Constructing Evolutionary Trees

Algorithms and Complexity

Anna Östlin

Department of Computer Science
Lund University

Department of Computer Science
Lund University
Box 118
S–221 00 Lund
Sweden

E-mail: Anna.Ostlin@cs.lth.se

© 2001 by Anna Östlin
Printed in Sweden

ISSN 1650-1276
ISBN 91–628–4772–4
Dissertation 17

Abstract

In this thesis three general problems concerning construction of evolutionary trees are considered. Algorithms for the problems are presented and the complexity of the problems is investigated.

The thesis consists of three corresponding parts. The first part is devoted to the problem of constructing evolutionary trees in the experiment model. Different algorithms for the problem are given, including an optimal algorithm for constructing evolutionary trees and an optimal algorithm for merging two evolutionary trees. Matching lower bounds are also provided.

The second part of the thesis presents results related to the inferred consensus tree problem. The optimization version of the problem is shown to be NP-complete and two heuristic algorithms are presented. Further, the ordered version of the problem is studied.

In the last part of the thesis the complexity of the maximum homeomorphic subtree problem is examined. The problem is shown to be hard to approximate, unless P=NP, even for trees of constant height, whereas a constant approximation ratio is obtained in case of a constant number of trees of constant height.

Contents

1 Introduction **1**
- 1.1 Computational Biology 1
- 1.2 Evolutionary Trees 3
- 1.3 Evolutionary Tree Reconstruction Methods 5
- 1.4 Thesis Outline 8

I Experiment Model **13**

2 Tree Construction Using Randomized Splitting **19**
- 2.1 Balanced Randomized Splitting of an Unknown Tree 20
- 2.2 Building Trees from Experiments 22

3 Merging Trees **27**
- 3.1 Merging Partial Evolutionary Trees 27
- 3.2 The Lower Bound on Merging Trees 32

4 Tight Bound for Constructing Trees **35**
- 4.1 Separator Trees 36
  - 4.1.1 Constructing Separator Trees 37
  - 4.1.2 Maintaining Separator Trees 41
  - 4.1.3 Ordered Separator Trees 46
- 4.2 Algorithm for Constructing and Maintaining Evolutionary Trees 49
- 4.3 Adversary for Constructing Evolutionary Trees 52
- 4.4 Lower Bound Analysis 54

II Local Consensus Trees **57**

5 Maximum Inferred Consensus Tree **65**
- 5.1 MICT is NP-Complete 66

5.2 A Simple Approximation Heuristics for MICT 68
5.3 Heuristic for MILCT 70

6 Ordered Consensus Trees 73
6.1 Preliminaries 74
6.2 A Simple Decremental Interval Union Algorithm 75
6.3 Efficient Construction of an Ordered Consensus Tree 78
6.4 A Cubic-Time Algorithm for the Optimization Problem 80

III Maximum Homeomorphic Subtree Problem 81

7 Complexity of Maximum Homeomorphic Subtree Problem 87
7.1 MHT is Hard to Approximate 88
7.2 Approximations of MHT with $O(1)$ Trees of Height $O(1)$ 89

Bibliography 95

Publications

The thesis is based upon the following publications.

- *On the Complexity of Constructing Evolutionary Trees.* Co-authors Leszek Gąsieniec, Jesper Jansson, and Andrzej Lingas. Journal of Combinatorial Optimization, 3:183–197, 1999 [22]. A preliminary version appeared in Proc. of the 3rd Annual Int. Computing and Combinatorics Conference, pages 134–145, 1997.
(Chapter 5 and Chapter 7)

- *Inferring Ordered Trees from Local Constraints.* Co-authors Leszek Gąsieniec, Jesper Jansson, and Andrzej Lingas. In Proc. of the 4th Australian Theory Symposium, pages 67–76, 1998 [21].
(Chapter 6)

- *Balanced Randomized Tree Splitting with Applications to Evolutionary Tree Constructions.* Co-authors Ming-Yang Kao and Andrzej Lingas. In Proc. of the 16th Annual Symposium on Theoretical Aspects of Computer Science, pages 184–196, 1999 [34].
(Chapter 2)

- *Efficient Merging, Construction, and Maintenance of Evolutionary Trees.* Co-authors Andrzej Lingas and Hans Olsson. To appear in Journal of Algorithms [41]. A preliminary version of the paper appeared in Proc. of the 26th Int. Colloquium on Automata, Languages and Programming, pages 544–553, 1999.
(Chapter 3)

- *The Complexity of Constructing Evolutionary Trees Using Experiments.* Co-authors Gerth Stølting Brodal, Rolf Fagerberg, and Christian N. S. Pedersen. To appear in Proc. of the 28th Int. Colloquium on Automata, Languages and Programming, 2001 [9].
(Chapter 4)

Acknowledgments

I have had a great time these years as a Ph.D. student at the Department of Computer Science at Lund University. I want to thank the department for giving me the opportunity to study here, and I want to thank all the people here for making these years so pleasant.

My advisor, Andrzej Lingas, has been very important to me on the way to the completion of this thesis. He has spent lots of hours discussing things with me, and not once have I felt he did not have time for me. Without your support and commitment I would not have been here now. Thank you, Andrzej!

I am also grateful to Christos Levcoloulos, among other things for introducing me to research in computer science in a way that both taught me a lot and made me interested in continuing my studies.

What would life be without lunch and coffee breaks, and what would breaks be without nice company? I am lucky not to know the answer. Andreas, Anna, Bengt, Björn, Jesper, Joachim, Kurt, Mikael, Per, and Thore, thank you for interesting discussions and for your support.

I thank my co-authors during my time as a Ph.D. student: Gerth Stølting Brodal, Bogdan S. Chlebus, Rolf Fagerberg, Leszek Gąsieniec, Jesper Jansson, Ming-Yang Kao, Hans Olsson, Christian N. S. Pedersen, and John Michael Robson. It has been a pleasure working with you and I have learned a lot.

Special thanks go to some special people in my life. Mum and dad, Rasmus, Joachim, and Åsa, thank you for your support and for being good friends!

Chapter 1

Introduction

The title of this thesis, *Constructing Evolutionary Trees : Algorithms and Complexity*, suggests that this is a thesis concerned both with theoretical computer science issues as well as biological issues. Computational biology is the field of research where computational problems arising in biology are solved using methods from mathematical and computer sciences. The problems studied in this thesis are related to construction of evolutionary trees, and hence the motivation comes from biology. However, this is a computer science thesis and the problems are formulated and treated as computational ones. Since the thesis is concerned mainly with computational aspects of evolutionary tree construction, we start by giving a short introduction to computational biology and evolutionary trees. This is followed by an overview of different methods for constructing evolutionary trees and a summary of the results presented in the thesis.

1.1 Computational Biology

Computational biology is still a young research area. Even though algorithms for combinatorial, data structuring, and string problems, with application in biology, have been around for quite some time, the field did not really expand until the nineties and the development in the last ten years has been extensive. Large investments have been done in bioinformatics, and the genome research is hot. There are many problems to study and many algorithms to develop in connection with this. What is really exciting is that computational problems motivated by molecular biology result in interesting problems in theoretical computer science at the same time as efficient solutions to the problems may result in interesting and even life saving progress in biology and medicine.

To give an idea of what computational biology is all about, we present a

few examples of some basic problems in the field. DNA, RNA, and proteins are chains of nucleotides (A, C, G, T), ribonucleotides (A, C, G, U), and amino acids (20 different), respectively. They are very important in molecular biology and give rise to a variety of problems on strings and sequences. Better and faster methods for sequencing (to determine the sequence, e.g. for a piece of DNA) are developed. The amount of sequence information available is huge and the rate of determining new sequences is constantly increasing. The sequences themselves are not that interesting, thus the information has to be processed in different ways. Due to the large data sets to process there is a great need for efficient algorithms and data structures in the data bases. This is one of the classical areas in computational biology. The study of algorithms and data structures for strings and sequences is also a classical area of research in computer science. Known results in this area have been proved useful in solving biological problems, at the same as biological applications have resulted in new computational problems to study related to strings and sequences. Among the things one wants to do with sequences is to compute how similar they are, find similar parts, and to align them.

Proteins are not only one dimensional sequences, they also fold into a three dimensional structure. The 3D shape of a protein is crucial for its function. To be able to determine the 3D shape given a sequence is therefore very useful. Even more interesting would be to determine what the sequence should look like in order to achieve a given 3D shape. The related computational problem is of combinatorial nature and far from being solved. Even very simplified versions of the problem result in NP-hard problems.

Physical maps show the distance between specific locations in a DNA sequence. At these locations the exact sequence for a small piece of the larger DNA sequence is known. A physical map can, for example, be used to locate the position of short DNA sequence in a larger one, for which the map is known. Efficient construction of physical maps is a challenging computational problem and can be viewed as a graph problem.

Construction of evolutionary trees is also a classical problem in computational biology that give rise to interesting computational problems. We discuss it in more detail in the following sections.

For those interested to know more about computational biology there are several introductory and overview books published recently. To mention a few of them (the ones in my bookshelf): *Introduction to Computational Molecular Biology* by Setubal and Meidanis [47] gives a quite basic introduction, *Introduction to Computational Biology : Maps, sequences and genomes* by Waterman [53] is oriented towards statistics and probability, *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology* by Gusfield [25] has

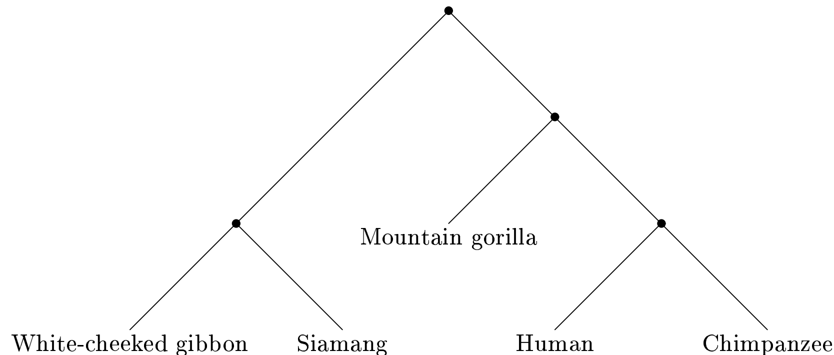

Figure 1.1: Example of a rooted evolutionary tree for five species.

a strong orientation towards string algorithms, and *Computational Molecular Biology : An algorithmic Approach* by Pevzner [45] covers some different areas, not including molecular evolution, which is most interesting for this thesis. The book *Molecular Evolution* by Li [39] gives the biological background needed for understanding molecular evolution, in a way accessible even to non-biologists, and it also includes a chapter on computational methods for tree reconstruction.

1.2 Evolutionary Trees

An evolutionary tree is a tree that models evolution for some set of objects. The objects can be biological entities such as species, proteins, genes, DNA, populations and individuals, but it can also be other things that evolve, like languages. Here we will refer to the objects as species, even though it can be any of the above mentioned types of objects.

Evolutionary trees are often also called phylogenetic trees, or simply phylogenies, when biological evolution is considered. Other names are also frequently used: species trees, when the objects are species, gene trees, when the underlying data comes from just one single gene from each species, etc.

Evolutionary trees are important in genetics and molecular biology, and have been studied since Darwin's days. All living organisms are related to each other by common ancestors and they all undergo a slow process of transformation, called evolution. Evolutionary trees describe this process back in time and show the relationship between present day species in terms of descent and ancestry. The true evolutionary tree that describes what actually happened in the past

is unique. An inferred evolutionary tree is a tree computed by using a certain method and certain data when reconstructing the evolutionary tree. By an evolutionary tree we shall mean an inferred evolutionary tree, unless stated otherwise.

Let S be a set of n species. An evolutionary tree for S is a tree T with n leaves, where each leaf is uniquely labeled by a species in S. The tree can be rooted or unrooted depending on if enough information to decide the orientation of the evolution is available. Internal nodes are common ancestors to the species at the leaves. If the tree is rooted then the root is a common ancestor for all species in the tree. An evolutionary tree is unordered, i.e. for rooted trees we do not distinguish between different orderings of the children of a node and for unrooted trees we do not distinguish between different orderings of the neighbors. The tree can be weighted or unweighted depending on if the edges in the tree have weights or not. For weighted trees the weight of an edge can be interpreted as proportional to the evolutionary time past along that edge, and the distance in the tree between two species can be seen as a measure of how similar two species are, the distance is smaller if they are more closely related.

When evolutionary trees are studied two different aspects are interesting. First, the topology of the tree, that is how the nodes in the tree are connected to each other and second, the distances in the tree, i.e. an estimation on how long it took for the species to evolve. In this thesis we are interested in the first aspect, to reconstruct the topology of evolutionary trees.

Two evolutionary trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$ for the same set of species are *(topologically) equal*, or *isomorphic*, if there exists a bijective mapping, $\phi : V_1 \to V_2$, of the vertices in the trees, where leaves labeled by the same species are pairwise mapped to each other, such that if $v, w \in V_1$, then there exists an edge $(v, w) \in E_1$ if and only if there exists an edge $(\phi(v), \phi(w)) \in E_2$.

Given two evolutionary trees $T_1 = (V_1, E_1)$ for a set S_1 of species and $T_2 = (V_2, E_2)$ for a set S_2 of species, where $S_1 \subseteq S_2$, T_1 is a *topological subtree* of T_2, if there exists a injective mapping $\phi : V_1 \to V_2$, where leaves labeled by the same species are pairwise mapped to each other, such that for each pair of edges in T_1, $(v_1, v_2), (w_1, w_2) \in E_1$, the two paths from $\phi(v_1)$ to $\phi(v_2)$ and from $\phi(w_1)$ to $\phi(w_2)$ in T_2 are edge disjoint. In the thesis we will also use the term *homeomorphic subtree* to mean the same thing as topological subtree.

An evolutionary tree T, leaf labeled by a set S of species, *restricted to* a set $S' \subseteq S$ is the tree $T' = (V, E)$, where V consists of all leaves in T labeled by a species in S' and all internal nodes v in T for which there exists a pair of leaves labeled by $a, b \in S'$ such that v is the lowest common ancestor of a and b in T, and E consists of those edges (u, v) for which there is a path from u to v in T, such that none of the vertices on the path, except u and v, are in V.

The *degree* of a rooted tree is the maximum number of children for any internal node in the tree. The degree of a unrooted tree is the maximum number of neighbors for any node in the tree. The true evolutionary tree is believed to be binary, but inferred trees may have large degrees.

1.3 Evolutionary Tree Reconstruction Methods

There are several methods for tree reconstruction, due to both biological and computational reasons. In the biological society there are different views and beliefs of how the classification should be done. Gusfield express this as follows [25]: *Although the technical side of tree building may appear to be a matter of pure graph theory and combinatorial optimization, the fundamental issue that determine the validity of these methods are sometimes discussed in terms more suited for religion.* This, of course, makes it hard as a computer scientist to decide which problems are most important to solve and the result is that several methods are investigated at the same time.

The most common algorithms used to build evolutionary trees can be classified into the following categories:

- Distance based methods
- Maximum parsimony methods
- Maximum likelihood methods,

and at a higher level

- Consensus methods

Distance based methods. In distance based methods distances between each pair of species are first computed, e.g. by using sequences where the distances depend on the number of substitutions or edits necessary to get one sequence from another one. A tree is then computed using the resulting distance matrix. In a tree that exactly fits the data, the distances in the edge weighted tree between each pair of species equal the corresponding distance in the matrix. A tree (or a distance matrix) is called additive in this case. If it also is possible to place a root in the tree such that all distances from the root to the leaves are equal then the tree is called ultrametric.

A tree can be computed in polynomial time from an additive or ultrametric distance matrix and the tree is unique. If the distance data for pairs of species is proportional to the time since they evolved into two different species then the matrix is ultrametric. However, there are several reasons why data (almost)

never is ultrametric or even additive. Two of them are that the data comes from experiments and experiments are not perfect and that the molecular clock assumption does not hold, at least not always, the rates of evolution along different branches in the tree are not the same.

Since the data hardly ever is additive or ultrametric, optimization versions of the problem of constructing evolutionary trees from distance matrices are considered. These most often give rise to NP-hard problems. Some of the optimization criteria used are to minimize $\sum_{i,j} |d(i,j) - M[i,j]|^k$, for different values of k, where $d(i,j)$ is the distance between species i and j in the tree and $M[i,j]$ is the corresponding value in the matrix, or to minimize the maximum of $|d(i,j) - M[i,j]|$ over all pairs of species.

Maximum parsimony methods. Maximum parsimony methods are character based, which means that the data from which the tree should be constructed are discrete characters. For example the characters can come from aligned molecular sequences, such as DNA. In this case the characters are A, C, G, T, one at each position in the aligned sequences. The characters can also be properties for the species, like presence or absence of certain restriction sites or morphological features. In any case each character is an element from a finite set of possible discrete values, called states. The input is a $n \times m$ matrix, where n is the number of species and m is the number of characters (e.g. positions in the aligned sequences).

The idea behind maximum parsimony methods is that the tree that requires the minimum total number of changes, from one state to another state, for a character during the evolution, is the tree wanted. One can think of it as if the internal nodes in the tree also were assigned characters, i.e. a row in the matrix. For each position we want as few changes as possible, from one state to another, for adjacent vertices in the tree. In the perfect phylogeny problem the question is whether a character matrix has a perfect tree, T, in the sense that for each character, c, and each state, s, all nodes (including internal nodes) with character c in state s form one connected subtree of T. In other words, is there a labeling of the internal nodes so that this requirement is fulfilled?

Maximum likelihood methods. Maximum likelihood methods are similar to maximum parsimony methods in the sense that both methods start with aligned sequences and search for the most likely tree, among all possible topologies, by imagining a labeling of the internal nodes in the tree. A difference is that in maximum likelihood methods a probabilistic model for the evolution is used to compute how likely a certain tree is. The probabilistic model is a model describing the probability that a certain nucleotide (in case aligned DNA se-

quences are used), during some time interval, changes into another nucleotide. Given a fixed tree topology, for a set of species, and a fixed labeling of all nodes in the tree for a specific position in the sequence, a probability for the tree can be computed. The likelihood function for a fixed tree topology, for a set of species, is the probability for the tree over all possible labelings of the internal nodes. The most likely tree topology is the topology for which the product of the likelihood function, for all positions in the sequence, is maximized, and this is the tree we are searching for.

Consensus methods. In consensus methods the molecular data is not used directly, as in maximum parsimony or maximum likelihood methods. Instead the input is in the form of a set of trees, leaf labeled by species, and possibly also a weight for each tree telling us how confident we are in the evolutionary relationships described by the trees. One can therefore say that consensus methods operate on a higher level, because they require some kind of preprocessing producing trees.

Consensus methods can both be seen as a way to become more confident in the evolutionary relationships described by the tree, since information in different trees are used when deciding the tree, and as a way to combine information from partly disjoint trees. Two methods that increase the confidence in the tree is to compute a maximum homeomorphic subtree or a strict consensus tree. A homeomorphic subtree for a set of leaf labeled trees, all on the same set of species, is a tree for a subset of the species such that the evolutionary relationships described by the tree also can be found in all of the input trees. In the maximum homeomorphic subtree problem we want to find a homeomorphic subtree for a subset of the species of maximum cardinality. A strict consensus tree for a set of leaf labeled trees only contains information common to all input trees, where information means an edge dividing the species into two subsets, i.e. a strict consensus tree may have few edges if the trees are not very similar.

To compute trees by using maximum likelihood methods is very time consuming, but on the other hand the quality of the trees produced is often high. Maximum likelihood trees for small sets of species, for example for four species, can be computed fast, but we are most often interested in trees for more than four species. Consensus methods can be used to combine the trees for the small subsets of species into a tree for all species we want a tree for. The consensus tree problem, or local consensus tree problem if all input trees have constant size, asks for a tree, T, such that each input tree is a topological subtree in T.

1.4 Thesis Outline

The thesis consists of three parts. Each part considers one general problem, all of them fall in the consensus method category. The parts begin with an introduction including problem motivation, its definition, overview of previous results related to the problem, and a description of the new results obtained in the thesis. The three problems and the corresponding new results presented in the thesis are briefly described below.

Experiment model. In Part I the problem of constructing evolutionary trees in the experiment model is considered. For three species a, b, and c, an experiment is defined as the topological subtree for those three species in the evolutionary tree. The experiment outcome denoted by $((a,b),c)$ means that the lowest common ancestor of a and b is below both the lowest common ancestor of a and c and the lowest common ancestor of b and c. When the lowest common ancestors of all pairs of species in the experiment are the same the outcome is denoted (a,b,c). There are four possible outcomes of an experiment, see Figure 1.2.

The problem of constructing an evolutionary tree, for a set S of species, in the experiment model is defined as constructing a tree T, where leaves are labeled by the species in S, and the topology of T is consistent with all experiments for triplets of species in S. The standard computational model is augmented by an oracle which, for any three species, returns the outcome of an experiment for those three species within one step.

The main result in this part is a tight bound for the number of experiments necessary to construct trees in the experiment model. An algorithm for the problem running in time $O(nd \log_d n)$ and using $n\lceil d/2\rceil(\log_{d-1} n + O(1))$ experiments for $d > 2$, and at most $n(\log n + O(1))$ experiments for $d = 2$, where d is the degree of the tree, is presented. A lower bound of $\Omega(nd \log_d n)$ on the number of experiments needed is also shown.

Two other algorithms for the problem are also presented. First a randomized

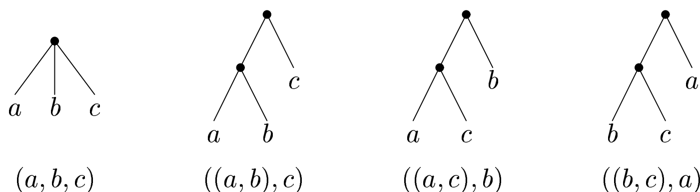

Figure 1.2: The four possible outcomes of an experiment for three species a, b and c.

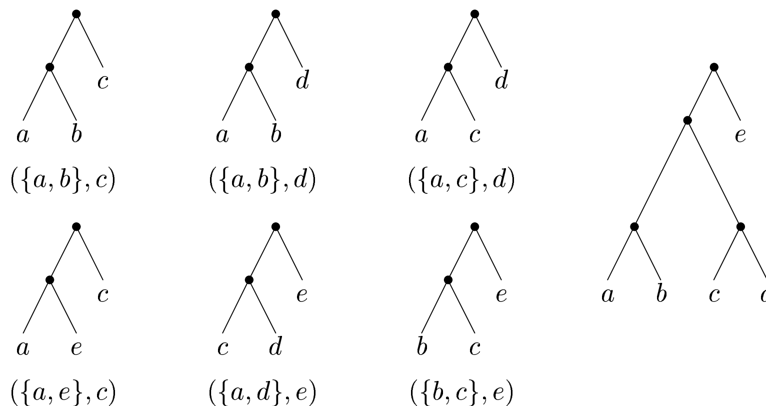

Figure 1.3: The consensus tree (right) is consistent with four out of six constraints (left).

algorithm for constructing trees in expected time $O(nd \log n \log \log n)$ is given. Even though this algorithm is less efficient than the aforementioned one, the technique used by the algorithm is of general interest.

The third algorithm is an algorithm for merging two leaf disjoint trees. Two trees can be merged in the experiment model in time $O(nd)$ using as many experiments. A matching lower bound is also shown.

Local consensus trees. In the consensus tree problem the input is in the form of a set of trees, each leaf labeled by a subset of the species in a set S, and the objective is to construct, if possible, a tree T, for the species in S, such that each input tree is a topological subtree in T. In Part II we look at some problems related to the consensus tree problem.

In the inferred consensus tree problem instead of trees the input is in the form of constraints on the lowest common ancestors for species in a set S. A constraint, denoted $\{i, j\} < \{k, l\}$, where $i, j, k, l \in S$, means that the lowest common ancestor for i and j is a proper descendant of the lowest common ancestor of k and l. Given a set of constraints on the lowest common ancestors, the inferred consensus tree problem asks for a tree consistent with all input constraints. We study the optimization version of this problem, called maximum inferred consensus tree problem, where a tree, for which the number of constraints consistent with the tree is maximized, is sought. Also a special case of the problem, called maximum inferred local consensus tree problem, is considered, where all constraints includes only three species, i.e. $\{i, j\} < \{i, k\}$, also denoted $(\{i, j\}, k)$ (cf. experiment outcomes). See Figure 1.3 for an example.

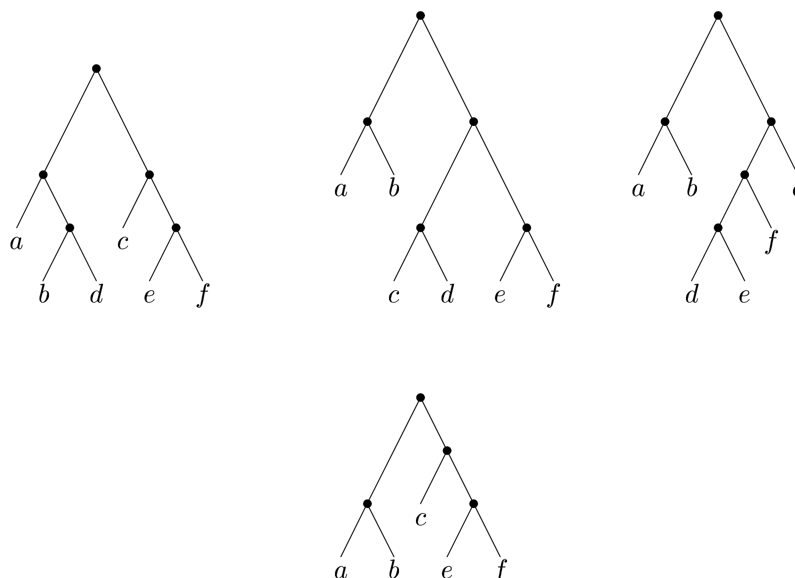

Figure 1.4: A maximum homeomorphic subtree (bottom) for three trees (top).

We show that the maximum inferred consensus tree problem is NP-complete and give a heuristic yielding solutions within one third of the optimum. For the maximum inferred local consensus tree problem a second heuristic is also presented. Both heuristics work also in case of weighted constraints.

Another special case of the inferred consensus tree problem is studied in Chapter 6, where a frontier of the leaves, i.e. a left-to-right ordering of the leaves in the tree, is given. For the ordered version of the inferred local consensus tree problem we give an $O((m+n)\log n)$-time algorithm, where n is the number of species and m is the number of constraints. The ordered version of the maximum inferred local consensus tree problem can, in contrast to the unordered version, be solved in polynomial time. We show this by giving an $O(n^3)$ time algorithm.

Maximum homeomorphic subtree. Given a set of rooted trees, $T_1, \ldots, T_k$, each leaf labeled by a set S, the homeomorphic subtree problem asks for a set $S' \subseteq S$, such that for $i = 1, \ldots, k$, the restrictions of the T_i's to S' are all equal. When the cardinality of S' is maximized the tree T_i restricted to S' is called a maximum homeomorphic subtree. The problem is often also called the maximum agreement subtree problem. See Figure 1.4 for an example.

Two results for the maximum homeomorphic subtree problem are presented in Part III. First we show that the problem is hard to approximate. Unless

P=NP, the maximum homeomorphic subtree problem can not be approximated within a factor N^ϵ, for any $0 \leq \epsilon < 1/9$, where N denotes the total size of all input trees. The second result shows that for a constant number of trees of constant height the maximum homeomorphic subtree problem can be approximated within a constant factor, and we give an algorithm achieving this approximation factor running in time $O(n \log n)$, where n is the number of species in S.

Part I

Experiment Model

Experiment model

Consider the problem of constructing an unknown tree using some partial information on the tree topology available at some cost. More precisely, the partial information is in the form of the topological subtree induced by a subset of the leaves and the cost corresponds to the time it takes to construct the subtree in a given model. This is the problem considered in this part of the thesis. In particular we are going to look at the problem of constructing evolutionary trees in the experiment model.

An *evolutionary tree* for a set S of species is a rooted tree with no unary nodes and n leaves labeled uniquely by the species in S. An *experiment* for three species a, b, and c is defined as the topological subtree in the evolutionary tree T for those three species. We denote by $((a,b),c)$ the experiment outcome where the lowest common ancestor of a and b is below both the lowest common ancestor of a and c and the lowest common ancestor of b and c. When the lowest common ancestors of all pairs of species in the experiment are the same we denote the outcome (a,b,c). The possible outcomes of an experiment are shown in Figure 1.5.

The problem of constructing an evolutionary tree, for a set S of species, in the *experiment model* is defined as constructing a tree T where all internal nodes have at least two children, and leaves are labeled by the species in S. Furthermore, the topology of T should be consistent with all experiments for triplets of species in S. The standard computational model is augmented by an oracle which, for any three species, returns the outcome of an experiment for those three species within one step. Figure 1.6 shows an example of a tree and the outcome of experiments. This model was suggested by Kannan, Lawler, and Warnow [31]. Let n denote the number of species in the tree, i.e. $|S| = n$, and define the degree d of T as the maximum degree of the nodes in T, where the degree of a node is the number of children.

The motivation for studying this model is twofold. First, biological experimental techniques, called DNA-DNA-hybridization, that yield a rooted tree for

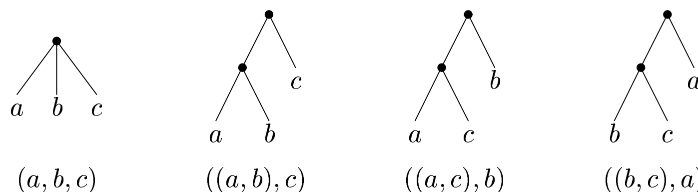

Figure 1.5: The four possible outcomes of an experiment for three species a, b and c.

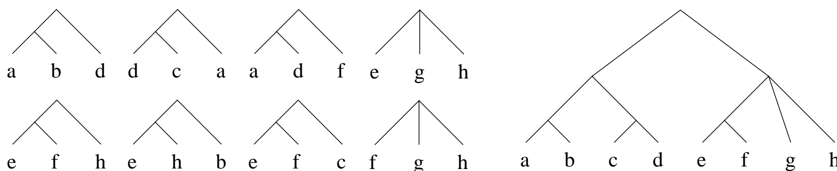

Figure 1.6: An example of experiment results and the corresponding evolutionary tree.

three species have been studied by Sibley and Ahlquist [48]. In these experiments one measures the temperature at which single stranded DNA from two different species bind together, the two species for which the DNA strands bind together at the highest temperature are likely to be close to each other in the tree.

The second motivation comes from the concept of noisy-ultrametric distance matrices. A distance matrix M for a set S of species is an $n \times n$ matrix where entry M_{ij} represents the evolutionary distance between species i and j. A distance matrix is additive if there exists an evolutionary tree for S such that we can assign weights to the edges in such a way that the distances between every pair of species in the tree equals the corresponding entries in the matrix. A matrix is ultrametric if it is additive and the tree can be rooted in such a way that the distances from the root to all species (leaves) are the same. Distances can for example be computed from alignments of molecular sequences, i.e. it is a measure of sequence similarity. The distance matrices we get are seldom ultrametric, but they may be almost ultrametric. The concept of noisy-ultrametric distance matrices, defined by Kannan *et al.* [31], is based on this. A matrix M is noisy ultrametric if there exists a rooted evolutionary tree such that for all triplets of species $a, b, c \in S$ it holds that $M_{ab} < \min\{M_{ac}, M_{bc}\}$ if and only if the lowest common ancestor of a and b is below the lowest common ancestor of a and c (which is also the lowest common ancestor of b and c) in the tree. An experiment can thus be seen as looking at three entries in a noisy-ultrametric distance matrix.

In the literature, another variant of the experiment model based on quartets of species is often considered, cf. [10]. In this variant, the input to the experiment consists of four different species, say a, b, c, and d, and the output is an unrooted tree on four leaves labeled with a, b, c, and d respectively, revealing the topology of the subtree of the evolutionary tree induced by the quartet. Note that if we fix some species as the root of the evolutionary tree then all the experiments on quartets including this particular species are equivalent to triple experiments for the other three species. By this simple observation, all upper bounds established

in the triple variant of the experiment model immediately carry over to the quartet variant. As for lower bounds, note that the output of an experiment on a, b, c, and d returning a rooted tree on four leaves labeled with a, b, c, d can easily be deduced from the output of $O(1)$ experiments on triples in $\{a,b,c,d\}$. Since such a rooted tree on a, b, c, d immediately yields the topology of an unrooted one on a, b, c, d, also the lower bounds valid in the triple variant of the experiment model carry over up to constants to the (unrooted) quartet variant.

Interestingly, the problem of sorting n distinct numbers in the comparison-based model can be modeled as a special case of the problem of constructing an evolutionary tree from experiments as follows. The binary tree to construct is like a caterpillar. It is composed of a single rooted path on n nodes and a set of n leaves, each pending from a unique node of the path and labeled uniquely by one of the input numbers, so the labels of the leaves along the path form, say, an decreasing sequence. The outcome of the experiment for a triple gives the maximum of the numbers in the triple. Thus, it can be obtained by two comparisons.

The problem of constructing evolutionary trees in the experiment model was first studied by Kannan, Lawler, and Warnow [31]. In their paper, they presented different algorithms for this problem. For binary trees they gave three algorithms, the fastest with running time $O(n \log n)$ using $4n \log n$ experiments (all logarithms are base two unless stated otherwise). Experiments are expensive and hence it is important to minimize their number. The other two algorithms use less experiments, they have running time $O(n^2)$ and $O(n \log^2 n)$ respectively, and use $n \log n$ and $n \log_{3/2} n$ experiments respectively. For trees of degree d an $O(n^2)$ time algorithm using $O(dn \log n)$ experiments was presented, and for the general case an algorithm using $O(n^2)$ experiments and $O(n^2)$ time together with a matching lower bound were given.

In this part three different algorithms for constructing trees in the experiment model are presented. They show the development of more efficient algorithms for the problem, ending with the asymptotically optimal algorithm in Chapter 4. Even though the algorithms presented in Chapter 2 and 3 are less efficient they contain interesting results and techniques.

By using our technique of balanced randomized tree splitting, presented in Chapter 2, an evolutionary tree for n species can be determined, using experiments, in expected time $O(nd \log n \log\log n)$, where d is the degree of the tree.

Given a tree T and a random sample of species in T, then removing the edges connecting the species in the sample will split the tree into disconnected components for the species not in the sample. We show that none of these components are large, with high probability. This fact can be used to achieve

the above mentioned result. The idea is that to decide which component the species belong to can be done faster then constructing the tree. The trees for the small components can then be computed fast. The result on randomized splitting is a general technique and could be applied to other problems involving tree construction.

The main result in Chapter 3 is an algorithm for merging two leaf-disjoint evolutionary trees, using additional experiments, running in time $O(dn)$, where n is the number of leaves in the resulting tree and d is its degree. Our merging algorithm yields a deterministic algorithm for constructing an evolutionary tree for n species from experiments running in time $O(dn \log n)$. A matching lower bound on the number of experiments needed for merging two trees is also shown.

In Chapter 4 we describe an algorithm which constructs an evolutionary tree of n species in time $O(nd \log_d n)$ using at most $n\lceil d/2 \rceil(\log_{d-1} n + O(1))$ 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, shown in Chapter 3 by a factor $\Theta(\log d)$. In this chapter we also show, by an explicit adversary argument, a matching lower bound, improving the previous best lower bound by a factor $\log_d n$.

Chapter 2

Tree Construction Using Randomized Splitting

Several of the known efficient algorithms for trees rely on their excellent separator properties. It is well known that each tree contains a vertex whose removal splits the tree into components of balanced size. Finding such a vertex usually requires the knowledge of the tree. In this paper, we consider a more general situation when the tree is unknown and we can obtain some partial information on its topology at some cost. More precisely, the partial information is in the form of the topological subtree induced by a subset of the leaves and the cost corresponds to the time taken by the construction of the subtree in a given model. We introduce an efficient randomized technique of balanced splitting of an unknown tree termed *balanced randomized tree splitting*. It can be used to construct an unknown tree recursively. To demonstrate the usefulness of our technique, we show an examples of its application to construction of evolutionary trees.

By using our new technique of balanced randomized tree splitting, we show that an evolutionary tree for n species can be determined, using experiments, in expected time $O(nd \log n \log\log n)$, where d is the maximum degree in the tree.

The technique of balanced randomized tree splitting presented in this chapter should be useful in finding efficient algorithms for several other problems involving construction of unknown trees (both within computational biology as well as outside it).

Section 2.1 presents some general results on balanced randomized tree splitting and in Section 2.2 the algorithm for constructing a tree from experiments is shown.

2.1 Balanced Randomized Splitting of an Unknown Tree

To find a balanced splitting of the unknown tree T, we randomly pick a sample of its leaves and build the topological subtree T' induced by the sample. The removal of the vertices of T' from T splits T into components of balanced number of leaves with high probability.

Given a tree T and a subset S of leaves of T, the *topological subtree* T' of T induced by S, denoted by $T \parallel S$, is the minimum size tree that has S as the set of its leaves and is homeomorphic to the subtree of T composed of paths in T joining each pair of leaves in S.

For an edge (u, w) of T, let $T_{u,w}$ denote the subtree of T rooted at u and composed of all nodes reachable from w via u.

For a pair of vertices w_1, w_2 of T, let $T(w_1, w_2)$ denote the forest composed of all distinct subtrees $T_{u,w}$ where w is an internal vertex on the path joining w_1 with w_2 in T and u is a vertex of T adjacent to w and outside this path.

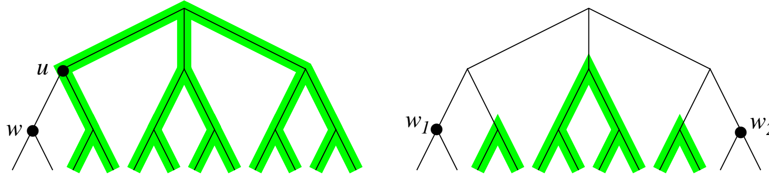

Figure 2.1: Example of $T_{u,w}$ and $T(w_1, w_2)$. Left: The subtree $T_{u,w}$ is shaded. Right: The three subtrees of the forest $T(w_1, w_2)$ are shaded.

A *component* of $T - T'$ is either (1) a subtree $T_{u,w}$ where $w \in V(T')$ and u does not belong to any path in T joining w with an adjacent vertex in T' or (2) a forest $T(w_1, w_2)$ where (w_1, w_2) is an edge of T'.

Theorem 2.1 *Let T be a tree on n leaves. Let $2 \leq n_0 < n$ and $k \geq 2$. For any sample S of at least $2k\frac{n}{n_0}\log n$ leaves chosen uniformly at random among the n leaves of T, and the topological subtree T' of T induced by S, each of the components of $T - T'$ contains less than n_0 leaves with probability at least $1 - n^{-k}$.*

Proof: Note that T has at least three edges, since $n > 2$. For a pair of vertices w_1, w_2 of T, suppose that $T(w_1, w_2)$ has at least n_0 leaves of T. The probability that none of the leaves is chosen in S is not greater than $(1 - \frac{n_0}{n})^{2k\frac{n}{n_0}\log n}$, i.e. it is not greater than $\frac{1}{n^{2k}}$. Since T has at most n vertices, there are less than

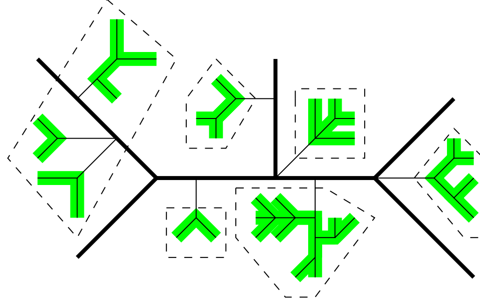

Figure 2.2: Example of the components in $T - T'$. Thick lines correspond to the tree T'. The content of the components are shaded. There are six different components, marked with dashed lines, two of type $T_{u,w}$ and four of type $T(w_1, w_2)$.

n^2 forests $T(w_1, w_2)$. Consequently, the probability that there is such a forest $T(w_1, w_2)$ with at least n_0 leaves of T none of which is chosen to S is less than $\frac{1}{n^{2k-2}}$ which is less than $\frac{1}{n^k}$, since $k \geq 2$. Each of the components of $T - T'$ is included in a forest of the form $T(w_1, w_2)$, therefore we conclude that each of them contains less than n_0 leaves with probability greater than $1 - n^{-k}$. □

Note that Theorem 2.1 implies the existence of a single vertex in T' whose removal from T partitions the leaves of T into balanced size components with high probability.

Corollary 2.2 *Let T be a tree on n leaves. Let $2 \leq n_0 \leq n/2$ and $k \geq 2$. For any sample S of at least $2k\frac{n}{n_0}\log n$ leaves chosen uniformly at random among the n leaves of T, the topological subtree T' of T induced by S contains a vertex of T whose removal disconnects T into subtrees none of which contains more than $n/2 + n_0$ leaves with probability at least $1 - n^{-k}$. Given T' and the size of the components of $T - T'$, such a vertex can be found in time $O((n/n_0)\log n)$.*

Proof: Transform T' to an auxiliary tree T^*, by breaking each edge $e = (w_1, w_2)$ into two edges (w_1, w_e), (w_e, w_2), where w_e is a new vertex uniquely associated with e, and adding for each component C of $T - T'$ a unique leaf l_C. If C is of the form $T_{u,w}$ then l_C is adjacent to the vertex w in T'. Otherwise, if C is of the form $T(w_1, w_2)$ then l_C is adjacent to w_e where $e = (w_1, w_2)$. For each leaf l_C of T^*, set its weight to the number of leaves of T in C. For each of the remaining vertices w of T^*, set the weight w to zero.

To prove the thesis, it is sufficient to find a vertex of T' whose removal disconnects T^* into subtrees none of which contains vertices of total weight greater than $n/2 + n_0$.

By Theorem 2.1, each vertex of T^* has weight not exceeding n_0, which does not exceed $n/2$ by our assumptions, with probability at least $1 - n^{-k}$.

Root T^* at a vertex of T'. If for each child of the root the total weight of descendant vertices is not greater than $n/2$ then we are done. Otherwise, walk down the tree always in direction of the child for which the total weight of descendant vertices is larger than $n/2$. If there is no such child, stop. Note that since each leaf of T^* has weight not larger than $n/2$, the above procedure stops at an internal vertex v of T^*. Also, by the definition of v, its removal disconnects T^* into subtrees each containing vertices of total weight not greater than $n/2$. If v is a vertex of T', we are done. Otherwise, we choose the parent w of v in the rooted T^*. Since v in this case has only two children, one of which is a leaf of weight less than n_0, it follows that w satisfies the thesis in this case.

Given T' and the size of the components of $T - T'$, the above procedure can be easily implemented in time linear in the size of T'. □

Corollary 2.2 is mostly interesting in the situation when the number of leaves in the components of T induced by the removal of the vertices in T' can be determined faster than it takes to build the whole T.

2.2 Building Trees from Experiments

We will use the result from Section 2.1 to derive a new upper time-bound on the construction of an evolutionary tree from experiments, depending on the maximum degree of the constructed tree. The following procedure will yield our upper bound.

Algorithm BuildTree(L)

Input: A set L of species for which experiments can be made.

Output: An evolutionary tree for L.

1. Pick a random sample S of species from L of size $8\lceil \log |L| \rceil$;
2. Build an evolutionary tree T' for S by using the quadratic-time algorithm from [31];
3. Determine the components $C_1, \ldots, C_j$ of $T - T'$ where T is the evolutionary tree to construct;
4. Augment each of the components of $T - T'$ of the form $T(x, y)$ by a single species labeling a leaf of T' lying below both x and y in T';

5. Augment each of the components of $T - T'$ of the form $T_{x,y}$, where y is the root of T', by a single species labeling a leaf of T';

6. **For** each component C_i (possibly augmented) **do**

   if C_i contains at most $\log n$ species then construct the evolutionary tree for C_i by using the quadratic-time algorithm

   else BuildTree(C_i)

7. Connect the evolutionary trees recursively computed for the components C_i with T' as follows:

   if C_i is of the form $T_{x,y}$ and y is not the root of T' **then** link the root of the evolutionary tree for C_i to y;

   if C_i is of the form $T_{x,y}$ and y is the root of T' **then** identify the root of T' with the father of the leaf labeled by the additional species in C_i;

   if C_i has been originally of the form $T(x,y)$ **then** identify the root of the evolutionary tree for the augmented C_i with the lowest common ancestor of x, y, and identify its leaf labeled by the additional species with the other vertex in $\{x, y\}$.

The next fact is useful for analyzing the time complexity of BuildTree(L).

Fact 2.3 *[see [31]] Let v be an internal node of the evolutionary tree for a set U of species. Let $W \subseteq U$, and let $W_1, \ldots, W_q$ be the splitting of W into non-empty components induced by v. For any $u \in U - W$, we can determine whether u belongs to any of the components of U induced by v that is a superset of one of the components $W_1, \ldots, W_q$, and if so, also the index of the component, in time $O(q)$ by performing $\lceil \frac{q}{2} \rceil$ experiments.*

Theorem 2.4 *An evolutionary tree for n species can be determined in expected $O(nd \log n \log\log n)$ time, where d is the maximum degree in the tree.*

Proof: Let L be the set of n species. We use BuildTree(L) to prove the theorem. The correctness of this procedure follows from the correctness of combining the evolutionary trees recursively computed for the augmented components in step 6.

As for the expected running time of BuildTree(L), it is easily seen to be dominated by steps 2, 3 and the "if" part of step 6.

To estimate the total work (including recursive calls) required in step 2 and the "if" part of step 6 note that each internal node of T can appear in at most one of the evolutionary trees T' built for the samples. Let $n_1, \ldots, n_q$ be the sizes of the samples drawn by the algorithm. We have $\sum_{i=1}^{q} n_i \leq n$, and $n_i \leq 8\lceil \log n \rceil$

for $i = 1, \ldots, q$. Hence, the total work for constructing the evolutionary trees T' for the samples (step 2) does not exceed

$$O\left(\max_{\sum_i n_i \le n, n_i \le 8\lceil \log n \rceil} (\sum n_i^2)\right) = O\left(\frac{n}{\log n} \log^2 n\right) = O\left(n \log n\right)$$

Analogously, the total time taken by the construction of the evolutionary trees for the components of size not greater than $\log n$ (the "if" part of step 6) is $O((n/\log n)\log^2 n) = O(n \log n)$.

In order to derive our upper bound on the expected time required by step 3, we consider the following method of determining the components of $T - T'$.

First, we find a (separator) vertex v of T' whose removal disconnects T' into subtrees T_i', $i = 1, \ldots, q$, none of which has more than $\frac{2}{3}$ of the vertices of T'. Now, we can determine the leaf sets of the subtrees of the form $T_{u,v}$, where (u, v) is an edge of T, by performing at most $\lceil d/2 \rceil n$ experiments in time $O(dn)$ by Fact 2.3. In this way, in particular we can determine the components of $T - T'$ of the form $T_{u,v}$. Let u_1 through u_l be the neighbors of v in T', and L_1 through L_l be the leaf sets assigned to the branches (v, u_1) through (v, u_l) by the aforementioned experiments. Note that for $i = 1, \ldots, l$, L_i is just the leaf set of a subtree $T_{u,v}$ where u is the second vertex on the path from v to u_i in T. Let T_i' be the subtree of T' corresponding to the branch (v, u_i), i.e. the maximal subtree of T' including u_i and excluding v. For $i = 1, \ldots, l$, we analogously find a vertex separator v_i of T_i' and perform at most $\lceil d/2 \rceil |L_i|$ experiments in order to split L_i into the subsets of leaf sets of the subtrees of the form T_{u,v_i}. Importantly, note that if $u = v$ then the subset of L_i assigned to the branch (v_i, v) yields the component $T(v, v_i)$. By proceeding in this way recursively, we can determine all the components of $T - T'$.

A recursive separator partition of T' in the form of a tree of vertex separators of T' can be found in time linear in the size of T', i.e. in time $O(\log n)$. Hence, the total time taken by finding such separator partitions including the recursive calls of the BuildTree procedure is easily seen to be $O(n)$.

In order to estimate the expected total work taken by determining the components of the tree T restricted to the current component with respect to the evolutionary tree T' for the sample for the current component let us make the following observation.

Since the recursive separator partition of T' has depth logarithmic in the size of T', each species labeling a leaf of T takes part in $O(\log \log n)$ leaf sets that are partitioned by the experiments in order to determine the components of $T - T'$.

For a species s, let $n_1, \ldots, n_h$ be the sizes of the components that s belongs to during the performance of the algorithm, where $n_1 = n$, $n_i > \log n$ for $i = 1, \ldots, h-1$, and $n_h \le \log n$. Let $h(n) = \log n - \log \log n$.

By Theorem 2.1 it follows that the probability that $n_{i+1} \leq n_i/2$, for $i = 1, \ldots, \min(h(n), h) - 1$, is at least $\prod_{i=1}^{\min(h(n),h)-1}(1 - \frac{1}{n_i}) \geq \prod_{i=1}^{h(n)-1}(1 - \frac{1}{\log n}) \geq e^{-1}$. Consequently, $\mathrm{Prob}(h \leq h(n)) \geq e^{-1}$ holds. Hence, by Markov inequality the expected value of h is at most $eh(n)$.

Thus, the expected number of leaf sets s belongs to during our algorithm is $O(\log n \log\log n)$. For each leaf set species s belongs to, s has to participate in $O(nd)$ experiments (determining the components of the tree T restricted to the current component with respect to the splitting trees). The expected total work and the expected total number of experiments are therefore $O(nd \log n \log\log n)$. □

Chapter 3

Merging Trees

In this chapter, we present a new technique of *efficiently merging partial evolutionary trees* in the experiment model. We are not aware of any prior methods for the important issue of efficiently merging trees.

The problem of merging two partial, leaf-disjoint evolutionary trees consists of constructing the partial evolutionary tree induced by the species, represented by the leaves of the two trees, by using both trees and access to an experiment oracle for triples of the aforementioned species.

We give an algorithm for this tree merging problem running in time $O(dn)$, where n is the number of leaves in the resulting tree and d is its maximum degree. Our merging algorithm yields a deterministic algorithm for constructing an evolutionary tree for n species from experiments running in time $O(dn \log n)$.

Section 3.1 describes our efficient method of merging partial evolutionary trees with disjoint sets of leaves. The merging is done in a top-down fashion, where we start at the roots of both trees. By making a few experiments the shape of the merged tree at the top level can be decided and the merging at lower levels of the tree can be done recursively. In section 3.2 a lower bound on the number of experiments necessary for merging two evolutionary trees is given. It asymptotically matches the upper time provided by our merging algorithm proving its asymptotic optimality.

3.1 Merging Partial Evolutionary Trees

Throughout the chapter, we shall denote by T an arbitrary fixed rooted evolutionary tree for a set of n species. We shall identify any leaf of T with the unique species labeling it. By the *degree* of a node v in a rooted tree, $deg(v)$ for short, we shall mean the number of children of v in the tree.

Given a subset S of leaves of T, the *partial evolutionary tree* induced by S, denoted by $T \parallel S$, is defined as the tree obtained by taking the minimum subtree of T connecting the species in S, and then contracting any degree 1 nodes in this subtree. The root of $T \parallel S$ is the node in the subtree closest to the root of T. For a partial evolutionary tree U, the set of its leaves will be denoted by S_U.

The problem of *merging two partial evolutionary trees* L and R, where $S_L \cap S_R = \emptyset$, consists of constructing the partial evolutionary tree $T \parallel (S_L \cup S_R)$, denoted by $L \cup R$, using L, R, and access to an experiment oracle for triples in $(S_L \cup S_R)^3$.

Theorem 3.1 *Let L and R be two partial evolutionary trees with disjoint sets of leaves. The merging of L and R into $L \cup R$ can be done in time $O(\sum_{v \in L \cup R} deg^2(v))$ using at most $\sum_{v \in L \cup R} deg^2(v)$ experiments.*

Proof: Let $L_1, \ldots, L_k$ denote the maximal subtrees of L respectively rooted at the children of the root of L, and let the subtrees $R_1, \ldots, R_m$ of R be defined symmetrically.

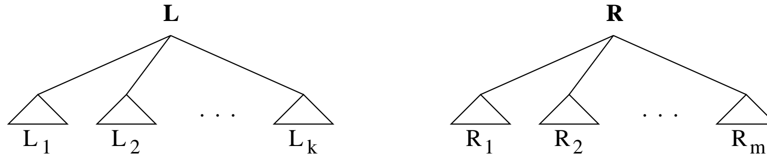

Figure 3.1: The trees L and R.

Six different types of the tree $L \cup R$ can be distinguished, see Figure 3.2.

1. The roots of L and R are exactly the children of the root of $L \cup R$.
2. The roots of $L_1, \ldots, L_k$ and R are exactly the children of the root of $L \cup R$.
3. The roots of L and $R_1, \ldots, R_m$ are exactly the children of the root of $L \cup R$.
4. The sequence of the maximal subtrees rooted at the children of the root of $L \cup R$ has the form $L_1, \ldots, L_{i-1}, L_i \cup R, L_{i+1}, \ldots, L_k$, where $1 \leq i \leq k$.
5. The sequence of the maximal subtrees rooted at the children of the root of $L \cup R$ has the form $R_1, \ldots, R_{i-1}, R_i \cup L, R_{i+1}, \ldots, R_m$, where $1 \leq i \leq m$.
6. The sequence of the maximal subtrees rooted at the children of the root of $L \cup R$ has the form $L_{i_1} \cup R_{j_1}, L_{i_2} \cup R_{j_2}, \ldots, L_{i_p} \cup R_{j_p}, L_{i_{p+1}}, \ldots, L_{i_k}$, $R_{j_{p+1}}, \ldots, R_{j_m}$.

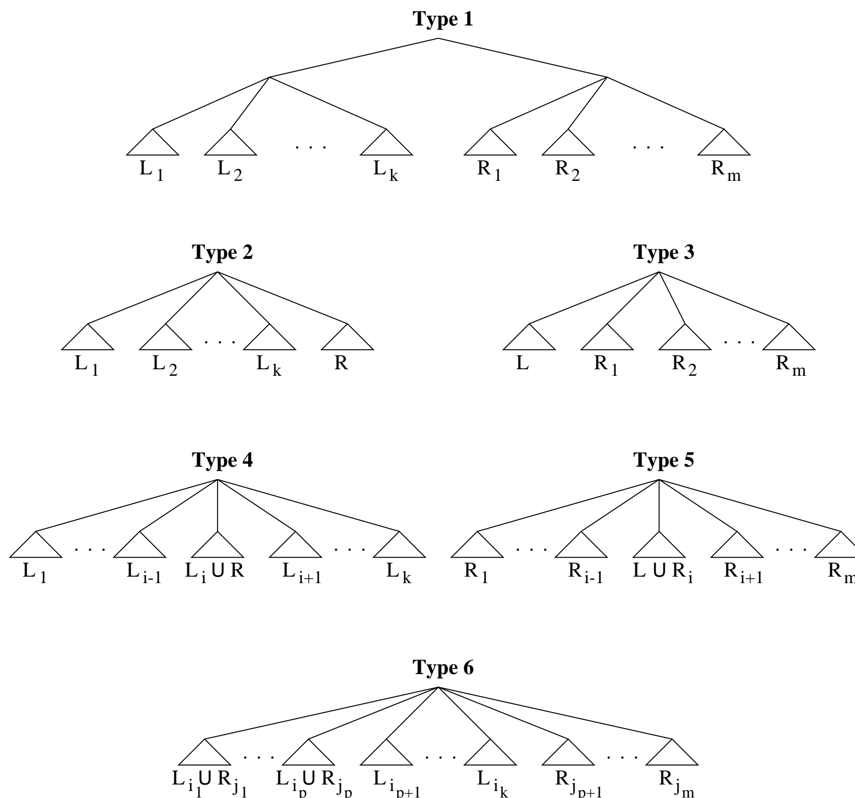

Figure 3.2: The six different types of the tree $L \cup R$.

Note that that the types 1, 2 and 3 are identical except for the placement of the root.

To see that $L \cup R$ is always of one of the six types observe that otherwise one of the maximal subtrees rooted at the children of the root of $L \cup R$ would exactly include some proper and non-empty subset of S_{L_i} or S_{R_j} for some $1 \leq i \leq k$ or some $1 \leq j \leq m$. Then, the root of L_i or R_j would overlap with that of $L \cup R$ which would contradict the topology of L or R.

For $i = 1, \ldots, k$, let l_i be a fixed leaf in S_{L_i}. Similarly, for $j = 1, \ldots, m$, let r_j be a fixed leaf in S_{R_j}.

The time complexity of the following multi-case merging method will depend only on the squares of the degrees of the nodes in $L \cup R$. The method first performs an experiment on the triple l_1, l_2, r_1, and then branches depending on the outcome of this experiment. In describing the possible outcomes we will

write $((a, b), c)$ to denote the fact that the lowest common ancestor of leaves a and b lies below the lowest common ancestor of leaves a and c in T. If the lowest common ancestors for all pairs from $\{a,\ b,\ c\}$ are the same node, we just write (a, b, c).

Case $((l_1, r_1), l_2)$. It is easy to verify that $L \cup R$ cannot be of type 1, 2 or 3. If it is of type 4 then one of the maximal subtrees rooted at the children of the root of $L \cup R$ is $L_1 \cup R$. If it is of type 5 then one of the maximal subtrees rooted at the children of the root of $L \cup R$ is $L \cup R_1$. Finally, if it is of type 6 then one of the maximal subtrees rooted at the children of the root of $L \cup R$ is $L_1 \cup R_1$.

To determine $L \cup R$, perform the experiment on the triple $l_1,\ l_2,\ r_2$. There are four possible outcomes:

$((l_1, r_2), l_2)$: $L \cup R$ is of type 4 and it remains to construct $L_1 \cup R$ in order to determine $L \cup R$.

$((l_1, l_2), r_2)$: $L \cup R$ is of type 5 and symmetrically it remains to construct $L \cup R_1$ in order to determine $L \cup R$.

$((l_2, r_2), l_1)$: $L \cup R$ is of type 6 and two of the maximal subtrees rooted at the children of the root of $L \cup R$ are $L_1 \cup R_1$ and $L_2 \cup R_2$, respectively. To determine the remaining subtrees $L_{i_q} \cup R_{j_q}$ perform the experiments on the triples $l_1,\ l_i,\ r_j$ for all $i,\ j$ satisfying $2 < i \leq k$ and $2 < j \leq m$. Whenever the outcome is $((l_i, r_j), l_1)$, $L_i \cup R_j$ must be one of the maximal subtrees rooted at the children of the root of $L \cup R$.

(l_1, l_2, r_2): $L \cup R$ is again of type 6, but $L_2 \cup R_2$ is not a child of the root, and similarly it is sufficient to perform the experiments on all the triples $l_1,\ l_i,\ r_j$ where $2 \leq i \leq k$ and $2 \leq j \leq m$.

Case $((l_2, r_1), l_1)$. This case reduces to the previous one by swapping L_1 and L_2.

Case (l_1, l_2, r_1). It is easy to verify by case analysis that $L \cup R$ cannot be of type 1 or 3 and if it is of type 5 then one of the maximal subtrees rooted at the children of the root of $L \cup R$ is $L \cup R_1$.

To determine $L \cup R$, perform the experiment on the triple $l_1,\ r_1,\ r_2$.

$((r_1, r_2), l_1)$: $L \cup R$ is of type 2 or 4. In this subcase if there is an index i such that $L_i \cup R$ is one of the maximal subtrees rooted at a child of the root of $L \cup R$ then $L \cup R$ is of type 4. We can easily verify whether

or not such an index exists, and if so, find it, e.g. by performing the experiments on the triples $l_i,\ l_{i+1},\ r_1$ for all i satisfying $1 \le i < k$.

$((l_1, r_1), r_2)$: $L \cup R$ is of type 5 and $L \cup R_1$ is one of the maximal subtrees rooted at the children of the root of $L \cup R$.

(l_1, r_1, r_2) or $((l_1, r_2), r_1)$: $L \cup R$ is of type 6. By performing the experiments on all the triples $l_i,\ r_j,\ r_{j+1}$ where $1 \le i \le k$ and $1 \le j \le m-1$, we can easily determine all the subtrees $L_{i_q} \cup R_{j_q}$, where $1 \le q \le k$.

Case $((l_1, l_2), r_1)$. $L \cup R$ can be only of type 1, 3 or 5. To determine $L \cup R$, perform the experiment on the triple $l_1,\ r_1,\ r_2$.

$((r_1, r_2), l_1)$: We are done since $L \cup R$ is simply of type 1.

(l_1, r_1, r_2): $L \cup R$ is of type 3 or 5. In this subcase, if there is an index j such that $L \cup R_j$ is one of the maximal subtrees rooted at a child of the root of $L \cup R$ then $L \cup R$ is of type 5, otherwise it is of type 3. We can easily verify whether or not such an index exists, and if so, find it, by performing the experiments on all the triples $l_1,\ r_j,\ r_{j+1}$, where $1 \le j \le m-1$.

$((l_1, r_1), r_2)$: (or **$((l_1, r_2), r_1)$** respectively): $L \cup R$ is of type 5 and $L \cup R_1$ (or $L \cup R_2$, respectively) is one of the maximal subtrees rooted at the children of the root of $L \cup R$.

Complexity analysis. Let w be the root of $L \cup R$. Observe that if $L \cup R$ is respectively of type 1, 2, 3, 4, 5, or 6 then $deg(w) = 2$, $deg(w) = k+1$, $deg(w) = m+1$, $deg(w) = k$, $deg(w) = m$, or $deg(w) \ge \max\{k, m\}$ respectively. Also, we have $k,\ m \ge 2$. Hence, by straightforward examination of all the cases, we infer that for any type of the resulting $L \cup R$, we use at most $deg(w)^2$ experiments apart from the experiments used to merge respective subtrees on subsequent recursion levels. The upper bound on the number of experiments easily follows by induction. To complete the proof it is sufficient to observe that our method requires a number of steps proportional to the number of experiments. □

Corollary 3.2 *Two evolutionary trees with disjoint sets of leaves can be merged in time $O(dn)$ using at most $2dn$ experiments, where n is the number of leaves in the resulting tree and d is its maximum degree.*

Proof: Let L and R be the two input trees. The maximum degree d of the resulting tree is not less than the maximum degrees of L and R. Hence, by Theorem 3.1, L and R can be merged in time $O(d \sum_{v \in L \cup R} deg(v))$ using at most $d \sum_{v \in L \cup R} deg(v)$ experiments. Now, it is sufficient to note that

$\sum_{v \in L \cup R} deg(v) \leq 2n$ since each non-leaf vertex of L or R has at least two children. □

Theorem 3.3 *An evolutionary tree, of maximum degree d, for n species can be constructed using experiments in time $O(dn \log n)$.*

Proof: Split the set of species (i.e. leaves) into two parts, of size $\lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor$, respectively. Next, we recursively construct the partial evolutionary tree for each of the parts, and merge the two resulting trees. Now, Corollary 3.2 together with the logarithmic recursion depth yield the theorem thesis. □

If the maximum degree d in the tree is known in advance it is possible to improve the upper time bound in Theorem 3.3. Simply, we can limit the recursion depth to $\lceil \log_2(n/d) \rceil$, and use the quadratic time algorithm by Kannan *et al.* [31] for the $O(n/d)$ subtrees for at most d species to reduce the upper bound to $O(dn + dn \log(n/d))$.

3.2 The Lower Bound on Merging Trees

To prove that $\Omega(dn)$ experiments are needed to merge two trees into one tree with n leaves and maximum degree d, we consider first the case of merging two d-stars. A d-star is a rooted tree with d leaves, which are the only children of the root.

Lemma 3.4 *Merging two evolutionary d-stars into one tree with $2d$ leaves requires, in the worst case, $\Omega(d^2)$ experiments.*

Proof: Consider two d-stars L and R. Next, consider an algorithm, say A, for merging evolutionary trees. We give an adversary argument showing that A must perform $\Omega(d^2)$ experiments to construct $L \cup R$.

Let $l_1, \ldots, l_d$ and $r_1, \ldots, r_d$ denote the leaves of L and R respectively. The subtrees rooted at the children of the root of $L \cup R$ will have two leaves, one from L and one from R, see Figure 3.3. We claim that in order to determine the pairs $(l_{i_1}, r_{j_1}), \ldots, (l_{i_d}, r_{j_d})$ forming the aforementioned subtrees, the algorithm A may have to perform enough experiments on triples $l_i,\ r_j,\ x$, where l_i is any leaf in L and r_i is any leaf in R, so that the number of different pairs of leaves (l_i, r_j) induced by the triples is at least $\binom{d}{2}$.

To prove the claim we play the following game with A as long as the following condition holds: *There are still at least two perfect pairings (matchings) of the l_i's with r_j's so that the respective pairs could be merged without contradicting experiments performed so far.* To model the game we use an originally complete, bipartite graph G on $\{l_1, \ldots, l_d\} \cup \{r_1, \ldots, r_d\}$.

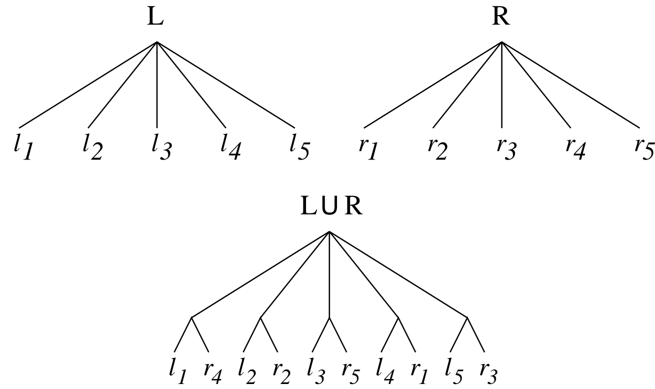

Figure 3.3: An example of two merged 5-stars.

Whenever A performs an experiment on a triple $l_i,\ r_j,\ x$, where x is any leaf, we set the outcome of the experiment to be $(l_i,\ r_j,\ x)$. Note that such an outcome excludes the possibility that the pairs of subtrees corresponding to the pairs of leaves in the experiment performed are subject of merging. Therefore, we remove the edge (if not already removed) connecting l_i with r_i from the bipartite graph G. Also, depending on whether x is a leaf $l_{i'}$ or some $r_{j'}$, we remove the edge connecting $l_{i'}$ with r_j or l_i with $r_{j'}$, respectively. The minimum number of experiments necessary to finish the game is clearly at least the minimum number of edges necessary to delete from G, originally isomorphic to $K_{d,d}$, so there is left a unique perfect matching. The latter number is in turn at least the number of pairs of edges in the perfect matching, i.e. $\binom{d}{2}$, since for each pair, say (l_{i_1}, r_{j_1}), (l_{i_2}, r_{j_2}), at least one of the two cross edges (l_{i_1}, r_{j_2}), (l_{i_2}, r_{j_1}) has to be deleted.

Each experiment triple induces at most two different pairs. We conclude that the algorithm A has to perform at least $\binom{d}{2}/2 = \Omega(d^2)$ experiments to merge the two trees. □

Theorem 3.5 *Merging two evolutionary trees with disjoint sets of leaves requires in the worst case $\Omega(dn)$ experiments where n is the number of leaves in the resulting tree and d is its maximum degree.*

Proof: Let l be the maximum integer such that $n = 2ld + x$, for some integer $x < 2d$. Consider an arbitrary tree L with maximum degree d, dl leaves and exactly l parents of the leaves. Observe that each of the l parents of the leaves is the root of a d-star shaped subtree of L. Let R be another tree whose root has two children, one is the root of any subtree with x leaves and maximum degree

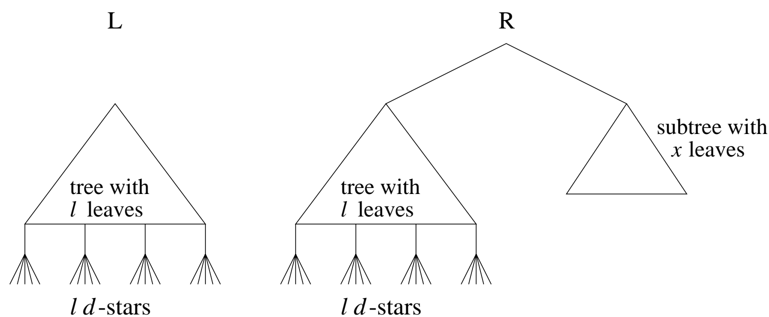

Figure 3.4: The input trees L and R.

d and the other is the root of a subtree that is isomorphic to L after neglecting leaf labels. See Figure 3.4 for an example.

As an adversary we can force an algorithm for merging L with R to pairwise merge the l d-star shaped subtrees of L with the subtrees of R of the same form. The algorithm then has to solve the problem of merging two d-stars l times. Note that experiments performed when merging one pair of d-stars cannot give any information useful when merging another pair of d-stars, since an experiment only includes three leaves, i.e. at most one pair of d-stars has at least two leaves in the experiment. From Lemma 3.4 it follows that any algorithm merging two d-stars into one tree with $2d$ leaves and maximum degree d, has to perform in the worst case $\Omega(d^2)$ experiments. Since this has to be done l times and $l \cdot \Omega(d^2) = \Omega(dn)$, the theorem holds. $\square$

Chapter 4

Tight Bound for Constructing Trees

In this chapter we present the first tight upper and lower bounds for the problem of constructing evolutionary trees 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 the dynamic algorithm in [40]. Our construction improves the previous best upper bound by a factor $\Theta(\log d)$. By an explicit adversary argument, we show a matching lower bound, improving the previous best lower bound by a factor $\Theta(\log_d n)$.

Our algorithm can also be used as a dynamic algorithm supporting insertion of new species, in which case it runs in 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. Central to our algorithm is the concept of separator trees of small height. In Section 4.1 we develop efficient algorithms for constructing and maintaining separator trees during the insertion of new nodes. These constructions may be of independent interest.

The chapter is organized as follows. In Section 4.1 we define the concept of separator trees and describe how to construct and efficiently maintain separator trees of small height. In Section 4.2 we present our algorithm for constructing and maintaining evolutionary trees. In Section 4.3 and 4.4 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 [7]

for proving a trade-off between the preprocessing time of a set of elements and membership queries, and Brodal, Chaudhuri, and Radhakrishnan [8] for proving a trade-off between the update time of a set of elements and the time for reporting the minimum of the set.

4.1 Separator Trees

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

Definition 4.1 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 4.1.

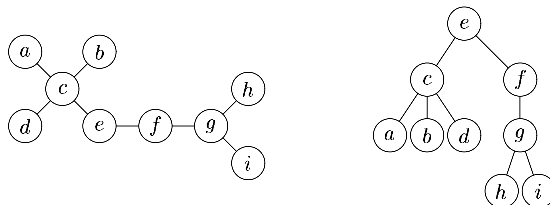

Figure 4.1: A tree T (left) and a separator tree S_T for T (right).

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

Fact 4.2 *Let S_T be a separator tree for T, and let r be a node in T. If S' denotes the subtree of S_T rooted at r, then:*

1. *The subgraph T' induced by the nodes in S' is a tree, and S' is a separator tree for T'.*
2. *For any edge from T with exactly one endpoint in T', the other endpoint is an ancestor of r in S_T, and each ancestor of r 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 4.1.

Definition 4.3 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.

In Section 4.1.1 we first show how to construct 1/2-separator trees in linear time. We then in Section 4.1.2 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 4.1.3 show how to extend the algorithms with a specific ordering of the children facilitating the use in Section 4.2 of separator trees for the efficient construction and maintenance of evolutionary trees in the experiment model.

4.1.1 Constructing Separator Trees

In Lemma 4.4 below we first give a simple algorithm for constructing 1/2-separator trees in time $O(n \log n)$. In Lemma 4.5 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 4.4 *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 $T(n) \leq cn(\log n + 1)$. □

The algorithm of Lemma 4.4 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 4.5 *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 4.4. 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: join, split, successor, and addpathcost. 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. 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 [51, 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 the insertion of an element with arbitrary priority and the deletion of the element with maximum priority in logarithmic time, and construction in linear time, e.g. binary heaps [20, 54]. The total time for constructing the initial priority queues at the nodes is $O(n)$.

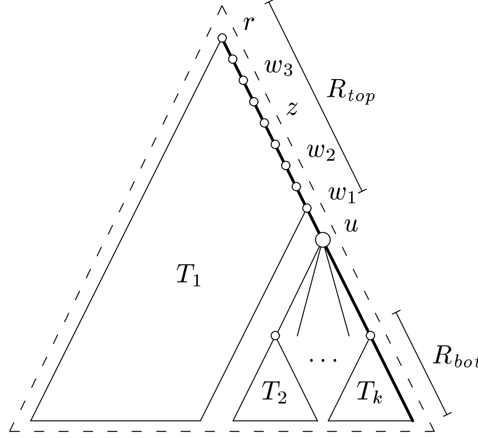

Figure 4.2: The separator node u on the heavy-path $R = R_{top} \cup \{u\} \cup R_{bot}$, and the nodes $w_1, \ldots, w_\ell$ where to update the left-to-right order of the children.

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 computed in time $O(\log n)$. 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 4.2. 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.

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

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

for some positive constant c, where $\sum_{i=1}^{k} n_i = n - 1$ and $n_i \leq n/2$ for all $i = 1, \ldots, k$. By induction it follows that $T(n) \leq cn + cn \sum_{i=0}^{\lfloor \log n \rfloor} i^2/2^i \leq 7cn$, 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 4.2. 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 equals 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 successor$(2\,|w_i|)$. See Figure 4.2. 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. □

4.1.2 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 4.3 shows a tree before and after one addition of each type, with new nodes in bold.

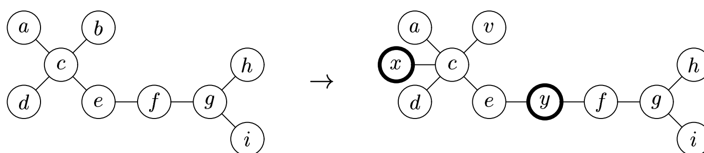

Figure 4.3: Insertions into a tree T.

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 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 resulting tree is easily seen to be a separator tree for the updated tree T. Figure 4.4 shows the insertions into S_T corresponding to the insertions into T shown in Figure 4.3.

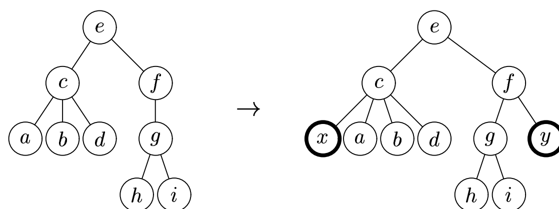

Figure 4.4: Insertions into S_T corresponding to Figure 4.3.

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 described in [4, 5] for maintaining small height in binary search trees. We first show how the linear time construction algorithm for 1/2-separator trees from Lemma 4.5 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, and experiments are likely to be expensive in practice [31]. 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).

Lemma 4.6 *For any $0 < \varepsilon < 1/4$, a $(1/2+\varepsilon)$-separator tree can be maintained in $O((\log n)/\varepsilon)$ amortized time per insertion.*

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 on the path, we rebuild the subtree rooted at the highest of these, using Lemma 4.5, and then restore the size and depth information by a traversal of the rebuilt subtree. Let v denote this node, and let u denote its largest child just before the rebuild. 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 $\Theta(1/\varepsilon)$ each will cover the $\Theta(|v|_{\text{now}})$ cost for rebuilding the subtree of v and restoring the information at the nodes. Thus, if an insertion is charged $\Theta(1/\varepsilon)$ 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 $O(\log_{1/(1/2+\varepsilon)} n)$, which is $O(\log n)$ by $\varepsilon < 1/4$, the stated time bound follows. □

Lemma 4.7 *A $(1/2 + o(1))$-separator tree can be maintained with a height bound of $\log(n) + 1$ in $O(\log^2 n)$ amortized time per insertion.*

Proof: In the method of Lemma 4.6, we maintain $\varepsilon < c/\log n$ for a constant $c > 0$, changing ε by a global rebuild of the entire separator tree whenever n has doubled.

The height h of a separator tree with threshold t satisfies $t^h n \geq 1$, implying $h \leq \log(n)/\log(1/t)$. For $x \to 0$, we have the first order approximations $\log(1+x) = \Theta(x)$ and $1/(1-x) = 1 + \Theta(x)$. Using these for $t = 1/2 + \varepsilon$ we get $1/\log(1/t) = -1/\log(t) = -1/\log(1/2+\varepsilon) = -1/(-1+\log(1+2\varepsilon)) = -1/(-1+\Theta(\varepsilon)) = 1 + \Theta(\varepsilon)$.

Choosing c small enough, we get a height bound of $\log(n)(1 + 1/\log(n)) = \log(n) + 1$, and an amortized time for insertions of $O(\log(n)/\varepsilon) = O(\log^2 n)$, as the amortized cost of the global rebuildings is $O(1)$ per insertion by Lemma 4.5. Specifically, the value $c = 1/6$ will do for $n \geq 2$, as can be verified by considering tangents to the graphs of $\log x$ and $1/(1-x)$. □

In the next theorem, we reduce the amortized time bound to $O(\log n)$. Statements about amortized complexity for data structures normally assume an initially empty structure—this is a special case of the statement below.

Theorem 4.8 *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 4.7 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 power of two between $(\log n)/2$ and $2 \log n$.

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 component of size one, which is added to the structure. For all other insertions, an existing (but possibly empty) 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 4.5. 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. This splits the violating new component, 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 4.7.

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 4.2). 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 easily 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 Lemma 4.2, 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 4.7, inspection of edge components will only add an additive logarithmic term to the amortized rebalancing cost for the separator tree for U, which therefore remains $O(\log^2 |U|)$.

To maintain the value of Δ, we rebuild the entire structure whenever n has doubled, adjusting Δ if it will go out of bounds before the next doubling of n. 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 4.5 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. This value is used below.

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. 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 4.4. 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 $O(\Delta/4)$ time, the generation of U can be performed in $O(n)$ time. By the time bound from Lemma 4.5, the entire separator tree for τ can be constructed in $O(n)$ time.

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 4.5 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 4.7, 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. □

4.1.3 Ordered Separator Trees

Lemma 4.9 *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 [36]. 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. □

We now extend the separator trees maintained by the algorithm from Theorem 4.8 with a specific ordering of the children, facilitating our use of separator trees in section 4.2 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.

Theorem 4.10 *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, the followings holds

$$\prod_{d_i \leq 2} 2 \cdot \prod_{d_i > 2} d_i < 9d(n+m) ,$$

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 $d = \max\{d_1, \ldots, d_{\ell-1}\}$.

Proof: The proof is by an extension of the construction from Theorem 4.8, 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 come 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 for 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.

This 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 4.5. 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 can be restored without affecting the time bound, by Lemma 4.9. 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, this can be done in constant time per node on the path, as the size of only one child per node changes, and only by 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 4.8. The height bound also follows from Theorem 4.8.

To prove the last claim of the lemma, note that the path 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 has a threshold of $1/2+\varepsilon$, where ε originates from the proof of Lemma 4.7, and has a value bounded by $c/\log|U|$ for some

constant $c \leq 1/6$. 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:

$$1 \;\leq\; |U| \cdot (1/2 + \varepsilon)^k \cdot \prod_{\substack{d_i \geq 2 \\ i<j}} \frac{1}{d_i} \,,$$

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

$$(1 + 2\varepsilon)^k \leq (1 + 2c/\log |U|)^{\log |U|+1} \leq (1 + 2c)e^{2c} \,,$$

since $|U| \geq 2$ when this part of the path exists. For $c = 1/6$, this value is below 1.861, and choosing e.g. $c = 1/40$ gives a value below $1 + 1/8$. We assume the latter choice of c here. Recalling that $|U| \leq 8(n + m)/\Delta$, we get

$$1 \;<\; 9(n + m)/\Delta \cdot (1/2)^k \cdot \prod_{\substack{d_i \geq 2 \\ i<j}} \frac{1}{d_i} \,.$$

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

$$1 \;\leq\; \Delta \cdot (1/2)^{k'} \cdot \prod_{\substack{d_i \geq 2 \\ i>j}} \frac{1}{d_i} \,,$$

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

$$d_j \;\leq\; d \,.$$

Multiplying left sides and right sides in the last three inequalities and rearranging the result proves the last statement of the lemma (since $d \geq 2$). □

4.2 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 construction algorithm is a further development of the dynamic algorithm by Lingas *et al.* in [40]. Our algorithm also yields a faster dynamic algorithm supporting online 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. The insertion 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 insertion of the new species starts at the root of S_T. We decide by experiments which subtree, rooted at a child of the root in S_T, the species a should be inserted into. This is repeated recursively until the correct position in T for a is found. We keep links between corresponding nodes in S_T and T for switching between the two trees. Furthermore 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.

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 new species a should be inserted directly below one of the nodes in S_v or should split an edge which is incident to a node in S_v, by creating a new internal node on the edge and make a a leaf below the new node. To decide this using experiments we proceed as described below.

Let v be the node in S_T for which we want to decide if the new species a should be inserted in the subtree above v in T, directly below v in T as a new child of v, or in a subtree rooted at a child $u_1, \ldots, u_k$ of v in T. The order of the children $u_1, \ldots, u_k$ is such that $u_1, \ldots, u_{k'}$ correspond to 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 correspond to 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 arbitrary 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. There are four possible outcomes of the i'th experiment corresponding to Figure 1.5:

1. (a, b, c) implies that a should be inserted below a u_j where a and b are not descendents of u_j, or a is a new leaf below v.

2. $((a, b), c)$ implies that a should be inserted below 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 a should be inserted into u_{2i} (u_1 for the $\lceil k/2 \rceil$'th experiment if k odd).

4. $((b, c), a)$ implies that a should be inserted into the subtree above v, since the least common ancestor of a and b is above v.

We perform experiments for increasing i until we get an outcome different from Case 1, or until we have performed all $\lceil k/2 \rceil$ experiments all with outcome as in 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 species a should be inserted into. 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 splits. Otherwise we continue recursively by inserting 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 be the one which species a should be inserted into. When the correct location of species a is found, the trees T and S_T are updated accordingly, as given by Theorem 4.10.

Lemma 4.11 *Given an evolutionary tree T for n species with degree d, and a separator tree S_T for T according to Theorem 4.10, 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 4.10 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 $i < \ell$, we at node v_i perform x_i experiments where $x_i \leq \lceil d/2 \rceil$ 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 4.10 have the constraint $\prod_{d_i \leq 2} 2 \cdot \prod_{d_i > 2} d_i \leq 9dn$. 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.

$$\begin{aligned}
\log(9dn) &\geq \sum_{d_i \leq 2} 1 + \sum_{d_i > 2} \log d_i \\
&\geq \sum_{x_i = 1} 1 + \sum_{x_i > 1} \log d_i \\
&\geq \sum_{x_i = 1} 1 + \sum_{x_i > 1} x_i \frac{1}{x_i} \log(2x_i - 1) \\
&\geq \frac{1}{\lceil d/2 \rceil} \log(2\lceil d/2 \rceil - 1) \sum_{i=1}^{\ell-1} x_i \ ,
\end{aligned}$$

since $x_i > 1$ implies $d_i \geq 3$, and for $f(x) = \frac{1}{x}\log(2x-1)$ it holds that $1 > f(2) > f(3)$ and $f(x)$ is decreasing for $x \geq 3$.

We conclude that $\sum_{i=1}^{\ell-1} x_i \leq \lceil d/2 \rceil \log_{2\lceil d/2 \rceil - 1}(9dn)$, i.e. for the total number of experiments we have $\sum_{i=1}^{\ell} x_i \leq \lceil d/2 \rceil (\log_{2\lceil d/2 \rceil - 1}(9dn) + 1)$.

The time needed for the insertion is proportional to the number of experiments performed plus the time to update S_T. By Theorem 4.10 the total time is thus $O(d \log_d n)$. □

From Lemma 4.11 and Theorem 4.10 we get the following bounds for constructing and maintaining an evolutionary tree under the insertion of new species in the experiment model.

Theorem 4.12 *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.*

4.3 Adversary for Constructing Evolutionary Trees

To prove a lower bound on the number of experiments required for constructing an evolutionary tree of n species, 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 (partially) represents the information revealed by the adversary. 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 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

$$|\mathrm{F}(a)| + |\mathrm{C}(a)| \leq d - 2\,. \tag{4.1}$$

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 (4.1) 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 such that c_i is added to the forbidden lists of all nodes in $\mathrm{C}(a)$. 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 $a \in \mathrm{C}(b)$.
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.

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 $\mathrm{Fix}(b, c)$. The operation $\mathrm{Fix}(a, b)$ ensures that $\mathrm{LCA}(a, b)$ is fixed by performing at most one of the following three updates.

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 $\mathrm{F}(a)$.

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 (4.1) is reestablished. 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 1.5.

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 $((b, c), a)$.

4.4 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 4.13 *Constructing an evolutionary tree for 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 $\text{Fix}(a, b)$ at most increases the size of the two conflicting lists, $\text{C}(a)$ and $\text{C}(b)$, by one, or the size of one of the forbidden list, $\text{F}(a)$ or $\text{F}(b)$, by one. If performing $\text{Fix}(a, b)$ increases the sum $|\text{F}(a)| + |\text{C}(a)|$ to $d - 1$, then species a is moved one level down in D and $\text{F}(a)$ and $\text{C}(a)$ 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 4.3, 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. After 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 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. $\square$

Part II

Local Consensus Trees

Local Consensus Trees

Different evolutionary trees for the same set of species or overlapping sets of species are considered for several reasons. For example different kinds of data gives us different trees, as in the case of gene trees where different genes give different trees that do not match the species tree exactly. Another reason we get different trees is that it often is NP-hard to find the optimal tree by using a certain method and thus we get approximate trees instead of an optimal one. It may also be hard to know what to optimize to find the true evolutionary tree. Therefore the optimal tree is not necessarily the tree we are looking for. These reasons, and others, like the fact that the data comes from biological experiments, which are not exact, result in that we end up having several trees, all saying something about the evolutionary tree we are looking for, but they may be inconsistent, include just subsets of the species, or not be as resolved as we would like. This is the motivation for consensus methods.

There are two basic consensus methods used. The first one is to find a maximum homeomorphic subtree (also called maximum agreement subtree), where a maximum size subset of the species is sought, such that all the available trees agree on the evolutionary relationship on this subset of species. The maximum homeomorphic subtree problem is considered in Part III. In the second approach the goal is to represent all input trees as a single tree for all available species. There are a number of methods of the second type, for example the tree compatibility problem [24, 52], strict consensus [13], median tree [6], asymmetric median tree [46], and consensus tree problem [28]. The objectives for these methods are described below.

A tree T is called the *compatibility tree* for a set $\{T_1, \ldots, T_k\}$ of trees, labeled by the species in a set S, if it holds that $C(T) = \bigcup_{i=1}^{k} C(T_i)$, where $C(T) = \{c_e : e \in E(T)\}$ and c_e is defined as the bipartition of S caused by removing edge e from T. The *tree compatibility problem*, sometimes also called *cladistic character problem*, is to find T, if it exists.

The compatibility tree combines all information in the input trees, while the *strict consensus tree* only contains the information common to all input trees. In the *strict consensus tree problem* a tree fulfilling the condition $C(T) = \bigcap_{i=1}^{k} C(T_i)$ is sought. The strict consensus tree can be computed in time $O(kn)$, which was shown by Day [13]. Contrary to the compatibility tree, the strict consensus tree always exists, but it may be very low resolved, i.e. have few edges.

The *median tree* is a tree T that minimizes the sum of symmetric differences between T and the input trees, denoted $d(T, T_i)$, i.e. $\sum_{i=1}^{k} d(T, T_i)$ is minimized, where $d(T, T_i) = |C(T) - C(T_i)| + |C(T_i) - C(T)|$. Since the median tree includes

bipartitions included in at least half of the input trees it is also called the *majority tree*. The median tree always exists and is, in general, more resolved than the strict consensus tree, but on the other hand it is still not as resolved as we would like. This is due to the fact that information not in at least half of the trees will not be included in the median tree. This was the motivation for Phillips and Warnow to suggest the *asymmetric median tree* [46].

In the *asymmetric median tree problem* each bipartition c of S is given a weight, $w(c)$, equal to the number of input trees c is included in, i.e. $w(c) = |\{i : c \in T_i\}|$. The asymmetric median tree is the tree T such that $\sum_{c \in C(T)} w(c)$ is maximized, under the condition that for every $c \in C(T)$ there is at least one tree T_i where $c \in C(T_i)$ and if $c \in C(T_i)$ for all $i = 1, \ldots, k$ then $c \in C(T)$. Phillips and Warnow [46] showed the problem to be NP-hard for three or more trees, but for two trees they gave a polynomial time algorithm. They also provided an approximation algorithm when the number of input trees is three or more.

Let T be a tree leaf labeled by a set $S = \{s_1, \ldots, s_n\}$ of species. Let $T|L$ denote the topological subtree of T for the species in $L \subseteq S$. For a set of binary rooted trees $\{T_1, \ldots, T_k\}$, each one leaf-labeled by a subset $L(T_i)$ of S, the *consensus tree problem* asks whether or not there is a tree T such that for $i = 1, \ldots, k$, T_i is homeomorphic to the subtree of $T|L(T_i)$. If all the input trees are of constant size it is termed the *local consensus tree problem*.

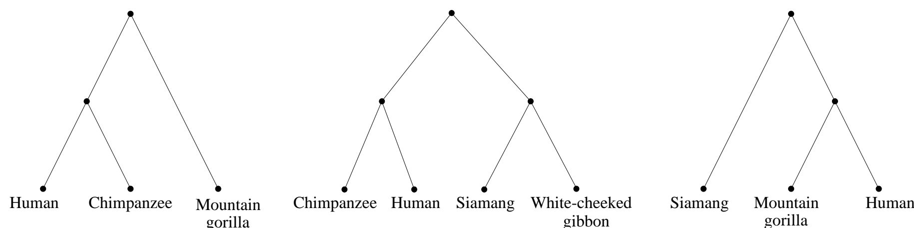

Figure 4.5: Is there a consensus tree for these trees? (The answer is yes!)

A related problem, studied by Aho *et al.*, is to construct trees from constraints on the lowest common ancestors of the leaves in the tree [1]. Let $\{x, y\}$ denote the lowest common ancestor of two leaves x and y in a rooted tree T, with leaves labeled uniquely by the numbers $1, \ldots, n$. The problem studied by Aho *et al.* was the following. Given a set of constraint of the form $\{i, j\} < \{k, l\}$, where $i \neq j$ and $k \neq l$, meaning that $\{i, j\}$ is a proper descendant of $\{k, l\}$ in T, reconstruct T or determine that no such tree exists. The motivation for studying this problem, mentioned in [1], came from an application to relational algebras used in data base theory. In the paper by Aho *et al.* they gave an algorithm for the problem running in time $O(mn \log n)$, where m is the number of constraints.

In the special case where all constraints are of the form $\{i,j\} < \{i,k\}$ they gave an algorithm running in time $O(mn)$. We will denote constraints only involving three distinct leaves, i.e. $\{i,j\} < \{i,k\}$, by $(\{i,j\},k)$.

A constraint of the form $(\{i,j\},k)$ is easily seen to be equivalent to the constraint imposed by a full binary tree on the leaves i,j, and k in the local consensus tree problem. For this reason, we shall call the tree inferring problem posed in [1] the *inferred consensus tree problem.*

A *local consensus rule* determines for three species in S the rooted topology for the three species based on a set of rooted trees. Suppose we define the local consensus rule, for each triple i,j,k, as the topological subtree for i, j, and k in each of the the input trees, $T_1,\ldots,T_k$, if it is the same in all of them and is fully resolved, i.e. binary. Since the binary rooted topology for a triple of species is equal to a constraint of the form $(\{i,j\},k)$ the consensus tree problem can be formulated as the problem of first applying this local consensus rule to each triple of species in S using the input trees $T_1,\ldots,T_k$ and then constructing a tree from constraints on the lowest common ancestors of the form $(\{i,j\},k)$, where i,j,k are species in the set S. This was the approach suggested by Henzinger *et al.* [28]. Using new dynamic data structures for graph connectivity, Henzinger *et al.* presented an $O(m\sqrt{n})$-time deterministic algorithm and an $O(m\log^3 n)$ expected-time randomized algorithm to test if a set of trees has a consensus tree, and if so construct it.

Unfortunately, it is often impossible to construct an exact consensus tree, since the trees often are inconsistent. This creates a need for an optimization version of the inferred consensus tree problem whose objective is to find a consensus tree for an as large as possible subset of the input set of constraints of the form $\{i,j\} < \{k,l\}$. For brevity, we term this optimization problem *the maximum inferred consensus tree problem* (MICT for short). We also distinguish the restricted case of MICT where all constraints are of the form $(\{i,j\},k)$ and call it *the maximum inferred local consensus tree problem* (MILCT).

In Chapter 5 we study the two problems MICT and MILCT. First, we provide an NP-completeness proof for MICT. Since the problem is NP-complete we also suggest a simple heuristic for MICT yielding solutions within one third of the optimum and running in time $O((n+m)\log n)$. For MILCT we present a more involved polynomial-time heuristic. Both heuristics work equally well for the weighted versions of MICT and/or MILCT where the constraints have weights and the objective is to find a consensus tree for a subset of the input constraints of maximum total weight.

Recently Jansson showed that the decision version of MILCT is NP-complete [30]. This was shown by a reduction from the NP-complete problem called *cyclic ordering.*

In Chapter 6 we focus on a special case of the local consensus tree problem for rooted, unweighted evolutionary trees in which a specified left-right ordering is imposed upon the species. Scanning the labels of the leaves of an ordered tree from left to right yields its *frontier*. The motivations for the ordered versions of the consensus tree problem come from tree drawing and from the desire to somehow order the leaves for ease of presentation, for example by the age of the species involved, or just alphabetically.

Another interesting motivation might come from the original local consensus tree problem under the condition that finding a possible frontier of the consensus tree to construct can be done efficiently. In fact, such a two phase approach, consisting of first finding the frontier and then the topology of the tree, has been successfully applied in the construction of evolutionary trees in the experiment model (studied in Part I), see [31]. One has to remember however that in the experiment model, one has the freedom of choosing triples of leaves for experiments revealing their topology, whereas in the local consensus tree problem the local tree constraints are given a priori.

If T, in addition to being in compliance with the given constraints, is required to be constructed in such a way that its frontier equals a certain ordering of the species, then we term the resulting problems *the ordered consensus tree problem* and *the ordered local consensus tree problem*, respectively. Figure 4.6 shows an ordered consensus tree for the species in Figure 4.5.

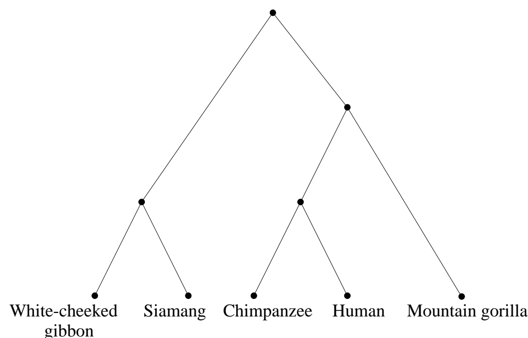

Figure 4.6: An ordered consensus tree for the trees depicted in Figure 4.5. Here, the species are ordered by size.

Our key observation is that the basic dynamic graph used by Henzinger *et al.* in their fast algorithms for the local consensus tree problem becomes an *interval graph* if the tree under construction is required to be ordered.

Using our decremental interval union algorithm, we show that the ordered local consensus tree problem can be deterministically solved in time $O((m +$

$n)\log n)$. We also show that the related optimization problem of constructing an ordered local consensus tree, for the maximum number of given 3-leaf binary trees, is solvable in cubic time.

Chapter 5

Maximum Inferred Consensus Tree

In this chapter two closely related problems, called *maximum inferred consensus problem*, MICT for short, and *maximum inferred local consensus tree*, MILCT for short, are studied.

Given a rooted tree T, leaf labeled by a set S of species, denote by $\{i,j\}$ the lowest common ancestor of the leaves labeled i and j in T. A constraint on the lowest common ancestor, denoted $\{i,j\} < \{k,l\}$, where $i \neq j$ and $k \neq l$, means that $\{i,j\}$ is a proper descendant of $\{k,l\}$ in T. If only three distinct leaves are involved in the constraint, it is called a 3-leaf constraint. A constraint $\{i,j\} < \{i,k\}$ is also denoted $(\{i,j\},k)$.

Maximum inferred consensus tree problem

Input: A set $C = \{c_1, \ldots, c_m\}$ of constraints on the lowest common ancestors of species from a set $S = \{s_1, \ldots, s_n\}$.

Output: A tree T, leaf labeled by S, such that the number of constraints in C, consistent with T, is maximized.

Maximum inferred local consensus tree problem

Input: A set $C = \{c_1, \ldots, c_m\}$ of 3-leaf constraints on the lowest common ancestors of species from a set $S = \{s_1, \ldots, s_n\}$.

Output: A tree T, leaf labeled by S, such that the number of constraints in C, consistent with T, is maximized.

For both MICT and MILCT we can define weighted versions of the problems. Then each constraint c_i has an associated weight $w(c_i)$ and and the objective is to construct a tree T consistent with a set $C_T \subseteq C$ of constraints such that $\sum_{c \in C_T} w(c)$ is maximized over all possible trees T.

The decision version of MICT is NP-complete, which is shown in Section 5.1. The proof is by a reduction from *3-partite perfect matching.* In Section 5.2 a simple heuristic for MICT and MILCT is provided, which produces a tree consistent with constraints of weight at least one third of the total weight of the input constraints. In the special case of MICT, where all constraints involves four different species, the tree produced will fulfill constraints of weight at least one half of the total weight of the input constraints. Finally, in Section 5.3 another heuristic is presented, only applicable to MILCT. Here we use the idea from the algorithm by Aho *et al.* [1], as well as the implementation presented by Henzinger *et al.* [28] to construct a tree consistent with constraints of weight at least $w - nt$, where w is the total weight of all input constraints and t is the total weight of the constraints not consistent with an optimal tree, i.e. the weight of the constraints necessary to remove to obtain an optimal tree.

5.1 MICT is NP-Complete

The problem of deciding whether or not a 3-partite hypergraph (V, E) has a perfect matching (3PM), i.e. if V is covered by a subset of pairwise disjoint edges in E, is known to be NP-complete [44]. To show the NP-completeness of MICT, we provide a reduction of 3PM to MICT.

Let $H = (V, E)$ be a 3-partite hypergraph and k a parameter that will be specified later on. We let each vertex in V label one leaf. Also, for each edge $e \in E$, we introduce $k+2$ leaves labeled e_i, where $i = 0, \ldots, k+1$. Let C be the minimal set of constraints satisfying:

1. for each $e, f \in E$ with $e \neq f$: the constraints $(\{e_i, e_l\}, f_j) \in C$, where $i = 0, 1,\ \ l = 2, \ldots, k+1,$ and $j = 2, \ldots, k+1$.
2. for each $e = (a, b, c) \in E$: the three constraints $\{a, b\} < \{e_0, e_1\}$, $\{a, c\} < \{e_0, e_1\}$, and $\{b, c\} < \{e_0, e_1\} \in C$.

Thus, C consists of $2k^2(|E|^2 - |E|)$ constraints of the first type and $3|E|$ constraints of the second type.

To characterize consensus trees for large subsets of C, we need the following definitions.

Definition 5.1 In a rooted tree T, the lowest common ancestor of a sequence of nodes $v_1, \ldots, v_m$ will be denoted by $lca(v_1, \ldots, v_m)$. Furthermore, the path from

a node v to the root of T will be denoted by $R(v)$. The subtree of T induced by a sequence of nodes $v_1, \ldots, v_m$ is the smallest subtree of T including the paths $R(v_i)$, $i = 1, \ldots, m$.

Definition 5.2 The full binary tree on four leaves a, b, c, d, where $lca(a, b)$ and $lca(c, d)$ form the intermediate level, will be denoted by $B_4(a, b, c, d)$.

Lemma 5.3 *If T is a consensus tree for at least $|C| - k^2 + 1$ constraints in C, then for each $e, f \in E$ with $e \neq f$, the subtree of T induced by $\{e_0, e_1, f_0, f_1\}$ is homeomorphic to $B_4(e_0, e_1, f_0, f_1)$.*

Proof: By the assumption on the number of constraints satisfied by T, for each $e, f \in E$ with $e \neq f$, there are indices $l, j \in \{2, \ldots, k+1\}$ such that for $i = 0, 1$, the constraints $(\{e_i, e_l\}, f_j)$, $(\{f_i, f_j\}, e_l)$ are satisfied by T. Because of $(\{e_0, e_l\}, f_j)$ and $(\{e_1, e_l\}, f_j)$, the path $R(lca(e_0, e_1, e_l))$ cannot be included in the path $R(f_j)$. Thus, $R(lca(e_0, e_1, e_l)) \not\subseteq R(lca(f_0, f_1, f_j))$. Similarly, by $(\{f_0, f_j\}, e_l)$ and $(\{f_1, f_j\}, e_l)$, we have $R(lca(f_0, f_1, f_j)) \not\subseteq R(lca(e_0, e_1, e_l))$. This means that the paths from $lca(e_0, e_1, e_l)$ to $lca(e_0, e_1, e_l, f_0, f_1, f_j)$ and from $lca(f_0, f_1, f_j)$ to $lca(e_0, e_1, e_l, f_0, f_1, f_j)$ must be edge-disjoint. □

Corollary 5.4 *Let T be a consensus tree for at least $|C| - k^2 + 1$ constraints in C. For each node $a \in V$ and two different edges $e, f \in E$, if T satisfies a constraint of the form $\{a, \cdot\} < \{e_0, e_1\}$ then T cannot satisfy any constraints of the form $\{a, \cdot\} < \{f_0, f_1\}$.*

Lemma 5.5 *Let $k > \sqrt{3|E| - |V|}$. The hypergraph H has a perfect matching if and only if there is a consensus tree for a subset of $2k^2(|E|^2 - |E|) + |V|$ constraints in C.*

Proof: Suppose first that H has a perfect matching M. We can construct a consensus tree T satisfying at least $2k^2(|E|^2 - |E|) + |V|$ of the constraints in C as follows. The root of T has $|E|$ children which are in one-to-one correspondence with the edges in E. For every $e \in E$, a subtree rooted in the corresponding child has as children the leaves $e_0, e_1, \ldots, e_{k+1}$. Furthermore, if $e = \{a, b, c\}$ is in M, then the subtree has another child which in turn is the parent of the leaves labeled a, b, c.

Suppose in turn that there is a consensus tree T satisfying $2k^2(|E|^2 - |E|) + |V|$ constraints in C. The total number of constraints in C is $2k^2(|E|^2 - |E|) + 3|E|$. It follows by $k > \sqrt{3|E| - |V|}$ that T satisfies at least $|C| - k^2 + 1$ constraints. Thus, by Corollary 5.4, for each node $a \in V$, there is at most one edge $e \in E$ such that some constraint of the form $\{a, \cdot\} < \{e_0, e_1\}$ is satisfied by T. On the other hand, for a given node a and a given edge e, at most two constraints

of the form $\{a, \cdot\} < \{e_0, e_1\}$ can be satisfied by T by the construction of C. Consequently, V can be partitioned into three disjoint subsets V_r, $r = 0, 1, 2$, respectively consisting of nodes $a \in V$ for which T satisfies r constraints of the form $\{a, \cdot\} < \{\cdot, \cdot\}$. The number of constraints of the form $\{\cdot, \cdot\} < \{\cdot, \cdot\}$ satisfied by T is hence $|V_2| + \frac{|V_1|}{2}$, since each constraint involves two nodes. There are only $2k^2(|E|^2 - |E|)$ constraints of the form $(\{\cdot, \cdot\}, \cdot)$ in C, therefore we conclude that V_2 has to be as large as possible, i.e. $V_2 = V$. It follows that for each edge $e \in E$, if a constraint of the form $\{\cdot, \cdot\} < \{e_0, e_1\}$ is satisfied by T, then all the three constraints of this form are satisfied by T. Hence, H has a perfect matching. □

The construction of C for k equal, say, to $|E|$ can easily be done in polynomial time. Hence, MICT is NP-hard by the NP-completeness of 3PM and Lemma 5.5. The membership of MICT in NP is obvious.

Theorem 5.6 *MICT is NP-complete.*

5.2 A Simple Approximation Heuristics for MICT

For a constraint $\{i, j\} < \{k, l\}$, where all the leaves are different, k and l are said to have an *upper occurrence* in the constraint, and i and j are said to have a *lower occurrence* in the constraint. For a constraint $\{i, j\} < \{i, k\}$, where i, j, k are different, i and j are said to have a lower occurrence in the constraint and k is said to have an upper occurrence in the constraint. The *total weight* of upper (or lower) occurrences for a leaf l is equal to the sum of the weights of all constraints in which l has upper (or lower) occurrences.

Lemma 5.7 *For any instance of MICT, the sum of all leaves' total weights of upper occurrences is at least one third (one half if all constraints contain four different leaves) of the sum of all leaves' total weight of upper and lower occurrences.*

Heuristic 1

input: a set C of m weighted constraints on leaves 1 through n;

output: a consensus tree T for a subset of C whose weight is at least one third (one half if all constraints contain four different leaves) of the total weight of the constraints in C;

1. $LEFT \leftarrow C$;
   $LEAVES \leftarrow \{1, \ldots, n\}$;
   $T \leftarrow \{v\}$;

2. if $LEFT = \emptyset$ then extend T by adding $|LEAVES|$ children to v, label them uniquely with elements in $LEAVES$, and return T;

3. pick a leaf y in $LEAVES$ which achieves the maximum ratio between the total weight of y's upper occurrences and the total weight of y's lower occurrences in the constraints in $LEFT$;

4. set Y to the set of constraints in $LEFT$ which contains y;

5. $LEFT \leftarrow LEFT \setminus Y$;

6. $LEAVES \leftarrow LEAVES \setminus \{y\}$;

7. extend T by adding two children to v; label the first child by y; set v to the second child;

8. go to 2

Theorem 5.8 *Heuristic 1 constructs a consensus tree for a subset of the input constraints C whose total weight is at least one third (one half if all constraints contain four different leaves) of the total weight of C, in time $O((m+n)\log n)$.*

Proof: By Lemma 5.7 and the choice of y, the ratio between the total weight of upper occurrences and lower occurrences of y in the constraints in $LEFT$ is at least one third (one half if all constraints contain four different leaves). All the constraints in Y in which y has an upper occurrence are satisfied by T by the construction of T.

To implement Steps 3 and 6 efficiently, we arrange $LEAVES$ in a priority queue partially ordered by the ratio between the total weight of their upper and lower occurrences in constraints in $LEFT$. All the priority queue operations, i.e. creating the priority queue, picking the y's, updating the priority queue after Step 5, take a total of $O((m+n)\log n)$ time.

To implement Steps 4 and 5, we lexicographically sort C four times according to four cyclic permutations of the four leaves in each constraint. For $i = 1, \ldots, 4$, the i-th permutation puts the i-th leaf as the first, the $(i+1)$-st (in the cyclic order) as the second, etc. Next, four search trees are built on the basis of the sorted lists. Using the search trees, we can find Y in $LEFT$ and remove it from $LEFT$ in time $O(|Y|\log n)$. We conclude that Steps 4 and 5 totally take time $O((m+n)\log n)$ (including the preprocessing). □

The absolute factors of one third and one half, respectively provided by Heuristic 1, are worst-case optimal. For example, any consensus tree can satisfy at most one constraint from each consecutive triple of constraints in a sequence

$(\{a_i, b_i\}, c_i)$, $(\{b_i, c_i\}, a_i)$, $(\{c_i, a_i\}, b_i)$, $i = 1, \ldots, k$. In case all constraints contain four different leaves, the sequence $\{a_i, b_i\} < \{c_i, d_i\}$, $\{c_i, d_i\} < \{a_i, b_i\}$, $i = 1, \ldots, k$, causes the lower bound $\frac{1}{2}$.

The consensus tree produced by Heuristic 1 has the form of a linear chain with singular leaves pending, where only the last chain node can have larger degree. It is easy to slightly modify Heuristic 1 to output the subset of the input constraints satisfied by the tree produced by Heuristic 1. A minimum height consensus tree for at least one third of the input constraints is then obtained in time $O(mn \log n)$ by running the algorithm of Aho *et al.* [1] for the inferred consensus tree problem on this set.

5.3 Heuristic for MILCT

In case the minimum number of constraints necessary to delete in order to build a consensus tree for the remaining part is very small, and the number m of constraints relative to the number of leaves is high (it is always $O(n^4)$), our second approach might be more useful than Heuristic 1.

Heuristic 2 for MILCT simply mimics the algorithm of Aho *et al.* [1] for the inferred consensus tree problem restricted to constraints of the form $(\{i, j\}, k)$. Their basic idea is simple. The input set of leaves $1, 2, \ldots, n$ is partitioned into a minimal set of blocks satisfying the following requirement:

(*) If $(\{i, j\}, k)$ is a constraint then i and j are in the same block.

Now, if the number of blocks in the minimal set is at least two, the algorithm of Aho *et al.* creates the consensus tree by connecting the roots of the consensus trees recursively computed for the respective blocks with a common parent root node. Otherwise, the number is one, and it returns a null consensus tree.

For a subset S of leaves, let $G(S)$ denote the auxiliary graph on S where the edges are induced by the requirement (*), and their weights are equal to the total weight of the constraints inducing them.

Whenever the algorithm of Aho *et al.* is stuck at a non-divisible subset S of the set of leaves and has to return a null tree, Heuristic 2 simply finds a minimum weight edge cut of the auxiliary graph $G(S)$ (with respect to the current set of constraints). Next, the edges of the min-cut are deleted from $G(S)$ and the connected components of $G(S)$ are computed. Consequently, the constraints corresponding to the edges of the min-cut are also deleted. Finally, the approximate consensus trees for the connected components are recursively computed and connected by a common parent node.

By using recent dynamic data structures for graph connectivity, Henzinger *et al.* gave efficient implementations of the algorithm of Aho *et al.* restricted to

constraints of the form $(\{i,j\},k)$ [28]. Their randomized implementation takes $O(m \log^3 n)$ expected time. They use an undirected graph U and a directed graph D defined as follows.

- $U = (V, E)$ with V equal to the set of leaves $\{1, 2, \ldots, n\}$, and where for each constraint $(\{a,b\},c)$ in C, edges $\{a,b\}$ and $\{b,c\}$ are in E.
- $D = (V', A)$, where for each constraint $(\{a,b\},c)$ in C, nodes $\{a,b\}$ and $\{b,c\}$ are in V', and the directed edge $\{a,b\} \to \{b,c\}$ is in A.

At the beginning, the graph U is colored yellow. U is used for finding yellow components. The consensus tree returned is found by combining the trees constructed for the yellow components. The graph D is used for finding edges in U that can be colored red; these edges correspond to the so-called maximal nodes in D.

A *maximal node* in D is a node with no outgoing edges, and a red edge whose endpoints are in different yellow components is called a *separable red edge*.

By slightly modifying the algorithm of Henzinger *et al.* and combining it with an algorithm for minimum weight edge cut [35], we can implement Heuristic 2 as follows.

Heuristic 2

1. Construct U and D. Add weights to the edges in U. The weight of an edge $\{a,b\}$ in U is equal to the sum of the weights of constraints of the form $(\{a,b\},\cdot)$. In particular, if for an edge in U no constraint of this form occurs, the weight of $\{a,b\}$ is zero. Color all nodes in D and edges in U yellow.

2. Identify maximal nodes in D. Recolor these nodes and the corresponding edges of U red.

3. If U has no edges, then return the consensus tree T with a root and all nodes in U children of the root. Otherwise, compute yellow components of U. If there is only one yellow component then find a minimum weight edge cut, delete the edges in the cut from U and the corresponding nodes from D, and recompute the yellow components. Let $\mathcal{C}_1, \ldots, \mathcal{C}_k$ be the current yellow components. Form a tree T by creating the root of T and connecting the root of the consensus trees recursively computed for the components $\mathcal{C}_1, \ldots, \mathcal{C}_k$ to it. For each current yellow component, identify the set E_{sep} of separable red edges incident to that new component. Delete these edges from U and the corresponding nodes from D. Go to Step 2.

Lemma 5.9 *Heuristic 2 can be implemented to run in* $O(n^3 \log n + m \log^3 n)$ *expected time.*

Proof: A minimum weight edge cut can be computed with high probability in time $O(n^2 \log n)$ [35]. In the worst case it has to be done n times; hence, the calls to minimum weight edge cut take a total of $O(n^3 \log n)$ expected time. All other operations can be performed in expected time $O(m \log^3 n)$ like in the algorithm of Henzinger *et al.* Thus, the total expected time is $O(n^3 \log n + m \log^3 n)$. □

Lemma 5.10 *Let* I *be an instance of MICT, and let* T *be the tree produced by Heuristic 2 for* I. *The total weight of constraints in* I *not satisfied by* T *is at most* $height(T)$ *times the minimum.*

Proof: Let J be a subset of I of minimum total weight such that $I \setminus J$ has a consensus tree. Next, let D be the set of connected components in the auxiliary graph where edges corresponding to the constraints in J are deleted. Suppose that Heuristic 2 at some stage finds a min-cut in a currently connected fragment C. Note that, C cannot be a subset of a simple component in D since then there wouldn't exist a consensus tree for $I \setminus J$. Hence, there is a subset J_C of J such that the set of edges corresponding to the constraints in J_C disconnects $G(C)$ into disjoint components. Clearly, the total weight of J_C is not smaller than the weight of a min-cut of $G(C)$. Now, it is sufficient to observe that the subsets J_C for distinct C's on the same recursion level of Heuristic 2 are pairwise disjoint. □

Theorem 5.11 *Let* n, w, t *be respectively the number of leaves, the total weight of constraints, and the minimum total weight of the constraints to remove in an instance* I *of MILCT. Heuristic 2 constructs a consensus tree for a subset of the constraints in* I *whose total weight is not smaller than* $w - nt$.

Note that the number of constraints in I might be cubic in n and that Heuristic 2 yields a better approximation factor than Heuristic 1 for MILCT whenever $t < \frac{2w}{3n}$.

Chapter 6

Ordered Consensus Trees

In this chapter we study a special case of the local consensus tree problem, called the *ordered local consensus tree problem.*

Given a rooted tree T, leaf labeled by a set S of species. A *3-leaf constraint* on the lowest common ancestors of three species $i, j, k \in S$, denoted $(\{i, j\}, k)$, means that the lowest common ancestor of i and j in T is a proper descendant of the lowest common ancestor of i and k in T (and consequently also a proper descendant of the lowest common ancestor of j and k in T).

Ordered local consensus tree problem (OLCT)

Input: A set $C = \{c_1, \ldots, c_m\}$ of 3-leaf constraints on the lowest common ancestors of species from a set $S = \{s_1, \ldots, s_n\}$ and an ordering O of the elements in S.

Output: A tree T, leaf labeled by S, such that all constraints in C are consistent with T and the left-to-right ordering of the leaves in T equals O.

Also an optimization version of the problem, called *maximum ordered local consensus tree problem (MOLCT)*, is studied. Here, instead of a tree consistent with all input constraints, we want to find a tree consistent with a subset of the constraints of maximum cardinality, or if the constraints are weighted, with a subset of the constraints of maximum total weight.

For the unordered version of the local consensus tree problem Henzinger *et al.* gave an algorithm running in time $O(m\sqrt{n})$ and an $O(m \log^3 n)$ expected-time randomized algorithm [28]. In this chapter we give an $O((m + n) \log n)$ time algorithm for OLCT. The idea for the implementation comes from the observation that the basic dynamic graph used by Henzinger *et al.* becomes an *interval graph* if the tree under construction is required to have a special ordering

of the leaves. For interval graphs with known interval representation, efficient dynamic connectivity algorithms are easily implemented by the known efficient dynamic interval union algorithms [11]. As we need only a decremental interval union algorithm, and the known efficient fully dynamic interval union algorithms are rather complicated, we provide a very simple and practical, decremental interval union algorithm matching the time performance of that from [11] in the decremental case. The decremental interval union algorithm is described in Section 6.2 and the algorithm for OLCT, using the results from Section 6.2, can be found in Section 6.3.

In Section 6.4 we show that the optimization version, MOLCT, is solvable in cubic time, in contrast to the unordered version of the problem, which is NP-hard [30].

6.1 Preliminaries

Segment trees, interval trees and interval tries will be used in our algorithms.

The following facts about segment trees and interval trees are shown in [42].

A segment tree is a useful data structure for storing a set S of intervals on the real line R together with some additional information. The leaves in a segment tree are $(-\infty, x_1)$, $[x_1, x_1]$, (x_1, x_2), $[x_2, x_2]$, ..., (x_N, ∞), where x_i, $i = 1, 2, ..., N$, are the legal endpoints for the segments and $x_i < x_{i+1}$ for all $i = 1, ..., N-1$.

Each node u of a segment tree is augmented by a node list, denoted $NL(u)$. To describe NL we will use $xrange(u)$, defined as follows. First, $xrange(root) = (-\infty, +\infty]$. Let r be the right child and l the left child of node u. The split value at node u, denoted $split(u)$, is any value that is at least as large as the values at the leaves in the left subtree of u and smaller than the values at the leaves in the right subtree of u. Next, we define $xrange(r) = xrange(u) \cap (split(u), +\infty]$ and $xrange(l) = xrange(u) \cap (-\infty, split(u)]$. We set $NL(u)$ to $\{I \in S \mid xrange(u) \subseteq I \ \& \ \ xrange(parent(u)) \not\subseteq I\}$. Each interval can be stored in the nodelist in at most $O(\log N)$ nodes.

Fact 6.1 *[42] A segment tree for a set of n intervals with both endpoints in a universe $\mathcal{U} \subset R$ of size N can be constructed in time $O(n \log N)$. The depth of the tree is $O(\log N)$ and its size is $O(N)$. An interval with both endpoints in $\mathcal{U}$ can be inserted or deleted from the segment tree in time $O(g(n) \log N)$, where $g(n)$ is the time required to insert or delete an interval from a node list of size n.*

In our application $g(n) = O(1)$, since we shall use a node counter instead of a node list. The value $NC(u)$ of the node counter for a node u is just the cardi-

nality of $NL(u)$, i.e. deletion or insertion corresponds to decreasing or increasing $NC(u)$ by one.

An interval tree is a leaf-oriented binary search tree that supports intersection queries for a set S of closed intervals on the real line in logarithmic time.

Fact 6.2 *[42] Suppose that the left endpoints of the intervals in S belong to a subset of R of size N and $|S| = n$. An interval tree T of depth $O(\log N)$ for S can be constructed in time $O(N + n \log Nn)$ using space $O(n + N)$. The insertion into T or deletion from T of an interval with left endpoint in $\mathcal{U}$ takes time $O(\log n + \log N)$. The intersection query is supported by T in time $O(\log N + r)$, where r is the number of the reported intervals.*

The interval trie is another data structure for maintaining a set S of intervals, efficiently supporting stabbing queries. The result of a stabbing query is a list of all intervals containing the query point. The next fact, following from [43], characterizes interval tries.

Fact 6.3 *[43] Let S be a set of n intervals with endpoints in $\{1, ..., N\}$ and let f be a function mapping natural numbers into natural numbers greater than 1. The interval trie data structure performs insertions and deletions of intervals in S in time $O(f(N) + \log \log N)$. The stabbing query can be answered in time $O(\frac{\log N}{\log f(N)} + r)$ where r is the number of reported component intervals. The total space required is $O(N \frac{\log N}{\log f(N)} + nf(N))$.*

6.2 A Simple Decremental Interval Union Algorithm

Here we consider the problem of maintaining the union of the set S of intervals under a sequence of interval deletions. Denote by n the number of intervals in S to start with and let $s_1, \ldots, s_m$ be the sequence of intervals to delete from S. We assume that at the beginning all intervals $s_1, \ldots, s_m$ are in S.

The first crucial step of our method consists of the construction of a special balanced segment tree T for S. Let $e_1, ..., e_l$ be the consecutive endpoints of the intervals in S. The consecutive leaves in T are $(-\infty, e_1)$, $[e_1, e_1]$, (e_1, e_2), $[e_2, e_2]$, ..., (e_l, ∞). To each node u of T we associate the node counter, $NC(u)$, instead of the standard node list, $NL(u)$. Recall that the value of $NC(u)$ is just the cardinality of $NL(u)$. Note that $l \leq 2n$. Hence, the segment tree T can be constructed in time $O(n \log n)$ by Fact 6.1.

Additionally, for each node u of T, we set a special bit $r(u)$ to 1 if and only if the node counters along the path from u to the root of T, including u and the

root of T, are all set to zero. Clearly, we can determine $r(u)$ for all nodes of T by traversing T in a top-down fashion in time proportional to the size of T, i.e. $O(n)$ time.

Remark 6.4 *There exist leaves l_1, l_2 and l_3, ordered from left to right in T, such that $r(l_1) = 0$, $r(l_2) = 1$, and $r(l_3) = 0$ if and only if the union of S is disconnected.*

Finally, we form a linear array A, to store the consecutive leaves of T, in linear time.

Besides the segment tree T, we initialize an interval tree P in order to maintain the component intervals in the union of the current set S. We can find the component intervals of the initial S, e.g. by using a standard sweep-line technique, in time $O(n \log n)$. Next, we can construct P on $O(n)$ leaves determined by the endpoints of the intervals in S and insert the initial component intervals in time $O(n \log n)$ by Fact 6.2.

To find the component intervals of the set resulting from the deletion of the interval s_1 from S, we proceed as follows.

To begin with, we find the component interval ci_1 that s_1 belongs to in time $O(\log n)$ by querying P with s_1.

Next, we find the $O(\log n)$ nodes u of T whose node lists would contain s_1, if we were keeping them, and decrease $NC(u)$ by one. This can be done in time $O(\log n)$ by Fact 6.1. If $NC(u)$ drops to 0 and $r(parent(u)) = 1$ then we set $r(u)$ to 1 and update $r(w)$ for the descendants w of u in a top-down and left-right fashion. We also produce the list $LL(u)$ of consecutive leaf-descendants w of u for which $r(w)$ become 1 due to the updating ($LL(u)$ can be empty). Further, we concatenate the lists $LL(u)$ in the order of the ranges of nodes u. Let L_1 denote the resulting list. Note that the updating of $r(w)$'s as well as the construction of the list L_1 takes time proportional to the number r_1 of nodes w that got $r(w)$ set to 1 for the first time.

Now the following observation is crucial: For any pair of consecutive leaves a, b in L_1 that are not neighbors in the array A of consecutive leaves of T, the right neighbor of a in A, denoted $rn(a)$, and the left neighbor of b in A, denoted $ln(b)$, yield the new component interval. Also, if the left endpoint of the old component interval ci_1, that s_1 belonged to, is not on L_1, then it together with the left neighbor $ln(a)$ of the first leaf on L_1 yields another new component interval. Analogously, the right neighbor $rn(z)$ of the last leaf on L_1 together with the right endpoint of ic_1 might yield yet another new component interval. Hence, we need to insert the new component intervals into P and delete ci_1 from P. Any of the aforementioned insertions and deletions take $O(\log n)$ time.

We conclude that we can update T and P in order to maintain the union of the set of intervals resulting from the deletion of s_1 from S in time $O((k+1)\log n + r_1)$, where k is the number of the new component intervals.

The next deletions d_i, $i = 2, ..., m$, are handled similarly. We define r_i for $i = 2, ..., m$ in the same way as we defined r_1 above. It follows that the sum $\sum_{i=1}^{m} r_i$ is bounded from above by the size of T, i.e. $O(n)$.

Summarizing, we obtain our main result in this section.

Theorem 6.5 *Let S be a set consisting of n closed intervals on the real line. After $O(n\log n)$ time preprocessing, the union of S can be maintained under a sequence of m interval deletions in time $O(m\log n + n + k\log n)$, where k is the final number of component intervals. After each deletion the newly created component intervals can be listed without increasing the asymptotic complexity. The intersection query can be answered in time $O(\log n + r)$, where r is the number of component intervals to report.*

If we need only a stabbing query and the endpoints of the intervals belong to $\{1, ..., N\}$ for a small N, we can improve our result using an interval trie instead of the interval tree P. By Fact 6.3, we obtain the following variant of the result.

Theorem 6.6 *Let S be a set of n closed intervals with endpoints in $\{1, ..., N\}$ and let $f(N)$ be an integer greater than 1. After $O(n\log n + N\frac{\log N}{\log f(N)} + nf(N))$-time preprocessing the union of S can be maintained under a sequence of m interval deletions in time $O(m\log n + n + k(f(N) + \log\log N))$, where k is the final number of component intervals. After each deletion the newly created component intervals can be listed without increasing the asymptotic complexity. The stabbing query can be answered in time $O(\frac{\log N}{\log f(N)} + r)$ where r is the number of reported intervals. The total space required is $O(N\frac{\log N}{\log f(N)} + nf(N))$.*

Our simple algorithm for maintaining the union of a set S of intervals under a sequence of interval deletions can also be interpreted (and will be used in the next section) as an algorithm for the connected components of the interval graph induced by S under a sequence of vertex deletions.

If we have an interval graph G and do not have its interval representation S, we can form it in time proportional to the size of G, e.g. by ordering the maximal cliques of G so that for each vertex v of G the maximal cliques containing v are consecutive [23, 37].

6.3 Efficient Construction of an Ordered Consensus Tree

A binary rooted tree T_i on $O(1)$ leaves in the local consensus tree problem corresponds to $O(1)$ 3-leaf constraints of the form $(\{i,j\},k)$ on the consensus tree T, where $(\{i,j\},k)$ implies that the lowest common ancestor of the leaves i and j has to be a proper descendant of the lowest common ancestor of the leaves i and k [49]. For this reason, we can consider the problem of inferring a local consensus tree from a set of 3-leaf constraints, instead of the local consensus tree problem.

By using recent dynamic data structures for graph connectivity, Henzinger *et al.* provide efficient algorithms for the aforementioned problem of inferring a local consensus tree. Their deterministic algorithm takes $O(m\sqrt{n})$ time whereas their randomized algorithm takes $O(m \log^3 n)$ expected-time, where m is the number of 3-leaf constraints on leaf-labels in $\{1,\ 2, ..., n\}$. In both cases, they use an undirected graph U and a directed graph D defined as follows.

- $U = (V, E)$ with V equal to the set of leaves $\{1, 2, ..., n\}$, and where for each constraint $(\{a,b\},c)$ in C, edges $\{a,b\}$ and $\{b,c\}$ are in E.

- $D = (V', A)$, where for each constraint $(\{a,b\},c)$ in C, nodes $\{a,b\}$ and $\{b,c\}$ are in V', and the directed edge $\{a,b\} \to \{b,c\}$ is in A.

The graph U is used for finding current (yellow in [28]) connected components. The consensus tree returned is found by combining the trees constructed for the current connected components. The graph D is used for finding edges in U that can be removed (colored red in [28]), these edges correspond to the so-called maximal nodes in D. A *maximal node* in D is a node with no outgoing edges.

Remark 6.7 *Whenever two leaves a and b are in the same current component of U, then not only a and b, but also all other leaves in $[a,b]$ have to be in the same subtree of the ordered consensus tree representing this component.*

By Remark 6.7, the vertices of U can be interpreted as segments inducing an interval graph. Hence, instead of computing the current components of U we can compute the corresponding components of the interval graph utilizing our efficient decremental interval union algorithm.

By combining this idea with that of the algorithm of Henzinger *et al.*, we can build an ordered consensus tree as follows.

Algorithm 1

1. $S \leftarrow \{[a,b] | (a,b) \in E\}$.
2. Construct D and initialize the interval trie Q on the universe $\{1,2,...,n\}$.
3. Identify maximal vertices in D. Move the corresponding intervals from S to Q.
4. If S is empty, then return the consensus tree O with a root and all nodes in U children of the root. Otherwise, compute the new component intervals of the union of S by using the decremental algorithm of Section 6.2. If there is no new component interval then return "no ordered consensus tree" and stop. Let $c_1, c_2, ..., c_k$ be the new component intervals in the order of their endpoints. For $i = 1, ..., k-1$, create the root of the ordered consensus tree to be computed for the component interval c_i and link it with that of the previous interval component including c_i, if such one exists. For $i = 1, ..., k-1$, if c_i and c_{i+1} are included in the same old component interval, then query Q with the median m_i of the right endpoint of c_i and the left endpoint c_{i+1}. Remove all reported (i.e. containing m_i) intervals from Q and the corresponding vertices from D. Go to Step 3.

Theorem 6.8 *For a set of m 3-leaf constraints on leaf-labels in $\{1,\ 2, ..., n\}$, we can decide whether there exists an ordered consensus tree, and if so, construct it, in time $O((m+n)\log n)$.*

Proof: If Algorithm 1 produces an ordered consensus tree then it is equivalent to the minimal tree produced by Aho *et al.* algorithm or the tree produced by the faster algorithm of Henzinger *et al.* (Algorithm 1 in [28]). To see this, note that the current component intervals of S correspond to the yellow components of the graph U in [28] and the intervals stored in Q correspond to the current red edges of U. Furthermore, an interval stored in Q corresponds to a red separable edge in [28] if and only if it overlaps with at least two new component intervals, i.e. if it is intersected by at least one of the medians $m_i,\ i = 1, ..., k-1$. If Algorithm 1 fails to produce an ordered consensus tree then there is no such a tree by Remark 6.4 and the correctness of the algorithms of Henzinger *et al.*

Analogously as in the algorithm of Henzinger *et al.*, the construction of the graph D takes time $O(m)$ and the overall time taken by determining the maximal vertices is proportional to the size of D, i.e. $O(m)$. The total time-cost of recomputing the component intervals including the preprocessing is $O((m+n)\log n)$ by Theorem 1. The total cost of the $O(m)$ insertions from and deletions into Q, and the $O(n)$ stabbing queries for Q is $O((m+n)\frac{\log n}{\log\log n})$ by Fact 6.3 for $f(n) = \Theta(\frac{\log n}{\log\log n})$. □

6.4 A Cubic-Time Algorithm for the Optimization Problem

Unfortunately, it is often impossible to construct an exact consensus tree. Therefore, the optimization version of the problem of inferring a local consensus tree in which the objective is to find a consensus tree for an as large as possible subset of an input set of 3-leaf constraints was considered in [22]. Approximation algorithms for this optimization problem were provided in Chapter 5, and the problem was recently shown to be NP-hard [30]. Here we show that the problem of finding an *ordered* consensus tree for a subset of the input constraints of maximum cardinality, is solvable, by dynamic programming, in cubic time.

Theorem 6.9 *For a set of m 3-leaf constraints on leaf-labels in $\{1, 2, ..., n\}$, we can find an ordered consensus tree for the maximum number of the constraints in time $O(n^3)$.*

Proof: For each pair of i, j in $\{1, 2, ..., n\}$, we recursively construct the ordered consensus tree $T_{i,j}$ for the maximum number $m_{i,j}$ of the input constraints on leaf-labels in $\{i, i+1, ..., j\}$. For $i = 1, 2, ..., n$, the tree $T_{i,i}$ simply consists of the leaf labeled with i and its parent which is the root of the tree. To determine a tree $T_{i,j}$ where $i < j$, we compute, for $k = i, ..., j-1$, the number $w_{i,k,j}$ of the input constraints $(\{l_1, l_2\}, l_3)$ where $\{l_1, l_2, l_3\} \subset \{i, ..., j\}$ and either $\{l_1, l_2\} \subset \{i, ..., k\}$ & $l_3 \in \{k+1, ..., j\}$ or $l_3 \in \{i, ..., k\}$ & $\{l_1, l_2\} \subset \{k+1, ..., j\}$. Next, we find a $k \in \{i, ..., j\}$ which maximizes the sum $m_{i,k} + m_{k+1,j} + w_{i,k,j}$ and form the tree $T_{i,j}$ by creating its root and making the roots of $T_{i,k}$ and $T_{k,j}$ its children. It follows by induction on $|j - i|$ that $m_{i,j} = m_{i,k} + m_{k+1,j} + w_{i,k,j}$, which proves the correctness of the construction of $T_{i,j}$. Assuming that the numbers $w_{i,k,j}$ are precomputed and the trees $T_{i',j'}$ where $j' - i' < j - i$ are already determined, the determination of $T_{i,j}$ takes $O(j - i + 1)$ time. The precomputation of all the numbers $w_{i,k,j}$, $i \leq k \leq j$, can be done by a single scan of the input constraint sequence in time $O(m + n^3)$, i.e. $O(n^3)$. The cubic upper time-bound follows. $\Box$

Part III

Maximum Homeomorphic Subtree Problem

Maximum Homeomorphic Subtree Problem

The *maximum agreement subtree problem* (MHT) can be considered to be a consensus method as described in Part II. Like in consensus methods, we are given a set of alternative evolutionary trees describing possible evolutions for sets of species, and the goal is to determine a single tree based on all the input trees. More exactly, we are given a set of leaf labeled tree, all for the same set of labels (species), and we want to identify a subtree contained within every given tree such that the number of leaves labeled by species is maximized.

There is no method for constructing evolutionary trees that is believed to be the correct one or the optimal one. Different methods with different optimization criteria are used, which result in different trees. Also different data sets give different trees. There is thus a need for a way to resolve differences and to compare different methods. MHT can be seen as way to compare methods of tree construction. Depending on if the trees computed using different methods give large or small MHT we may be able draw conclusions about the methods, like how similar the methods are and how reliable trees they produce. MHT can also be seen as a way to increase the confidence for the result. The intersection of the trees should have a larger probability to be true then each of the input trees by themselves. In [12] also an application to automatic translation between languages is mentioned.

The maximum homeomorphic subtree problem is defined as follows. Given a set of k rooted trees $T_1,\ T_2, \ldots,\ T_k$, each with n leaves labeled distinctly with elements chosen from a set A of cardinality n, find a maximum cardinality subset B of A such that the minimal homeomorphic subtrees of $T_1,\ T_2, \ldots,\ T_k$ restricted to B are all isomorphic. See Figure 6.1 for an example.

This problem is frequently also called *maximum agreement subtree problem*, especially when the number of trees is equal to 2. This special case is studied

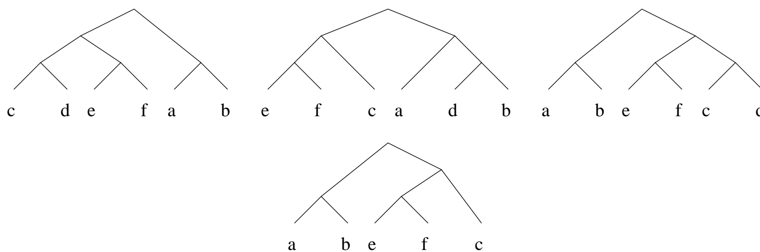

Figure 6.1: Three binary input trees with six species yield a maximum agreement subtree with five species.

more than the general problem since it is solvable in polynomial time, in contrast to the former problem, which is known to be NP-hard.

The first algorithm for MHT for two binary trees was a heuristic algorithm with running time $O(n^5)$ invented by Gordon and Finden [19]. Kubicka *et al.* gave an superpolynomial algorithm for the same problem, running in time $O(n^{1/2+\epsilon \log n})$ [38]. The first polynomial time algorithms are due to Steel and Warnow [50]. Their algorithm for two trees with arbitrary degree runs in time $O(n^{4.5} \log n)$, and for two trees of maximum degree d they gave algorithms with running time $O(d!n^2)$ and $O(d^{2.5} n^2 \log n)$. Their algorithms work equally well for rooted as unrooted trees. After this several papers have presented improved algorithms for MHT for two trees. Farach and Thorup first gave an algorithm running in time $O(n^{2+o(1)})$ for unrooted trees and in time $O(n^2)$ for rooted trees of arbitrary degree [17]. They soon after improved this result by giving an $O(n^{1.5} \log n)$ algorithm for general degree trees and an $O(n^{1+o(1)})$ algorithm for trees with bounded degree [18]. An $O(n\sqrt{d} \log^3 n)$ algorithm was presented by Farach *et al.*, in particular running in time $O(n \log^3 n)$ time for binary trees [15]. The same technique was used by Cole *et al.* in their $O(n \log n)$ time algorithm for two rooted binary trees [12]. In between this two results Kao presented an $O(n \log^2 n)$ algorithm for two trees of constant degree, running in time $\min(O(nd^2 \log d \log^2 n), O(nd\sqrt{d} \log^3 n))$ for trees of degree d [32]. Kao *et al.* showed that also for two unrooted trees of constant degree MHT can be computed in $O(n \log n)$ time [33].

The known results for MHT for k trees begun with the paper by Amir and Keselman [3]. They showed that MHT is NP-complete for three trees of unbounded degree. They also gave an algorithm running in time $O(kn^{d+1} n^{2d})$ for the problem and an approximation algorithm which returns a tree with the number of removed species at most 4 times the minimum number of species necessary to remove. Farach *et al.* improved the running time of the construction algorithm to $O(kn^3 + n^d)$ [16]. Hein *et al.* showed that MHT can not be approximated within a factor $2^{\log^{\delta} n}$ for any $\delta < 1$ unless NP$\subseteq$DTIME$[2^{polylog(n)}]$, even for $k = 3$ [27].

Hein *et al.* also looked at a related problem, which they called MAST-EC, maximum agreement subtree with edge contractions. In this problem an agreement subtree with the maximum number of edges is sought, while all species are still in the tree, but edge contractions are made. They showed that this problem is also NP-complete [27].

Another related problem is the largest common subtree problem, LCST, where a common connected vertex induced subtree, of maximum size, for a set of vertex labeled trees is sought. The complexity of this problem is similar to that for MHT. LCST is solvable in polynomial time for two trees and for a fixed

number of trees of bounded degree, while it is NP-hard for three trees in general and it is also hard to approximate, see e.g. [2].

In Chapter 7 we present two new result for MHT. The first is an inapproximability result that says that MHT can not be approximated within a factor N^ϵ for any $0 \leq \epsilon < \frac{1}{9}$ in polynomial time, unless P=NP. Here N is the total size of all input trees. This result is valid even if all input trees are of height two. The second result is an approximation algorithm and we show that MHT for instances with $O(1)$ trees of height $O(1)$ can be approximated within a constant factor in time $O(n \log n)$.

Chapter 7

Complexity of Maximum Homeomorphic Subtree Problem

Given a set of k rooted trees $T_1, T_2, \ldots, T_k$, each with n leaves labeled distinctly with elements chosen from a set A of cardinality n, the *maximum homeomorphic subtree problem (MHT)* asks for a maximum cardinality subset B of A such that the minimal homeomorphic subtrees of $T_1, T_2, \ldots, T_k$ restricted to B are all isomorphic.

In this chapter we study the approximability of this problem. In Section 7.1 we show that MHT is hard to approximate even if the trees are restricted to have height three. Let N denote the total total size of all input trees, i.e. $N = \Theta(nk)$, then MHT can not be approximated within a factor N^ϵ for any $0 \leq \epsilon < \frac{1}{9}$, unless P=NP. This is shown by making a reduction from maximum independent set, which has been shown by Håstad to be hard to approximate within a factor $l^{1/3-\delta}$ for any $\delta > 0$, where l is the number of vertices in the graph [26]. The recent result, by Engebretsen and Holmerin, that max clique, and hence also maximum independent set, cannot be approximated in polynomial time within a factor $n^{1-O(1/\sqrt{\log\log n})}$, unless NP$\subseteq$ZPTIME$(2^{O(\log n(\log n \log\log n)^{3/2})})$ [14], can be used to improve our result accordingly. The second result, presented in Section 7.2, shows that although it is hard to approximate MHT for trees of constant height, and also for a constant number of trees [27], it can be approximated within a constant factor in polynomial time, when we have a constant number of trees of constant height. Let h denote the maximum height in the given trees, then the approximation factor is k^h and the algorithm runs in time $O(n \log n)$.

7.1 MHT is Hard to Approximate

Our main result in this section is the following theorem.

Theorem 7.1 *For any $0 \leq \epsilon < \frac{1}{9}$, MHT, even if restricted to trees of height 2, cannot be approximated within a factor of N^ϵ in polynomial time, unless P=NP.*

Proof: First, we describe a reduction from the maximum independent set problem to MHT. Next, we show that if MHT can be approximated within a factor of N^ϵ in polynomial time then the problem of finding a maximum independent set in a graph with l nodes can be approximated within a factor of $l^{3\epsilon+o(1)}$. Finally, we apply known results about the inapproximability of the maximum independent set problem to get our result. In part, our reduction can be seen as a generalization of the reduction of three-dimensional perfect matching to MHT restricted to instances with three trees used in [3].

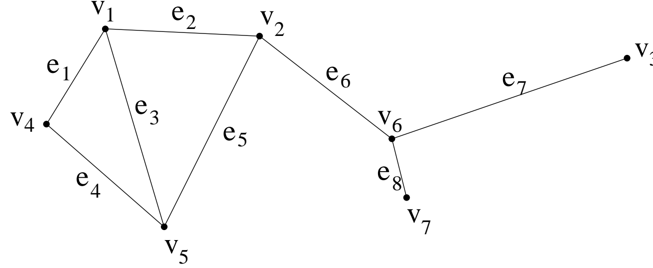

Figure 7.1: An instance of the maximum independent set problem with $l = 7$ and $k = 8$.

Let $G = (V, E)$ be a graph where $V = \{v_1, \ldots, v_l\}$ and $E = \{e_1, \ldots, e_k\}$ with $k > 1$. Construct k rooted trees $T_1, \ldots, T_k$ on $l + q$ labeled leaves that contain all the adjacency information about the nodes of G as follows. For each edge $e_i = (v_a, v_b) \in E$, build a rooted tree T_i on the set of leaves labeled by $w_1, \ldots, w_l, w_{l+1}, \ldots, w_{l+q}$. Let the root r_i of T_i be the parent of $(l-1)+q$ children, where the first child ("the non-leaf child") is a node with two children leaves labeled w_a and w_b, and the remaining children of r_i are leaves labeled by the elements in $\{w_j \mid 1 \leq j \leq l+q \text{ and } j \notin \{a, b\}\}$. Thus, r_i has exactly one pair of grandchildren, and we write $GC(T_i) = \{w_a, w_b\}$.

Now, let T be a maximum homeomorphic agreement subtree of $T_1, \ldots, T_k$. We choose q large enough to guarantee that each of the roots $r_1, \ldots, r_k$ will correspond to the root r of T. Actually, $q = 2$ is sufficient. (To see this, assume that the non-leaf child of r_i turned out to be the root for some i. By the construction above, all non-leaf children have two leaf children, so the number

of leaves in this agreement subtree can be no larger than two. But we can always find an agreement subtree with three leaves by selecting r_i as root and including w_{l+1} and w_{l+2} in addition to the path from the root to a fixed leaf w_j, where $1 \leq j \leq l$.)

T has no non-leaf children because if it did, then there would exist some x and y such that for each i, where $1 \leq i \leq k$, $GC(T_i)$ would be equal to $\{w_x, w_y\}$. Consequently, G would only have one edge, which contradicts $k > 1$. T's children are $m+q$ $(= m+2)$ leaves labeled $w_{\mu_1}, w_{\mu_2}, \ldots, w_{\mu_m}, w_{l+1}, w_{l+2}$. If v_a is adjacent to v_b in G then at most one of w_a and w_b can be a child of T. Otherwise, $GC(T_i)$ wouldn't be equal to $\{w_a, w_b\}$ for any T_i, and $e_i \neq (v_a, v_b)$ would hold for all i, contradicting the adjacency of v_a and v_b in G. Thus, the nodes $v_{\mu_1}, v_{\mu_2}, \ldots, v_{\mu_m}$ form an independent set in G. Conversely, given an independent set I of nodes in G, we can easily construct an agreement subtree T_I in the form of a rooted tree with $|I| + 2$ leaves uniquely labeled with w_j, where $v_j \in I$, and $w_{l+1},\ w_{l+2}$. By the maximality of T, m equals the cardinality of the maximum independent set of G. Thus, an algorithm for MHT would immediately imply an algorithm for the maximum independent set problem. See Figures 7.1–7.3 for an example of the reduction.

The total size N of the trees $T_1, \ldots,\ T_k$ is $k \cdot O(l) = O(l^3) = l^{3+o(1)}$. Clearly, they can be constructed from G in polynomial time. Also, note that they are of height 2. Below, we only consider approximations that can be carried out in polynomial time. If MHT could be approximated within a factor of N^ϵ, then $\frac{OPT+2}{s+2} \leq N^\epsilon$, where $OPT + 2$ refers to the number of leaves in an optimal solution for a given instance of MHT and $s + 2$ is the number of leaves in its corresponding, approximative solution. For $s \geq 1$, it follows that $\frac{OPT}{s} \leq 3 \cdot \frac{OPT+2}{s+2} \leq 3N^\epsilon = l^{3\epsilon+o(1)}$, which would imply that the problem of finding a maximum independent set in a graph could be approximated within a factor of $l^{3\epsilon+o(1)}$. However, Håstad proved in [26] that this problem is not approximable within $l^{1/3-\delta}$ for any $\delta > 0$, unless P=NP. Hence, if P $\neq$ NP, MHT cannot be approximated within a factor of N^ϵ for any $0 \leq \epsilon \leq \frac{1}{9} - o(1)$. Finally, since $\frac{1}{9} - o(1)$ can be made arbitrarily close to $\frac{1}{9}$ by choosing N large enough, there exist instances of MHT which cannot be approximated within a factor of N^ϵ for any constant $0 \leq \epsilon < \frac{1}{9}$ in polynomial time (unless P=NP). □

7.2 Approximations of MHT with $O(1)$ Trees of Height $O(1)$

We know that MHT is hard to approximate, both for instances with three trees [27] and for instances with an arbitrary number of trees of height 2 or

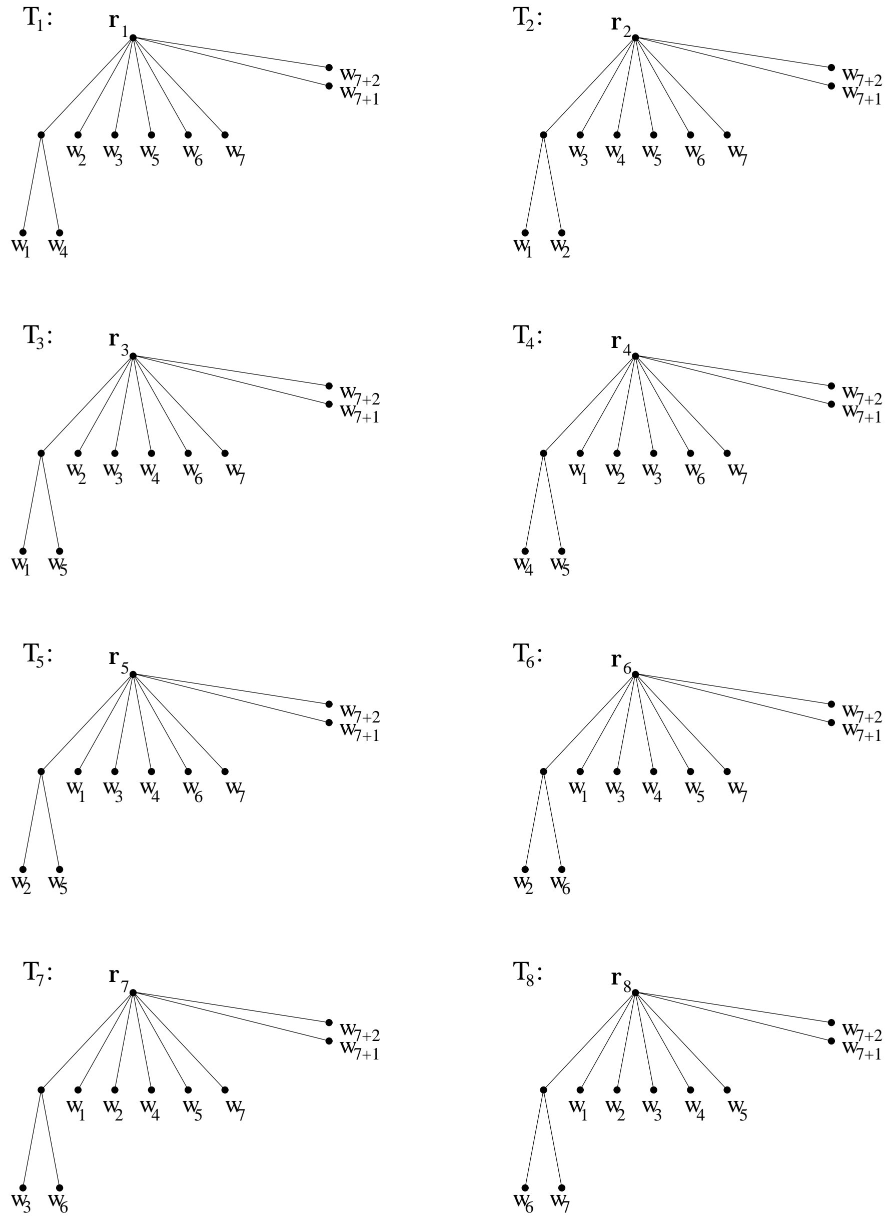

Figure 7.2: The trees T_i corresponding to the graph in Figure 7.1.

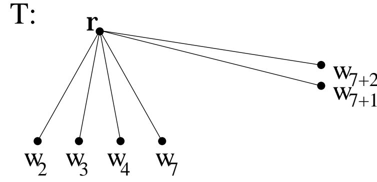

Figure 7.3: The maximum homeomorphic agreement subtree of $T_1, \ldots, T_8$ tells us that $\{v_2, v_3, v_4, v_7\}$ is a maximum independent set of the graph in Figure 7.1.

more by Theorem 7.1. The natural question arises whether or not MHT for instances with a bounded number of trees, each one of bounded height, can be tightly approximated in polynomial time. The following result, together with Theorem 7.1, yields a characterization of the approximability of MHT restricted to instances with trees of $O(1)$ height.

Theorem 7.2 *MHT restricted to instances with k trees of height not exceeding h can be approximated within a factor of k^h in time $O(n \log n)$.*

To begin the proof of Theorem 7.2, we need to introduce the following notation. For a tree T, $V(T)$ stands for the set of nodes of T. Let v be a node of a rooted tree T. The minimal subtree of T rooted at v, including v and all its descendants is denoted by T_v. $L(T_v)$ stands for the set of labels of the leaves in T_v. The set of children of v in T is denoted by $C(v)$. Furthermore, by a *k-partite hypergraph* H we shall mean a pair $(V_1 \cup \cdots \cup V_k,\ E)$, where V_1 through V_k are pairwise disjoint sets and E is a subset of $V_1 \times \cdots \times V_k$. The elements of $V_1 \cup \cdots \cup V_k$ are called the *nodes* of H whereas the elements of E are called the *edges* of H. A *matching* of H is a subset of E in which no pair of edges includes a common node.

Let $T_1, \ldots, T_k$ be the input trees. For $(v_1, \ldots, v_k)$, where $v_i \in V(T_i)$ for $i = 1, \ldots, k$, let $Mht(v_1, \ldots, v_k)$ denote the maximum size of an agreement subtree of the trees $T_1, \ldots, T_k$ restricted to $B = L((T_1)_{v_1}) \cap \cdots \cap L((T_k)_{v_k})$. We can view $Mht(v_1, \ldots, v_k)$ as the solution of MHT for $(T_1)_{v_1}, \ldots, (T_k)_{v_k}$. Next, let $H(v_1, \ldots, v_k)$ denote the k-partite hypergraph $(C(v_1) \cup \cdots \cup C(v_k), C(v_1) \times \cdots \times C(v_k))$ whose edges $(w_1, \ldots, w_k)$ have weight $Mht(w_1, \ldots, w_k)$. Finally, let $Match(v_1, \ldots, v_k)$ be the maximum weight of a matching in the hypergraph $H(v_1, \ldots, v_k)$ and let $Diag(v_1, \ldots, v_k) = \max\{Mht(w_1, \ldots, w_k) \mid (w_1, \ldots, w_k) \in (\{v_1\} \cup C(v_1)) \times \cdots \times (\{v_k\} \cup C(v_k)) - \{(v_1, \ldots, v_k)\}\}$.

Intuitively, in the final agreement subtree of $(T_1)_{v_1}, \ldots, (T_k)_{v_k}$ either the roots of the trees, i.e., v_1 through v_k, are matched together which forces their children to be optimally matched together (*Match*), or only some of the roots

are matched together with some children of the remaining roots (*Diag*). This yields the following lemma which is a straightforward generalization of the basic lemma in the dynamic programming approach to MAST in [50].

Lemma 7.3 *For any* $(v_1, \ldots, v_k)$, *where* $v_i \in V(T_i)$ *for* $i = 1, \ldots, k$, *if at least one of the* v_i*'s is a leaf then* $Mht(v_1, \ldots, v_k) = |L((T_1)_{v_1}) \cap \cdots \cap L((T_k)_{v_k})|$, *else* $Mht(v_1, \ldots, v_k) = \max\{Match(v_1, \ldots, v_k), Diag(v_1, \ldots, v_k)\}$.

The recursive computation of $Mht(v_1, \ldots, v_k)$ for $(v_1, \ldots, v_k) \in V(T_1) \times \cdots \times V(T_k)$ in the lemma above can be bottom-up ordered by $Hs(v_1, \ldots, v_k) = \sum_{i=1}^{k} height((T_i)_{v_i})$. Hence, we have the following algorithm for MHT.

Algorithm 1

1. input $T_1, \ldots, T_k$
2. for each $(v_1, \ldots, v_k) \in V(T_1) \times \cdots \times V(T_k)$, in increasing order of $Hs(v_1, \ldots, v_k)$
   do compute $Mht(v_1, \ldots, v_k)$ by using the expression in Lemma 7.3.
3. output $Mht(r_1, \ldots, r_k)$, where r_i is the root of T_i for $i = 1, \ldots, k$.

It is hard to compute the exact value of $Match(v_1, \ldots, v_k)$ in the expression of Lemma 7.3 since the problem of computing maximum matching in a 3-partite hypergraph is NP-complete [44]. For this reason, we shall rely on a greedy method for approximating $Match(v_1, \ldots, v_k)$ yielding an approximation of $Mht(v_1, \ldots, v_k)$. The greedy method consists of repeatedly picking the heaviest edge e and removing all edges overlapping e. It can be implemented easily using a priority queue. The edge e can overlap with at most k edges in an optimum solution, and since their total weight $\leq k \cdot weight(e)$, we obtain:

Lemma 7.4 *Let* $H = (V, E)$ *be a* k*-partite hypergraph on* m *edges with positive integer weights. A matching in* H *of total weight within a factor* k *of the maximum can be constructed in a greedy fashion in time* $O(k|E| + |V| + m \log m)$.

Interestingly, in the unweighted case there are known (much slower, but still) polynomial-time heuristics yielding solutions within almost $\frac{k}{2}$ of the optimum [29].

Combining Algorithm 1 with the greedy method for approximating the value of $Match(v_1, \ldots, v_k)$, we obtain the following lemma, yielding Theorem 7.2.

Lemma 7.5 *For all* $(v_1, \ldots, v_k) \in V(T_1) \times \cdots \times V(T_k)$, *we can approximate* $Mht(v_1, \ldots, v_k)$ *within a factor of* k^h, *where* $h = \max\{height((T_i)_{v_i}) \mid 1 \leq i \leq k\}$, *in* $O(n \log n)$ *time.*

Proof: For $(v_1, \ldots, v_k) \in V(T_1) \times \cdots \times V(T_k)$, let $s(v_1, \ldots, v_k)$ denote the size of the intersections $L((T_1)_{v_1}) \cap \cdots \cap L((T_k)_{v_k})$. Clearly, we have $Mht(v_1, \ldots, v_k) \leq s(v_1, \ldots, v_k)$, and in particular if one of the v_i's is a leaf then $Mht(v_1, \ldots, v_k) = s(v_1, \ldots, v_k)$. For a leaf label j, we determine all k-tuples $(v_1, \ldots, v_k)$ for which $j \in L((T_1)_{v_1}) \cap \cdots \cap L((T_k)_{v_k})$ by finding, in each T_i, $i = 1, \ldots, k$, the nodes on the path of length $\leq h$ from the leaf labeled j to the root. It follows that the number of these tuples is $(h+1)^k$. Consequently, the set L of all k-tuples for which $s(v_1, \ldots, v_k) > 0$ has size not exceeding $n(h+1)^k$. To list L efficiently, we sort the pointers to leaves in T_1 through T_k by the leaf labels. Such a sorted list of pointers can be produced in time $O(|V(T_1)| + \cdots + |V(T_k)|)$. Using it, we can generate L by finding appropriate tree paths in time $O(|V(T_1)| + \cdots + |V(T_k)| + (h+1)^k)$.

For the k-tuples $(v_1, \ldots, v_k)$ in the set L that include at least one leaf, we have $s(v_1, \ldots, v_k) = 1$ and $Mht(v_1, \ldots, v_k) = 1$. To compute approximations of $Mht(v_1, \ldots, v_k)$ for the remaining k-tuples in L, we build a balanced search tree S_L for L, with respect to the lexicographic order of k-tuples in $V(T_1) \times \cdots \times V(T_k)$, in time $O(|L| \log |L|)$. Next, we follow the scheme of Algorithm 1 using the greedy method to approximate $Match(v_1, \ldots, v_k)$ in the hypergraph $H_L(v_1, \ldots, v_k)$ which is the hypergraph $H(v_1, \ldots, v_k)$ defined in Lemma 7.3 restricted to edges in L.

Each k-tuple $(w_1, \ldots, w_k) \in L$ occurs at most once as an edge in the hypergraphs $H_L(v_1, \ldots, v_k)$ for $(v_1, \ldots, v_k) \in L$ (only when $w_i \in C(v_i)$ for $i = 1, \ldots, k$). Hence, the hypergraphs $H_L(v_1, \ldots, v_k)$ for $(w_1, \ldots, w_k) \in L$, have no more than $|L|$ edges totally and can be constructed (without weights) by scanning L and using S_L in total time $O(|L| \log |L|)$. Clearly, each $H_L(v_1, \ldots, v_k)$ has at most $s(v_1, \ldots, v_k)$ edges with positive weights. For each of its edges $(w_1, \ldots, w_k)$, we have $Hs(w_1, \ldots, w_k) = Hs(v_1, \ldots, v_k) - k$ and $\max\{height(w_i) | 1 \leq i \leq k\} = h - 1$. Hence, we may inductively assume that we have already k^{h-1} approximations of $Mht(w_1, \ldots, w_k)$, i.e., of the weights of $(w_1, \ldots, w_k)$ in the hypergraph. Consequently, we obtain an approximation of $Match(v_1, \ldots, v_k)$ within a factor of k^h by applying the greedy method. Due to Lemma 7.4, the total time complexity of the greedy method is bounded by $O(k|L| + (\sum_{(v_1, \ldots, v_k) \in L} s(v_1, \ldots, v_k)) \log n)$. By induction on $Hs(v_1, \ldots, v_k)$, we obtain an approximation of $Diag(v_1, \ldots, v_k)$ within a factor of k^h by considering solely k-tuples $(w_1, \ldots, w_k)$ in $L \cap ((\{v_1\} \cup C(v_1)) \times \cdots \times (\{v_k\} \cup C(v_k)) - \{(v_1, \ldots, v_k)\})$. Each $(w_1, \ldots, w_k) \in L$ can contribute to the value of $Diag(v_1, \ldots, v_k)$ for at most $2^k - 1$ k-tuples $(v_1, \ldots, v_k) \in L$. Hence, the total size of the subsets of L contributing to $Diag(v_1, \ldots, v_k)$ over all $(v_1, \ldots, v_k) \in L$, and consequently the total cost of finding maxima of Mht-approximations over these subsets, is $O(2^k |L|)$. We can build these subsets, again by scanning L and using S_L, in total time $O(2^k |L| \log |L|)$.

Each of the trees T_1 through T_k has size not exceeding $2n$ by its binarity. Hence, by $|L| \leq (h+1)^k n$, $\sum_{(v_1,\ldots,v_k)\in L} s(v_1,\ldots,v_k) \leq (h+1)^k n$, $h = O(1)$, $k = O(1)$, and straightforward calculations, we obtain the $O(n \log n)$ bound. $\square$

Bibliography

[1] A.V. Aho, Y. Sagiv, T.G. Szymanski, and J.D. Ullman. Inferring a tree from lowest common ancestors with an application to the optimization of relational expressions. *SIAM Journal on Computing*, 10:405–421, 1981.

[2] T. Akutsu and M.M. Halldórsson. On the approximation of largest common subtrees and largest common point sets. *Theoretical Computer Science*, 233:33–50, 2000.

[3] A. Amir and D. Keselman. Maximum agreement subtree in a set of evolutionary trees: Metrics and efficient algorithms. *SIAM Journal on Computing*, 26(6):1656–1669, 1997.

[4] A. Andersson. Improving partial rebuilding by using simple balance criteria. In *Proceedings of Algorithms and Data Structures, Workshop, WADS'89*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer-Verlag, 1989.

[5] A. Andersson and T.W. Lai. Fast updating of well-balanced trees. In *Proceedings of the 2nd Scandinavian Workshop on Algorithm Theory, SWAT'90*, volume 447 of *Lecture Notes in Computer Science*, pages 111–121. Springer-Verlag, 1990.

[6] J.-P. Barthélemy and F.R. McMorris. The median procedure for n-trees. *Journal of Classification*, 3:329–334, 1986.

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

[8] 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'96)*, 3(4):337–351, 1996.

[9] G.S. Brodal, R. Fagerberg, C.N.S. Pedersen, and A. Östlin. The complexity of constructing evolutionary trees using experiments. In *Proceedings of 28th Int. Colloquium on Automata, Languages and Programming, ICALP'01*. 2001. (to appear).

[10] D. Bryant and M. Steel. Fast algorithms for constructing optimal trees from quartets. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'99*, pages 147–155, 1999.

[11] S.W. Cheng and R. Janardan. Efficient maintenance of the union of intervals on a line, with applications. *Journal of Algorithms*, 12:57–74, 1991.

[12] R. Cole, M. Farach-Colton, R. Hariharan, T. Przytycka, and M. Thorup. An $O(n \log n)$ *algorithm for the maximum agreement subtree problem for binary trees.* SIAM Journal on Computing, *30(5):1385–1404, 2000.*

[13] W.H.E. Day. Optimal algorithms for comparing trees with labeled leaves. Journal of Classification, *2:7–28, 1985.*

[14] L. Engebretsen and J. Holmerin. Clique is hard to approximate within $n^{1-o(1)}$. *In* Proceedings of 27th Int. Colloquium on Automata, Languages and Programming, ICALP'00, *volume 1853 of* Lecture Notes in Computer Science, *pages 2–12. Springer-Verlag, 2000.*

[15] M. Farach, T.M. Przytycka, and M. Thorup. Computing the agreement of trees with bounded degrees. In Proceedings of the Third Annual European Symposium on Algorithms, ESA'95, *volume 979 of* Lecture Notes in Computer Science, *pages 381–393. Springer-Verlag, 1995.*

[16] M. Farach, T.M. Przytycka, and M. Thorup. On the agreement of many trees. Information Processing Letters, *55:297–301, 1995.*

[17] M. Farach and M. Thorup. Fast comparison of evolutionary trees. Information and Computation, *123:29–37, 1995.*

[18] M. Farach and M. Thorup. Sparse dynamic programming for evolutionary-tree comparison. SIAM Journal on Computing, *26(1):210–230, 1997.*

[19] C.R. Finden and A.D. Gordon. Obtaining common pruned trees. Journal of Classification, *2:255–276, 1985.*

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

[21] L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin. Inferring ordered trees from local constraints. In Proceedings of the 4th Australian Theory Symposium, CATS'98, pages 67–76, 1998.

[22] L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin. On the complexity of constructing evolutionary trees. Journal of Combinatorial Optimization, 3:183–197, 1999.

[23] M.C. Golumbic. Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York, 1980.

[24] D. Gusfield. Efficient algorithms for inferring evolutionary trees. Networks, 21:19–28, 1991.

[25] D. Gusfield. Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology. Cambridge university press, 1997.

[26] J. Håstad. Testing of the long code and hardness for clique. In Proceedings of the 28th Annual ACM Symposium on Theory of Computing, STOC'96, pages 11–19, 1996.

[27] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. Discrete Applied Mathematics, 71:153–169, 1996.

[28] M.R. Henzinger, V. King, and T.J. Warnow. Constructing a tree from homeomorphic subtrees, with application to computational evolutionary biology. Journal of Algorithms, 24:1–13, 1999.

[29] C.A.J. Hurkens and A. Schrijver. On the size of systems of sets every t of which have an sdr, with an application to the worst-case ratio of heuristics for packing problems. SIAM Journal of Discrete Mathematics, 2:68–72, 1989.

[30] J. Jansson. On the complexity of inferring rooted evolutionary trees. In Proceedings of the Brazilian Symposium on Graphs, Algorithms, and Combinatorics, GRACO'01, pages 121–125, 2001.

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

[32] M.Y. Kao. Tree contractions and evolutionary trees. SIAM Journal on Computing, 27(6):1592–1616, 1998.

[33] M.Y. Kao, T.W. Lam, T.M. Przytycka, W.K. Sung, and H.F. Ting. General techniques for comparing unrooted evolutionary trees. In Proceedings of the

29th Annual ACM Symposium on Theory of Computing, STOC'97, *pages 54–65, 1997.*

[34] *M.Y. Kao, A. Lingas, and A. Östlin. Balanced randomized tree splitting with applications to evolutionary tree constructions. In* Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science, STACS'99, *volume 1563 of* Lecture Notes in Computer Science, *pages 184–196. Springer-Verlag, 1999.*

[35] *D.R. Karger. Minimum cuts in near-linear time. In* Proceedings of the 28th Annual ACM Symposium on Theory of Computing, STOC'96, *pages 56–63, 1996.*

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

[37] *N. Korte and R.H. Möhring. An incremental linear time algorithm for recognizing interval graphs.* SIAM Journal on Computing, *18:68–81, 1989.*

[38] *E. Kubicka, G. Kubicki, and F.R. McMorris. An algorithm to find agreement subtrees.* Journal of Classification, *12:91–99, 1995.*

[39] *W.-H. Li.* Molecular Evolution. *Sinauer Associates, Inc., 1997.*

[40] *A. Lingas, H. Olsson, and A. Östlin. Efficient merging, construction, and maintenance of evolutionary trees. In* Proceedings of 26th Int. Colloquium on Automata, Languages and Programming, ICALP'99, *volume 1644 of* Lecture Notes in Computer Science, *pages 544–553. Springer-Verlag, 1999.*

[41] *A. Lingas, H. Olsson, and A. Östlin. Efficient merging and construction of evolutionary trees.* Journal of Algorithms, *(to appear).*

[42] *K. Mehlhorn.* Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry. *EATCS Monographs on Theoretical Computer Science, Springer Verlag, Berlin, 1984.*

[43] *M. Overmars. Computational geometry on a grid (an overview).* Theoretical Foundations of Computer Graphics and CAD, *F40:167–184, 1987.*

[44] *C.H. Papadimitriou.* Computational Complexity. *Addison-Wesley, Reading, 1994.*

[45] *P.A. Pevzner.* Computational Molecular Biology : An Algorithmic Approach. *The MIT Press, 2000.*

[46] *C. Phillips and T.J. Warnow. The asymmetric median tree - a new model for building consensus trees. In* Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching, CPM'96, *volume 1075 of* Lecture Notes in Computer Science, *pages 234–252. Springer-Verlag, 1996.*

[47] *J. Setubal and J. Meidanis.* Introduction to Computational Biology. *PWS publishing company, 1997.*

[48] *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.*

[49] *M. Steel. The complexity of reconstructing trees from qualitative characters and subtrees.* Journal of Classification, *9:91–116, 1992.*

[50] *M. Steel and T.J. Warnow. Kaikoura tree theorems: Computing the maximum agreement subtree.* Information Processing Letters, *48:77–82, 1993.*

[51] *R.E. Tarjan.* Data Structures and Network Algorithms. *Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.*

[52] *T. Warnow. Tree compatibility and inferring evolutionary history.* Journal of Algorithms, *16:388–407, 1994.*

[53] *M.S. Waterman.* Introduction to Computational Biology : Maps, sequences and genomes. *Chapman & Hall, 1995.*

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