. Schmid，《可编程数据平面：抽象、架构、算法与应用》，发表于*ACM CSUR*，2021年。

[4] W. Kellerer、P. Kalmbach、A. Blenk、A. Basta、M. Reisslein 和 S. Schmid，《可适配、数据驱动的软件化网络：综述、机遇与挑战》，发表于 IEEE PIEEE，2019年。[5] A. Fischer、J. F. Botero、M. T. Beck、H. de Meer 和 X. Hesselbach，《虚拟网络嵌入：综述》，IEEE通信调查与教程，2013年。[6] M. N. Hall、K.-T. Foerster、S. Schmid 和 R. Durairajan，《可重构光网络综述》，发表于 OSN，2021年。[7] A. Borodin 和 R. El-Yaniv，《在线计算与竞争分析》，剑桥大学出版社，2005年。[8] C. Avin、K. Mondal 和 S. Schmid，《有界度的需求感知网络设计》，发表于 DISC，2017年。[9] D. D. Sleator 和 R. E. Tarjan，《自调整二叉搜索树》，《ACM杂志》，1985年。[10] C. Avin、K. Mondal 和 S. Schmid，《下推树：最优自调整完全树》，发表于 IEEE/ACM、TON，2022年。[11] J. Iacono，《键无关最优性》，《算法学》，2005年。[12] I. Stoica、R. T. Morris、D. Liben-Nowell、D. R. Karger、M. F. Kaashoek、F. Dabek 和 H. Balakrishnan，《Chord：一种可扩展的互联网应用点对点查找协议》，IEEE/ACM Trans. Netw.，2003年。[13] V. S. Mirrokni、M. Thorup 和 M. Zadimoghaddam，《具有有界负载的一致性哈希》，发表于 ACM-SIAM离散数学与理论计算机科学研讨会，2018年。[14] A. Aamand、J. B. T. Knudsen 和 M. Thorup，《动态球箱集合的负载均衡》，发表于 ACM SIGACT计算机科学理论研讨会，2021年。[15] C. Griner、J. Zerwas、A. Blenk、S. Schmid、M. Ghobadi 和 C. Avin，《Cerberus：数据中心拓扑设计中的选择之力（吞吐量视角）》，发表于 ACM SIGMETRICS，2021年。[16] M. L. Waskom，《seaborn：统计数据可视化》，J. of Open Source Softw.，2021年。[17] J. D. Hunter，《Matplotlib：二维图形环境》，Comput. Sci. Eng.，2007年。[18] C. Avin、M. Bienkowski、I. Salem、R. Sama、S. Schmid 和 P. Schmidt，《使用转子漫步的确定性自调整树网络》，发表于 IEEE国际分布式计算系统会议，2022年。[19] C. Dobraunig、M. Eichlseder 和 F. Mendel，《SHA-512/224 和 SHA-512/256 的分析》，IACR Cryptol. ePrint Arch.，2016年。[20] D. D. Sleator 和 R. E. Tarjan，《列表更新与分页规则的摊销效率》，《美国计算机学会通讯》，1985年。[21] S. Albers，《带前瞻的列表更新问题竞争分析》，MFCS，1994年。[22] S. Kamali 和 A. L´opez-Ortiz，《列表更新算法与模型综述》，发表于 LNTCS，2013年。[23] S. Albers 和 M. Janke，《付费交换模型下随机化列表更新的新界》，发表于 STACS，2020年。[24] S. Albers、B. Von Stengel 和 R. Werchner，《用于列表更新问题的组合式位与时间戳算法》，《信息处理快报》，1995年。[25] T. Garefalakis，《一类新的随机化列表访问算法》，发表于 ESA，1997年。[26] N. Reingold、J. R. Westbrook 和 D. D. Sleator，《用于列表更新问题的随机化竞争算法》，《算法学》，1994年。[27] S. Albers 和 S. Lauer，《具有引用局部性的列表更新》，发表于 ICALP，2008年。[28] J. M. Lucas，《竞争性二叉搜索树算法的标准形式》，罗格斯大学，1988年。[29] E. D. Demaine、D. Harmon、J. Iacono 和 M. Patrascu，《动态最优性——几乎实现》，发表于 IEEE计算机科学基础研讨会，2004年。[30] G. V. Cormack 和 R. N. Horspool，《自适应霍夫曼编码算法》，《信息处理快报》，1984年。[31] P. Bose、K. Douıeb、V. Dujmovi´c 和 R. Fagerberg，《一种具有最优最坏情况访问时间的 o (log log n)-竞争性二叉搜索树》，发表于 SWAT，2010年。[32] C. C. Wang、J. Derryberry 和 D. D. Sleator，《O (log log n)-竞争性动态二叉搜索树》，发表于 ACM-SIAM离散数学与理论计算机科学研讨会，2006年。[33] G. F. Georgakopoulos，《链式伸展树，或如何通过伸展操作实现并证明 loglogn-竞争比》，《信息处理快报》，2008年。[34] A. Blum、S. Chawla 和 A. Kalai，《列表与树的静态最优性和动态搜索最优性》，发表于 ACM-SIAM离散数学与理论计算机科学研讨会，2002年。[35] J. I. Munro，《线性搜索的竞争比》，发表于 ESA，2000年。[36] E. D. Demaine、D. Harmon、J. Iacono、D. M. Kane 和 M. Patrascu，《二叉搜索树的几何》，发表于 ACM-SIAM离散数学与理论计算机科学研讨会，2009年。[37] K. Fox，《最大贪心二叉搜索树的上界》，发表于 WADS，2011年。[38] J. Iacono，《探寻动态最优性猜想》，载于《空间高效数据结构、数据流与算法》，2013年。[39] D. E. Knuth，《动态霍夫曼编码》，《算法杂志》，1985年。[40] R. L. Milidi´u、E. S. Laber 和 A. A. Pessoa，《FGK算法压缩损失的界定》，《算法杂志》，1999年。[41] A. Moffat，《霍夫曼编码》，ACM CSUR，2019年。[42] J. S. Vitter，《动态霍夫曼编码的设计与分析》，《美国计算机学会会刊》，1987年。[43] J. Kujala 和 T. Elomaa，《Poketree：一种动态竞争性强且最坏情况性能良好的数据结构》，载于 ISAAC，2006年。[44] W. Pugh，《跳表：平衡树的一种概率替代方案》，《美国计算机学会通讯》，1990年。[45] C. Avin、I. Salem 和 S. Schmid，《自调整跳表网络路由的工作集定理》，载于 IEEE INFOCOM，2020年。[46] A. Bagchi、A. L. Buchsbaum 和 M. T. Goodrich，《偏置跳表》，《算法学》，2005年。[47] V. Ciriani、P. Ferragina、F. Luccio 和 S. Muthukrishnan，《一种用于外存中字符串访问序列的数据结构》，《美国计算机学会算法汇刊》，2007年。[48] P. Bose、K. Douıeb 和 S. Langerman，《跳表和B树的动态最优性》，载于 ACM-SIAM离散数学与理论计算机科学研讨会，2008年。[49] J. Iacono，《伸展树的替代方案：最坏情况访问时间为 o(log n)》，载于 ACM-SIAM离散数学与理论计算机科学研讨会，2001年。[50] C. Avin、K. Mondal 和 S. Schmid，《以最小拥塞和路由长度进行需求感知网络设计》，载于 IEEE INFOCOM，2019年。[51] H. Ballani、P. Costa、R. Behrendt、D. Cletheroe、I. Haller、K. Jozwik、F. Karinou、S. Lange 等，《Sirius：采用纳秒级光交换的扁平数据中心网络》，载于 ACM SIGCOMM，2020年。[52] K. Chen、A. Singla、A. Singh、K. Ramachandran、L. Xu、Y. Zhang、X. Wen 和 Y. Chen，《Osa：一种具有前所未有灵活性的数据中心网络光交换架构》，IEEE/ACM网络汇刊，2014年。[53] F. Douglis、S. Robertson、E. Van den Berg、J. Micallef、M. Pucci、A. Aiken、M. Hattink、M. Seok 和 K. Bergman，《Fleet——以每秒10太比特的速度通过快速通道加速执行：项目概述》，《IEEE互联网计算》，2021年。[54] K.-T. Foerster、M. Ghobadi 和 S. Schmid，《可重构数据中心架构的算法复杂度特征》，载于 ACM/IEEE网络与通信系统架构研讨会，2018年。[55] M. Ghobadi、R. Mahajan、A. Phanishayee、N. Devanur、J. Kulkarni、G. Ranade、P.-A. Blanche、H. Rastegarfar 等，《Projector：敏捷可重构数据中心互连》，载于 ACM SIGCOMM，2016年。[56] J. Kulkarni、S. Schmid 和 P. Schmidt，《双层可重构数据中心中机会链路的调度》，载于 ACM并行算法与体系结构研讨会，2021年。[57] W. M. Mellette、R. Das、Y. Guo、R. McGuinness、A. C. Snoeren 和 G. Porter，《跨时间扩展以实现带宽效率和低延迟》，载于 USENIX网络系统设计与实现研讨会，2020年。[58] W. M. Mellette、R. McGuinness、A. Roy、A. Forencich、G. Papen、A. C. Snoeren 和 G. Porter，《Rotornet：一种可扩展、低复杂度的光数据中心网络》，载于 ACM SIGCOMM，2017年。[59] K.-T. Foerster 和 S. Schmid，《可重构数据中心网络综述：使能技术、算法与复杂度》，载于《SIGACT新闻》，2019年。[60] E. Feder、I. Rathod、P. Shyamsukha、R. Sama、V. Aksenov、I. Salem 和 S. Schmid，《面向匹配模型的惰性自调整有界度网络》，载于 IEEE INFOCOM，2022年。[61] C. Avin 和 S. Schmid，《迈向需求感知网络：自调整网络理论》，ACM SIGCOMM CCR，2018年。[62] ——，《Renets：静态最优需求感知网络》，载于 SIAM算法与离散优化研讨会，2021年。[63] B. S. Peres、O. A. de Oliveira Souza、O. Goussevskaia、C. Avin 和 S. Schmid，《分布式自调整树网络》，载于 IEEE INFOCOM，2019年。[64] D. R. Karger、E. Lehman、F. T. Leighton、R. Panigrahy、M. S. Levine 和 D. Lewin，《一致性哈希与随机树：用于缓解万维网热点的分布式缓存协议》，载于 ACM计算理论研讨会，1997年。[65] A. Rodland，《使用新的一致性哈希算法改进负载均衡》，Vimeo工程博客，Medium，2016年。