Alcom-FT Technical Report Series
ALCOMFT-TR-01-130

The Complexity of Constructing Evolutionary Trees Using Experiments

Gerth Stølting Brodal* Rolf Fagerberg*

Christian N. S. Pedersen* Anna Östlin†

Abstract

We present tight upper and lower bounds for the problem of constructing evolutionary trees in the experiment model. We describe an algorithm which constructs an evolutionary tree of n species in time $O(nd \log_d n)$ using at most $n\lceil d/2 \rceil(\log_{2\lceil d/2 \rceil -1} n + O(1))$ experiments for $d > 2$, and at most $n(\log n + O(1))$ experiments for $d = 2$, where d is the degree of the tree. This improves the previous best upper bound by a factor $\Theta(\log d)$. For $d = 2$ the previously best algorithm with running time $O(n \log n)$ had a bound of $4n \log n$ on the number of experiments. By an explicit adversary argument, we show an $\Omega(nd \log_d n)$ lower bound, matching our upper bounds and improving the previous best lower bound by a factor $\Theta(\log_d n)$. Central to our algorithm is the construction and maintenance of separator trees of small height. We present how to maintain separator trees with height $\log n + O(1)$ under the insertion of new nodes in amortized time $O(\log n)$. Part of our dynamic algorithm is an algorithm for computing a centroid tree in optimal time $O(n)$.

Keywords: Evolutionary trees, Experiment model, Separator trees, Centroid tree, Lower bounds

*BRICS (Basic Research in Computer Science, www.brics.dk, funded by the Danish National Research Foundation), Department of Computer Science, University of Aarhus, Ny Munkegade, DK-8000 Århus C, Denmark. E-mail: {gerth,rolf,cstorm}@brics.dk. Partially supported by the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

†Department of Computer Science, Lund University, Box 118, S-221 00 Lund, Sweden. E-mail: Anna.Ostlin@cs.lth.se. Partially supported by TFR grant 1999-344.

1 Introduction

The evolutionary relationship for a set of species is commonly described by an evolutionary tree, where the leaves correspond to the species, the root corresponds to the most recent common ancestor for the species, and the internal nodes correspond to the points in time where the evolution has diverged in different directions. The evolutionary history for a set of species is rarely known, hence estimating the true evolutionary tree for a set of species from obtainable information about the species is of great interest. Estimating the true evolutionary tree computationally requires a model describing how to use available information about species to estimate aspects of the true evolutionary tree. Given a model, the problem of estimating the true evolutionary tree is often referred to as constructing the evolutionary tree in that model.

In this paper we study the problem of constructing evolutionary trees in the *experiment model* proposed by Kannan, Lawler and Warnow in [16]. In this model the information about the species is obtained by *experiments* which can yield the evolutionary tree for any triplet of species, cf. Figure 1. The problem of constructing an evolutionary tree for a set of n species in the experiment model is to construct a rooted tree with no unary internal nodes and n leaves labeled with the species such that the topology of the constructed tree is consistent with all possible experiments involving the species. Hence, the topology of the constructed tree should be such that the induced tree for any three species is equal to the tree returned by an experiment on those three species.

The relevance of the experiment model depends on the possibility of performing experiments. A standard way to express phylogenetic information is by a distance matrix. A distance matrix for a set of species is a matrix where entry M_{ij} represents the evolutionary distance between species i and j, measured by some biological method (see [16] for further details). For three species a, b and c where $M_{ab} < \min\{M_{ac}, M_{bc}\}$ it is natural to conclude that the least common ancestor of a and b is below the least common ancestor of a and c, i.e. the outcome of an experiment on a, b and c can be decided by inspecting M_{ab}, M_{ac} and M_{bc}. The consistency of experiments performed by inspecting a distance matrix depends entirely on the distance matrix. Kannan *et al.* in [16] define a distance matrix as noisy-ultrametric if there exists a rooted evolutionary tree such that for all triplets of species a, b and c it holds that $M_{ab} < \min\{M_{ac}, M_{bc}\}$ if and only if the least common ancestor of a and b is below the least common ancestor of a and c in the rooted evolutionary tree. Hence, if a noisy-ultrametric distance matrix for the set of species can be obtained, it can be used to perform experiments consistently. Another and more direct method for performing experiments is DNA-DNA hybridization as described by Sibley and Ahlquist in [23]. In this experimental technique one measures the temperature at which single stranded DNA from two different species bind together. The binding temperature is correlated to the evolutionary distance, i.e. by measuring the binding temperatures between DNA strands from three species one can decide the outcome of the experiment by deciding which pair of the three species bind together at the highest temperature.

Kannan *et al.* introduce and study the experiment model in [16] under the assumption that experiments are flawless in the sense that they do not contradict each other, i.e. it is always possible to construct an evolutionary tree for a set of species that is consistent with all possible experiments involving the species. They present algorithms for constructing evolutionary trees with bounded as well as unbounded degree, where the degree of a tree is the maximum number of children for an internal node. For constructing binary evolutionary trees they present three different algorithms with running times $O(n \log n)$, $O(n \log^2 n)$ and $O(n^2)$ respectively, using $4n \log n$, $n \log_{3/2} n$ and $n \log n$ experiments respectively, where $\log n$ denotes $\log_2 n$. For constructing an evolutionary tree of degree d they present an algorithm with running time $O(n^2)$ using $O(dn \log n)$ experiments. Finally, for the general case they present an algorithm with running time $O(n^2)$ using $O(n^2)$ experiments together with a matching lower bound. Kao, Lingas, and Östlin in [17] present a randomized algorithm for constructing evolutionary trees of degree d with expected running time $O(nd \log n \log \log n)$. They also prove a lower bound $\Omega(n \log n + nd)$ on the number of experiments. The best algorithm so far for constructing evolutionary trees of degree d is due to Lingas, Olsson, and Östlin, who in [19] present an algorithm with running time $O(nd \log n)$ using the same number of experiments.

In this paper we present the first tight upper and lower bounds for the problem of constructing evolutionary trees of degree d in the experiment model. We present an algorithm which constructs an evolutionary tree for n species in time $O(nd \log_d n)$ using at most $n\lceil d/2 \rceil(\log_{2\lceil d/2 \rceil - 1} n + O(1))$ experiments for $d > 2$, and at most $n(\log n + O(1))$ experiments for $d = 2$, where d is the degree of the constructed tree. The algorithm is a further development of an algorithm from [19]. Our construction improves the previous best upper bound by a factor $\Theta(\log d)$. For $d = 2$ the previously best algorithm with running time $O(n \log n)$ had a bound of $4n \log n$ on the number of experiments. The improved constant factors on the number of experiments are important because experiments are likely to be expensive in practice, cf. Kannan *et al.* [16]. By an explicit adversary argument, we show an $\Omega(nd \log_d n)$ lower bound, matching our upper bounds and improving the previous best lower bound by a factor $\Theta(\log_d n)$.

Our algorithm also supports the insertion of new species with a running time of $O(md \log_d(n + m))$ using at most $m\lceil d/2 \rceil(\log_{2\lceil d/2 \rceil - 1}(n + m) + O(1))$ experiments for $d > 2$, and at most $m(\log(n+m)+O(1))$ experiments for $d = 2$, where n is the number of species in the tree to begin with, m is the number of insertions, and d is the maximum degree of the tree during the sequence of insertions.

Central to our algorithm is the construction and maintenance of *separator trees* of small height. We refer the reader to Section 2 for a detailed definition. The special class of separator trees we in Section 2 denote $1/2$-separator trees can also be denoted *centroid trees*, since the separating nodes are then centroids. A centroid of a tree is a node whose removal disconnects the tree into components each containing at most half of the nodes in the tree. Jordan's classical result establishes that any tree has either one or two centroid [15, 14]. Goldman [12] and Megiddo *et al.* [21] showed how to compute a centroid of a tree in $O(n)$ time.

Recursively locating centroids for each resulting component gives a centroid tree. By recursive applications of the algorithms from [12, 21] it follows that a centroid tree can be constructed in time $O(n \log n)$ (see Lemma 1 for further details). In Section 2, Lemma 2, we present an algorithm for constructing centroid trees, i.e. 1/2-separator trees, with optimal running time $O(n)$. Schwarz, Smid and Snoeyink [22] describe how to compute 1/2-separator trees (in [22] denoted 1/2-decomposition trees) for the case of binary trees in time $O(n)$, by modifying the algorithm of Guibas, Hershberger, Leven, Sharir and Tarjan [13] for computing centroid decompositions (in [13], centroid refers to a centroid edge in a binary tree).

In general, separator trees are a relaxation of centroid trees where the components resulting from deleting a node are not required to contain at most half of the nodes. Schwarz *et al.* [22] showed how to maintain 3/4-separator trees in amortized time $O(\log n)$ per insertion for the case of binary trees. The height of a 3/4-separator tree is bounded by $\log_{4/3} n$. In Section 2 we show how to maintain separator trees in amortized logarithmic time under the insertion of new nodes, such that the height of the separator tree is bounded by $\log n + O(1)$. Our main result for the dynamic case is summarized in Theorem 2. Inequality (1) is the essential bound required in the analysis of the number of experiments performed in our application to evolutionary trees.

The basic idea of transforming a tree into a new tree with logarithmic height is a fundamental approach used in many algorithms. For designing dynamic algorithms on trees several other general tree transformation techniques exist: Frederickson's topology trees [10, 11], Sleator and Tarjan's dynamic trees [24], and Alstrup *et al.*'s top trees [1, 2]. One application of such a tree transformation is in Cohen and Tamassia's algorithm for dynamic expression tree evaluation [7]. For parallel algorithms on trees related techniques exist, e.g. the centroid decomposition technique of Megiddo [20] and the accelerated centroid decomposition technique of Cole and Vishkin [8] (in [8, 20], centroid refers to the centroid paths in a tree).

The rest of this paper is organized as follows. In Section 2 we define separator trees and describe how to construct and efficiently maintain separator trees of small height. In Section 3 we present our algorithm for constructing and maintaining evolutionary trees. In Section 4 and 5 the lower bound is proved using an explicit adversary argument. The adversary strategy used is an extension of an adversary used by Borodin, Guibas, Lynch, and Yao [5] for proving a trade-off between the preprocessing time of a set of elements and membership queries, and Brodal, Chaudhuri, and Radhakrishnan [6] for proving a trade-off between the update time of a set of elements and the time for reporting the minimum of the set.

2 Separator Trees

In this section we define separator trees and present efficient algorithms for their constructing and maintenance.

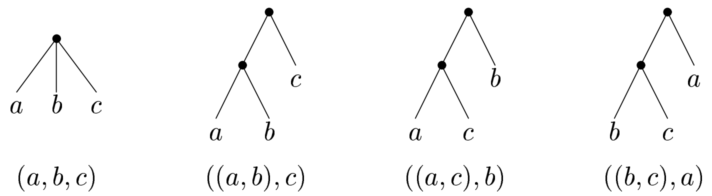

Figure 1: The four possible outcomes of an experiment for three species a, b and c.

Definition 1 *Let T be an unrooted tree with n nodes. A* separator tree S_T *for T is a rooted tree on the same set of nodes, defined recursively as follows: The root of S_T is a node u in T, called the* separator node. *The removal of u from T disconnects T into disjoint trees $T_1, \ldots, T_k$, where k is the number of edges incident to u in T. The children of u in S_T are the roots of separator trees for $T_1, \ldots, T_k$.*

Clearly, there are many possible separator trees S_T for a given tree T. An example is shown in Figure 2.

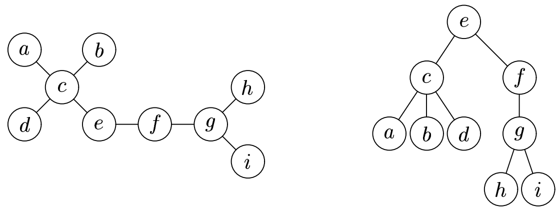

Figure 2: A tree T (left) and a separator tree S_T for T (right).

For later use, we note the following facts for separator trees:

Fact 1 *Let S_T be a separator tree for T, and let v be a node in T. If S_v denotes the subtree of S_T rooted at v, then:*

1. *The subgraph T_v induced by the nodes in S_v is a tree, and S_v is a separator tree for T_v.*
2. *For any edge from T with exactly one endpoint in T_v, the other endpoint is an ancestor of v in S_T, and each ancestor of v can be the endpoint of at most one such edge.*

The main point of a separator tree S_T is that it may be balanced, even when the underlying tree T is not balanced for any choice of root. The notion of balanced separator trees is contained in the following definition, where the size $|T|$ of a tree T denotes the number of nodes in T, and where T_i refers to the trees $T_1, \ldots, T_k$ from Definition 1.

Definition 2 *A separator tree is a* t-separator tree, *for a threshold* $t \in [1/2, 1]$, *if* $|T_i| \leq t|T|$ *for each* T_i *and the separator tree for each* T_i *is also a* t*-separator tree.*

Note that a t-separator tree is also a t'-separator tree for all $t' \geq t$. In Section 2.1 we first show how to construct $1/2$-separator trees in linear time. Such a tree has height at most $\lfloor \log n \rfloor$. We then in Section 2.2 consider dynamic separator trees and show how to maintain separators trees with small height in logarithmic time per insertion. A simple algorithm yields height $O(\log n)$ and a more involved algorithm improves the height bound to $\log n + O(1)$. Finally, we in Section 2.3 show how to extend the algorithms with a specific ordering of the children facilitating the use in Section 3 of separator trees for the efficient construction and maintenance of evolutionary trees in the experiment model.

2.1 Constructing Separator Trees

In Lemma 1 below we first give a simple algorithm for constructing $1/2$-separator trees in time $O(n \log n)$. In Lemma 2 we then improve the running time of the algorithm to $O(n)$ by adopting additional data structures.

We need the following definitions for our algorithms. For a node v in a rooted tree T, we define the *size* of v, denoted $|v|$, to be the number of nodes in the subtree rooted at v. We let the *heavy-child* of a node be a child of maximum size, where ties are broken arbitrarily. The edges to the heavy-children define a decomposition of T into disjoint *heavy-paths.* All nodes on a heavy-path, except the first node, are heavy-children, and the last node is a leaf.

Lemma 1 *Given a tree* T *with* n *nodes, a 1/2-separator tree for* T *can be constructed in time* $O(n \log n)$.

Proof. We first make T a rooted tree by letting an arbitrary node of T be the root. For all nodes v in T we compute $|v|$ and identify the heavy-paths in T in one traversal of T in time $O(n)$. We identify the root of the $1/2$-separator tree S_T as follows: We start at the root r of T and follow the heavy-path from r to the lowest node u where $|u| \geq n/2$ (possibly $u = r$), i.e. $|v| < n/2$ for all children v of u. The node u becomes the root of S_T. By removing u from T, the tree T splits into disjoint trees $T_1, \ldots, T_k$, where each tree T_i has size $n_i \leq n/2$, since the tree T_j containing the parent of u has size at most $n - |u| \leq n/2$. We recursively compute $1/2$-separator trees for each T_i. The root of each recursively constructed $1/2$-separator tree becomes a child of u in S_T.

Locating u takes time $O(n)$ since the heavy-path starting at the root of T contains at most n nodes. This implies that the construction time is bounded by $T(n)$, where $T(n)$ is given by the recurrence

$$T(n) \;\leq\; cn + \sum_{i=1}^{k} T(n_i) \;,$$

for some positive constant c, where $\sum_{i=1}^{k} n_i = n-1$ and $n_i \leq n/2$ for all $i = 1, \ldots, k$. By induction it follows that $T(n) \leq cn(\log n + 1)$. $\square$

The algorithm of Lemma 1 recomputes the sizes of all nodes and the heavy-paths for each recursive call. Furthermore it does not exploit that the sizes along a heavy-path is monotonically decreasing when searching for the root of the separator tree. The following lemma shows how to exploit these two observations to reduce the construction time to $O(n)$.

Lemma 2 *Given a tree T with n nodes, a 1/2-separator tree for T can be constructed in time $O(n)$.*

Proof. The basic algorithm is identical to the algorithm described in the proof of Lemma 1. To improve the search for separator nodes we keep track of the heavy-paths as balanced search trees. Each heavy-path is stored in a search tree where the elements are the nodes on the heavy-path and the keys are the sizes of the nodes. The search trees should support the operations: key, join, split, successor, and addpathcost. Given a pointer to an element, key returns the key of the element. The operation join concatenates two search trees, provided that the keys in one search tree are all smaller than the keys in the other search tree, and split splits a search tree at a particular element. The operation addpathcost adds the same value to all keys in a search trees. Given a key, successor finds the element with the smallest key larger than, or equal to, the given key. As described by Tarjan [25, Chapter 5], all these operations can be supported in time $O(\log n)$, where n is the number of elements in the search tree. Given a sorted list, the corresponding search tree can be constructed in linear time.

Initially, we make T rooted, compute $|v|$ for all nodes v in T, identify heavy-paths in T, and construct a search tree for each heavy path. In total this takes time $O(n)$. At each node which is the head of a heavy-path, we store a link to the search tree storing the heavy-path starting at that node. For each node we store a link to a priority queue which stores the children of the node, except the heavy-child, with priorities equal to their sizes. The priority queues should support insertion of an element with arbitrary priority and deletion of the element with maximum priority in logarithmic time, and construction of a queue in linear time, as e.g. binary heaps [9, 26] do. The total time for constructing the initial priority queues at the nodes is $O(n)$.

We find the root of the 1/2-separator tree S_T using the search tree R storing the heavy-path starting at the root r of T. We first observe that $|r|$ is the maximal key in R, which can be found in time $O(\log n)$ by the operation key. To find the root of S_T we perform the query successor($|r|/2$) on R, which by construction locates a node u in T where $|u| \geq |r|/2$ and all children v of u have $|v| < |r|/2$, i.e. u is a valid node for the root of S_T. Removing u from T splits T into disjoint trees $T_1, \ldots, T_k$, where each subtree T_i has size $n_i \leq n/2$. See Figure 3. We recursively compute a 1/2-separator tree for each T_i. The root of each constructed 1/2-separator tree becomes a child of u in the separator tree S_T.

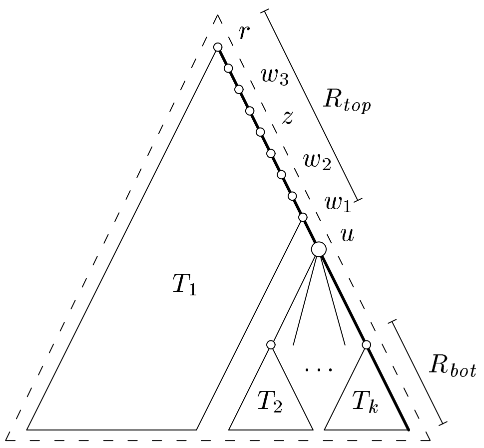

Figure 3: The separator node u on the heavy-path $R = R_{top} \cup \{u\} \cup R_{bot}$, and the nodes $w_1, \dots, w_\ell$ where to update the left-to-right order of the children.

To avoid recomputing the heavy-paths for each of the recursive calls we update the already computed heavy-paths, and corresponding search trees, as described below in time $O(\log^2 n)$. This implies that the total construction time is bounded by $O(n + T(n))$, where

$$T(n) \;\le\; c(1 + \lfloor \log n \rfloor^2) + \sum_{i=1}^{k} T(n_i) \;,$$

for some positive constant c, where $\sum_{i=1}^{k} n_i = n - 1$ and $n_i \le n/2$ for all $i = 1, \dots, k$. By induction it follows that $T(n) \le cn + cn \sum_{i=0}^{\lfloor \log n \rfloor} i^2/2^i \le 7cn$, since $\sum_{i=0}^{\infty} i^2/2^i = 6$. We conclude that the total construction time is $O(n)$.

To update the heavy-paths, we start by splitting the search tree R containing u into three parts, R_{top}, u, and R_{bot}, where R_{top} stores the part of the heavy-path above u, and R_{bot} stores the part of the heavy-path below u. See Figure 3. This can be done in time $O(\log n)$ by applying the split operation twice. By adding a link from the heavy child of u in T, i.e. the node in R_{bot} with maximum key, to the search tree R_{bot}, it follows that for all the T_i trees that were rooted at the children of u the heavy-paths are correctly stored as search trees.

What remains is to update the search trees storing the heavy-paths in the tree T_j that contains the parent of u from T, i.e. the part of T above u. First we update the keys (i.e. sizes) of all nodes in R_{top} by subtracting the size of the subtree of T that was rooted at u, i.e. the key of u. This takes time $O(\log n)$ by the addpathcost operation, and ensures that the keys of all nodes in T_j equal their new sizes. What remains is to reorder the search trees for the paths in T_j such that they represent the heavy-paths in T_j, i.e. to identify the new heavy-children of the nodes in R_{top}.

We define nodes $w_1, w_2, \ldots, w_\ell$ as follows. Let w_1 be the parent of u in T, and w_{i+1} the ancestor of w_i in R_{top} determined by $\mathsf{successor}(2\,|w_i|)$, where $|w_i| = \mathsf{key}(w_i)$. See Figure 3. Since $|w_{i+1}| \geq 2\,|w_i|$ and $|T_j| \leq n/2$, it follows that $|w_i| \geq 2^{i-1}$ and $\ell \leq \log n$. We now argue that $w_1, \ldots, w_\ell$ are the only nodes in R_{top} where the child also in R_{top} is no longer a heavy-child. Consider a node z in R_{top} between w_i and w_{i+1}. Since $|w_i| < |z| < 2\,|w_i|$, it follows that the child of z in R_{top} is still the heavy child of z in T_j since it has at least size $|w_i| > |z|/2$, i.e. the children of z are correctly placed.

Now consider w_i. Let x be the heaviest child of w_i in T and let Q be the priority queue storing the remaining children of w_i. If Q is empty no updates are necessary at w_i. Otherwise let y be the child of w_i with maximum key in Q, i.e. the second heaviest child of w_i in T. If $i = 1$, then $x = u$ and y becomes the new heavy child of w_1. We delete the maximum element y from Q; join R_{top} with the search tree storing the heavy-path starting in y; and let R_{top} be the resulting search tree. We continue recursively updating R_{top} at w_{i+1}.

Otherwise $i \geq 2$. If $|x| \geq |y|$ in T_j, i.e. if $|x|$ is larger than or equal to the key of y in Q, then x is also the heavy-child of w_i in T_j. Otherwise, x is not the heavy-child of w_i in T_j, and we must update the heavy-paths accordingly. First, we split R_{top} between x and w_i, this results in two search trees R'_{top}, storing the nodes on the path from the root to w_i, and R''_{top}, storing the heavy-path which starts at x. We then delete the maximum element y from Q; insert x into Q; and let x have a pointer to R''_{top}. The node y is the new heavy-child of w_i. We join R'_{top} with the search tree storing the heavy-path starting at y, and let R_{top} be the resulting search tree. We continue recursively updating R_{top} at w_{i+1}.

It takes time $O(\log n)$ to find each w_i, and at each w_i we use time $O(\log n)$ to update the heavy child information. Since $\ell \leq \log n$, the total time for reestablishing the heavy-paths is $O(\log^2 n)$, which concludes the proof. $\square$

2.2 Maintaining Separator Trees

In this section, we first discuss how to insert new nodes into a tree T and its corresponding separator tree S_T, and then present methods for maintaining balance and height in a separator tree S_T during such insertions.

We allow two types of node insertions in T: Type 1, which is the addition of a new leaf node connected to an existing node in T by a new edge, and Type 2, which is the addition of a new node by breaking an existing edge into two edges. Figure 4 shows a tree before and after one addition of each type, with new nodes in bold.

In the separator tree S_T for T, we for a Type 1 insertion insert the new node as a child of the single node in T to which it is connected, and for a Type 2 insertion we insert the new node as a child of the deepest node in S_T among the two nodes in T to which it is connected. The two nodes are on the same root to leaf path follows from 2. in Fact 1 The resulting tree is easily seen to be a separator tree for the updated tree T. Figure 5 shows the insertions into S_T corresponding to the insertions into T shown in Figure 4.

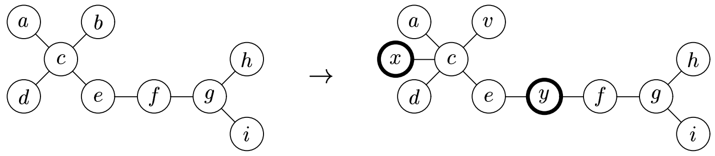

Figure 4: Insertions into a tree T.

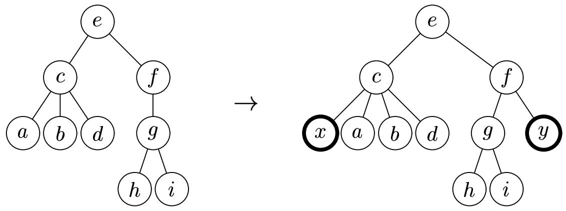

Figure 5: Insertions into S_T corresponding to Figure 4.

The methods we now present for maintaining balance and height in separator trees during insertions of new nodes are based on rebuilding of subtrees, and are inspired by methods of Andersson and Lai described in [3, 4] for maintaining small height in binary search trees. We first show how the linear time construction algorithm for 1/2-separator trees from Lemma 2 leads to a simple algorithm for keeping separator trees well balanced. The height bound achieved by this algorithm is $O(\log n)$, using $O(\log n)$ amortized time per update. We then use a two-layered structure to improve the height bound to $\log n + O(1)$ without sacrificing the time bound. The improved constant factor in the height bound is significant for our use of separator trees for maintaining evolutionary trees in the experiment model, since the number of experiments for an insertion of a new species will turn out to be proportional to the height of the separator tree. Furthermore, this height bound is within an additive constant of the best bound possible, as trees exist where any separator tree must have height at least $\lfloor \log n \rfloor$, e.g. a tree which is a single path.

Statements about amortized complexity for data structures normally assume an initially empty structure—this is a special case of the statements below.

Lemma 3 *For any $0 < \varepsilon < 1/4$, a $(1/2 + \varepsilon)$-separator tree can be maintained in amortized time $O((\log n)/\varepsilon)$ per insertion, provided that the initial tree is a 1/2-separator tree.*

Proof. We let each node v in the separator tree store the size $|v|$ of its subtree (its number of descendants in the separator tree, including v itself), as well as its depth (the number of edges on the path to the root in the separator tree).

During insertions, we update this information along the path to the root, and check for violations of the threshold. If any violating nodes are found, we rebuild the subtree rooted at the highest node v among these, using Lemma 2, and then restore the size and depth information by a traversal of the rebuilt subtree. Let u denote the largest child of v just before the rebuild. We have $|u| > (1/2+\varepsilon)|v|$. Immediately after the last time we did a rebuild involving v, either u was not present, or we had $|u|_{\text{then}} \leq |v|_{\text{then}}/\,2 \leq |v|_{\text{now}}/\,2$. As $|u|_{\text{now}} > (1/2+\varepsilon)|v|_{\text{now}}$, at least $\varepsilon|v|_{\text{now}}$ insertions have taken place below v since then. Charging these insertions $O(1/\varepsilon)$ each will cover the $O(|v|_{\text{now}})$ cost for rebuilding the subtree of v and restoring the information at the nodes. Thus, if an insertion is charged $O(1/\varepsilon)$ for each node on the path from the new node to the root, the cost of all rebuildings are covered. Since the height of the separator tree is at most $\log_{1/(1/2+\varepsilon)} n$, which is $O(\log n)$ by $\varepsilon < 1/4$, the stated time bound follows. □

Lemma 4 *A $\left(\frac{1}{2}+\frac{1}{3\lceil\log n\rceil}\right)$-separator tree can be maintained with a height bound of $\lceil\log n\rceil$ in amortized time $O(\log^2 n)$ per insertion, provided that the initial tree is a 1/2-separator tree.*

Proof. In the method of Lemma 3, we maintain $\varepsilon = \frac{1}{3\lceil\log N\rceil}$, where N denotes a power of two larger than or equal to n. Initially $N = 2^{\lceil\log n\rceil+1}$, i.e. the smallest power of two larger than or equal to $2n$. Whenever n exceeds N, we double N, which causes ε to change, and we rebuild the entire separator tree as a new 1/2-separator tree by applying the algorithm of Lemma 2. Note that n must at least be doubled before the first rebuild can occur and between two rebuilds, i.e. we can charge the preceding insertions the cost of a rebulding

For a separator tree with threshold t, the size of a subtree rooted at depth i is at most $n \cdot t^i$. Using the standard inequality $(1+x/y)^y \leq e^x$, we have

$$n\left(\frac{1}{2}+\frac{1}{3\lceil\log N\rceil}\right)^{\lceil\log n\rceil} \leq n\frac{1}{2^{\lceil\log n\rceil}}\left(1+\frac{2}{3\lceil\log n\rceil}\right)^{\lceil\log n\rceil} \leq e^{2/3} < 2\ ,$$

i.e. a subtree rooted at depth $\lceil\log n\rceil$ must be a single node. It follows that the height of a separator tree is at most $\lceil\log n\rceil$.

By Lemma 3 the amortized time for insertions is $O((\log n)/\varepsilon) = O(\log^2 n)$, as the amortized cost of the global rebuildings is $O(1)$ per insertion by Lemma 2. □

In the next theorem, we reduce the amortized time bound to $O(\log n)$.

Theorem 1 *Let T be an unrooted tree initially containing n nodes. After $O(n)$ time preprocessing, a separator tree for T with a height bound of $\log(n+m)+5$ can be maintained during m insertions in time $O(m\log(n+m))$.*

Proof. We use a two-layered rebalancing mechanism to reduce the time bound from Lemma 4 by a factor of $\Theta(\log n)$. The top rebalancing scheme will work on a *sample* U of the nodes of the underlying tree T. If the nodes in U and all the edges with which they are incident are removed from T, it will break into a set of connected components. We denote these *the components induced by U*.

We maintain the following invariants on U, where Δ is a multiple of four within $\Theta(\log n)$.

1. Each component induced by U contains less than Δ nodes.
2. Each component induced by U is connected to at most two nodes from U.

We view U as a graph by letting two nodes in U be connected by an edge if they in T are connected to the same induced component, or if they are already neighbors in T. By Invariant 2, each component is connected to either one or two nodes in U (unless U is empty, in which case T itself is a single component). The components connected to only one node in U we denote *leaf components*. The components connected to two nodes in U may be associated with the corresponding edge in U, and we denote these *edge components*. Assigning an empty edge component to edges in T which connect two nodes in U, we obtain a one-to-one correspondence between the edges of U and the edge components. Using this, it is easy to see that since T is a tree, U is also a tree.

The separator tree for T will be a separator tree for U where separator trees for the induced components are attached as extra children of the nodes. The separator tree for a leaf component is attached as a child of the single node in U to which it is connected in T. The separator tree for an edge component is attached as a child of the node of largest depth in the separator tree for U, among the two nodes in U to which it is connected in T.

We remark that this combined structure really does constitute a separator tree for T: removing the root r of the structure (i.e. the root of the separator tree for U) from T breaks T into pieces, of which the pieces containing no nodes from U exactly are the leaf components attached as children of r, and the pieces containing nodes from U are in one-to-one correspondence with the pieces of U left when removing r from U. Continuing recursively proves the remark true.

We now discuss how to update the separator tree for T after an insertion into T. For a Type 1 insertion, the existing node to which the new node is connected may belong to U. In this case, the new node will form a new leaf component of size one, which is added to the structure. For all other insertions, an existing (but possibly empty) leaf or edge component C will grow by exactly one node. After inserting into C, the component is rebuilt to threshold $1/2$ by the algorithm from Lemma 2. If the number of nodes in C has reached Δ due to the insertion, it is now split into components of size at most $\Delta/2$ by adding the root v of the separator tree for C to the sample U. For edge components, one of the new components formed by the split may be connected to *three* nodes in U. Specifically, this happens if and only if v is not located on the unique path in T between the two nodes $u_1, u_2 \in U$ to which C is connected. To maintain Invariant 2, we also add to U the node w located where the paths from v to u_1 and from v to u_2 separate. In total, this splits the violating component into three or more components each being connected to at most two nodes in U, reestablishing the invariant.

We build a $1/2$-separator tree for each of the components which arise by the inclusion of w in U, let these components be children of w, and let w be the single child of v in the separator tree for U.

The addition of v and w into U constitutes two insertions into the separator tree for U, below the node of which C was a child. To maintain balance in the separator tree for U after these insertions, we use the rebalancing scheme from Lemma 4.

After a rebuild of a subtree S in the separator tree for U during such rebalancing, the depth of each node in S may have changed. As said, an edge component in the separator tree for T should be a child of the node of largest depth in the separator tree for U, among the two nodes in U to which it is connected (these nodes are ancestors of each other in the separator tree for U, as follows from Fact 1). Therefore, for edge components connected to at least one node in S we must after the rebuild check the updated depth information of these nodes, and change parent of the component if necessary. This is done by a traversal of S during which we inspect all edge components connected to nodes in it. By the one-to-one correspondence between edge components and edges of U, the number of components to inspect is equal to the number of edges in U with at least one endpoint in S. By Fact 1, this number is bounded by $|S| - 1$ plus the depth of the root of S in the separator tree for U. Thus, by the height bound in Lemma 4, inspection of edge components will only add an additive logarithmic term to the rebalancing cost for the separator tree for U, which therefore remains amortized $O(\log^2 |U|)$.

To maintain the value of Δ, we rebuild the entire structure whenever n has doubled, setting Δ to $4\lceil(\log n)/4\rceil$. We now discuss how to perform such a global rebuilding in $O(n)$ time. The same algorithm is also used as preprocessing to construct the separator tree for the initial tree T. Thus, preprocessing takes $O(n)$ time.

To construct the separator tree for some existing tree τ, we first generate the sample U and its induced components. We then use the algorithm from Lemma 2 to construct a separator tree for U and for each component. Finally, we attach each leaf component to the single node from U to which it is connected, and attach each edge component as a child of the lowest of the two nodes in U to which it is connected. In the case of the preprocessing, we will need the generated U to fulfill Invariant 1 with a value of $\Delta/2$ instead of Δ in order to obtain the stated time bound for the first n insertions. We use this value in the description here.

The sample U is generated by a traversal of τ using e.g. a depth first search, during which we maintain a sample and its induced components for the part of τ traversed so far. The algorithm for this is similar to the insertion procedure described above, except that no separator trees are maintained for neither U nor the edge and leaf components. Specifically, when a new node v is encountered during the traversal, we consider the node w from which it was reached. If w is in U, we start a new component. If not, v is added to the component of w. If the number of nodes in a component reaches $\Delta/2$, we split it into components containing at most $\Delta/4$ nodes each by adding one of its nodes to U. To locate this node, we use the method described in the first lines of the proof of Lemma 1. If necessary, we also split one of the new components to maintain Invariant 2.

When a component overflows, at least $\Delta/4$ nodes have been inserted into it since it was created by a component split or by the start of a new component. Hence, at most $4n/\Delta$ overflows can occur during the generation of U. As each overflow can be handled in time $O(\Delta)$, the generation of U can be performed in time $O(n)$. By the time bound from Lemma 2, the entire separator tree for τ can be constructed in $O(n)$ time. This concludes our description of the global rebuilding of the structure.

We now analyze the time for m insertions in the separator tree. Clearly, we only need to consider the case $m < n$, as the rebalancing scheme is reset by a global rebuild each time n has doubled, and as each such rebuild except the initial construction amounts to $O(1)$ amortized work per insertion. The insertion into an induced component and the rebuilding of its separator tree by Lemma 2 takes $O(\Delta) = O(\log n)$ time, including any splitting of the component due to overflow. Each overflowing component gives rise to at most two insertions into the separator tree for U. When a component is created by a component split or by the start of a new component, it contains at most $\Delta/2$ nodes. The size of the components after the construction of the initial separator tree is also bounded by $\Delta/2$. Hence, after m insertions, at most $2m/\Delta$ overflows of components can have occurred. Each overflow gives rise to at most two insertions into the separator tree for U, each of which costs $O(\log^2 |U|) = O(\log^2 n)$. The total cost of these insertions is then $O((m \log^2 n)/\Delta) = O(m \log n)$. The stated time bound follows.

To prove the stated height bound, note that in the initial tree, U contains at most $8n/\Delta$ nodes. At most $2m/\Delta$ overflows of components have occurred during insertions, each of which inserts at most two more nodes into U. Hence, the size of U is bounded by $8(n+m)/\Delta$. By Lemma 4, the height of the separator tree for U is most $\log(8(n+m)/\Delta) + 1 = \log(n+m) + 4 - \log\Delta$. By Invariant 1, the height of the separator trees for the induced components is at most $\log\Delta$, as these are 1/2-separator trees. Adding one to the height to account for the edges connecting the root of the separator trees for components to nodes in the separator tree for U gives the stated height bound. □

2.3 Ordered Separator Trees

We now extend the separator trees maintained by the algorithm from Theorem 1 with a specific ordering of the children, facilitating our use of separator trees in Section 3 for finding insertion points for new species in evolutionary trees. The basic idea is to speed up the search in the separator tree by considering the children of the nodes in decreasing size-order. This ensures a larger reduction of subtree size in the case that many children have to be considered before the subtree to proceed the search in is found.

The below lemma shows that size order can be assumed after a rebuild of a separator tree.

Lemma 5 *A separator tree of size n can be processed in time $O(n)$ such that children of nodes are sorted in decreasing size-order.*

Proof. We first traverse the separator tree in linear time and compute the size of all nodes. Since the sizes are bounded by n, a list of all nodes can be sorted in decreasing size order in linear time using bucket-sort [18]. By scanning through the sorted list of nodes in increasing size order making the nodes visited the first child of their respective parents, we in linear time update the order of children at each node in the separator tree such that they are sorted in decreasing size-order. □

However, for the two layered structure from Theorem 1, further details are needed to achieve the following.

Theorem 2 *Let T be an unrooted tree initially containing n nodes. After $O(n)$ time preprocessing, an ordered separator tree for T can in time $O(m \log(n+m))$ be maintained during m insertions in a way such that the height is bounded by $\log(n+m)+5$ and such that for any path $(v_1, v_2, \ldots, v_\ell)$ from the root v_1 to a node v_ℓ in the separator tree, it holds that*

$$\prod_{d_i \leq 2} 2 \cdot \prod_{d_i > 2} d_i \quad < \quad 16d(n+m) \ , \tag{1}$$

where d_i is the number which v_{i+1} has in the ordering of the children of v_i, for $1 \leq i < \ell$, and d is $\max\{d_1, \ldots, d_{\ell-1}\}$.

Proof. The proof is by an extension of the construction from Theorem 1, and familiarity with the proof of this theorem is assumed here.

We extend the construction by an ordering of the children of the nodes of the separator tree as follows. For a node v in U, the children which belong to U will be first in the ordering, followed by the the children not in U. Furthermore, the children belonging to U will be in decreasing order in terms of the size of their subtrees in the separator tree for U (which is not the same as the size of their subtrees in the entire separator tree for T). For a node v in U, we do not define any particular order among the children not in U. For a node v not in U, the children (none of which can be in U), will be in decreasing order in terms of the size of their subtree in the separator tree for the induced component in which they are contained.

The above ordering must be maintained during insertions and rebalancing of the structure. Whenever an insertion occurs in an induced component, it is completely rebuilt by the algorithm from Lemma 2. This algorithm is also used as the fundamental operation in the rebalancing of the separator tree for U. After an invocation of this algorithm, the order order can be restored without affecting the time bound, by Lemma 5. When an insertion into U occurs due to the splitting of a component, the ordering may have to change among children of nodes on the path from the insertion point to the root in the separator tree for U. With a proper linked list representation of the children of a node in groups of children with equal size, this can be done in constant time per node on the path, as the size of only one child per node changes, and the increase in size is only one. Thus, this takes time proportional to the height of the separator tree of U. All in all, the ordering can be maintained without affecting the time bound from Theorem 1. The height bound also follows from Theorem 1.

To prove the last claim of the lemma, i.e. inequality (1), note that a path $(v_1, v_2, \ldots, v_\ell)$ will first pass through nodes from U, then through nodes from a single induced component. Let v_j be the last node from U on the path.

We first consider the part $(v_1, v_2, \ldots, v_j)$ of the path lying within the separator tree for U. This separator tree by Lemma 4 has a threshold of $\frac{1}{2} + \frac{1}{3\lceil \log |U| \rceil}$. For $d_i \geq 2$, a descent into the d_i'th child must reduce by a factor of at least d_i the number of nodes in the current subtree of the separator tree for U. For $d_i = 1$, we can only claim a factor given by the threshold of the separator tree. Since this part of the path ends at the latest when there is a single node left in the subtree of the separator tree for U, we have the following for this part of the path:

$$1 \ \leq \ |U| \cdot \left(\frac{1}{2} + \frac{1}{3\lceil \log |U| \rceil} \right)^k \cdot \prod_{\substack{d_i \geq 2 \\ i<j}} \frac{1}{d_i} \ ,$$

where $k = |\{ i < j \mid d_i = 1 \}|$. From Lemma 4 the height of the separator tree for U is bounded by $\lceil \log |U| \rceil$. Using this and the inequality $(1 + x/y)^y \leq e^x$, we get

$$\left(\frac{1}{2} + \frac{1}{3\lceil \log |U| \rceil} \right)^k \leq \frac{1}{2^k} \cdot \left(1 + \frac{2}{3\lceil \log |U| \rceil} \right)^{\lceil \log |U| \rceil} \leq \frac{1}{2^k} \cdot e^{2/3} < \frac{2}{2^k} \ .$$

Recalling that $|U| \leq 8(n+m)/\Delta$, we get

$$1 \ < \ 16(n+m)/\Delta \cdot \frac{1}{2^k} \cdot \prod_{\substack{d_i \geq 2 \\ i<j}} \frac{1}{d_i} \ . \tag{2}$$

The part $(v_{j+1}, \ldots, v_\ell)$ of the path lies within a separator tree for an induced component, which has a threshold of exactly $1/2$. By a similar but simpler argument, we get

$$1 \ \leq \ \Delta \cdot \frac{1}{2^{k'}} \cdot \prod_{\substack{d_i \geq 2 \\ i>j}} \frac{1}{d_i} \ , \tag{3}$$

where $k' = |\{ i > j \mid d_i = 1 \}|$.

At v_j, the ordering of the children not in U is arbitrary, and the measure of size in the above argument changes, hence the above argument is not valid. By definition we have the inequality

$$d_j \ \leq \ d \ . \tag{4}$$

Multiplying left sides and right sides in the inequalities (2), (3) and (4), and rearranging the result proves (1). $\square$

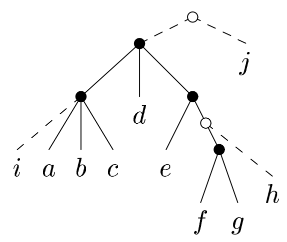

Figure 6: The three possible changes to an evolutionary tree when inserting a new species i, j or h.

3 Algorithm for Constructing and Maintaining Evolutionary Trees

In this section we describe an algorithm for constructing an evolutionary tree T in the experiment model for a set of n species in time $O(nd \log_d n)$, where d is the degree of the tree. Note that d is not known by the algorithm in advance. The algorithm is a further development of an algorithm by Lingas *et al.* in [19]. Our algorithm also supports the insertion of new species with running time $O(md \log_d(n+m))$ using at most $m\lceil d/2\rceil(\log_{2\lceil d/2\rceil-1}(n+m)+O(1))$ experiments for $d>2$, and at most $m(\log(n+m)+O(1))$ experiments for $d=2$, where n is the number of species in the tree to begin with, m is the number of insertions, and d is the maximum degree of the tree during the sequence of insertions.

The construction algorithm inserts one species at the time into the tree in time $O(d \log_d n)$ until all n species have been inserted. Figure 6 shows the three possible changes to an evolutionary tree when inserting a new species: (i) The new species is a leaf below an existing node; (j) the species causes a new root to be created; (h) an existing edge is split by creating a new internal node.

The search for the insertion point of a new species a is guided by a separator tree S_T for the internal nodes of the evolutionary tree T for the species inserted so far. The search starts at the root of S_T. In a manner to be described below, we decide by experiments which subtree, rooted at a child of the root in S_T, the search should continue in. This is repeated recursively until the correct insertion point in T for a is found. We keep links between corresponding nodes in S_T and T for switching between the two trees. To facilitate the experiments, we for each internal node in T maintain a pointer to an arbitrary leaf in its subtree. When inserting a new internal node in T this pointer is set to point to the new leaf which caused the insertion of the node.

We say that the insertion point of a is *incident* to a node v, if

1. a should be inserted directly below v, or

2. a should split an edge which is incident to v by creating a new internal node on the edge and make a a leaf below the new node, or

3. if v is the root of T, a new root of T should be created with a and v as its two children.

The invariant for the search is the following. Assume we have reached node v in the separator tree for the internal nodes in T, and let S_v be the internal nodes of T which are contained in the subtree of S_T rooted at v (including v). Then the insertion point of the new species a is incident to a node in S_v.

Let v be the node in S_T for which we want to decide if the insertion point for the new species a is in the subtree above v in T; if it is in a subtree rooted at a child of v in T; or if a should be inserted as a new child of v. We denote by $u_1, \ldots, u_k$ the children of v in T, where $u_1, \ldots, u_{k'}$ are nodes in distinct subtrees $T_1, \ldots, T_{k'}$ below v in S_T, whereas $u_{k'+1}, \ldots, u_k$ are leaves in T or are nodes above v in S_T. The order of the subtrees $T_1, \ldots, T_{k'}$ below v in S_T is given by the ordered separator tree S_T and determines the order of $u_1, \ldots, u_{k'}$. The remaining children $u_{k'+1}, \ldots, u_k$ of v may appear in any order.

We perform at most $\lceil k/2 \rceil$ experiments at v. The i'th experiment is on the species a, b and c, where b and c are leaves in T below u_{2i-1} and u_{2i} respectively. The leaves b and c can be located using the pointers stored at u_{2i-1} and u_{2i}. Note that the least common ancestor of b and c in T is v. If k is odd then the species b and c in the $\lceil k/2 \rceil$'th experiment is chosen as leaves in T below u_k and u_1 respectively, and note that the two leaves are distinct because $k \geq 2$ by definition. There are four possible outcomes of the i'th experiment corresponding to Figure 1:

1. (a, b, c) implies that the insertion point for a is incident to aădescendent of u_j, where b and c are not descendents of u_j, or a is a new leaf below v.
2. $((a, b), c)$ implies that the insertion point for a is incident to a descendent of u_{2i-1}, since the least common ancestor of a and b is below v in T.
3. $((a, c), b)$ is symmetric to the above case and the insertion point of a is incident to a descendent of u_{2i} (u_1 for the $\lceil k/2 \rceil$'th experiment if k odd).
4. $((b, c), a)$ implies that the insertion point of a is in the subtree above v, since the least common ancestor of a and b is above v. If v is the present root of T, a new root should be created with children a and v.

We perform experiments for increasing i until we get an outcome difference from Case 1, or until we have performed all $\lceil k/2 \rceil$ experiments all with outcome cf. Case 1. In the latter case species a should be inserted directly below v in T as a new child. In the former case, when the outcome of an experiment is different from Case 1, we know in which subtree adjacent to v in T the insertion point for species a is located. If there is no corresponding subtree below v in S_T, then we have identified the edge incident to v in T which the insertion of species a should split. Otherwise we continue recursively searching for the insertion point for species a at the child of v in S_T which roots the separator tree for the subtree adjacent to v which has been identified to contain the insertion point for a. When the insertion point for species a is found, we insert one leaf and at most one internal node into T, and S_T is updated according to Theorem 2.

Lemma 6 *Given an evolutionary tree T for n species with degree d, and a separator tree S_T for T according to Theorem 2, then a new species a can be inserted into T and S_T in amortized time $O(d \log_d n)$ using at most $\lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1} n + O(1))$ experiments for $d > 2$, and at most $\log n + O(1)$ experiments for $d = 2$.*

Proof. Let $v_1, \ldots, v_\ell$ be the nodes in S_T (and T) visited by the algorithm while inserting species a, where v_1 is the root of S_T and v_{j+1} is a child of v_j in S_T. Define d_i by v_{i+1} being the d_i'th child of v_i in S_T, for $1 \leq i < \ell$.

For $d = 2$ we perform exactly one experiment at each v_i. The total number of experiments is thus bounded by the height of the separator tree. By Theorem 2 it follows that the number of experiments is bounded by $\log n + O(1)$. In the following we consider the case where $d \geq 3$.

For $i < \ell$, let x_i denote the number of experiments performed at node v_i. We have $x_i \leq \lceil d/2 \rceil$ and $d_i \geq 2x_i - 1$, since each experiment considers two children of v_i in T and the first experiment also identifies if a should be inserted into the subtree above v_i. At v_ℓ we perform at most $\lceil d/2 \rceil$ experiments.

For $d_1, \ldots, d_{\ell-1}$ we from Theorem 2 have the constraint $\prod_{d_i \leq 2} 2 \cdot \prod_{d_i > 2} d_i \leq 16dn$, since $|S_T| \leq n - 1$. To prove the stated bound on the worst case number of experiments we must maximize $\sum_{i=0}^{\ell} x_i$ under the above constraints. We have

$$
\begin{aligned}
\log(16dn) &\geq \sum_{d_i \leq 2} 1 + \sum_{d_i > 2} \log d_i \\
&\geq \sum_{x_i = 1} 1 + \sum_{x_i > 1} \log d_i \\
&\geq \sum_{x_i = 1} x_i + \sum_{x_i > 1} x_i \frac{1}{x_i} \log(2x_i - 1) \\
&\geq \frac{1}{\lceil d/2 \rceil} \log(2\lceil d/2 \rceil - 1) \sum_{i=1}^{\ell-1} x_i \ ,
\end{aligned}
$$

where the second inequality holds since $x_i > 1$ implies $d_i \geq 3$. The last inequality holds since for $f(x) = \frac{1}{x} \log(2x - 1)$ we have $1 > f(2) > f(3)$ and $f(x)$ is decreasing for $x \geq 3$, i.e. $f(x)$ is minimized when x is maximized.

We conclude that $\sum_{i=1}^{\ell-1} x_i \leq \lceil d/2 \rceil \log_{2\lceil d/2 \rceil - 1}(16dn)$, i.e. for the total number of experiments we have $\sum_{i=1}^{\ell} x_i \leq \lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1}(16dn) + 1)$.

The time needed for the insertion is proportional to the number of experiments performed plus the time to update S_T. By Theorem 2 the total time is thus $O(d \log_d n)$. □

From Lemma 6 and Theorem 2 we get the following bounds for constructing and maintaining an evolutionary tree under the insertion of new species in the experiment model.

Theorem 3 *After $O(n)$ preprocessing time an evolutionary tree T for n species can be maintained under m insertions in time $O(dm \log_d (n + m))$ using at most*

$m\lceil d/2\rceil(\log_{2\lceil d/2\rceil-1}(n+m)+O(1))$ *experiments for* $d>2$*, and at most* $m(\log(n+m)+O(1))$ *experiments for* $d=2$*, where* d *is the maximum degree of the tree during the sequence of insertions.*

4 Adversary for Constructing Evolutionary Trees

To prove a lower bound on the number of experiments required for constructing an evolutionary tree of n species with degree at most d, we describe an adversary strategy for deciding the outcome of experiments. The adversary is required to give consistent answers, i.e. the reported outcome of an experiment is not allowed to contradict the outcome of previously performed experiments. A construction algorithm is able to construct an unambiguous evolutionary tree based on the performed experiments when the adversary is not able to answer any additional experiments in such a way that it contradicts the constructed evolutionary tree. The role of the adversary is to force any construction algorithm to perform provably many experiments in order to construct an unambiguous evolutionary tree.

To implement the adversary strategy for deciding the outcome of experiments in a consistent way, the adversary maintains a rooted infinite d-ary tree, D, where each of the n species are stored at one of the nodes, allowing nodes to store several species. Initially all n species are stored at the root. For each experiment performed, the adversary can move the species downwards by performing a sequence of *moves*, where each move shifts a species from the node it is currently stored at to a child of the node.

By deciding the outcome of experiments, the adversary reveals information about the evolutionary relationships between the species to the construction algorithm performing the experiments. The distribution of the n species on D represents the information revealed by the adversary (together with the forbidden and conflicting lists introduced below). The evolutionary tree T to be established by the construction algorithm will be a connected subset of nodes of D including the root. Initially, when all species are stored at the root, the construction algorithm has no information about the evolutionary relationships. The evolutionary relationships revealed to the construction algorithm by the current distribution of the species on D corresponds to the tree formed by the paths from the root of D to the nodes storing at least one species. More precisely, the correspondence between the final evolutionary tree T and the current distribution of the species on D is that if v is a leaf of T labeled a then species a is stored at some node on the path in D from the root to the node v.

Our objective is to prove that if an algorithm computes T, then the n species on average must have been moved $\Omega(\log_d n)$ levels down by the adversary, and that the number of moves by the adversary is a fraction $O(1/d)$ of the number of experiments performed. These two facts imply the $\Omega(nd\log_d n)$ lower bound on the number of experiments required.

To control its strategy for moving species on D, the adversary maintains for each species a a *forbidden list* $\mathrm{F}(a)$ of nodes and a *conflicting list* $\mathrm{C}(a)$ of species. If a is stored at node v, then $\mathrm{F}(a)$ is a subset of the children $c_1, \ldots, c_d$ of v, and $\mathrm{C}(a)$ is a subset of the other species stored at v. If $c_i \in \mathrm{F}(a)$, then a is not allowed to be moved to child c_i, and if $b \in \mathrm{C}(a)$ then a and b must be moved to two distinct children of v. It will be an invariant that $b \in \mathrm{C}(a)$ if and only if $a \in \mathrm{C}(b)$. Initially all forbidden and conflicting lists are empty. The adversary maintains the forbidden and conflicting lists such that the size of the forbidden and conflicting lists of a species a is bounded by the invariant

$$|\mathrm{F}(a)| + |\mathrm{C}(a)| \leq d - 2 \; . \tag{5}$$

The adversary uses the sum $|\mathrm{F}(a)|+|\mathrm{C}(a)|$ to decide when to move a species a one level down in D. Whenever the invariant (5) becomes violated because $|\mathrm{F}(a)| + |\mathrm{C}(a)| = d-1$, for a species a stored at a node v, the adversary moves a to a child $c_i \notin \mathrm{F}(a)$ of v. Since $|\mathrm{F}(a)| \leq d-1$, such a $c_i \notin \mathrm{F}(a)$ is guaranteed to exist. When moving a from v to c_i, the adversary updates the forbidden and conflicting lists as follows: For all $b \in \mathrm{C}(a)$, a is deleted from $\mathrm{C}(b)$ and c_i is inserted into $\mathrm{F}(b)$. If c_i was already in $\mathrm{F}(b)$, the sum $|\mathrm{F}(b)|+|\mathrm{C}(b)|$ decreases by one, if c_i was not in $\mathrm{F}(b)$ the sum remains unchanged. Finally, $\mathrm{F}(a)$ and $\mathrm{C}(a)$ are assigned the empty set.

For two species a and b, we define their *least common ancestor*, $\mathrm{LCA}(a, b)$, to be the least common ancestor of the two nodes storing a and b in D. We denote $\mathrm{LCA}(a, b)$ as *fixed* if it cannot be changed by future moves of a and b by the adversary. If $\mathrm{LCA}(a, b)$ is fixed then the least common ancestor of the two species a and b in T is the node $\mathrm{LCA}(a, b)$. If a is stored at node v_a and b is stored at node v_b, it follows that $\mathrm{LCA}(a, b)$ is fixed if and only if one of the following four conditions is satisfied.

1. $v_a = \mathrm{LCA}(a, b) = v_b$ and $a \in \mathrm{C}(b)$ (and $b \in \mathrm{C}(a)$).
2. $v_a \neq \mathrm{LCA}(a, b) = v_b$ and $c_i \in \mathrm{F}(b)$, where c_i is the child of v_b such that the subtree rooted at c_i contains v_a.
3. $v_a = \mathrm{LCA}(a, b) \neq v_b$ and $c_i \in \mathrm{F}(a)$, where c_i is the child of v_a such that the subtree rooted at c_i contains v_b.
4. $v_a \neq \mathrm{LCA}(a, b) \neq v_b$.

In Case 1, species a and b are stored at the same node and cannot be moved to the same child because $a \in \mathrm{C}(b)$, i.e. $\mathrm{LCA}(a, b)$ is fixed as the node which currently stores a and b. Cases 2 and 3 are symmetric. In Case 2, species a is stored at a descendant of a child c_i of the node storing b, and b cannot be moved to c_i because $c_i \in \mathrm{F}(b)$, i.e. $\mathrm{LCA}(a, b)$ is fixed as the node which currently stores b. Finally, in Case 4, species a and b are stored at nodes in disjoint subtrees, i.e. $\mathrm{LCA}(a, b)$ is already fixed.

The operation $\mathrm{Fix}(a, b)$ ensures that $\mathrm{LCA}(a, b)$ is fixed as follows:

1. If $v_a = \mathrm{LCA}(a, b) = v_b$ and $a \notin \mathrm{C}(b)$ then insert a into $\mathrm{C}(b)$ and insert b into $\mathrm{C}(a)$.

2. If $v_a \neq \mathrm{LCA}(a,b) = v_b$ and $c_i \notin \mathrm{F}(b)$, where c_i is the child of v_b such that the subtree rooted at c_i contains v_a, then insert c_i into $\mathrm{F}(b)$.

3. If $v_a = \mathrm{LCA}(a,b) \neq v_b$ and $c_i \notin \mathrm{F}(a)$, where c_i is the child of v_a such that the subtree rooted at c_i contains v_b, then insert c_i into $\mathrm{F}(a)$.

Otherwise $\mathrm{Fix}(a,b)$ does nothing. If performing $\mathrm{Fix}(a,b)$ increases $|\mathrm{F}(a)|$ such that $|\mathrm{F}(a)| + |\mathrm{C}(a)| = d-1$, then a is moved one level down as described above. Similarly, if $|\mathrm{F}(b)| + |\mathrm{C}(b)| = d-1$ then b is moved one level down. After performing $\mathrm{Fix}(a,b)$ we thus have that $|\mathrm{F}(a)|+|\mathrm{C}(a)| \leq d-2$ and $|\mathrm{F}(b)|+|\mathrm{C}(b)| \leq d-2$, which ensures that the invariant (5) is not violated.

When the construction algorithm performs an experiment on three species a, b and c, the adversary decides the outcome of the experiment based on the current distribution of the species on D and the content of the conflicting and forbidden lists. To ensure the consistency of future answers, the adversary first fix the least common ancestors of a, b and c by applying the operation Fix three times: $\mathrm{Fix}(a,b)$, $\mathrm{Fix}(a,c)$ and $\mathrm{Fix}(b,c)$. After having fixed $\mathrm{LCA}(a,b)$, $\mathrm{LCA}(a,c)$, and $\mathrm{LCA}(b,c)$, the adversary decides the outcome of the experiment by examining $\mathrm{LCA}(a,b)$, $\mathrm{LCA}(a,c)$, and $\mathrm{LCA}(b,c)$ in D as described below. The four cases correspond to the four possible outcomes of an experiment cf. Figure 1.

1. If $\mathrm{LCA}(a,b) = \mathrm{LCA}(b,c) = \mathrm{LCA}(a,c)$ then return (a,b,c).

2. If $\mathrm{LCA}(a,b) \neq \mathrm{LCA}(b,c) = \mathrm{LCA}(a,c)$ then return $((a,b),c)$.

3. If $\mathrm{LCA}(a,c) \neq \mathrm{LCA}(a,b) = \mathrm{LCA}(b,c)$ then return $((a,c),b)$.

4. If $\mathrm{LCA}(b,c) \neq \mathrm{LCA}(a,b) = \mathrm{LCA}(a,c)$ then return $((b,c),a)$.

5 Lower Bound Analysis

We will argue that the above adversary strategy forces any construction algorithm to perform at least $\Omega(nd \log_d n)$ experiments before being able to conclude unambiguously the evolutionary relationships between the n species.

Theorem 4 *Constructing an evolutionary tree of n species requires $\Omega(nd\log_d n)$ experiments, where d is the degree of the constructed tree.*

Proof. We first observe that an application of $\mathrm{Fix}(a,b)$ at most increases the size of the two conflicting lists, $\mathrm{C}(a)$ and $\mathrm{C}(b)$, by one, or the size of one of the forbidden list, $\mathrm{F}(a)$ or $\mathrm{F}(b)$, by one. If performing $\mathrm{Fix}(a,b)$ increases the sum $|\mathrm{F}(a)| + |\mathrm{C}(a)|$ to $d-1$, then species a is moved one level down in D and $\mathrm{F}(a)$ and $\mathrm{C}(a)$ are emptied, which causes the overall sum of the sizes of forbidden and conflicting lists to decrease by $d-1$. This implies that a total of k Fix operations, starting with the initial configuration where all conflicting and forbidden lists are empty, can cause at most $2k/(d-1)$ moves. Since an experiment involves three Fix operations, we can bound the total number of moves during m experiments by $6m/(d-1)$.

Now consider the configuration, i.e. the distribution of species and the content of conflicting and forbidden lists, when the construction algorithm computing the evolutionary tree terminates. Some species may have nonempty forbidden lists or conflicting lists. By forcing one additional move on each of these species as described in Section 4, we can guarantee that all forbidden and conflicting lists are empty. At most n additional moves must be performed.

Let T' be the tree formed by the paths in D from the root to the nodes storing at least one species. We first argue that all internal nodes of T' have at least two children. If a species has been moved to a child of a node, then the forbidden list or conflicting list of the species was nonempty. If the forbidden list was nonempty, then each of the forbidden subtrees already contained at least one species, and if the conflicting list was nonempty there was at least one species on the same node that was required to be moved to another subtree, at the latest by the n additional moves. It follows that if a species has been moved to a child of a node then at least one species has been moved to another child of the node, implying that T' has no node with only one child.

We next argue that all n species are stored at the leaves of T' and that each leaf of T' stores either one or two species. If there is a non-leaf node in T' that still contains a species, then this species can be moved to at least two children already storing at least one species in the respective subtrees, implying that the adversary can force at least two distinct evolutionary trees which are consistent with the answers returned. This is a contradiction. It follows that all species are stored at leaves of T'. If a leaf of T' stores three or more species, then an experiment on three of these species can generate different evolutionary trees, which again is a contradiction. We conclude that each leaf of T' stores exactly one or two species, and all internal nodes of T' store no species. It follows that T' has at least $n/2$ leaves.

For a tree with k leaves and degree d, the sum of the depths of the leaves is at least $k \log_d k$. Since each leaf of T' stores at most two species, the n species can be partitioned into two disjoint sets of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ such that in each set all species are on distinct leaves of T'. The sum of the depths of all species is thus at least $\lceil n/2 \rceil \log_d \lceil n/2 \rceil + \lfloor n/2 \rfloor \log_d \lfloor n/2 \rfloor \geq n \log_d (n/2)$. Since the depth of a species in D is equal to the number of times the species has been moved one level down in D, and since m experiments generate at most $6m/(d-1)$ moves and we perform at most n additional moves, we get the inequality

$$n \log_d (n/2) \;\leq\; 6m/(d-1) + n \;,$$

from which the lower bound $m \geq (d-1)n(\log_d(n/2) - 1)/6$ follows. □

References

[1] S. Alstrup, J. Holm, K. de Lichtenberg, and M. Thorup. Minimizing diameters of dynamic trees. In *Proc. 24th Int. Colloquium on Automata,*

Languages and Programming (ICALP), volume 1256 of *Lecture Notes in Computer Science*, pages 270–280. Springer-Verlag, 1997.

[2] S. Alstrup, J. Holm, and M. Thorup. Maintaining center and median in dynamic trees. In *Proc. 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1851 of *Lecture Notes in Computer Science*, pages 46–56. Springer-Verlag, 2000.

[3] A. Andersson. Improving partial rebuilding by using simple balance criteria. In *Proc. 1st Workshop on Algorithms and Data Structures (WADS)*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer-Verlag, 1989.

[4] A. Andersson and T. W. Lai. Fast updating of well-balanced trees. In *Proc. 2nd Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer-Verlag, 1990.

[5] A. Borodin, L. J. Guibas, N. A. Lynch, and A. C. Yao. Efficient searching using partial ordering. *Information Processing Letters*, 12:71–75, 1981.

[6] G. S. Brodal, S. Chaudhuri, and J. Radhakrishnan. The randomized complexity of maintaining the minimum. *Nordic Journal of Computing, Selected Papers of the 5th Scandinavian Workshop on Algorithm Theory (SWAT)*, 3(4):337–351, 1996.

[7] R. F. Cohen and R. Tamassia. Dynamic expression trees. *Algorithmica*, 13(3):245–265, 1995.

[8] R. Cole and U. Vishkin. The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time. *Algorithmica*, 3:329–346, 1988.

[9] R. W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[10] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985.

[11] G. N. Frederickson. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. *SIAM J. Comput.*, 26(2):484–538, 1997.

[12] A. J. Goldman. Optimal center location in simple networks. *Transportation Sci.*, 5:212–221, 1971.

[13] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan. Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[14] F. Harary. *Graph Theory.* Addison-Wesley, Mass., 1969.

[15] C. Jordan. Sur les assemblages de lignes. *J. Reine Angew. Math.*, 70:185–190, 1869.

[16] S. K. Kannan, E. L. Lawler, and T. J. Warnow. Determining the evolutionary tree using experiments. *Journal of Algorithms*, 21:26–50, 1996.

[17] M. Y. Kao, A. Lingas, and A. Östlin. Balanced randomized tree splitting with applications to evolutionary tree constructions. In *Proc. 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 1563 of *Lecture Notes in Computer Science*, pages 184–196. Springer-Verlag, 1999.

[18] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison–Wesley, third edition, 1998.

[19] A. Lingas, H. Olsson, and A. Östlin. Efficient merging, construction, and maintenance of evolutionary trees. In *Proc. 26th Int. Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, pages 544–553. Springer-Verlag, 1999.

[20] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *J. ACM*, 30(4):852–865, 1983.

[21] N. Megiddo, A. Tamir, E. Zemel, and R. Chandrasekaran. An $O(n \log^2 n)$ algorithm for the kth longest path in a tree with applications to location problems. *SIAM J. Comput.*, 10(2):328–337, 1981.

[22] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest pair problem. *Algorithmica*, 12(1):18–29, 1994.

[23] C. G. Sibley and J. E. Ahlquist. Phylogeny and classification of birds based on the data of DNA-DNA-hybridization. *Current Ornithology*, 1:245–292, 1983.

[24] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[25] R. E. Tarjan. *Data Structures and Network Algortihms.* Society for Industrial and Applied Matehmatics, Philadelphia, Pennsylvania, 1983.

[26] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.