Consensus Algorithms for Trees and Strings

Jansson, Jesper

2003

Consensus Algorithms for Trees and Strings

Jesper Jansson

Doctoral dissertation, 2003

Department of Computer Science
Lund Institute of Technology
Lund University

This thesis has been submitted to the Board of Research – FIME (Fysik, Informationsteknik, Matematik, Elektroteknik) at Lund Institute of Technology, Lund University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

Jesper Jansson
Department of Computer Science
Lund University
Box 118
SE-221 00 Lund
Sweden

E-mail: `Jesper.Jansson@cs.lth.se`
Webpage: `http://www.cs.lth.se/~jj`

ISBN 91-628-5586-7
ISSN 1404-1219
Dissertation 17, 2003
LU-CS-DISS:2003-2

Abstract

This thesis studies the computational complexity and polynomial-time approximability of a number of discrete combinatorial optimization problems involving labeled trees and strings. The problems considered have applications to computational molecular biology, pattern matching, and many other areas of computer science.

The thesis is divided into three parts. In the first part, we study some problems in which the goal is to infer a leaf-labeled tree from a set of constraints on lowest common ancestor relations. Our NP-hardness proofs, polynomial-time approximation algorithms, and polynomial-time exact algorithms indicate that these problems become computationally easier if the resulting tree is required to comply with a prespecified left-to-right ordering of the leaves.

The second part of the thesis deals with two problems related to identifying shared substructures in labeled trees. We first investigate how the polynomial-time approximability of the maximum agreement subtree problem depends on the maximum height of the input trees. Then, we show how the running time of the currently fastest known algorithm for the alignment between ordered trees problem can be reduced for problem instances in which the two input trees are similar and the scoring scheme satisfies some natural assumptions.

The third part is devoted to radius and diameter clustering problems for binary strings where distances between strings are measured using the Hamming metric. We present new inapproximability results and various types of approximation algorithms as well as exact polynomial-time algorithms for certain restrictions of the problems.

List of Publications

Most of the results presented here have been or will be published in:

- L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin. On the complexity of constructing evolutionary trees. *Journal of Combinatorial Optimization*, 3(2–3):183–197, 1999. A preliminary version appeared in *Proceedings of the 3rd Annual International Computing and Combinatorics Conference* (COCOON'97), volume 1276 of *Lecture Notes in Computer Science*, pages 134–145. Springer-Verlag Berlin Heidelberg, 1997.
- L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin. Inferring ordered trees from local constraints. In *Proceedings of Computing: the 4th Australasian Theory Symposium* (CATS'98), volume 20(3) of *Australian Computer Science Communications*, pages 67–76. Springer-Verlag Singapore, 1998.
- L. Gąsieniec, J. Jansson, and A. Lingas. Efficient approximation algorithms for the Hamming center problem. Technical Report LU-CS-TR:99-211, Lund University, 1999. A short form version of this article was published in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), pages S905–S906, 1999.
- L. Gąsieniec, J. Jansson, and A. Lingas. Approximation algorithms for Hamming clustering problems. *Journal of Discrete Algorithms*, to appear. A preliminary version appeared in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching* (CPM 2000), volume 1848 of *Lecture Notes in Computer Science*, pages 108–118. Springer-Verlag Berlin Heidelberg, 2000.
- J. Jansson. On the complexity of inferring rooted evolutionary trees. In *Proceedings of the Brazilian Symposium on Graphs, Algorithms, and Combinatorics* (GRACO 2001), volume 7 of *Electronic Notes in Discrete Mathematics*, pages 121–125. Elsevier, 2001.
- J. Jansson and A. Lingas. A fast algorithm for optimal alignment between similar ordered trees. *Fundamenta Informaticae*, to appear. A preliminary version appeared in *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching* (CPM 2001), volume 2089 of *Lecture Notes*

in Computer Science, pages 232–240. Springer-Verlag Berlin Heidelberg, 2001.

Acknowledgments

First of all, I would like to thank my thesis supervisor Andrzej Lingas for his invaluable advice and guidance, his endless patience, and for sharing his broad knowledge. I also thank Rolf Karlsson who introduced me to algorithm theory for convincing me to pursue my graduate studies in theoretical computer science, Leszek Gąsieniec for providing challenging problems to work on, Anna Östlin for successful collaboration, and Takeshi Tokuyama for some rewarding discussions which helped to improve this thesis.

I would also like to express my gratitude to Akiyoshi, Albert, Alf, Alfred, Anders, Andrea, Andreas, Andy, Anna, Anne-Marie, Aramis, Ariane, Aura, Bengt, Bertil, Birgitta, Björn, Cecilia, Chano, Chet, Christer, Christos, Cornelia, Dale, Daniel, Eivor, Elina, Emilia, Emir, Eric, Eva, Eva-Marta, Fred, Fredrik, Gert, Gertie, Gunilla, Hanna, Harald, Hasse, Igor, Ingegärd, Ingemar, Jack, Jakob, Jan, Jan-Arne, Jennifer, Jenny, Jens, Jimmy, Jinhee, Joachim, Johan, Jonas, Karim, Katarina, Kazue, Kenichiro, Kirk, Kunihiko, Lars, Lena, Lennart, Linus, Magdalene, Magnus, Maja, Marcus, Margit, Maria, Marjo, Martin, Matti, Mattias, Mia, Mikael, Mikiko, Nadia, Nancy, Nee, Ola, Olof, Patrik, Paul, Per, Petra, Renato, Sakura, Sami, Seth, Sonja, Stefan, Susanna, Sven, Sylvia, Thore, Tim, Tomas, Tony, Veena, Victoria, Wenli, Willy, Wing-Kin, and Yohei for inspiration and for contributing to the completion of this thesis in various ways. Thank you!

Greetings to noice, Stext, and TRIAD ...

Contents

Chapter 1

Introduction

This thesis studies a number of discrete combinatorial optimization problems of the form:

> Given a set S of objects, compute an object that summarizes the information contained in S in the best way possible.

A wide range of combinatorial optimization problems involving different kinds of objects and using different definitions of "the best way possible" fit the description above. We refer to problems of this general type as *consensus problems*, and algorithms for solving or approximating consensus problems as *consensus algorithms*. Three important categories of consensus problems are those whose respective goals are:

1. To merge two or more labeled trees into one larger tree so that no (or as little as possible) branching information is lost.
2. To identify a subtree contained in all members of a set of labeled trees.
3. To find a representative that resembles all of the strings in a given set.

Some problems from the first category considered here are *the maximum 3-leaf constraints consistency problem* (M3LC), *the ordered 3-leaf constraints consistency problem* (O3LC), and *the maximum ordered 3-leaf constraints consistency problem* (MO3LC). In these problems, the objective is to combine a set of overlapping, rooted, binary trees having precisely three labeled leaves each into one distinctly leaf-labeled tree; applications can be found in the construction of evolutionary trees. Another problem which can be classified as belonging to the first category is *the alignment between ordered trees problem* where two given node-labeled, ordered trees S and T have to be augmented with nodes labeled by the blank symbol until they both become isomorphic to (when node labels are ignored) some larger tree U in such a way that parts of S and T which are alike correspond to the same nodes in U. Efficient algorithms for computing alignments between trees may be useful in computational molecular biology, software construction, change detection in structured data, chemical structure analysis, information retrieval, pattern matching, and automated natural language translation systems.

An example of a consensus problem from the second category is *the maximum agreement subtree problem* (MAST). Given a set of rooted, unordered, leaf-labeled trees, MAST asks for a tree with the maximum possible number of leaves contained in all of the input trees. The primary use of MAST is to compare a set of alternative evolutionary trees for a fixed set of objects, obtained by using different tree construction methods or different sets of data. Another example of a consensus problem from the second category is *the tree inclusion problem* which can be formulated as a special case of the alignment between trees problem.

The Hamming center problem (HCP) is a consensus problem from the third category. The input to HCP is a set S of binary strings of equal length, and the objective is find a binary string (not necessarily in S) that minimizes the maximum Hamming distance to the strings in S. In a generalization of HCP called *the Hamming p-radius clustering problem* (HRC), the representative is not just *one* string, but a *set of* strings. A related problem where the objective is to partition the input strings into p groups so that the maximum of the group diameters is minimized is named *the Hamming p-diameter clustering problem* (HDC). HCP, HRC, and HDC have applications to coding theory, computational molecular biology, and clustering.

1.1 Thesis Objectives

The general goal of this thesis is to contribute to the understanding of what makes certain consensus problems solvable by efficient algorithms and others intractable. For this purpose, a number of specific, well-defined problems involving labeled trees and strings are studied from a computational complexity point of view.

The computational resource that we focus on here is *time*. When measuring the efficiency of an algorithm, we are concerned with the asymptotic behavior of its worst-case running time as the size of its input tends to infinity. An algorithm is said to be efficient if its asymptotic worst-case running time is upper bounded by a polynomial in the size of its input, and a problem is called efficiently solvable if it can be solved by an efficient algorithm. From here on, an algorithm's asymptotic worst-case running time is normally just referred to as its running time.

For each problem considered, if it is efficiently solvable, we try to provide an exact algorithm with as low running time as possible. On the other hand, if a problem can be shown to be NP-hard[1], we look for polynomial-time approxi-

[1] If a problem is NP-hard then it is highly unlikely that an efficient, exact algorithm for solving it can ever be constructed; see [10, 32, 49, 90, 105].

mation algorithms with as good approximation factors as possible, and attempt to find matching lower bounds on the polynomial-time approximability. For the problems which are NP-hard, we also try to clarify whether any non-trivial restrictions lead to simpler problems which we can solve exactly in polynomial time, or at least obtain better polynomial-time approximation factors for.

1.2 Organization of Thesis

The thesis is divided into three parts that may be read separately. Each part is in turn divided into two self-contained chapters treating one or two specific problems in detail, as outlined below.

Part I: Inferring Leaf-Labeled Trees from LCA Constraints

Chapter 2	The maximum LCA constraints consistency problem (MLC) The maximum 3-leaf constraints consistency problem (M3LC)
Chapter 3	The ordered 3-leaf constraints consistency problem (O3LC) The maximum ordered 3-leaf constraints consistency problem (MO3LC)

Part II: Identifying Shared Substructures in Labeled Trees

Chapter 4	The maximum agreement subtree problem (MAST)
Chapter 5	The alignment between ordered trees problem

Part III: Clustering under the Hamming Metric

Chapter 6	The Hamming center problem (HCP)
Chapter 7	The Hamming p-radius clustering problem (HRC) The Hamming p-diameter clustering problem (HDC)

Every chapter begins with an introduction which formally defines the investigated problems, argues why the problems are worth studying (Motivation), provides some background information (Previous Results), and briefly describes the new results that will be presented in that chapter (Our Contributions). The last section of every chapter (Concluding Remarks) summarizes our results and discusses related open questions.

The reader is assumed to possess a basic knowledge of algorithm theory and computational complexity equivalent to that covered by the widely used textbooks [32] and [105]. Some knowledge of approximation algorithms for NP-hard problems will also be helpful; good introductions to this subject can be found in the books [10, 63, 128].

1.3 Main Results

Here, we list the main results presented in this thesis along with references to where they have been published.

Part I: Inferring Leaf-Labeled Trees from LCA Constraints

- A new proof that the maximum LCA constraints consistency problem (MLC) and the maximum 3-leaf constraints consistency problem (M3LC) are NP-hard.

  (Theorem 2.2 and Corollary 2.3 in Section 2.3; published in [67]. A more complicated NP-hardness proof for MLC was published in [53], but it is not reproduced here since the result follows directly from the NP-hardness of M3LC. M3LC was independently proved to be NP-hard by Bryant in [21].)

- MLC and M3LC (as well as their weighted versions) can be approximated within a factor of 3 of the optimum in polynomial time.

  (Algorithm *Approximation A* in Section 2.4.1; published in [53].)

- A better polynomial-time approximation factor than the above can be obtained for instances of M3LC (and its weighted version) in which the optimal solution contains a large number of the input constraints.

  (Algorithm *Approximation B* in Section 2.4.2; an older version containing some minor errors was published in [53].)

- An algorithm for maintaining the union of a set of closed intervals under a sequence of deletions from the set.

  (Theorem 3.5 in Section 3.3; published in [52].)

- A polynomial-time algorithm for the ordered 3-leaf constraints consistency problem (O3LC) which is in general asymptotically faster than the currently fastest known algorithms for the corresponding problem for unordered trees.

  (Algorithm *Fast O3LC* in Section 3.4; published in [52].)

- The maximum ordered 3-leaf constraints consistency problem (MO3LC) can be solved in cubic time.

  (Theorem 3.11 in Section 3.5; preliminary version published in [52].)

Part II: Identifying Shared Substructures in Labeled Trees

- The maximum agreement subtree problem (MAST) can be approximated within a factor of $(n/\log n)$ in $O(kn^2)$ time, where k is the number of input trees and n is the total number of different leaf labels.

(Algorithm *Simple MAST-Approx* in Section 4.2.)

- If P$\neq$NP then MAST cannot be approximated within a factor of n^{ϵ} for any constant ϵ where $0 \leq \epsilon < \frac{1}{2}$ in polynomial time, even for instances containing only trees of height 2. An even stronger inapproximability result holds if ZPP$\neq$NP.

  (Theorem 4.2 in Section 4.3; published in [53].)

- MAST restricted to instances where both the number of input trees and the maximum height of at least one tree are bounded by constants can be approximated within a constant factor in polynomial time.

  (Theorem 4.3 in Section 4.4; published in [53].)

- The running time of the algorithm of Jiang, Wang, and Zhang [70] for the alignment between ordered trees problem can be reduced for instances in which the two input trees are similar and the scoring scheme satisfies some natural assumptions. In particular, if there exists an optimal alignment with at most d blank symbols and d is known in advance, the problem can be solved in $O(n \cdot (\log n + \Delta^3) \cdot d^2)$ time, where n is the number of nodes in the largest input tree and Δ is the maximum degree of the trees.

  (Algorithms *Fast Score* and *Unspecified d* in Sections 5.6 and 5.7; published in [68].)

Part III: Clustering under the Hamming Metric

- Several restrictions of the Hamming center problem (HCP) can be solved exactly in polynomial time.

  (Corollary 6.6 in Section 6.3; published in [50].)

- A randomized $(\frac{4}{3} + \varepsilon)$-approximation algorithm for HCP, where ε can be selected to be any constant > 0.

  (Theorem 6.14 in Section 6.5; published in [50].)

- The Hamming p-radius clustering problem (HRC) and the Hamming p-diameter clustering problem (HDC) are NP-hard to approximate within a factor of $2 - \varepsilon$ for any constant $\varepsilon > 0$.

  (Theorem 7.5 in Section 7.3.1; published in [51].)

- An inapproximability result for the version of HDC in which the constraint on the number of produced clusters is relaxed.

  (Theorem 7.6 in Section 7.3.2 published in [51].)

- HDC is NP-hard for every fixed $p \geq 3$.

  (Corollary 7.7 in Section 7.3.2 published in [51].)

- Several restrictions of HRC and HDC can be solved exactly in polynomial time.

  (Corollary 7.13 in Section 7.4; published in [51].)

- Approximation algorithms for HRC and HDC.

  (Algorithms *Farthest-Point Clustering* and *HRC Approximation Scheme* in Sections 7.5.1 and 7.5.2; published in [51]. Algorithm *HRC Randomized PTAS* in Section 7.5.3 was obtained by combining the randomized PTAS of Ostrovsky and Rabani [101] and the PTAS of Li, Ma, and Wang [91].)

- An approximation algorithm for HRC which approximates the p-radius within a factor of $(1 + \varepsilon)$ for any constant $0 < \varepsilon < 1$ by slightly increasing the number of output strings. Its running time is polynomial as long as the p-radius is not too large.

  (Theorem 7.22 in Section 7.6; published in [51].)

Part I

Inferring Leaf-Labeled Trees from LCA Constraints

Chapter 2

Inferring Unordered Trees from Lowest Common Ancestor Constraints

In this chapter, we study an optimization problem related to evolutionary tree construction called *the maximum LCA constraints consistency problem* (MLC) in which a set of constraints on lowest common ancestor relations is given and the goal is to construct an unordered, leaf-labeled tree which maximizes the number of satisfied constraints. Special attention is paid to the case called *the maximum 3-leaf constraints consistency problem* (M3LC) where each of the input LCA constraints involves three leaves only.

2.1 Introduction

We begin with some definitions.

Let S be a finite set of elements. A *lowest common ancestor constraint on S* (*LCA constraint on S*) is a constraint of the form $\{i, j\} < \{k, l\}$, where $i, j, k, l \in S$, which specifies that the lowest common ancestor of i and j is a proper descendant of the lowest common ancestor of k and l. An LCA constraint of the form $\{i, j\} < \{i, k\}$ is called a *3-leaf constraint on S* or a *rooted triple on S*, and is written as $(\{i, j\}, k)$ for short.

A tree whose leaves are labeled by elements in S in such a way that no two leaves have the same label is said to be *distinctly leaf-labeled by S*. When S is used to distinctly label the leaves of a tree R, each leaf of R is identified with its corresponding element in S. Therefore, the elements of S are commonly referred to as *leaves*[1]. A rooted, unordered tree which is distinctly leaf-labeled by S and

[1]In some applications, the elements of S are called *species*.

an LCA constraint on S which is satisfied in the tree are *consistent* with each other. See Figure 2.1 for two examples. Note that a 3-leaf constraint $(\{i,j\},k)$ uniquely determines the relative topology of i, j, and k in any unordered tree which is consistent with it; conversely, any rooted, unordered, binary tree with three distinctly labeled leaves corresponds to a unique 3-leaf constraint.

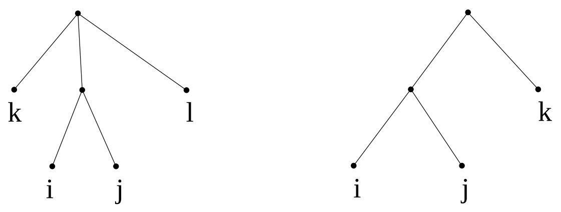

Figure 2.1: The tree on the left is one of several unordered trees with four leaves consistent with the LCA constraint $\{i,j\} < \{k,l\}$. The tree on the right is the unique unordered tree with three leaves which is consistent with the 3-leaf constraint $(\{i,j\},k)$.

Aho, Sagiv, Szymanski, and Ullman [3] studied the following problem which we call *the LCA constraints consistency problem* (LC):

The LCA constraints consistency problem (LC)

Instance: Finite set S, set T of LCA constraints on S.

Output: A rooted, unordered tree with leaves distinctly labeled by S which is consistent with all of the constraints in T, if one exists; otherwise, null.

They also studied the special case where each constraint is a 3-leaf constraint:

The 3-leaf constraints consistency problem (3LC)

Instance: Finite set S, set T of 3-leaf constraints on S.

Output: A rooted, unordered tree with leaves distinctly labeled by S which is consistent with all of the constraints in T, if one exists; otherwise, null.

LC and 3LC can be turned into optimization problems:

The maximum LCA constraints consistency problem (MLC)

Instance: Finite set S, set T of LCA constraints on S.

Output: A rooted, unordered tree with leaves distinctly labeled by S which is consistent with as many of the constraints in T as possible.

The maximum 3-leaf constraints consistency problem (M3LC)

Instance: Finite set S, set T of 3-leaf constraints on S.

Output: A rooted, unordered tree with leaves distinctly labeled by S which is consistent with as many of the constraints in T as possible.

Throughout this chapter, we let n denote the cardinality of the set of leaves and m the number of constraints, i.e., $n = |S|$ and $m = |T|$ in the problem definitions above. Observe that $m = O(n^4)$ in MLC, and $m = O(n^3)$ in M3LC.

As an example, let $S = \{a, b, c, d\}$ and $T = \{(\{a, b\}, c), (\{a, c\}, d), (\{c, d\}, b)\}$ be an instance of M3LC. There is no tree which is consistent with all of T and whose leaves are distinctly labeled by S (as can be verified by applying the algorithm of Aho *et al.* described in Section 2.2), but each of the unordered trees shown in Figure 2.2 is consistent with two constraints in T. Hence, this problem instance has three optimal solutions.

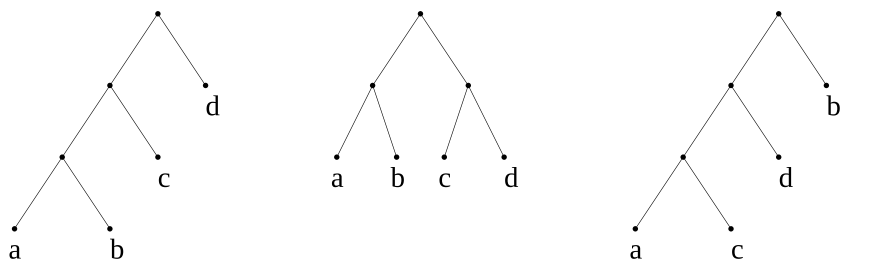

Figure 2.2: $T = \{(\{a, b\}, c), (\{a, c\}, d), (\{c, d\}, b)\}$. The first tree is consistent with $(\{a, b\}, c)$ and $(\{a, c\}, d)$, the second tree is consistent with $(\{a, b\}, c)$ and $(\{c, d\}, b)$, and the third tree is consistent with $(\{a, c\}, d)$ and $(\{c, d\}, b)$.

2.1.1 Motivation

Aho, Sagiv, Szymanski, and Ullman introduced LC and 3LC in [3]. Their motivation for studying these problems originated from a problem in the theory of relational databases in which an SPJ-expression is given and the goal is to construct an equivalent SPJ-expression containing the minimum possible number of join operators (see [3] for details). Their proposed solution represents the given SPJ-expression by a tableau A, transforms A into an equivalent minimum row tableau A', and then constructs an SPJ-expression from A'; in the important special case of simple tableaux, an efficient algorithm for solving LC can be used to efficiently construct SPJ-expressions from row-optimized tableaux in the last step.

3LC and M3LC are also of interest because of their connection to the construction of evolutionary trees[2]. Reliable methods for determining evolutionary history are of fundamental importance to biology because before the mechanisms of evolution can be explained, its effects must be examined and well understood. By inferring evolutionary trees for different sets of species, biologists can deduce in which order and at what rate genetic changes have taken place, which yields clues about the causes of evolution. Secondly, classifications of species based on their evolutionary history are indicative and meaningful, and help scientists organize and exchange information. Note that the "species" do not necessarily have to be distinct biological species; they may be entire populations or categories of species, or any other entities assumed to have been subjected to an evolutionary process such as proteins, nucleic acids, languages, chain letters, or medieval manuscripts [15, 18, 23, 92, 111]. Therefore, fields like historical linguistics also need good methods for constructing evolutionary trees.

To describe evolutionary relationships with trees is an old idea. Long before the present-day theories of molecular biology, Charles Darwin, Edward Blyth, Ernst Haeckel, and other 19th century natural scientists compared the development of different species originating from a common ancestor to a branching tree. Initially, biologists built evolutionary trees from data based on observable morphological features (and intuition). Later, biochemical data and results from molecular sequence comparisons became popular as well. The assumptions behind what distinguishes good optimization criteria vary according to the particular application, the hypothesized model of evolution, and the type of available data[3], which explains why so many kinds of evolutionary trees and methods for reconstructing evolutionary history have been invented. No single method appears to work well all the time; hence, many different methods are still commonly used. Roughly, existing techniques for inferring evolutionary trees can be divided into:

Character-state methods: Represent each object by a vector of character states and look for a tree that clusters objects with a lot in common.

Distance-based methods: Compute the evolutionary distance between every

[2]An *evolutionary tree* (or a *phylogenetic tree*) is an unordered, leaf-labeled tree that describes how a set of objects produced by some evolutionary process are believed to be related. The objects are represented by leaves and common ancestors by internal nodes so that the branching structure of the tree reflects the assumed evolutionary relationships. Sometimes weights are assigned to the edges to illustrate evolutionary distance, i.e., estimates of the time taken for an ancestral object to evolve into other objects. In some settings, the data does not uniquely determine a root, which leads to unrooted (as opposed to rooted) trees. Here, we concentrate on unweighted, rooted trees.

[3]For instance, the parsimony principle (which says that one should attempt to minimize the number of state changes induced by the constructed tree) is often used when treating morphological data but may be less appropriate for DNA sequence data. This is because DNA sequence positions do not evolve independently and because changes during evolution in DNA are much more frequent than changes in morphological characters, implying that many back substitutions should be expected.

pair of objects, then try to build an edge-weighted tree which complies with these values.

Maximum likelihood methods: View evolution as a stochastic process and use statistical methods to find the most probable tree.

Consensus methods: Combine a set of (possibly conflicting) trees into one final tree.

Comprehensive surveys of the first three types of methods can be found in [92, 111, 122, 130]. Many evolutionary tree construction algorithms belonging to these three categories have been implemented in PHYLIP [44] (a package of computer programs available for free on the Internet) and in a commercial software package called PAUP$*$ [121]. The fourth category refers to methods such as the *quartet* approach (see [81] for a recent survey), rooted triple-based methods [71, 72], and methods based on *splits/clusters* contained in the input trees (see Section 6.2 in [21]) which all take as input a set of evolutionary trees, obtained by some other tree inference method, and output a single tree which summarizes the branching information in the input trees. The rationale for using consensus methods to infer new trees is that although computationally expensive tree construction methods such as maximum likelihood or maximum parsimony are infeasible for large sets of objects, they can be applied to obtain highly accurate trees for smaller, overlapping subsets of the objects which can subsequently be merged into one tree for all the objects by using less computationally intensive techniques. Ideally, the optimization criteria for determining the final tree should be selected in accordance with the assumed model of evolution to guarantee the practical relevance of the optimal solution. But before this is possible, one must learn about the underlying combinatorial problems.

This is where 3LC and M3LC come in. Given a set T of rooted, binary evolutionary trees where each tree contains exactly three leaves (for example, obtained by Sibley-Ahlquist-style DNA-DNA hybridization experiments [71] or by maximum likelihood methods), the problem of constructing a rooted, unordered tree consistent with all of T (if one exists) is precisely 3LC. However, data obtained experimentally often contains errors, implying that there probably will not exist a tree consistent with *all* of T. Since a single erroneous tree in the input results in the algorithms for 3LC returning the null tree, the optimization version (M3LC) is perhaps even more important for real applications. Therefore, we are especially interested in the computational complexity and polynomial-time approximability of M3LC.

2.1.2 Previous Results

Aho, Sagiv, Szymanski, and Ullman [3] presented an algorithm for solving LC which runs in $O(mn \log n)$ time. For 3LC, its running time is $O(mn)$. The algorithm returns a tree of minimum height which is consistent with all of the input constraints, if such a tree exists.

Henzinger, King, and Warnow [62] showed how to solve the special case 3LC more efficiently. Assuming that $m = \Omega(n)$, their adaptation of the algorithm of Aho *et al.* runs in $O(mn^{1/2})$ time (deterministic version), or in $O(m \log^3 n)$ expected time (randomized version). Henzinger *et al.* also gave a deterministic algorithm for 3LC with running time $O(m+n^2 \log n)$. It is asymptotically faster than the deterministic algorithm above if $m = \omega(n^{3/2} \log n)$, but does not always construct a tree of minimum height.

M3LC was first proved to be NP-hard by Bryant in [21].

The analog of 3LC for unrooted trees is called *the quartets consistency problem* (QC). In QC, the input is a set $\mathcal{Q}$ of *resolved quartets* (unrooted, leaf-labeled trees each having 4 leaves and no nodes of degree 2), and the goal is to find an unrooted, leaf-labeled tree which is consistent with all of the quartets in $\mathcal{Q}$, if one exists. *The maximum quartets consistency problem* (MQC) is the optimization version of QC. 3LC and M3LC can be viewed as the restrictions of QC and MQC to instances where all the given quartets have one leaf in common, corresponding to the root [116]. For a brief discussion of how the computational complexity results for 3LC and M3LC compare to those known for QC and MQC, see the footnote in Section 2.5. For further references on quartets, see [81].

2.1.3 Our Contributions

We provide a new proof for the NP-hardness of M3LC and MLC. Then, since no algorithm can solve MLC or M3LC exactly in polynomial time unless P=NP, we present two polynomial-time approximation algorithms called Algorithm *Approximation A* and Algorithm *Approximation B*. The first one approximates both MLC and M3LC within a constant factor, and the second one yields a better approximation factor for instances of M3LC in which the optimal solution contains a large number of the input constraints. To be more precise, we show that:

- MLC and M3LC (as well as their weighted versions) can be approximated within a factor of 3 of the optimum in $O((m+n)\log n)$ time.

  (Algorithm *Approximation A* in Section 2.4.1.)

- M3LC (and its weighted version) can be approximated in the following way. An approximate solution consistent with a subset of the input constraints whose total weight is at least $W - nt$, where W is the total weight of all input constraints and t is the minimum total weight of constraints needed to remove in order to achieve consistency, can be constructed in $\min\{O(mn^2 + n^3 \log n),\, O(n^4)\}$ time, or with high probability in $\min\{O(mn \log^3 n),\, O(n^3 \log n)\}$ time.

  (Algorithm *Approximation B* in Section 2.4.2.)

To our knowledge, these are the only results on the polynomial-time approximability of MLC and M3LC that have ever been published.

The rest of this chapter is organized as follows. We review existing algorithms for LC and 3LC in Section 2.2. In Section 2.3, we investigate the computational complexities of MLC and M3LC. We present Algorithm *Approximation A* and Algorithm *Approximation B* in Section 2.4. Finally, in Section 2.5, some open problems are discussed.

2.2 Preliminaries

Below, we describe the algorithms of Aho, Sagiv, Szymanski, and Ullman [3] and Henzinger, King, and Warnow [62] since they form the basis for the approximation algorithm given in Section 2.4.2 and the exact algorithm in Section 3.4.

2.2.1 The Algorithm of Aho, Sagiv, Szymanski, and Ullman

The algorithm of Aho *et al.* [3] for LC partitions the leaves in S into *blocks* using the set of input constraints T. The blocks are chosen so that each block consists of all leaves that will be descendants of (or equal to) one child of the root in the tree being constructed. To partition the leaves, the algorithm looks for the largest possible set of blocks obeying the rules:

(1) If $\{i,j\} < \{k,l\}$ is an input constraint then i and j are in the same block[4].

(2) If $\{i,j\} < \{k,l\}$ is an input constraint and k and l are in the same block then i,j,k,l are all in this block[5].

For 3LC, only one rule is needed:

(1′) If $(\{i,j\},k)$ is an input constraint then i and j are in the same block.

If the number of blocks is at least two, the algorithm recursively constructs a tree for each block, attaches these trees to a common parent node, and returns the resulting tree. When recursing on a block, only constraints consisting entirely of leaves in that block are considered; all other constraints are ignored.

Otherwise, there is just one block. If the block consists of a single leaf i, the algorithm returns a tree with one leaf labeled by i. If the block contains more than one leaf, the algorithm aborts its execution and returns the null tree since no tree can be consistent with all of the constraints (see [3] for proofs).

[4]This is because for any $i,j,k,l \in S$, if $\{i,j\} < \{k,l\}$ is an input constraint then the lowest common ancestor of i and j cannot be at the root of the tree (otherwise, it could not be a proper descendant of the lowest common ancestor of k and l), so i and j cannot belong to two different blocks.

[5]Similarly to the above, for any $i,j,k,l \in S$, if $\{i,j\} < \{k,l\}$ is an input constraint then either k and l must be in different blocks or all four leaves must belong to the same block.

In [3], Aho *et al.* explained how their algorithm for LC can be implemented to run in $O(mn \log n)$ time. They also showed that when restricted to 3LC, it can be implemented to run in $O(mn)$ time as follows.

For any subset L of leaves in S, let $G(L)$ be the undirected graph with vertex set L and edge set $E(L)$, where $E(L)$ is the set of edges induced by rule $(1')$ applied to the set of constraints on leaves entirely contained in L, i.e., if there is a constraint $(\{i,j\},k)$ and $i,j,k \in L$, then edge $\{i,j\}$ is included in $E(L)$. Now, the connected components of $G(L)$ represent the blocks of leaves that L is partitioned into by rule $(1')$.

Thus, given an instance (S,T) of 3LC, the algorithm builds $G(S)$ and calculates the connected components $\mathcal{C}_1, ..., \mathcal{C}_q$ of $G(S)$. If $q \geq 2$, it then makes q recursive calls to itself on instances (S_1,T_1),...,(S_q,T_q), where for $1 \leq p \leq q$, S_p is the set of leaves in $\mathcal{C}_p$, and the sets $T_1, ..., T_q$ are obtained by scanning T (for each $(\{i,j\},k) \in T$, if all of i, j, and k belong to the same S_p then $(\{i,j\},k)$ is placed in T_p; otherwise it is deleted).

At each of the $O(n)$ recursion levels, the total time required to build all graphs and to find their connected components is $O(m)$. Scanning the constraints to compute the sets T_p also takes $O(m)$ time on each level. Therefore, this implementation has a running time which is $O(mn)$.

2.2.2 The Algorithm of Henzinger, King, and Warnow

The running time of the algorithm of Aho *et al.* for 3LC as stated is $O(mn)$; Henzinger *et al.* [62] subsequently improved its efficiency. By employing an auxiliary data structure consisting of two graphs U and D described below and an algorithm for the deletions-only dynamic graph connectivity problem that keeps track of the connected components in a graph under a sequence of edge deletions, they speeded up two bottlenecks in Aho *et al.*'s algorithm: (1) determining which of the input constraints contain leaves from a specified block only, and (2) finding the connected components of the graphs $G(S_1), G(S_2), ..., G(S_q)$ (i.e., recomputing the connected components of the graph $G(S)$ after a set of edges has been deleted).

The undirected graph U and the directed graph D are defined as:

- $U = (S,E)$ with vertex set equal to the input set of leaves S, and where for each constraint $(\{a,b\},c)$ in T, the edges $\{a,b\}$ and $\{b,c\}$ are in E.
- $D = (S',A)$, where for each constraint $(\{a,b\},c)$ in T, the vertices $\{a,b\}$ and $\{b,c\}$ are in S' and the directed edge $\{a,b\} \rightarrow \{b,c\}$ is in A.

Current blocks of leaves in the algorithm of Aho *et al.* correspond to current *yellow components* in U. At the beginning, all edges of U are colored yellow. During the course of the algorithm, D is used to find edges of U that are colored red. Checking if coloring an edge red results in a yellow component being split into two yellow components is done with a dynamic graph connectivity algorithm which uses some internal data structure to represent U.

A *maximal vertex* in D is a vertex with no outgoing edges. A red edge of U whose endpoints are in different yellow components is called a *separable red edge.*

Each input constraint is represented by a directed edge in D. Hence, deleting a vertex in D and its incident edges corresponds to deleting one or more of the input constraints. Consider an input constraint $(\{a,b\},c)$. In the algorithm of Aho *et al.*, it is deleted when b and c no longer belong to the same block; in the algorithm of Henzinger *et al.*, the vertex $\{b,c\}$ and the directed edge $\{a,b\} \rightarrow \{b,c\}$ are deleted from D and the edge $\{b,c\}$ is deleted from U when $\{b,c\}$ becomes a separable red edge in U, i.e., when b and c belong to different yellow components for the first time.

```
Algorithm   Fast 3LC
Input:   An instance of 3LC.
Output:  A rooted, unordered tree $R$ of minimum height which is consistent with
         all of the constraints, if one exists; the null tree, otherwise.

1  Construct $U$ and $D$.
2  Color all edges in $U$ yellow and initialize the dynamic graph connectivity
   algorithm.
3  Create the root of $R$. Initialize components information.
4  for each maximal vertex $\{x,y\}$ in $D$ do
       If $\{x,y\}$ is a yellow edge in $U$ then color it red, delete it in the dynamic
       graph connectivity data structure, and query the dynamic connectivity
       algorithm "are $x$ and $y$ connected?"; if "no" then update components in-
       formation and $R$.
   endfor
5  For each old yellow component $C_i$, if $|C_i| > 1$ and $C_i$ was not split into at
   least two components during step 4 in this iteration then return the null tree.
6  If $|C_i| = 1$ for every yellow component $C_i$ then return $R$.
   Otherwise, find all separable red edges and delete these edges from $U$ and the
   corresponding vertices (plus their incident edges) from $D$.
7  Go to step 4.
End  Fast 3LC
```

Figure 2.3: Henzinger, King, and Warnow's adaptation of the algorithm of Aho, Sagiv, Szymanski, and Ullman.

Henzinger *et al.*'s algorithm is outlined in Figure 2.3. The tree constructed by the algorithm is denoted by R. If a yellow component C splits into c yellow components $C_1, C_2, ..., C_c$ in one iteration of the algorithm, then the node in R corresponding to C will have c children corresponding to $C_1, C_2, ..., C_c$, respectively. The algorithm keeps information about the current yellow components so that new nodes can be created and attached to their correct parent nodes when updating R in step 4. We omit the implementation details here.

Updating R in step 4 also includes checking the new yellow components to see if any of them consist of a single vertex v. If this is the case, then the corresponding node in R is labeled by v.

In any given iteration of the algorithm, the yellow components of U right before the execution of step 4 are called *old*. When step 5 is reached, if an old yellow component consisting of more than just one leaf was not split into two or more new yellow components then the algorithm gives up since some constraints involving the leaves in that component contradict each other.

To find the separable red edges efficiently, Henzinger *et al.* extended an idea from [39]. In step 4, if the dynamic graph connectivity algorithm reports that x and y are not connected any longer then the yellow component C that previously contained both x and y has been split into two. The number of vertices in at least one of the two resulting components must be less than or equal to half of the number of vertices in C; such a component is called *lesser*. When C is split into two, a lesser component can be discovered by searching in U from x and y, alternating between the two searches until one component is completely visited. This is then used in step 6, where it would be too time-consuming to locate the separable red edges by traversing *all* the yellow components in every iteration. Instead, the algorithm only searches in newly created yellow components which are lesser; this way, each edge is visited at most $O(\log n)$ times in total until it becomes a separable red edge and is deleted from U.

Henzinger *et al.* employed two different algorithms for dynamic graph connectivity. The first one [37, 47] is deterministic and allows each update and connectivity query to be performed in $O(n^{1/2})$ and $O(1)$ time, respectively, whereas the second one [61] is randomized and allows each update to be carried out in $O(\log^3 n)$ amortized expected time and each query in $O(\log n / \log\log n)$ time. All other operations in Henzinger *et al.*'s algorithm take a total of $O(n + m + M \cdot \log n)$ time, where M is the initial number of edges in the constructed graph U. Since $M = O(m)$, the 3LC algorithm can thus be implemented to run in $O(n + mn^{1/2})$ time (deterministically), or in $O(n + m \log^3 n)$ expected time (randomized).

We note that the running time can be improved simply by substituting the used algorithms for dynamic graph connectivity with more recent ones. It suffices to use a *decremental* dynamic graph connectivity algorithm since all updates are edge deletions. Furthermore, the number of edges in U is bounded not only by $O(m)$, but also by $O(n^2)$ because $|S| = n$, so $M = \min\{O(m), O(n^2)\}$. In general, we have the following theorem.

Theorem 2.1 *Given an algorithm for decremental dynamic graph connectivity which takes $u(n)$ amortized time per update and answers each connectivity query in $q(n)$ time, Henzinger* et al.*'s algorithm for 3LC can be implemented to run in $O(n + m + M \cdot (\log n + u(n) + q(n)))$ time, where $M = \min\{O(m), O(n^2)\}$.*

For example, if we use the deterministic algorithm for fully dynamic graph connectivity due to Holm, de Lichtenberg, and Thorup [65] with $u(n) = O(\log^2 n)$ and $q(n) = O(\log n / \log\log n)$, we obtain a deterministic algorithm for solving 3LC whose running time is $\min\{O(n + m\log^2 n),\, O(m + n^2\log^2 n)\}$. Alternatively, we can take the randomized algorithm for fully dynamic graph connectivity of Thorup [127] with $u(n) = O(\log n(\log\log n)^3)$ amortized expected time and $q(n) = O(\log n/\log\log\log n)$ to get a randomized algorithm for 3LC with $\min\{O(n + m\log n(\log\log n)^3),\, O(m + n^2\log n(\log\log n)^3)\}$ expected running time. A specialized randomized algorithm for decremental dynamic graph connectivity by Thorup [126] is even more efficient if U initially contains many edges; if $M = \Omega(n(\log n\log\log n)^2)$, then $u(n) = O(\log n)$ amortized expected time and $q(n) = O(1)$, resulting in an algorithm with $\min\{O(n + m\log n),\, O(m + n^2\log n)\} = \min\{O(m\log n),\, O(m + n^2\log n)\}$ expected running time for these special instances of 3LC.

In [62], Henzinger *et al.* also presented a variant of the above algorithm which solves 3LC in $O(m + n^2\log n)$ time. It constructs trees which are binary, and hence not necessarily equivalent to the trees of minimum height constructed by the algorithm of Aho *et al.* The main difference is that in each iteration, at most one new yellow component is discovered so that R is extended with at most two new nodes. In step 4, rather than processing each edge individually (i.e., deleting it in the dynamic graph connectivity data structure and testing if its endpoints still are connected), the algorithm first deletes all those edges using a special batch deletion algorithm and then asks for one new component. Not all newly created yellow components are discovered immediately; thus, a component which is reported as "new" does not need to have been caused by the most recent batch of edge deletions. As before, R is updated and a lesser yellow component is identified in order to find separable red edges later on. Step 5 is modified to return the null tree if no new component was reported in step 4.

2.3 MLC and M3LC are NP-Hard

In this section, we prove that MLC and M3LC are NP-hard problems[6].

Denote the decision problem versions of MLC and M3LC by MLC(D) and M3LC(D), respectively. Here, a positive integer D is also given as part of the input, and the output is the answer to the question "Does there exist a rooted, unordered tree that is consistent with D of the input constraints?".

[6]The NP-hardness proof for M3LC given here was published in 2001 [67] in response to an open question from 1999 [53]. Unknown to us at that time, M3LC had already been proven to be NP-hard by Bryant in 1997 [21]. Nevertheless, we have decided to include our proof from [67] here since alternative reductions (Bryant's proof uses a reduction from *the feedback arc set problem* to M3LC whereas we reduce from *cyclic ordering* to M3LC) may be helpful, e.g., for proving inapproximability results for M3LC in the future.

To determine the computational complexity of M3LC(D), we will use a result of Galil and Megiddo [48] stating that the following problem (listed as problem MS2 in [49]) is NP-complete.

Cyclic ordering

Instance: Finite set A, collection C of ordered triples (a, b, c) of distinct elements from A.

Question: Is there a one-to-one function $f : A \rightarrow \{1, 2, ..., |A|\}$ such that, for each $(a, b, c) \in C$, we have either $f(a) < f(b) < f(c)$, or $f(b) < f(c) < f(a)$, or $f(c) < f(a) < f(b)$?

We are now ready for the main result of this section.

Theorem 2.2 *M3LC(D) is NP-complete.*

Proof: M3LC(D) is in NP since verifying if there exists a rooted tree that is consistent with a given subset of T can be done in polynomial time with the algorithm of Aho *et al.* (see Section 2.2).

To show the NP-hardness of M3LC(D), we provide a polynomial-time reduction from cyclic ordering to M3LC(D). Given an instance (A, C) of cyclic ordering, let $S = A \cup \{x_0, x_1, x_2, ..., x_{|C|}\}$ and let $D = \frac{|A| \cdot (|A|-1)}{2} + 2 \cdot |C|$. For each $a, b \in A$ with $a \neq b$, include the two constraints $(\{x_0, a\}, b)$ and $(\{x_0, b\}, a)$ in T. Next, for every i in $\{1, 2, ..., |C|\}$, add to T the three constraints $(\{x_i, a\}, b)$, $(\{x_i, b\}, c)$, and $(\{x_i, c\}, a)$, where (a, b, c) is the ith ordered triple in C. Observe that at most one of $(\{x_0, a\}, b)$ and $(\{x_0, b\}, a)$ and at most two of $(\{x_i, a\}, b)$, $(\{x_i, b\}, c)$, and $(\{x_i, c\}, a)$ can be consistent with any rooted tree, so the number of constraints in T that can be satisfied at the same time must be $\leq D$.

Claim: (A, C) has a cyclic ordering if and only if there exists a rooted tree that is consistent with D of the constraints in T.

Proof of claim: Suppose the answer to the cyclic ordering instance is yes. Then there exists a one-to-one function $f : A \rightarrow \{1, 2, ..., |A|\}$ such that for each ordered triple $(a, b, c) \in C$, we have either $f(a) < f(b) < f(c)$, or $f(b) < f(c) < f(a)$, or $f(c) < f(a) < f(b)$. We can construct a rooted tree consistent with D constraints as in Figure 2.4. (If $f(\alpha_i) < f(\beta_i) < f(\gamma_i)$ for the ith ordered triple in C, then $(\{x_i, \alpha_i\}, \beta_i)$ and $(\{x_i, \beta_i\}, \gamma_i)$ are consistent with the tree in Figure 2.4. Also, for each pair $a, b \in A$ with $a \neq b$, exactly one of $(\{x_0, a\}, b)$ and $(\{x_0, b\}, a)$ is consistent with the tree. Thus, the tree is consistent with $2 \cdot |C| + \frac{|A| \cdot (|A|-1)}{2}$ of the constraints in T.)

Conversely, suppose there exists a rooted tree R consistent with $\frac{|A| \cdot (|A|-1)}{2} + 2 \cdot |C|$ of the constraints. At most $\frac{|A| \cdot (|A|-1)}{2}$ constraints of type $(\{x_0, a\}, b)$ and at most $2 \cdot |C|$ constraints of type $(\{x_i, a\}, b)$ with $i \neq 0$ can be consistent

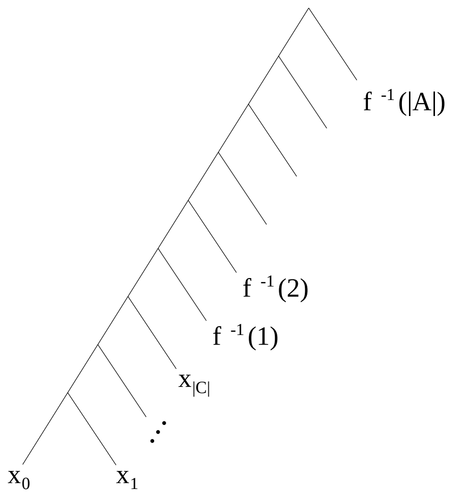

Figure 2.4: This tree is consistent with D constraints.

with R, so R must be consistent with precisely this many constraints of each type, respectively. $\frac{|A|\cdot(|A|-1)}{2}$ constraints of the former type can only be satisfied if the subtree of R induced by $A \cup \{x_0\}$ is a rooted caterpillar tree whose root is the parent of a leaf and an internal node, and one of the two leaves at maximum distance from the root is labeled x_0 (otherwise, for some pair $a, b \in A$, neither $(\{x_0, a\}, b)$ nor $(\{x_0, b\}, a)$ would be consistent with R). For each $a \in A$, let $f(a)$ be the number of internal nodes on the path from a to x_0 in the subtree of R induced by $A \cup \{x_0\}$. Next, because of the constraints of the second type, for every ordered triple $(a, b, c) \in C$, exactly two of the three corresponding constraints in T are consistent with R (if, for some ordered triple, just one constraint was consistent with R, then the number of constraints of this type consistent with R could not add up to $2 \cdot |C|$). Therefore, either (1) a is closer to x_0 than b is to x_0 and b is closer to x_0 than c is to x_0, implying $f(a) < f(b) < f(c)$, or (2) b is closer to x_0 than c is to x_0 and c is closer to x_0 than a is to x_0, implying $f(b) < f(c) < f(a)$, or (3) c is closer to x_0 than a is to x_0 and a is closer to x_0 than b is to x_0, implying $f(c) < f(a) < f(b)$.

Hence, M3LC(D) is NP-complete. □

Corollary 2.3 *MLC(D) is NP-complete.*

Proof: MLC(D) is in NP because the algorithm of Aho *et al.* can check any given subset of the LCA constraints for consistency in polynomial time. MLC(D) is NP-hard since it admits a direct reduction from M3LC(D); just replace each 3-leaf constraint $(\{a, b\}, c)$ in the given instance by $\{a, b\} < \{a, c\}$. □

2.4 Polynomial-Time Approximation Algorithms for MLC and M3LC

The approximation algorithms in this section also work for the natural generalizations of MLC and M3LC in which a positive weight $w(c)$ is associated to each input constraint c, and the objective is to construct a rooted tree which is consistent with a subset of the constraints of maximum total weight.

If no weights have been assigned to the constraints, they are all assumed to have weight 1.

2.4.1 Algorithm *Approximation A*

In this subsection, we give a constant-factor approximation algorithm for MLC and M3LC which runs in polynomial time.

Definition 2.4 For an LCA constraint $\{i, j\} < \{k, l\}$, where all of the four 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 3-leaf constraint $(\{i, j\}, k)$, 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.

Definition 2.5 The *total weight of upper (lower) occurrences* for a leaf l is the sum of the weights of all constraints in which l has upper (lower) occurrences.

We immediately obtain the following lemma.

Lemma 2.6 *In any instance of MLC/M3LC, the sum of all leaves' total weight of upper occurrences is at least one third (exactly one half if all constraints contain four different leaves) of the sum of all leaves' total weight of upper and lower occurrences.*

Proof: Denote the sum of all leaves' total weight of upper occurrences by $\mathcal{Y}$ and the sum of all leaves' total weight of upper and lower occurrences by $\mathcal{X}$. Let A be the set of 3-leaf constraints in T, and let B be the set of LCA constraints in T with four different leaves. For every constraint $c \in T$, let $w(c)$ be the weight of c. If $c \in A$, then it contributes $w(c)$ to $\mathcal{Y}$ and $3w(c)$ to $\mathcal{X}$. Otherwise, $c \in B$ and hence contributes $2w(c)$ to $\mathcal{Y}$ and $4w(c)$ to $\mathcal{X}$. Therefore, $\mathcal{Y} = \sum_{c\in A} w(c) + \sum_{c\in B} 2w(c)$ and $\mathcal{X} = \sum_{c\in A} 3w(c) + \sum_{c\in B} 4w(c)$. Now,

$$\begin{cases} \mathcal{Y} \geq \sum_{c\in A} w(c) + \frac{4}{3}\cdot \sum_{c\in B} w(c) = \frac{1}{3}\cdot\mathcal{X} \\ \\ \mathcal{Y} \leq \frac{3}{2}\cdot \sum_{c\in A} w(c) + 2\cdot \sum_{c\in B} w(c) = \frac{1}{2}\cdot\mathcal{X} \end{cases}$$

which shows that $\frac{1}{3}\cdot\mathcal{X} \leq \mathcal{Y} \leq \frac{1}{2}\cdot\mathcal{X}$. If all constraints have four different leaves then $A = \emptyset$ and $\mathcal{Y} = \frac{1}{2}\cdot\mathcal{X}$. □

In particular, Lemma 2.6 implies that given a nonempty set of constraints, there always exists a leaf for which the total weight of its upper occurrences divided by the total weight of its upper and lower occurrences is $\geq \frac{1}{3}$. This is used by Algorithm *Approximation A*, shown in Figure 2.5, to obtain a factor 3 approximation algorithm for MLC/M3LC.

```
Algorithm   Approximation A
Input:   An instance of MLC or M3LC.
Output:  A rooted, unordered tree which is consistent with a subset of the con-
         straints whose total weight is at least one third (at least one half if
         all constraints contain four different leaves) of the total weight of all
         constraints.
1   $REMAINING \leftarrow T$
2   $LEAVES \leftarrow S$
3   $R \leftarrow$ a rooted tree consisting of a single, unlabeled node $v$
4   while  $REMAINING \neq \emptyset$  do
4.1     Pick a leaf $\ell$ in $LEAVES$ which achieves the maximum ratio between
        its total weight of upper occurrences and its total weight of upper and
        lower occurrences in the constraints in $REMAINING$.
4.2     $L \leftarrow$ the set of constraints in $REMAINING$ which contain $\ell$
4.3     $REMAINING \leftarrow REMAINING \setminus L$
4.4     $LEAVES \leftarrow LEAVES \setminus \{\ell\}$
4.5     Extend $R$ by adding two children to $v$; label the first child by $\ell$ and
        set $v$ to the second child.
    endwhile
5   Extend $R$ by adding $|LEAVES|$ children to $v$, label them uniquely with ele-
    ments in $LEAVES$, and return $R$.
End  Approximation A
```

Figure 2.5: A polynomial-time 3-approximation algorithm for MLC/M3LC.

Theorem 2.7 *Algorithm* Approximation A *constructs a tree which is consistent with a subset of the constraints whose total weight is at least one third (at least one half if all constraints contain four different leaves) of the total weight of all the constraints in* $O((m+n)\log n)$ *time.*

Proof: By Lemma 2.6 and the choice of ℓ in step 4.1, the ratio between the total weight of upper occurrences and the total weight of upper and lower occurrences for ℓ in the constraints in $REMAINING$ is at least one third. All constraints in L in which ℓ has an upper occurrence are consistent with R by the construction of R. Thus, every time the algorithm has performed step 4.5, R is consistent with a subset of $T \setminus REMAINING$ whose total weight is at least one third of the total weight of all the constraints in $T \setminus REMAINING$.

To implement steps 4.1 and 4.4 efficiently, we arrange $LEAVES$ in a priority queue ordered by the ratio between the total weight of their upper occurrences and the total weight of their upper and lower occurrences in constraints in $REMAINING$. All priority queue operations (creating the priority queue, selecting the ℓ's, and updating the priority queue after step 4.3) take a total of $O((n+m)\log n)$ time.

To implement steps 4.2 and 4.3, we lexicographically sort T four times according to four cyclic permutations of the four leaves in each constraint. For $i = 1, ..., 4$, the ith permutation puts the ith leaf as the first, the $(i+1)$st (in the cyclic order) as the second, etc. Next, four search trees are built based on the sorted lists. Using the search trees, we can find L in $REMAINING$ and remove it from $REMAINING$ in $O(|L|\log n)$ time. We conclude that steps 4.2 and 4.3 take a total of $O((m+n)\log n)$ time, including the preprocessing. □

The rooted tree produced by Algorithm *Approximation A* has the form of a linear chain with singular leaves pending, where only the last internal node on the chain can have degree larger than two. Algorithm *Approximation A* can be used to obtain a more balanced tree by modifying it to return the subset of input constraints which are satisfied by the constructed tree instead of the tree itself; a rooted tree of minimum height consistent with at least one third of the input constraints is then obtained in $O(mn\log n)$ time by running the algorithm of Aho *et al.* (see Section 2.2) on the instance consisting of these constraints.

The absolute approximation factors of three and two, respectively, for Algorithm *Approximation A* are worst-case optimal since any tree can satisfy at most one of the three constraints $(\{a,b\},c)$, $(\{b,c\},a)$, and $(\{c,a\},b)$, and therefore at most one third of the constraints from a sequence $(\{a_i,b_i\},c_i)$, $(\{b_i,c_i\},a_i)$, $(\{c_i,a_i\},b_i)$, $i = 1, ..., k$; similarly, for the case in which all constraints contain four different leaves, the sequence $\{a_i,b_i\} < \{c_i,d_i\}$, $\{c_i,d_i\} < \{a_i,b_i\}$, $i = 1, ..., k$, causes a lower bound of two.

However, if the quality of an approximation is measured relative to the total weight of constraints consistent with an optimal solution, polynomial-time approximation algorithms with better approximation factors might exist. If the minimum number of constraints necessary to delete in order to build a tree for the remaining constraints is small and the number of constraints is high compared to the number of leaves, an approach different from that of Algorithm *Approximation A* can be more useful. In the next subsection, we give an approximation algorithm for M3LC with better relative performance than 3 in such cases.

2.4.2 Algorithm *Approximation B*

Here, we present a polynomial-time approximation algorithm for M3LC which is based on the algorithm of Aho *et al.* (see Section 2.2). We analyze its worst-case performance and explain how to implement it with the techniques of Henzinger *et al.* (also described in Section 2.2).

High-level description of Algorithm *Approximation B*

Our approximation algorithm for M3LC called Algorithm *Approximation B* mimics the algorithm of Aho *et al.* for 3LC with two modifications:

- In Algorithm *Approximation B*, the graphs $G(L)$ are edge-weighted. Every time a graph $G(L)$ is built for some subset L of S, we set the weight of each edge in $E(L)$ equal to the total weight of the constraints which induce it.

- Whenever the algorithm of Aho *et al.* is stuck at a non-divisible subset L of the set of leaves and has to return the null tree, Algorithm *Approximation B* finds a minimum weight edge cut of $G(L)$ with respect to the current set of constraints. Next, the edges of the min-cut are deleted from $G(L)$ and the resulting connected components of $G(L)$ are computed. Consequently, the constraints that induce the edges of the min-cut are also deleted[7]. Approximate trees for the new components are then constructed and connected by a common parent node.

Worst-case performance of Algorithm *Approximation B*

Let (S, T) be an instance of M3LC, and let R be the tree produced by Algorithm *Approximation B* on (S, T). Define t as the minimum total weight of constraints needed to remove from T such that there exists a rooted tree consistent with all remaining constraints.

Lemma 2.8 *The total weight of constraints in T which are not consistent with R is at most height(R) times t.*

Proof: Let J be a subset of T with minimum total weight (i.e., equal to t) such that there exists a rooted tree which is consistent with all constraints in $T \setminus J$.

Suppose that Algorithm *Approximation B* at some stage looks for a min-cut in a currently connected component A. Let T_A be the subset of T consisting of constraints with all three leaves belonging to the set of vertices in A, and let J_A equal $J \cap T_A$.

Assume that deleting the set of edges corresponding to constraints in J_A would not disconnect A. Then even more constraints belonging to T_A than just J_A would have to be deleted in order to disconnect A. Let i be the first recursion level at which the vertices of A are placed in at least two different components by the algorithm of Aho *et al.* when applied to the instance $(S,\ T \setminus J)$. Since this is the first time that the vertices in A are separated, the algorithm will not delete any constraints in T_A until after it has reached recursion level i (recall that the algorithm deletes constraints only when their leaves are contained in different components). Thus, all constraints in T_A except those in J_A remain

[7]By the construction of $G(L)$, each constraint contributes to just one edge, so deleting an edge in $G(L)$ corresponds to deleting one or more constraints from T.

when reaching recursion level i, and the assumption above implies that the vertices belonging to A would still be connected. Contradiction. Hence, A will be split into at least two components if J_A is deleted.

Clearly, the weight of a min-cut of A is less than or equal to the total weight of J_A. Now, it is sufficient to observe that the subsets J_A for distinct A's on the same recursion level of Algorithm *Approximation B* are pairwise disjoint so that on each recursion level, the total weight of deleted constraints which are not consistent with R is $\leq t$. $\square$

Since $height(R) \leq n$, we have:

Theorem 2.9 *Algorithm* Approximation B *constructs a rooted tree which is consistent with a subset of the constraints in T whose total weight is greater than or equal to $W - nt$, where W is the total weight of all the input constraints.*

Note that the number of constraints in T might be cubic in n and that Algorithm *Approximation B* yields a better approximation factor than Algorithm *Approximation A* for M3LC whenever $t < \frac{2W}{3n}$, i.e., when the optimal solution contains a large number of the input constraints.

Implementing Algorithm *Approximation B*

Algorithm *Approximation B* can be implemented by modifying steps 1 and 5 in the algorithm of Henzinger *et al.* (see Section 2.2). The result is displayed in Figure 2.6. Whenever a minimum weight edge cut is computed in step 5, those edges are deleted from U and the dynamic graph connectivity data structure. Also, the corresponding vertices and their incident edges are deleted from D.

To find minimum weight edge cuts in step 5, we can use a deterministic algorithm by Nagamochi and Ibaraki [98] or Stoer and Wagner [118], or a randomized Monte Carlo-algorithm by Karger [79].

Let M be the initial number of edges in the graph U. As pointed out in Section 2.2, $M = \min\{O(m),\, O(n^2)\}$ follows directly by the construction of U. The next lemma provides a lower bound on M which is used below when analyzing the running time of Algorithm *Approximation B*.

Lemma 2.10 $M \geq \frac{m}{n}$.

Proof: Partition the m input constraints into n sets $T_1, ..., T_n$ by placing each constraint of the form $(\{\cdot,\cdot\}, x)$ into T_x. By the pigeonhole principle, at least one of the resulting sets contains $\geq m/n$ constraints; let T_k be such a set. Initially, there is at least one edge in U for each constraint in T_k (e.g., the edge $\{a,b\}$ if $(\{a,b\},k) \in T_k$). Thus, $M \geq |T_k| \geq \frac{m}{n}$. $\square$

```
Algorithm   Approximation B
Input:      An instance of M3LC.
Output:     A rooted, unordered tree which is consistent with a subset of the con-
            straints whose total weight is greater than or equal to $W - nt$, where
            $W$ is the total weight of all input constraints and $t$ is the minimum
            total weight of constraints needed to remove to achieve consistency.

1  Construct $U$ and $D$. Assign 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
   $(\{a, b\}, \cdot)$.
2  Color all edges in $U$ yellow and initialize the dynamic graph connectivity
   algorithm.
3  Create the root of $R$. Initialize components information.
4  for each maximal vertex $\{x, y\}$ in $D$ do
       If $\{x, y\}$ is a yellow edge in $U$ then color it red, delete it in the dynamic
       graph connectivity data structure, and query the dynamic connectivity
       algorithm "are $x$ and $y$ connected?"; if "no" then update components in-
       formation and $R$.
   endfor
5  For each old yellow component $C_i$, if $|C_i| > 1$ and $C_i$ was not split into at least
   two components during step 4 in this iteration then find a minimum weight
   edge cut of $C_i$ with respect to currently yellow edges, delete the edges in the
   cut, and update components information and $R$.
6  If $|C_i| = 1$ for every yellow component $C_i$ then return $R$.
   Otherwise, find all separable red edges and delete these edges from $U$ and the
   corresponding vertices (plus their incident edges) from $D$.
7  Go to step 4.
End Approximation B
```

Figure 2.6: Using the techniques of Henzinger *et al.* to implement Algorithm *Approximation B*.

Theorem 2.11 *Algorithm* Approximation B *can be implemented to run in*

1. $\min\{O(mn^2 + n^3 \log n),\, O(n^4)\}$ *time, or*

2. $\min\{O(mn \log^3 n),\, O(n^3 \log n)\}$ *time, giving a solution that with high probability*[8] *attains the approximation factor stated in Theorem 2.9.*

Proof: A minimum weight edge cut of U can be computed deterministically in $O(Mn + n^2 \log n)$ time [98, 118] or with high probability in $\min\{O(M \log^3 n),\, O(n^2 \log n)\}$ time [79]. In the worst case, this has to be done $n - 1$ times. Thus, the calls to the min-cut procedure take a total of $O(Mn + n^2 \log n) \cdot O(n) =$

[8] *With high probability* means with probability greater than $(1 - \frac{1}{n^c})$ for some constant $c > 1$.

$O(Mn^2+n^3\log n)$ or $\min\{O(M\log^3 n),\, O(n^2\log n)\}\cdot O(n) = \min\{O(Mn\log^3 n),\, O(n^3\log n)\}$ time, respectively. All other operations are performed as in the algorithm of Henzinger *et al.*, and therefore take $O(n+m+M\log^2 n)$ time if we use the deterministic algorithm for dynamic graph connectivity of Holm *et al.* [65] by the comments following Theorem 2.1.

In the deterministic case, the total running time of Algorithm *Approximation B* is $O(Mn^2+n^3\log n)+O(n+m+M\log^2 n) = O(Mn^2+n^3\log n) = \min\{O(mn^2+n^3\log n),\, O(n^4)\}$.

In the randomized case, the total running time becomes $\min\{O(Mn\log^3 n),\, O(n^3\log n)\}+O(n+m+M\log^2 n)$. Since $O(Mn\log^3 n)+O(n+m+M\log^2 n) = O(Mn\log^3 n+m)$, which is $O(Mn\log^3 n)$ by Lemma 2.10, and $O(n^3\log n)+O(n+m+M\log^2 n) = O(n^3\log n)$, we see that the total running time is $\min\{O(mn\log^3 n),\, O(n^3\log n)\}$.

To ensure that Algorithm *Approximation B* succeeds with high probability in the randomized case, we utilize an amplified version of Karger's minimum weight edge cut algorithm [79] running in $\min\{O(M\log^3 n),\, O(n^2\log n)\}$ time and having success probability at least $1-\frac{1}{n^3}$ (the success probability can be amplified from at least $1-\frac{1}{n}$ to at least $1-\frac{1}{n^c}$ for any constant $c>1$ without increasing the asymptotic running time by making c independent calls to the min-cut algorithm and selecting the best solution found; the probability of the min-cut algorithm failing all c times is $\le \frac{1}{n^c}$). During its execution, Algorithm *Approximation B* calls the amplified min-cut algorithm d times, where $d \le n-1$. Let A_i denote the event that the ith call fails. Algorithm *Approximation B* fails if and only if the amplified min-cut algorithm fails one or more times; hence, the probability that Algorithm *Approximation B* fails is $\mathbf{Pr}\left[\bigcup_{i=1}^{d} A_i\right] \le \sum_{i=1}^{d}\mathbf{Pr}[A_i] \le (n-1)\cdot\frac{1}{n^3} < \frac{1}{n^2}$. □

2.5 Concluding Remarks

The fastest known algorithm for LC is still the one by Aho *et al.* [3] with $O(mn\log n)$ running time. In Section 2.2, we observed that 3LC can be solved in $\min\{O(n+m\log^2 n),\, O(m+n^2\log^2 n)\}$ time with the algorithm of Henzinger *et al.* [62] by substituting the used decremental graph connectivity algorithm with a more efficient one by Holm *et al.* [65].

We have given a new proof for the NP-hardness of MLC and M3LC in Section 2.3, and then, in Section 2.4, shown that MLC and M3LC and their weighted versions can be approximated within factor of 3 of the optimum in $O((m+n)\log n)$ time (Algorithm *Approximation A*). Moreover, M3LC and its weighted version can be approximated as follows: An approximate solution consistent with a subset of the input constraints whose total weight is at least $W-nt$, where W is the total weight of all input constraints and t is the minimum total weight of constraints needed to remove in order to achieve consistency, can be constructed in $\min\left\{O(mn^2 + n^3 \log n),\ O(n^4)\right\}$ time, or with high probability in $\min\left\{O(mn \log^3 n),\ O(n^3 \log n)\right\}$ time (Algorithm *Approximation B*). Here, Algorithm *Approximation B* has a better approximation ratio than Algorithm *Approximation A* for instances of M3LC with $t < \frac{2W}{3n}$.

The main open problem concerns the approximability of MLC and M3LC. Our Algorithm *Approximation A* approximates both problems within a constant factor in polynomial time, but we do not know whether it is possible to find a polynomial-time approximation scheme (PTAS) for either of them. Even if the problems are MAX SNP-hard, implying that no PTAS exists unless P=NP [8], it might still be useful to find polynomial-time algorithms with better approximation factors than Algorithms *Approximation A* and *Approximation B*. Special cases of the problems might be easier to approximate; for example, we believe that it is possible to construct a PTAS for the restriction of M3LC to *complete* instances in which the input contains one 3-leaf constraint for each cardinality 3 subset of the leaves[9].

Semple and Steel [110] have independently developed a heuristic for combining a set of rooted, unordered leaf-labeled trees with overlapping leaf sets which uses the same basic idea of finding minimum weight edge cuts in the graphs $G(L)$ as our Algorithm *Approximation B*, with some modifications. However, no approximation factor for their algorithm in terms of how many of the input trees that are consistent with the output tree was given in [110] as their main focus was on proving how well nestings shared by all the input trees can be preserved (and some other related properties) rather than trying to maximize the number of consistent 3-leaf constraints in the output tree.

The other open question is: How much can the asymptotic running times of Algorithm *Approximation B* and the exact polynomial-time algorithms for LC and 3LC be improved? The running time of Algorithm *Approximation B* is currently dominated by the time it takes to compute minimum weight edge cuts in undirected graphs. If faster algorithms for computing min-cuts are invented, the efficiency of Algorithm *Approximation B* can be improved accordingly. Faster algorithms for decremental dynamic graph connectivity will not help the asymptotic running time of Algorithm *Approximation B* unless faster min-cut algorithms are also employed. On the other hand, by Theorem 2.1, faster algorithms

[9]The motivation for this is as follows. In *the quartets consistency problem* (QC), the input is a set $\mathcal{Q}$ of *resolved quartets* (unrooted, leaf-labeled trees each having 4 leaves and no nodes of degree 2), and the goal is to find an unrooted, leaf-labeled tree which is consistent with all of the quartets in $\mathcal{Q}$, if one exists. *The maximum quartets consistency problem* (MQC) is the optimization version of QC, and *complete* MQC is MQC restricted to instances where one quartet for each cardinality 4 subset of the leaves is included in $\mathcal{Q}$. QC is NP-hard [116], and although MQC is MAX SNP-hard in general [69, 116] and complete MQC remains NP-hard [69], complete MQC admits a PTAS [69]. In contrast, 3LC can be solved exactly in polynomial time by the algorithms described in Section 2.2. Thus, since 3LC seems so much easier than QC, it would be surprising if complete M3LC was much harder to approximate than complete MQC. Therefore, we conjecture that complete M3LC has a PTAS as well.

for decremental dynamic graph connectivity will directly improve the running time of the algorithm of Henzinger *et al.* for 3LC.

Chapter 3

Inferring Ordered Trees from Rooted Triples

We now consider the problem of inferring an *ordered*, leaf-labeled tree from a set of rooted triples, where in addition to the input set of rooted triples, a given left-to-right ordering is imposed on the leaves. We call this problem *the ordered 3-leaf constraints consistency problem* (O3LC). As in Chapter 2, we are also concerned with the corresponding maximization problem, here termed *the maximum ordered 3-leaf constraints consistency problem* (MO3LC).

The algorithms for inferring ordered trees presented in this chapter are in general more efficient than the corresponding fastest known algorithms for inferring unordered trees. In fact, our algorithm for MO3LC runs in polynomial time whereas the analogous maximization problem for unordered trees is NP-hard (as proved in Chapter 2).

An essential part of our algorithm for O3LC is an algorithm for maintaining the union of a set of closed intervals under a sequence of deletions from the set. Therefore, in this chapter we also develop an efficient decremental interval union algorithm.

3.1 Introduction

Let S be a finite set of elements. A *3-leaf constraint on S*, also referred to as a *rooted triple on S*, is a constraint of the form $(\{i, j\}, k)$, where $i, j, k \in S$, which specifies that the lowest common ancestor of i and j is a proper descendant of the lowest common ancestor of i and k[1]. A rooted tree whose leaves are distinctly labeled by elements in S (i.e., no two leaves have the same label) and a 3-leaf

[1] Or equivalently, that the lowest common ancestor of i and j is a proper descendant of the lowest common ancestor of j and k.

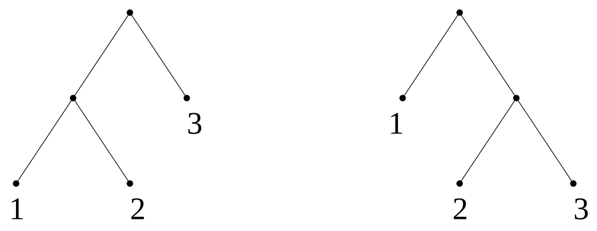

Figure 3.1: Let $\mathcal{O}$ be the ordering $1, 2, 3$. The tree on the left complies with $\mathcal{O}$ and is consistent with the 3-leaf constraint $(\{1, 2\}, 3)$. The tree on the right is consistent with the constraint $(\{2, 3\}, 1)$ and Note that no tree can satisfy the constraint $(\{1, 3\}, 2)$ and comply with $\mathcal{O}$ at the same time.

constraint on S which is satisfied in the tree are said to be *consistent* with each other.

The *leaf ordering* of a rooted, ordered, leaf-labeled tree is the sequence of labels obtained by scanning its leaves from left to right.

Next, we define two computational problems called O3LC and MO3LC, similar to the problems 3LC and M3LC studied in Chapter 2. The difference is that in O3LC and MO3LC, the leaf ordering of the constructed tree is required to comply with a specified ordering. See Figure 3.1 for an example.

The ordered 3-leaf constraints consistency problem (O3LC)

Instance: Finite set S, set T of 3-leaf constraints on S, ordering $\mathcal{O}$ of S.

Output: A rooted, ordered tree with leaves distinctly labeled by S whose leaf ordering equals $\mathcal{O}$ and which is consistent with all of the constraints in T, if one exists; otherwise, null.

The maximum ordered 3-leaf constraints consistency problem (MO3LC)

Instance: Finite set S, set T of 3-leaf constraints on S, ordering $\mathcal{O}$ of S.

Output: A rooted, ordered tree with leaves distinctly labeled by S whose leaf ordering equals $\mathcal{O}$ and which is consistent with as many of the constraints in T as possible.

Henceforth, we assume without loss of generality that the elements of S are named $\{1, 2, ..., n\}$ and that the given ordering $\mathcal{O}$ is precisely the sequence $1, 2, ..., n$. (Any given instance can be transformed into an instance of this type by relabeling before running our algorithms.)

As in Chapter 2, we denote the number of constraints in T by m. It follows from the problem definitions that $m = O(n^3)$ for O3LC and MO3LC.

3.1.1 Motivation

Applications of 3LC and M3LC (the unordered versions of O3LC and MO3LC) were discussed in Section 2.1.1. We study O3LC and MO3LC primarily because they are special cases of 3LC and M3LC which turn out to be more efficiently solvable. As mentioned in Chapter 2, it is not always possible to construct a rooted tree which is consistent with all of the 3-leaf constraints in a given set since some constraints may contradict each other. In such cases, an algorithm which produces a tree consistent with as many of the 3-leaf constraints as possible is more useful. Unfortunately, to construct an *unordered* tree for the maximum number of 3-leaf constraints was proved to be NP-hard in Section 2.3, implying that no efficient algorithm for this problem exists unless P=NP. However, if we are given the leaf ordering of the final tree in advance, the situation improves drastically: dynamic programming can be applied to solve MO3LC in polynomial time, as we will see later in this chapter.

In certain evolutionary tree construction situations, it may be possible to determine or accurately estimate the leaf ordering of a planar embedding of the true tree by taking into account other kinds of data such as the geographical distributions of the species or data based on some measurable quantitative characteristic (average life span, size, etc.) which can be sorted to obtain a linear ordering of the species. O3LC and MO3LC might also arise in graph drawing applications where a leaf-labeled tree has to be inferred from a set of 3-leaf constraints and additional restrictions are placed on the leaves (for example, that they must be ordered alphabetically) for ease of presentation.

3.1.2 Previous Results

No algorithms or computational complexity results for O3LC and MO3LC can be found in the literature since these problems have not been studied before. For known results related to 3LC, M3LC, and the corresponding quartet consistency problems QC and MQC, please refer to Chapter 2 and the references therein.

As for the problem of maintaining the union of a set Y of closed intervals under a sequence of deletions from Y, a general, fully dynamic interval union algorithm which also supports insertions of new intervals into Y was given by Cheng and Janardan in [26]. Their algorithm allows interval insertions and deletions be carried out in $O(\log n)$ time and the list of intervals in the union to be reported in $O(k)$ time, where n is the number of intervals currently in Y and k is the current number of maximal nonoverlapping intervals in the union of Y. However, since we only need a decremental interval union algorithm in our algorithm for O3LC and the algorithm of Cheng and Janardan is somewhat complicated, we will provide a simpler solution for the decremental case.

3.1.3 Our Contributions

We observe that in the algorithm of Henzinger *et al.* [62] for 3LC (see Section 2.2), if the tree being constructed is required to have a specified leaf ordering then yellow edges and yellow components in the auxiliary graph U can be represented as closed intervals and maximal nonoverlapping intervals in their union, respectively. Coloring yellow edges of U red thus corresponds to deleting intervals from a given set of closed intervals, which allows us to implement the decremental dynamic graph connectivity computations needed in the algorithm of Henzinger *et al.* with a (faster) decremental interval union algorithm. We develop such a decremental interval union algorithm and use it to obtain an algorithm called Algorithm *Fast O3LC* which solves O3LC in $O((m+n)\log n)$ time.

Given a set Y of M closed intervals on the real line, our decremental interval union algorithm uses $O(M \log M)$ time for preprocessing, and then maintains the union of Y under a sequence of δ interval deletions in $O(\delta \log M + M + k_\delta \log M)$ time, where k_δ is the final number of maximal nonoverlapping intervals in the union. After each deletion, the newly created maximal nonoverlapping intervals in the union can be listed without increasing the asymptotic time complexity. The interval intersection query (i.e., given an interval q, which maximal nonoverlapping intervals in the union of the current Y intersect q?) can be answered in $O(\log M + R)$ time, where R is the number of maximal nonoverlapping intervals in the union to report.

We also give a dynamic programming-based algorithm for MO3LC which runs in $O(n^3)$ time.

In Section 3.2, we recall the definitions of segment trees, interval trees, and interval tries and state some useful facts about these data structures. In Section 3.3, we describe our decremental interval union algorithm which is the basis of Algorithm *Fast O3LC* presented in Section 3.4. We present the cubic-time algorithm for MO3LC in Section 3.5. Section 3.6 summarizes our results and proposes some generalizations of the considered problems.

3.2 Preliminaries

The decremental interval union algorithm in Section 3.3 uses segment trees, interval trees, and interval tries. Below, we briefly review these three data structures and state some known facts from [94] and [102].

3.2.1 Segment Trees

The *segment tree* is a data structure for storing a set of intervals on the real line $\mathbb{R}$ along with some additional information.

Let Y be a set of intervals on $\mathbb{R}$ whose endpoints belong to a set $\mathcal{U} = \{x_i\}_{i=1}^{n} \subset \mathbb{R}$, where $x_i < x_{i+1}$ for all $i = 1, 2, ..., n-1$. Denote by (x_i, x_{i+1}) the open interval from x_i to x_{i+1} and let $[x_i, x_i]$ denote the point x_i. The segment tree for Y with respect to $\mathcal{U}$ is a balanced binary search tree of depth $O(\log n)$ with $2n+1$ leaves corresponding to (from left to right) the intervals $(-\infty, x_1)$, $[x_1, x_1]$, (x_1, x_2), $[x_2, x_2]$, ..., (x_n, ∞) called its *atomic segments*.

Every node u of the segment tree is associated with a *node list* $NL(u)$. To describe $NL(u)$, we first define $xrange(u)$ as follows. If u is a leaf, $xrange(u)$ is simply the corresponding atomic segment. If u is an internal node, let $xrange(u) = xrange(l) \cup xrange(r)$, where l and r are the left and right child of u. Now, define $NL(u)$ as $\{I \in Y \mid xrange(u) \subseteq I \ \& \ xrange(parent(u)) \nsubseteq I\}$.

The following is proved in [94] (see also [33] or [107]).

Fact 3.1 [94] *A segment tree for a set of M intervals with both endpoints in a subset $\mathcal{U}$ of $\mathbb{R}$ of size n can be constructed in $O(M \log n)$ time. An interval with both endpoints in $\mathcal{U}$ can be inserted into or deleted from the segment tree in $O(g(M) \log n)$ time, where $g(M)$ is the time required to insert or delete an interval from a node list of size M. Every interval in Y is stored in at most two different node lists at each level of the tree, i.e., in a total of $O(\log n)$ different node lists. Furthermore, given an interval I in Y, all nodes u with $I \in NL(u)$ can be found in $O(\log n)$ time.*

In our application, we will use a modified segment tree in which we only need to keep track of how many intervals from Y that belong to each node list and do not have to worry about which ones they are. Hence, rather than explicitly storing the node lists, we employ *node counters*; the value $NC(u)$ of the node counter for a node u is defined as the cardinality of the corresponding node list $NL(u)$, if we were keeping it. Every deletion from or insertion into $NL(u)$ corresponds to decreasing or increasing $NC(u)$ by one, which can be done in $O(1)$ time. Thus, we have $g(M) = O(1)$.

3.2.2 Interval Trees

The *interval tree* is a kind of binary search tree for storing a set Y of intervals whose left endpoints belong to a fixed finite subset $\mathcal{U}$ of $\mathbb{R}$. It allows intervals to be deleted from Y as well as new intervals with left endpoints in $\mathcal{U}$ to be inserted into Y in logarithmic time. It can report all intervals in Y with nonempty intersection with a given query interval; such a request is called an *interval intersection query*.

The following fact summarizes what we need to know about interval trees for our purposes. For a description of how information is stored in interval trees, how interval trees are constructed and updated efficiently, how queries are handled, etc., see [33], [94], or [107].

Fact 3.2 [94] *Suppose that the left endpoints of the intervals in a set Y belong to a subset $\mathcal{U}$ of $\mathbb{R}$ of size n and $|Y| = M$. An interval tree Q of depth $O(\log n)$ for Y can be constructed in $O(n + M \log nM)$ time. Each insertion into Y of an interval with left endpoint in $\mathcal{U}$ and each deletion from Y can be performed by Q in $O(\log M + \log n)$ time. The interval intersection query is supported by Q in $O(\log n + R)$ time, where R is the number of reported intervals.*

3.2.3 Interval Tries

The *interval trie* is another data structure for maintaining a set Y of intervals under a sequence of insertions and deletions. It requires that all interval endpoints have integer coordinates between 1 and a fixed number n. The interval trie can be employed to efficiently answer *stabbing queries* in which a query point q is given and the object is to return a list of all intervals in Y that contain q. The next fact, following from [102], characterizes interval tries.

Fact 3.3 [102] *Let Y be a set of M intervals with both endpoints in $\{1, 2, ..., n\}$. For any constant $\epsilon > 0$, there is an interval trie for Y which allows each insertion into Y and each deletion from Y to be performed in $O(\log^{\epsilon} n + \log\log n)$ time and which answers the stabbing query in $O(\frac{\log n}{\epsilon \cdot \log\log n} + R)$ time, where R is the number of reported intervals. It can be constructed in $O(n \cdot \frac{\log n}{\epsilon \cdot \log\log n} + M \cdot \log^{\epsilon} n)$ time.*

The parameter ϵ can be chosen to get the desired tradeoff between update time and query time.

3.3 A Decremental Interval Union Algorithm

Here we present an algorithm for maintaining the union of a set Y of closed intervals under a sequence of deletions from Y.

For the rest of this section, let M be the number of intervals in Y at start, and let $s_1, ..., s_\delta$ be the sequence of intervals to delete from Y, where all of $s_1, ..., s_\delta$ initially belong to Y. Denote by $\mathcal{U}$ the set of endpoints of the intervals in Y; clearly, $|\mathcal{U}| \leq 2M$. The maximal nonoverlapping intervals covered by Y are called *interval union components*.

The first step of our method consists of the construction of a segment tree W for Y. Since $|\mathcal{U}| = O(M)$, W can be constructed in $O(M \log M)$ time by Fact 3.1.
For efficiency, we use node counters (denoted by NC) instead of the standard node lists NL, as explained at the end of Section 3.2.1. Recall that $NC(u)$ for a node u of W is defined as the cardinality of $NL(u)$. Additionally, for each node u of W, we set a special bit $r(u)$ to 1 if and only if all NC counters along the path from the root of W to u, including $NC(u)$, are set to zero. We can

determine the initial values of $r(u)$ for all nodes of W by traversing W in a top-down fashion in time proportional to the size of W, i.e., in $O(M)$ time. The usefulness of the r-bits stems from the next lemma.

Lemma 3.4 *For any leaf w of W, $r(w) = 0$ if and only if the atomic segment corresponding to w is covered by an interval union component.*

Proof: Let $\mathbf{p}$ be the path in W from the root to w. If $r(w) = 0$ then there exists a node t on $\mathbf{p}$ with $NC(t) > 0$. Since $xrange(t) \subseteq I$ for some interval I in Y and $xrange(w) \subseteq xrange(t)$, it follows that $xrange(w)$ is covered by I and thus by one of the interval union components. If $r(w) = 1$ then since every node in the tree whose $xrange$ overlaps with $xrange(w)$ lies on $\mathbf{p}$ and therefore has its NC counter set to zero, $xrange(w)$ is not contained in any interval in Y. $\square$

Next, we augment each leaf w of W with two pointers which point to the *left* and *right neighbor of* w, defined as the predecessor of w and the successor of w in the consecutive left-to-right ordering of W's leaves. This can be done in $O(M)$ time.

Besides the segment tree W, we use an interval tree Q in order to maintain the interval union components of the current Y. The set of left endpoints of intervals to be stored in Q is a subset of the set $\mathcal{U}$ of all endpoints of intervals in Y, and hence $O(M)$. We can find the interval union components for the initial Y in $O(M \log M)$ time by sorting all endpoints in Y and using a standard sweep-line technique. By Fact 3.2, we can then construct Q and insert the initial interval union components in $O(M \log M)$ time.

In total, the preprocessing steps described above take $O(M \log M)$ time.

Deleting the interval s_1 from Y may affect the current interval union components, as illustrated by the example in Figure 3.2. W and Q need to be modified accordingly; see Figure 3.3. To update W and Q, we proceed as follows.

To begin with, we locate the $O(\log M)$ nodes of W whose node lists would contain s_1 if we were keeping them, and decrease their NC counters by one. This can be done in $O(\log M)$ time by Fact 3.1. If, for some node u, $NC(u)$ drops to 0 and $r(parent(u)) = 1$, 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 $r(w)$ becomes 1 due to the updating ($LL(u)$ can be empty). Next, we concatenate the lists $LL(u)$ in the order of the ranges of the nodes u and let L denote the resulting list. The updating of the $r(w)$'s as well as the construction of the list L take time proportional to the number r_1 of nodes that got their $r(w)$-bit set to 1 for the first time.

We then find the interval union component $\mathcal{C}$ that s_1 belongs to by querying Q with s_1. Let x be the left neighbor of the leaf in W corresponding to the left endpoint of $\mathcal{C}$ and let y be the right neighbor of the leaf in W corresponding to the right endpoint of $\mathcal{C}$. We insert x first in L, and similarly, insert y last in L.

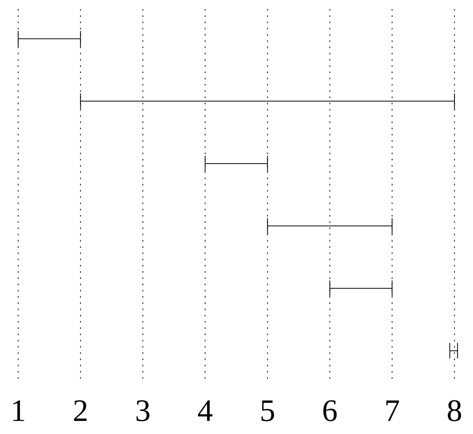

Figure 3.2: The union of the intervals $\{[1,2],[2,8],[4,5],[5,7],[6,7],[8,8]\}$ is a single interval union component $\mathcal{C}=[1,8]$. Deleting the interval $[2,8]$ results in $\mathcal{C}$ being split into three interval union components: $[1,2]$, $[4,7]$, and $[8,8]$.

Now, Lemma 3.4 implies that for any pair of consecutive leaves a, b in L that are not neighbors of each other, the right neighbor of a and the left neighbor of b yield a new interval union component. Hence, we delete $\mathcal{C}$ from Q and insert the new interval union components into Q. Let k_0 be the number of interval union components of the initial Y, and let k_1 be the number of interval union components after the deletion of s_1. The interval intersection query takes $O(\log M)$ time (there is exactly one interval union component with nonempty intersection with s_1), and each deletion and insertion takes $O(\log M)$ time by Fact 3.2. Thus, updating Q takes $O(\log M + \log M + (k_1 - k_0 + 1)\log M) = O((k_1 - k_0 + 1)\log M)$ time.

We conclude that we can update W and Q to maintain the union of the set of intervals resulting from the deletion of s_1 from Y in $O(\log M + r_1)$ and $O((k_1 - k_0 + 1)\log M)$ time, respectively.

The next interval deletions are handled similarly. We define r_i and k_i for $i = 2, ..., \delta$ in the same way as we defined r_1 and k_1 above. It follows that the sum $\sum\limits_{i=1}^{\delta} r_i$ is bounded from above by the size of W, i.e., $O(M)$. Furthermore, by telescoping, we have

$$\sum_{i=1}^{\delta}(k_i - k_{i-1} + 1) \;=\; k_\delta - k_0 + \delta \;<\; k_\delta + \delta.$$

Summarizing, we obtain the main result of this section:

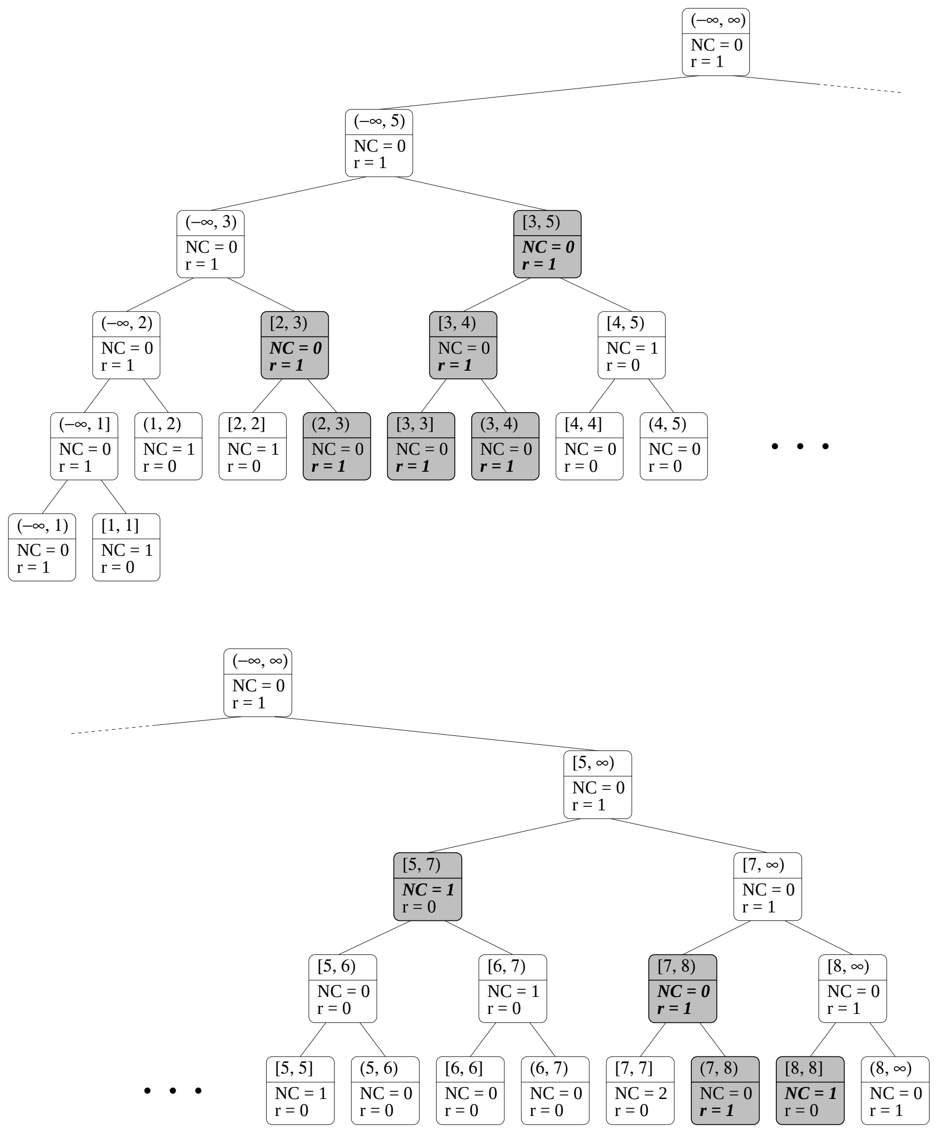

Figure 3.3: The segment tree for the example in Figure 3.2 after the deletion of the interval $[2, 8]$. Affected nodes are shaded, and changes in NC counters and r-bits are italicized. Here, $L = ((-\infty, 1), (2, 3), [3, 3], (3, 4), (7, 8), (8, \infty))$, so the left and right endpoints of the new interval union components to be inserted in Q are given by $[1, 1]$ and $[2, 2]$, $[4, 4]$ and $[7, 7]$, and $[8, 8]$ and $[8, 8]$.

Theorem 3.5 *Let Y be a set consisting of M closed intervals on the real line. After $O(M \log M)$ time preprocessing, the union of Y can be maintained under a sequence of δ interval deletions in $O(\delta \log M + M + k_\delta \log M)$ time, where k_δ is the final number of interval union components. After each deletion, the newly created interval union components can be listed without increasing the asymptotic time complexity. The interval intersection query can be answered in $O(\log M + R)$ time, where R is the number of interval union components to report.*

If we only need to answer stabbing queries and all endpoints of the intervals in Y belong to a set $\{1, 2, ..., n\}$, the result can be improved by replacing the interval tree Q with an interval trie. By Fact 3.3, we obtain the following variant of the result.

Theorem 3.6 *Let Y be a set of M closed intervals with both endpoints in $\{1, 2, ..., n\}$. For any constant $\epsilon > 0$, the following holds. After $O(M \log nM + n \cdot \frac{\log n}{\epsilon \cdot \log\log n} + M \cdot \log^\epsilon n)$ time preprocessing, the union of Y can be maintained under a sequence of δ interval deletions in $O(n + \delta(\log n + \log^\epsilon n) + k_\delta(\log^\epsilon n + \log\log n))$ time, where k_δ is the final number of interval union components. After each deletion, the newly created interval union components can be listed without increasing the asymptotic time complexity. The stabbing query can be answered in $O(\frac{\log n}{\epsilon \cdot \log\log n})$ time.*

3.4 Algorithm *Fast O3LC*

In this section, we give an algorithm for O3LC which runs in $O((m + n) \log n)$ time. It combines the algorithm of Henzinger, King, and Warnow [62] for 3LC described in Section 2.2 with our decremental interval union algorithm from Section 3.3.

Recall that given a set $S = \{1, 2, ..., n\}$ and a set T of 3-leaf constraints on S, the algorithm of Henzinger *et al.* constructs a rooted, unordered tree R whose leaves are distinctly labeled by S and which is consistent with all of the input constraints in T, if such a tree exists. It uses an auxiliary data structure consisting of two graphs U and D defined as:

- $U = (S, E)$ whose vertex set is the input set of leaves $S = \{1, 2, ..., n\}$, and where for each constraint $(\{a, b\}, c)$ in T, the edges $\{a, b\}$ and $\{b, c\}$ are in E.
- $D = (S', A)$, where for each constraint $(\{a, b\}, c)$ in T, the vertices $\{a, b\}$ and $\{b, c\}$ are in S', and the directed edge $\{a, b\} \rightarrow \{b, c\}$ is in A.

All edges of U are initially colored yellow. The graph D is used for finding edges in U that are to be colored red; these edges correspond to vertices in D called *maximal vertices* which have no outgoing edges. A *separable red edge* is a red edge of U whose endpoints belong to two different yellow components. A maximal vertex in D and its corresponding red edge e in U are deleted when e becomes a separable red edge.

The tree R is constructed level by level. During the algorithm's execution, each current yellow component of U contains all leaves from the input set S that will belong to the same subtree in the final R, rooted at one of the leaves of the current R.

See Section 2.2 for more details on the algorithm of Henzinger *et al.*

For ordered trees, we make the following crucial observation.

Observation 3.7 *If R' is a rooted, ordered tree, leaf-labeled by S, then for any two leaves a and b it holds that any subtree of R' containing a and b must also contain all leaves in the interval $[a, b]$.*

Below, we denote the ordered tree which we are constructing by R'. By Observation 3.7, a yellow edge in U forces all leaves in the interval defined by its two endpoints to belong to the same subtree of R'. Transitivity implies that the set of yellow edges in U induces a set of intervals such that each of the maximal nonoverlapping intervals in the union of these intervals contains all leaves from S which should end up in one subtree of R'. Hence, to construct R', instead of maintaining the current yellow components of U as in the algorithm of Henzinger *et al.*, we maintain the corresponding interval union components.

Our algorithm is called Algorithm *Fast O3LC* and is listed in Figure 3.4. In addition to the directed graph D defined above, it uses the decremental interval union algorithm from Section 3.3 to keep track of interval union components in the set Y of intervals induced by U, and an interval trie Z to store intervals corresponding to edges in U that have been colored red. By $|C_i|$ we mean the number of leaves included in the interval union component C_i.

The algorithm continues until all interval union components have been split into interval union components which cover a single leaf each, or until it discovers that no ordered tree consistent with all of T exists.

Theorem 3.8 *Algorithm* Fast O3LC *solves O3LC in $O((m + n) \log n)$ time.*

Proof: If Algorithm *Fast O3LC* produces an ordered tree then it is consistent with all constraints in T. To prove this, note that intervals currently in Y covering at least two leaves correspond to current yellow edges of U in the algorithm of Henzinger *et al.* By the remarks following Observation 3.7, each current interval union component of Y contains all leaves which will belong to one subtree of R'. Intervals in Z correspond to red edges of U; furthermore, an

```
Algorithm   Fast O3LC
Input:   An instance of O3LC.
Output:  A rooted, ordered tree $R'$ which is consistent with all of the constraints,
         if one exists; the null tree, otherwise.

1  Construct $D$.
2  Let $Y = \{[a,b] \mid \{a,b\} \in E\} \cup \{[a,a] \mid a \in S\}$ and initialize the decremental
   interval union algorithm on $Y$. Create an (initially empty) interval trie $Z$ for
   storing a set of intervals with both endpoints in $\{1, 2, ..., n\}$.
3  Create the root of $R'$. Initialize components information.
4  for each maximal vertex $\{x,y\}$ in $D$ do
       If $[x,y]$ belongs to $Y$ then delete it from $Y$ and insert it into $Z$, and query
       the decremental interval union algorithm "are $x$ and $y$ in the same interval
       union component of $Y$?"; if "no" then update components information
       and $R'$.
   endfor
5  For each old interval union component $C_i$ of $Y$, if $|C_i| > 1$ and $C_i$ was not
   split into at least two components during step 4 in this iteration then return
   the null tree.
6  If $|C_i| = 1$ for every interval union component $C_i$ of $Y$ then return $R'$.
   Otherwise, for each old interval union component $C_i$ of $Y$ with $|C_i| > 1$, let
   $c_1, c_2, ..., c_j$ be the new interval union components of $Y$ created from $C_i$. For
   $k = 1, ..., j-1$, query $Z$ with the median $m_k$ of the right endpoint of $c_k$ and
   the left endpoint of $c_{k+1}$, and delete all reported (i.e., containing $m_k$) intervals
   from $Z$ and the corresponding vertices (plus their incident edges) from $D$.
7  Go to step 4.
End Fast O3LC
```

Figure 3.4: An algorithm for solving O3LC which combines Henzinger, King, and Warnow's algorithm with our decremental interval union algorithm.

interval corresponds to a separable red edge if and only if it overlaps with at least two new interval union components, i.e., if and only if it is intersected by at least one of the medians m_i, $i = 1, 2, ..., k-1$. If Algorithm *Fast O3LC* fails to produce an ordered tree then there is no such tree by Observation 3.7 and the correctness of the algorithm of Henzinger *et al.*

The construction of the graph D takes $O(m)$ time and the overall time taken to determine maximal vertices is proportional to the size of D, i.e., $O(m)$. By Theorem 3.5, the total time used to compute all interval union components during the course of the algorithm (including the preprocessing) is $O((m+n)\log n)$ since $M = O(m+n)$, $\log M = O(\log n)$, $\delta = O(m)$, and $k_\delta = n$. By Fact 3.3, the total time needed to construct Z, to perform the $O(m)$ insertions into and deletions from Z, and to answer the $O(n)$ stabbing queries to Z is $O(n \cdot \frac{\log n}{\epsilon \cdot \log\log n} + m \cdot (\log^\epsilon n + \log\log n) + n \cdot \frac{\log n}{\epsilon \cdot \log\log n} + m) = O((m+n) \cdot \frac{\log n}{\log\log n})$ (choose, e.g., $\epsilon = 0.5$). $\square$

3.5 A Cubic-Time Algorithm for MO3LC

Here, we present an algorithm for MO3LC which runs in $O(n^3)$ time. It bears resemblance to the well-known cubic-time dynamic programming algorithms for computing a minimum weight triangulation of a simple polygon [32] and for recognizing which strings belong to a given context-free language [90].

We first introduce some new notation.

Definition 3.9 For all $i, j \in \{1, ..., n\}$ with $i \le j$, denote the subset of input constraints on leaf-labels entirely in the interval $\{i, i+1, ..., j\}$ by $T_{i,j}$, and let $m_{i,j}$ be the maximum number of constraints in $T_{i,j}$ consistent with any ordered tree.

Definition 3.10 For all $i, j, k \in \{1, ..., n\}$ with $1 \le i \le k < j \le n$, let $w_{i,k,j}$ be the number of constraints $(\{l_1, l_2\}, l_3) \in T$ which satisfy either

- $\{l_1, l_2\} \subseteq \{i, ..., k\}$ & $l_3 \in \{k+1, ..., j\}$

or

- $l_3 \in \{i, ..., k\}$ & $\{l_1, l_2\} \subseteq \{k+1, ..., j\}$.

The algorithm for MO3LC is based on the following observation. Let $i, j \in \{1, ..., n\}$ with $i < j$. Consider an ordered tree R which is consistent with the maximum number of input constraints in $T_{i,j}$. Without loss of generality, assume that R is binary. The root of R has two children; denote the two subtrees rooted at these nodes by R_1 and R_2, and let K be the label of the rightmost leaf in R_1 (see Figure 3.5). The input constraints in $T_{i,j}$ can be partitioned into three sets: $T_{i,K}$, $T_{K+1,j}$, and those constraints that involve at least one leaf label less than or equal to K and at least one leaf label strictly greater

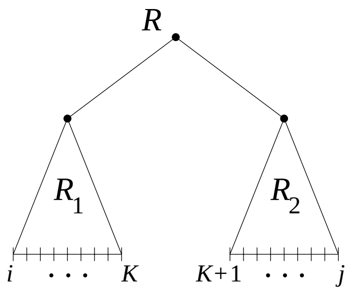

Figure 3.5: R is an optimal ordered tree for $T_{i,j}$. The rightmost leaf in the left subtree R_1 is labeled K, and the leftmost leaf in the right subtree R_2 is labeled $K+1$.

than K. R_1 and R_2 must be consistent with the largest possible number of input constraints in $T_{i,K}$ and $T_{K+1,j}$, respectively, since otherwise there would exist another tree R' which was consistent with more constraints in $T_{i,j}$ than R, contradicting the maximality of R. Next, note that R is consistent with $w_{i,K,j}$ constraints in $T_{i,j} \setminus (T_{i,K} \cup T_{K+1,j})$. We thus obtain a recursive formula for $m_{i,j}$:

$$m_{i,j} = m_{i,K} + m_{K+1,j} + w_{i,K,j}$$

When calculating $m_{i,j}$, we can assume that all values of $w_{i,k,j}$, $m_{i,k}$, and $m_{k+1,j}$, where $k \in \{i, ..., j-1\}$, have been computed already. We don't know K beforehand, but since it must belong to $\{i, ..., j-1\}$, we can try all possibilities, evaluate each resulting tree, and select the best one. This yields:

$$m_{i,j} = \max_{k \in \{i, ..., j-1\}} \{m_{i,k} + m_{k+1,j} + w_{i,k,j}\} \tag{3.1}$$

Equation (3.1) leads to a dynamic programming-based algorithm for MO3LC. It uses an auxiliary $(n \times n)$-matrix M whose (i,j)th entry contains the k which maximizes the value of $m_{i,j}$ in equation (3.1). Initially, all values of $w_{i,k,j}$, where $1 \le i \le k < j \le n$, are computed in $O(n^3)$ time as explained below. Then, for all $i \in \{1, ..., n\}$, $m_{i,i}$ is set to 0 and $M[i,i]$ is set to i. Next, letting l vary from 1 to $n-1$, for every pair $i, j \in \{1, ..., n\}$ with $j - i = l$, the algorithm finds a $k \in \{i, ..., j-1\}$ that maximizes $m_{i,k} + m_{k+1,j} + w_{i,k,j}$, stores this maximal value in $m_{i,j}$, and sets $M[i,j]$ to k. Since every $w_{i,k,j}$ and every previously computed value of $m_{i,k}$ or $m_{k+1,j}$ is accessible in constant time, this step takes a total of $O(n^3)$ time. After all entries of M have been calculated, an optimal ordered tree for any $T_{i,j}$ can be obtained by doing a traceback in $O(j-i+1)$ time. (If $i = j$ then return a tree with a single node labeled by i. Otherwise, let $k := M[i,j]$, recursively construct optimal trees R_1 and R_2 for $T_{i,k}$ and $T_{k+1,j}$, and return a tree consisting of a root node with R_1 and R_2 as its children.) The algorithm returns an optimal ordered tree for $T_{1,n}$ as the final solution.

This proves the next theorem.

Theorem 3.11 *MO3LC is solvable in $O(n^3)$ time.*

It remains to describe how to precompute $w_{i,k,j}$ for all $1 \le i \le k < j \le n$ in $O(n^3)$ time[2]. By Definition 3.10, each input constraint $(\{a,b\},c)$ with $a \le b < c$ increments by one those $w_{i,k,j}$ that satisfy $1 \le i \le a$, $b \le k \le c-1$, and $c \le j \le n$. Similarly, each input constraint $(\{a,b\},c)$ with $c < a \le b$ increments by one those $w_{i,k,j}$ that satisfy $1 \le i \le c$, $c \le k \le a-1$, and $b \le j \le n$. Every input constraint thus defines a rectilinear region called a ***constraint box*** in the finite three-dimensional space $\{1, 2, ..., n\} \times \{1, 2, ..., n\} \times \{1, 2, ..., n\}$ whose axes correspond to indices i, k, j of $w_{i,k,j}$. If a point with coordinates (i,k,j) lies

[2]A naive approach will accomplish this in $O(mn^3) = O(n^6)$ time, but we wish to do better.

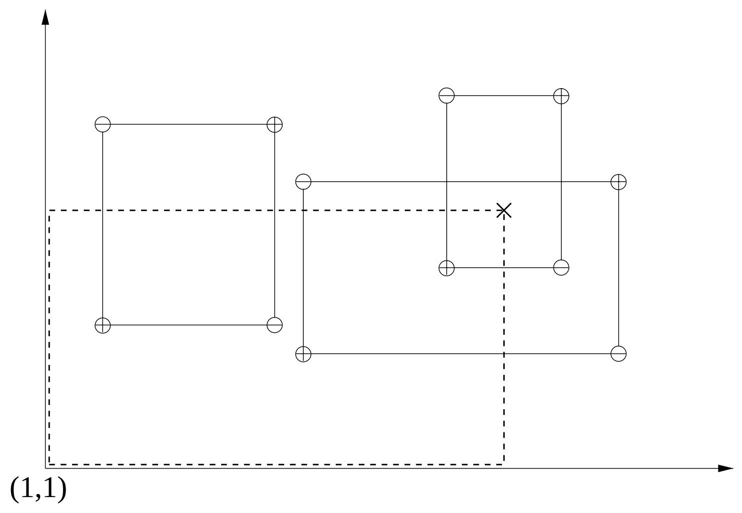

Figure 3.6: The number of rectangles that contain the query point $\times$ equals the number of $\oplus$-corners minus the number of $\ominus$-corners inside the query rectangle with corners at $(1, 1)$ and $\times$.

inside a constraint box, then $w_{i,k,j}$ is affected by the corresponding constraint, so the value of $w_{i,k,j}$ equals the total number of constraint boxes that contain the point (i, k, j).

To efficiently determine how many boxes contain any given point in the space, we reduce the problem to a range searching problem. The strategy is to insert "markers" for the corners of the constraint boxes, and then deduce how many constraint boxes that contain a query point (i, k, j) by looking at the number of markers that lie within a query box with corners at $(1, 1, 1)$ and (i, k, j). For this purpose, we distinguish between two types of constraint box corners, denoted by $\oplus$ and $\ominus$. Figure 3.6 illustrates the idea for two dimensions. Let $A_{\oplus}$, $A_{\ominus}$, $B_{\oplus}$, and $B_{\ominus}$ be four $((n + 1) \times (n + 1) \times (n + 1))$-sized arrays of counters, all initially set to zero. The A-counters keep track of the number of constraint box corners in each point, and the B-counters are used to count how many constraint box corners of each type that are located in the rectilinear region stretching from $(1, 1, 1)$ to (i, k, j) for every $i, k, j \in \{1, 2, ..., n + 1\}$. This way, $w_{i,k,j}$ for any i, k, j with $1 \leq i \leq k < j \leq n$ will equal $B_{\oplus}[i,\, k,\, j] \,-\, B_{\ominus}[i,\, k,\, j]$.

The new range searching problem can be solved as follows. First, scan the set of input constraints to assign correct values to the A-counters. The eight corners of the constraint box induced by the constraint $(\{a, b\}, c)$ are located at $(1,\, b,\, c)$, $(1,\, b,\, n)$, $(1,\, c-1,\, c)$, $(1,\, c-1,\, n)$, $(a,\, b,\, c)$, $(a,\, b,\, n)$, $(a,\, c-1,\, c)$, and $(a,\, c-1,\, n)$ if $a \leq b < c$, or at $(1,\, c,\, b)$, $(1,\, c,\, n)$, $(1,\, a-1,\, b)$, $(1,\, a-1,\, n)$, $(c,\, c,\, b)$, $(c,\, c,\, n)$, $(c,\, a-1,\, b)$, and $(c,\, a-1,\, n)$ if $c < a \leq b$. However, when handling a query point located on the boundary of some constraint box, that constraint should be counted as well, so constraint box corners which mark the end of a constraint must be inserted one step farther away from the origin. More precisely, for every input constraint $(\{a, b\}, c)$, increase each of the following A-counters by one:

$$\left.\begin{array}{l} A_{\oplus}[1,\, b,\, c], \\ A_{\ominus}[1,\, b,\, n+1], \\ A_{\ominus}[1,\, c,\, c], \\ A_{\oplus}[1,\, c,\, n+1], \\ A_{\ominus}[a+1,\, b,\, c], \\ A_{\oplus}[a+1,\, b,\, n+1], \\ A_{\oplus}[a+1,\, c,\, c], \\ A_{\ominus}[a+1,\, c,\, n+1] \end{array}\right\} \text{if } a \le b < c \qquad \left.\begin{array}{l} A_{\oplus}[1,\, c,\, b], \\ A_{\ominus}[1,\, c,\, n+1], \\ A_{\ominus}[1,\, a,\, b], \\ A_{\oplus}[1,\, a,\, n+1], \\ A_{\ominus}[c+1,\, c,\, b], \\ A_{\oplus}[c+1,\, c,\, n+1], \\ A_{\oplus}[c+1,\, a,\, b], \\ A_{\ominus}[c+1,\, a,\, n+1] \end{array}\right\} \text{if } c < a \le b$$

Next, update the B-counters in order. Here, any strategy which ensures that $B[i,\, k,\, j-1]$, $B[i,\, k-1,\, j]$, $B[i,\, k-1,\, j-1]$, $B[i-1,\, k,\, j]$, $B[i-1,\, k,\, j-1]$, $B[i-1,\, k-1,\, j]$, and $B[i-1,\, k-1,\, j-1]$ are taken care of before $B[i,\, k,\, j]$ for every (i, k, j) can be employed. By the principle of inclusion-exclusion, we have:

$$\begin{aligned} B_{\oplus}[i,\, k,\, j] \;:=\; & A_{\oplus}[i,\, k,\, j] \\ & + B_{\oplus}[i,\, k,\, j-1] + B_{\oplus}[i,\, k-1,\, j] + B_{\oplus}[i-1,\, k,\, j] \\ & - B_{\oplus}[i,\, k-1,\, j-1] - B_{\oplus}[i-1,\, k,\, j-1] - B_{\oplus}[i-1,\, k-1,\, j] \\ & + B_{\oplus}[i-1,\, k-1,\, j-1], \end{aligned}$$

where $B_{\oplus}[i,\, k,\, j]$ is defined to equal 0 if at least one of the three indices is 0. Proceed analogously for $B_{\ominus}[i,\, k,\, j]$.

Finally, $w_{i,k,j}$ for every i, k, j with $1 \le i \le k < j \le n$ is simply equal to $B_{\oplus}[i,\, k,\, j] - B_{\ominus}[i,\, k,\, j]$.

Initializing the A- and B-counters takes $O(n^3)$ time, scanning the input constraints and inserting all corresponding constraint box corners in the A-counters takes $O(m) = O(n^3)$ time, and updating all B-counters takes a total of $O(n^3)$ time because the operations for each B-counter take $O(1)$ time to execute and there are $O(n^3)$ B-counters. Thus, the total time complexity to calculate all $w_{i,k,j}$ is $O(n^3)$. (The space complexity is also $O(n^3)$ since each index i, k, j requires $O(1)$ words for its counters. To improve the space usage by a constant factor, only compute $B_{\oplus}[i,\, k,\, j]$, $B_{\ominus}[i,\, k,\, j]$, and $w_{i,k,j}$ for indices satisfying $i \le k < j$. Also, note that $A_{\oplus}[i,\, k,\, j]$ can be used to hold the value of $B_{\oplus}[i,\, k,\, j]$ after it has been calculated because the original value of $A_{\oplus}[i,\, k,\, j]$ is not needed again from then on; therefore, rather than using four arrays of counters, we can manage with only two: one for $\oplus$ and one for $\ominus$.) We have:

Lemma 3.12 *All values of $w_{i,k,j}$, where $1 \le i \le k < j \le n$, can be precomputed in $O(n^3)$ time.*

3.6 Concluding Remarks

The following table summarizes and compares what is known about the computational complexities of the problems studied in Chapters 2 and 3:

Problem	*Result*	*Reference*
LC	$O(mn \log n)$	Aho *et al.* [3]
3LC	$\min\{O(n + m \log^2 n), O(m + n^2 \log^2 n)\}$	Henzinger *et al.* [62], Holm *et al.* [65], and Theorem 2.1[3]
O3LC	$\min\{O((m + n) \log n), O(n^3)\}$	Theorem 3.8 and Theorem 3.11
MLC	NP-hard	Corollary 2.3
M3LC	NP-hard	Theorem 2.2; alternative proof by Bryant [21]
MO3LC	$O(n^3)$	Theorem 3.11

When $m = \omega(\frac{n^3}{\log n})$, the $O(n^3)$-time algorithm for MO3LC given in Section 3.5 is asymptotically faster than Algorithm *Fast O3LC* from Section 3.4. We can run both algorithms in parallel until one of them is finished (and if the MO3LC algorithm finishes first, check if the produced solution satisfied all of the constraints) in order to solve O3LC in $\min\{O((m + n) \log n), O(n^3)\}$ time.

Our algorithm for O3LC is asymptotically faster than the currently best algorithm for 3LC if $m = o(n^2 \log n)$. It is an open question whether O3LC can be solved more efficiently than 3LC when $m = \Omega(n^2 \log n)$.

It is noteworthy that MO3LC can be solved in polynomial time while M3LC is NP-hard. In Chapter 5, we consider another problem which is NP-hard for unordered trees yet polynomial-time solvable for ordered trees.

Our cubic-time algorithm for MO3LC can be generalized to include other forms of lowest common ancestor constraints, e.g., constraints of the form "the lowest common ancestor of i and j has to be a proper descendant of the lowest common ancestor of k and l" considered by Aho *et al.* in [3]. In this case, equation (3.1) still holds, but the definition of $w_{i,k,j}$ needs to be modified accordingly, resulting in a slower (but still polynomial time) algorithm.

It would also be interesting to consider the more general situation where only a partial ordering of the leaf labels is given a priori. One could look here for efficient algorithms for constructing ordered trees that would take advantage of the input partial ordering as much as possible.

[3]By Theorem 2.1, a faster algorithm for decremental dynamic graph connectivity than the one in [65] automatically improves the result for 3LC.

Part II

Identifying Shared Substructures in Labeled Trees

Chapter 4

On the Approximability of the Maximum Agreement Subtree Problem

Given a set of rooted, unordered, leaf-labeled trees, *the maximum agreement subtree problem* (MAST) asks for a tree contained in all of the input trees with as many labeled leaves as possible. The computational complexity of MAST restricted to instances where the number of input trees is bounded and/or the maximum degree of the input trees is bounded has been studied previously in the literature; here, we investigate how the polynomial-time approximability of MAST depends on another important parameter, namely the *maximum height* of the input trees.

4.1 Introduction

A tree whose leaves are labeled by elements belonging to a finite set S in such a way that no two leaves have the same label is said to be *distinctly leaf-labeled by S*. Below, each leaf in such a tree is identified with its corresponding element in S. The lowest common ancestor of any two leaves a and b in a rooted tree T is denoted by $\mathrm{lca}_T(a,b)$, and we define $\mathrm{lca}_T(a,a) = a$. The *degree of a node u in a rooted tree* is the number of children of u, and the *degree of a node u in an unrooted tree* is the number of edges incident to u. The *degree of a tree T* is the maximum degree of all nodes in T.

Let S be a finite set and let T be a rooted, unordered tree distinctly leaf-labeled by S. For any subset S' of S, $T|S'$ is the rooted, unordered tree with node set $\{\mathrm{lca}_T(a,b) \,|\, (a,b) \in S' \times S'\}$ and edges defined so that $\mathrm{lca}_{T|S'}(a,b) = \mathrm{lca}_T(a,b)$ for every $(a,b) \in S' \times S'$. Algorithmically, $T|S'$ can be obtained by first deleting from T all leaves which are not in S' and all internal nodes without any descendants in S' along with their incident edges, and then contracting every edge between a node having just one child and its child (see Figure 4.1). $T|S'$ is uniquely determined by S' [40].

Given a set $\mathcal{T} = \{T_1, T_2, ..., T_k\}$ of rooted, unordered trees, each distinctly leaf-labeled by S, an *agreement subtree of* $\mathcal{T}$ is a rooted, unordered tree U such that for some $S' \subseteq S$ it holds that U is distinctly leaf-labeled by S' and $U = T_1|S' = T_2|S' = ... = T_k|S'$. A *maximum agreement subtree of* $\mathcal{T}$ is an agreement subtree of $\mathcal{T}$ with the maximum possible number of leaves. See Figure 4.2 for an example[1].

The maximum agreement subtree problem (MAST), also referred to as *the maximum homeomorphic agreement subtree problem* or *the maximum homeomorphic subtree problem* (MHT) by some researchers, is defined as follows:

The maximum agreement subtree problem (MAST)

Instance: Finite set S, set $\mathcal{T} = \{T_1, T_2, ..., T_k\}$ of rooted, unordered trees, where each $T_i \in \mathcal{T}$ is distinctly leaf-labeled by S and no $T_i \in \mathcal{T}$ has a node of degree 1.

Output: A maximum agreement subtree of $\mathcal{T}$.

In this chapter, n and k represent the cardinalities of S and $\mathcal{T}$. The minimum and the maximum of the heights of the trees in $\mathcal{T}$ are denoted by h and H, i.e., $h = \min\{height(T_i) \mid 1 \leq i \leq k\}$ and $H = \max\{height(T_i) \mid 1 \leq i \leq k\}$.

The problem definition requires that no tree in $\mathcal{T}$ has a node with a single child. Thus, the number of nodes in each input tree is always $O(n)$. Also, $H = O(n)$. (Note that given an invalid instance $I = (S, \mathcal{T})$ with one or more degree 1 nodes, we can replace T_i by $T_i|S$ for all input trees in total time which is linear in the size of I to make it valid since U is an agreement subtree of $\mathcal{T}$ if and only if U is an agreement subtree of $\{T_1|S, T_2|S, ..., T_k|S\}$.)

An algorithm $\mathcal{A}$ is said to approximate MAST within a factor of f if for any instance $(S, \mathcal{T})$ of the problem, $\mathcal{A}$ outputs an agreement subtree with at least $|S^*|/f$ leaves, where $|S^*|$ is the number of leaves in a maximum agreement subtree for $(S, \mathcal{T})$. In this case, $\mathcal{A}$ is also called a factor f approximation algorithm (or just an f-approximation algorithm) for MAST.

4.1.1 Motivation

An agreement subtree represents branching structure shared by two or more leaf-labeled trees in a given set. Hence, one of the main motivations for studying

[1]The data used in this example is fictitious. Any resemblance to real-life data is purely coincidental.

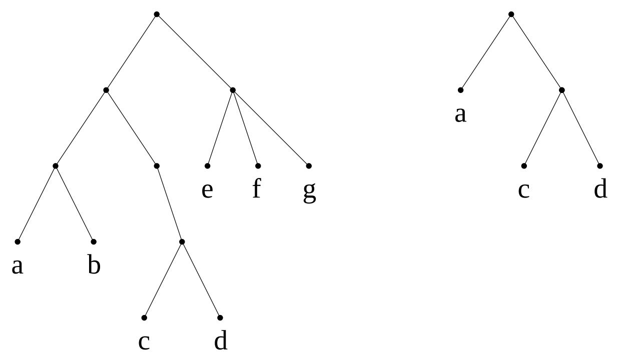

Figure 4.1: Let T be the tree on the left. Then $T|\{a, c, d\}$ is the tree shown on the right.

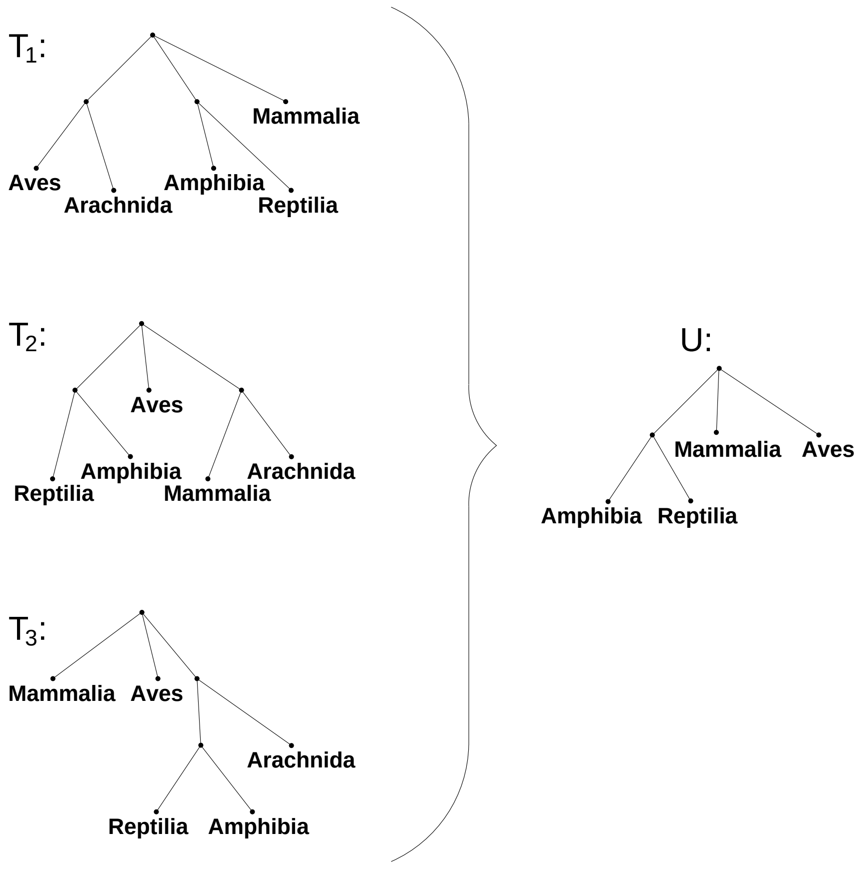

Figure 4.2: U is a maximum agreement subtree of the set of trees $\{T_1, T_2, T_3\}$.

algorithms for constructing maximum agreement subtrees is the following: Suppose a number of trees, each one describing the possible evolution of a fixed set of biological species, have been obtained by applying different tree construction methods or different clustering criteria to some available data. (Another possibility is that one such method has been used on several alternative sets of data originating from different sources or repeated experiments conducted on one source.) Furthermore, suppose that these trees do not completely agree because of distortions due to assumptions inherent to the methods used or because of measurement errors. It would then be informative to find a subtree contained in every one of the trees with as many leaves labeled by species as possible since such a subtree more likely represents genuine evolutionary structure in the data [45]. In this way, one would get an indication of which ancestral relationships can be regarded as resolved and which species need to be subjected to further experiments.

Moreover, maximum agreement subtrees can be used to measure the similarity of the input trees [40, 45, 78] or to estimate a classification's stability to small changes in the data [45]. MAST-based algorithms have also been used to prepare and improve bilingual context-using dictionaries for automated language translation systems [28, 95].

Unfortunately, MAST is NP-hard [6] and thus cannot be solved exactly in polynomial time in the full generality in which it was defined (unless P=NP). We are therefore interested in determining whether certain special cases of MAST admit polynomial-time algorithms. In particular, we would like to find out how restricting various parameters for MAST affects the computational complexity. Then, for the application mentioned above, if the method used to obtain the trees provides sufficiently strong upper bounds on these parameters, we can be certain that we can compute a maximum agreement subtree efficiently even if the number of leaves is large.

Previous research on MAST has mostly focused on restricting the number of input trees and/or their maximum degrees (see Section 4.1.2 for a survey of known results). However, sometimes there are thousands of trees to compare [54]. Also, many of the tree-construction methods used today do not guarantee any upper bounds on the degrees of the produced trees [42]. Therefore, it may be beneficial to study the dependency of MAST's computational complexity on other parameters besides these two. Our main goal in this chapter is to establish how the computational complexity of MAST is related to the *maximum height* of the input trees.

4.1.2 Previous Results

Various aspects of MAST and related problems have been studied in the literature. We summarize some of the algorithmic and combinatorial highlights below.

Most of the published results on MAST involve algorithms for the special case $k = 2$. This line of research was initiated by Finden and Gordon [45] who presented a polynomial-time heuristic (not guaranteed to find an optimal solution) for MAST restricted to instances consisting of two binary trees. A few years later, Kubicka, Kubicki, and McMorris [86] gave an exact algorithm with superpolynomial running time in the worst case for the *unrooted* maximum agreement subtree problem (UMAST)[2] for two binary, unrooted trees. Steel and Warnow [117] presented the first exact polynomial-time algorithms to solve MAST and UMAST for two trees with unbounded degrees. Since then, a plethora of improvements have been published (e.g., [28, 41, 42, 54, 73, 74, 75, 76, 77, 78, 87, 108, 119]). The fastest currently known algorithm for MAST with $k = 2$, invented by Kao, Lam, Sung, and Ting [78], runs in $O(\sqrt{D}\, n \log(2n/D))$ time, where D is the maximum degree of the two input trees[3]. Note that this is $O(n \log n)$ for trees with maximum degree bounded by a constant and $O(n^{1.5})$ for trees with unbounded degrees; incidentally, this matches the running times of the fastest algorithms for UMAST so far: $O(n \log n)$ for two unrooted trees with maximum degree bounded by a constant [74], and $O(n^{1.5})$ for two unrooted trees with unbounded degrees [119]. Finally, for two rooted, *ordered* trees, a maximum agreement subtree can be computed in $O(n \log^2 n)$ time [119].

Amir and Keselman [6] considered the more general case $k \geq 2$. They proved that MAST is NP-hard already for three trees with unbounded degrees, but solvable in polynomial time for three or more trees if the degree of at least one of the input trees is bounded by a constant. For the latter case, Farach, Przytycka, and Thorup [40] gave an algorithm with improved efficiency running in $O(kn^3 + n^d)$ time, where d is an upper bound on at least one of the input trees' degrees; Bryant [21] proposed a conceptually different algorithm with the same running time. Bryant's approach led to a recent result in the field of parameterized complexity theory stating that it is possible to determine whether an instance of MAST has an agreement subtree with at least $n - \mu$ leaves for any integer $0 \leq \mu \leq n$ in $O(kn^3 + 2.270^{\mu})$ time[4] (see [5]).

In [6], Amir and Keselman also presented a factor 4 approximation algorithm with $O(kn^5)$ running time for the problem of finding a subset of S of minimum cardinality whose removal leaves a set $\tilde{S}$ such that $T_1|\tilde{S} = T_2|\tilde{S} = \ldots = T_k|\tilde{S}$, where all trees are unrooted.

Hein, Jiang, Wang, and Zhang [60] proved the following inapproximability result: MAST with three trees with unbounded degrees cannot be approximated within a factor of $2^{\log^{\delta} n}$ in polynomial time for any constant $\delta < 1$, unless NP $\subseteq$ DTIME$[2^{\mathrm{polylog}\, n}]$. This inapproximability result also holds for

[2]UMAST is defined like MAST except that all trees are unrooted and $T|S'$ now denotes the tree obtained by first deleting from T all nodes (and their incident edges) which are not on any path between two leaves in S', and then contracting every node with degree 2.

[3]In fact, the result still holds for D equal to the smaller of the two input trees' degrees [120].

[4]Note that $O(kn^3 + 2.270^{\mu})$ running time might be preferable to $O(kn^3 + n^d)$ if d is unrestricted and the number of leaves we are willing to exclude is small.

UMAST [60]. Bonizzoni, Della Vedova, and Mauri [19] showed that it can be carried over to *the maximum isomorphic agreement subtree problem* (MIT)[5] restricted to three trees with unbounded degrees as well, and that even stronger bounds can be proved for MIT in the general case. Akutsu and Halldórsson [4] and Khanna, Motwani, and Yao [82] proved lower and upper bounds on the approximability of another related problem known as *the largest common subtree problem* (LCST)[6].

Finally, we mention some known results of a more enumerative nature. Kubicka, Kubicki, and McMorris [85] demonstrated that even if UMAST is restricted to two binary, unrooted trees, there exist instances which have $(6^{1/4})^n$, i.e., an exponential number of different maximum agreement subtrees (adjusting the idea used in their construction to two binary, *rooted* trees yields an analogous exponential worst-case lower bound for MAST). In the same paper, the authors gave lower bounds on the number of leaves in any maximum agreement subtree of two binary, unrooted trees; these bounds were subsequently strengthened (and extended to cover the case of binary, rooted trees) by Goddard and Kubicki [55].

4.1.3 Our Contributions

In Section 4.2, we show how Akutsu and Halldórsson's general-purpose approximation algorithm for the largest common subtree problem given in [4] can be modified to obtain a simple factor $(n/\log n)$ approximation algorithm for MAST whose running time is $O(kn^2)$.

Then, in Section 4.3, we prove that: (1) if P$\neq$NP then MAST cannot be approximated within a factor of n^ϵ for any constant ϵ where $0 \leq \epsilon < \frac{1}{2}$ in polynomial time, even for instances containing only trees of height 2; and (2) if ZPP$\neq$NP then MAST cannot be approximated within a factor of n^ϵ for any constant ϵ where $0 \leq \epsilon < 1$ in polynomial time, even for instances containing only trees of height 2. This implies that if we only restrict the maximum height of the input trees then MAST remains hard to approximate.

On the other hand, we show in Section 4.4 that if both the number of input trees and the height of at least one tree are bounded by constants then MAST can approximated efficiently. More precisely, we prove that MAST restricted to instances with $k = O(1)$ and $h = O(1)$ can be approximated within a constant factor in polynomial time. Furthermore, if all of the input trees' heights are required to be bounded by a constant ($H = O(1)$), then MAST can be approximated within a constant factor in $O(n \log n)$ time.

[5]MIT is defined like MAST except that when computing $T|S'$, nodes having just one child are left that way, i.e., no edges are contracted.

[6]In LCST, the input is a set of rooted, unordered trees in which all nodes are labeled and the same label may be assigned to more than one node, and the object is to find a node-labeled tree with the maximum possible number of nodes that is isomorphic to an induced connected subgraph in each of the input trees.

4.2 A Polynomial-Time $(n/\log n)$-Approximation Algorithm for MAST

Akutsu and Halldórsson's general-purpose approximation algorithm for the largest common subtree problem from [4] can be adapted to obtain a polynomial-time $(n/\log n)$-approximation algorithm for MAST. Observe that this does not contradict the inapproximability result of Hein *et al.* [60] (see Section 4.1.2) since $n/\log n = 2^{\log n - \log\log n} = \omega(2^{\log^{\delta} n})$ for any fixed $\delta < 1$. The resulting algorithm is called Algorithm *Simple MAST-Approx* and is presented in Figure 4.3.

```
Algorithm   Simple MAST-Approx
Input:  An instance of MAST.
Output: An agreement subtree of $\mathcal{T}$ whose number of leaves is at least $\frac{\log n}{n}$
        times the number of leaves in a maximum agreement subtree of $\mathcal{T}$.

1  Arbitrarily partition $S$ into $\lfloor n/\log n \rfloor$ sets $S_1, S_2, ..., S_{\lfloor n/\log n \rfloor}$, each of size at
   most $\lceil \log n \rceil + 1$.
2  Let $Z := \emptyset$.
3  for each subset $S_i'$ of each set $S_i$ do
       If $|S_i'| > |Z|$ and $T_1|S_i' = T_2|S_i' = ... = T_k|S_i'$ then let $Z := S_i'$.
   endfor
4  return $T_1|Z$.
End  Simple MAST-Approx
```

Figure 4.3: Akutsu and Halldórsson's approximation algorithm applied to MAST.

Theorem 4.1 *Algorithm* Simple MAST-Approx *is an $\frac{n}{\log n}$-approximation algorithm for MAST and can be implemented to run in $O(kn^2)$ time.*

Proof: Let S^* be the leaves in a maximum agreement subtree of $\mathcal{T}$. Because of the pigeonhole principle, at least one of the sets $S_1, S_2, ..., S_{\lfloor n/\log n \rfloor}$ contains $\geq \frac{1}{\lfloor n/\log n \rfloor}$ of the elements in S^*; thus, $|Z| \geq \frac{|S^*|}{\lfloor n/\log n \rfloor} \geq \frac{|S^*|}{n/\log n}$.

To implement step 3 of the algorithm, first construct $T_1|S_i$, $T_2|S_i$, ..., $T_k|S_i$ for all the sets S_i. Each tree $T_j|S_i$ can be obtained in $O(n)$ time and contains at most $\log n + 2$ leaves and hence $O(\log n)$ nodes in total since every internal node of $T_j|S_i$ has at least two children. Next, every set S_i has at most $2^{\log n + 2} = O(n)$ subsets to be considered by the algorithm; each such subset S_i' can be evaluated in $O(k \log n)$ time by checking if $(T_1|S_i)|S_i' = (T_2|S_i)|S_i' = ... = (T_k|S_i)|S_i'$. Thus, Algorithm *Simple MAST-Approx* can be implemented to run in $O(\frac{n}{\log n} \cdot k \cdot n + \frac{n}{\log n} \cdot n \cdot k \log n) = O(kn^2)$ time. $\square$

As in [4], if at least one of the input trees is known to contain much fewer than n leaves, the running time of Algorithm *Simple MAST-Approx* can be reduced by only considering the leaves in that tree rather than all of the leaves in S (replace Step 1 by "Arbitrarily partition the m leaves of the smallest tree in $\mathcal{T}$ into $\lfloor m/\log m \rfloor$ sets $S_1, S_2, ..., S_{\lfloor m/\log m \rfloor}$, each of size $\leq \lceil \log m \rceil + 1$.") since any leaf which belongs to S^* must also belong to all of the input trees and, in particular, to the smallest input tree.

4.3 MAST Restricted to Trees of Height 2 is Hard to Approximate

Our main result in this section is the next theorem.

Theorem 4.2 *For any constant ϵ where $0 \leq \epsilon < \frac{1}{2}$, MAST, even if restricted to trees of height 2, cannot be approximated within a factor of n^ϵ in polynomial time, unless P=NP. Furthermore, for any constant ϵ where $0 \leq \epsilon < 1$, MAST, even if restricted to trees of height 2, cannot be approximated within a factor of n^ϵ in polynomial time, unless ZPP=NP.*

Proof: We first describe a reduction from the maximum independent set problem to MAST. Then, we show that if MAST could 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 vertices could be approximated within a factor of $l^{\epsilon+o(1)}$. Finally, we apply a known result about the inapproximability of the maximum independent set problem to get our result.

Given a graph $G = (V, E)$ where $V = \{v_1, ..., v_l\}$ and $E = \{e_1, ..., e_k\}$ with $k > 1$, construct k rooted trees $T_1, ..., T_k$ on $l + q$ labeled leaves (q is an integer that will be specified below) containing all the adjacency information about the vertices 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 $w_1, ..., w_l, w_{l+1}, ..., w_{l+q}$. 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 $GC(T_i) = \{w_a, w_b\}$.

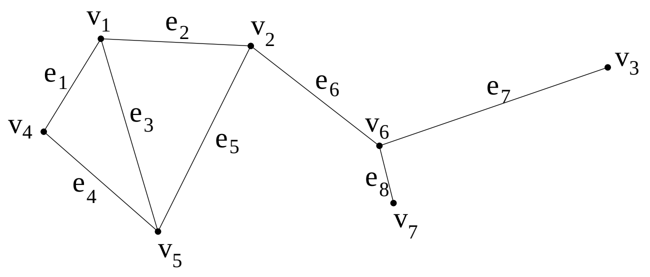

Figure 4.4: An instance of the maximum independent set problem with $l = 7$, $k = 8$.

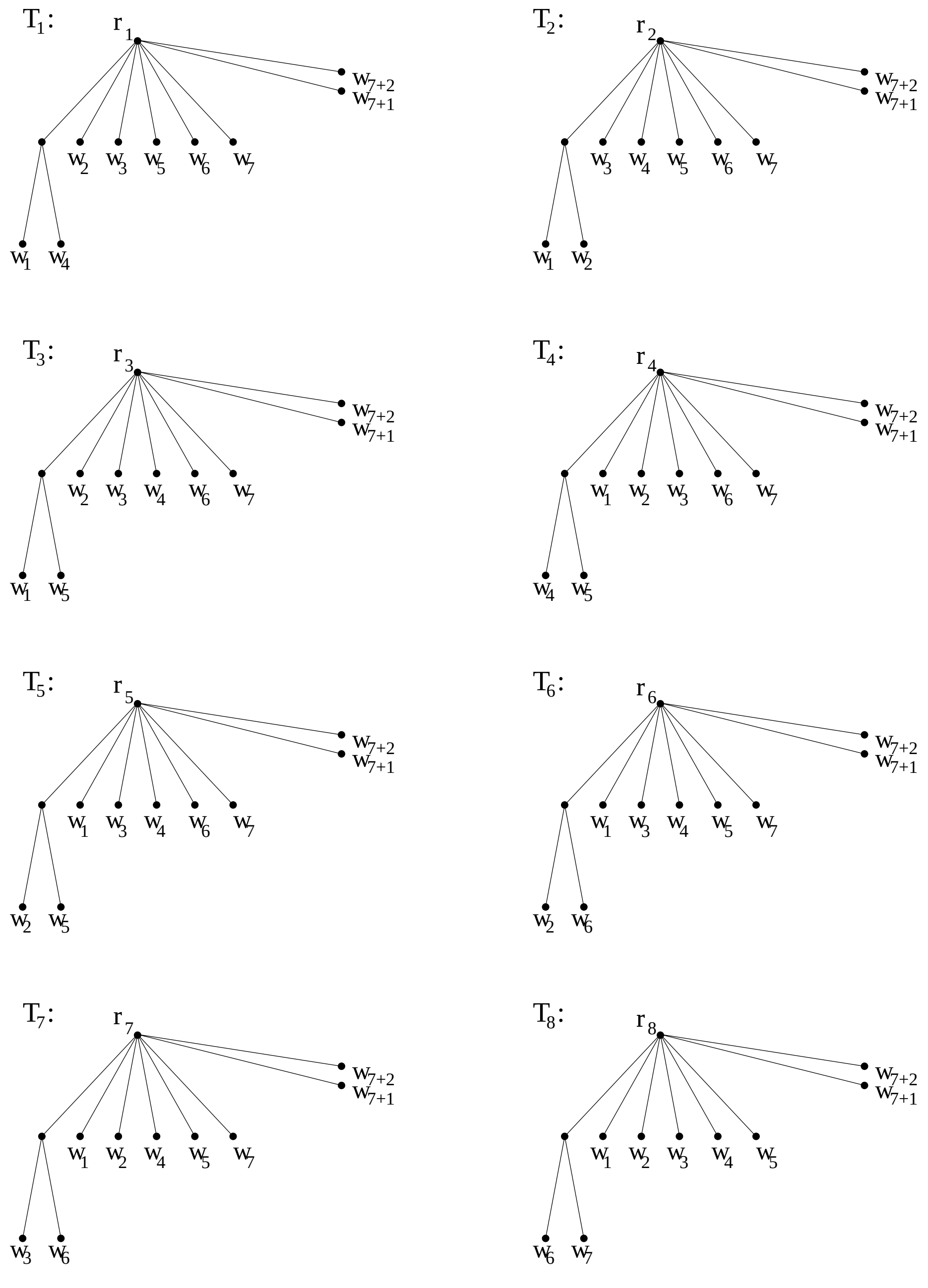
T1: r1 w7+2 w7+1 w2 w3 w5 w6 w7 w1 w4
T2: r2 w7+2 w7+1 w3 w4 w5 w6 w7 w1 w2
T3: r3 w7+2 w7+1 w2 w3 w4 w6 w7 w1 w5
T4: r4 w7+2 w7+1 w1 w2 w3 w6 w7 w4 w5
T5: r5 w7+2 w7+1 w1 w3 w4 w6 w7 w2 w5
T6: r6 w7+2 w7+1 w1 w3 w4 w5 w7 w2 w6
T7: r7 w7+2 w7+1 w1 w2 w4 w5 w7 w3 w6
T8: r8 w7+2 w7+1 w1 w2 w3 w4 w5 w6 w7

Figure 4.5: The trees T_i corresponding to the graph in Figure 4.4.

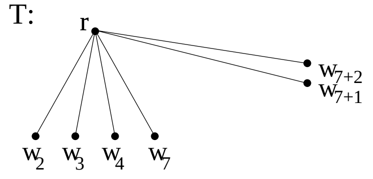

Figure 4.6: The maximum agreement subtree of $T_1, ..., T_8$ shown above tells us that $\{v_2, v_3, v_4, v_7\}$ is a maximum independent set of the graph in Figure 4.4.

Now, let T be a maximum agreement subtree of $T_1, ..., T_k$. Denote the set of leaves in T by S'. We choose q large enough to guarantee that each of the roots $r_1, ..., r_k$ will correspond to the root of T, i.e., so that r_i is still the root of $T_i|S'$ for every $i \in \{1, ..., k\}$. Actually, $q = 2$ is sufficient. (To see this, assume that for some i, the non-leaf child of r_i turns out to be the root of $T_i|S'$. 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 any fixed leaf w_j, where $1 \leq j \leq l$. Contradiction. In the same way, if any of the other descendants of r_i becomes the root of $T_i|S'$ then the resulting agreement subtree can not be maximal.)

The root r of 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 $k > 1$. Thus, the children of r are $m + q$ $(= m + 2)$ leaves labeled $w_{\mu_1}, w_{\mu_2}, ..., w_{\mu_m}, w_{l+1}, w_{l+2}$. 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 r. Otherwise, $GC(T_i)$ would not 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. Hence, the vertices $v_{\mu_1}, v_{\mu_2}, ..., v_{\mu_m}$ form an independent set in G. Conversely, given an independent set I in G, we can construct an agreement subtree with $|I| + 2$ leaves consisting of a root node with $|I|$ children distinctly labeled by $\{w_j : v_j \in I\}$, and two children labeled by w_{l+1} and w_{l+2}. By the maximality of T, m equals the cardinality of a maximum independent set in G. Thus, an exact algorithm for MAST would immediately yield an exact algorithm for the maximum independent set problem. See Figures 4.4–4.6 for an example of the reduction.

The trees $T_1, ..., T_k$ can clearly 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. Assume that MAST could be approximated within a factor of n^ϵ for some $\epsilon \geq 0$. Then $\frac{OPT}{APPR} \leq n^\epsilon$, where OPT refers to the number of leaves in an optimal solution for a given instance of MAST and $APPR$ is the number of leaves in its approximate solution. In particular, for the instance of MAST obtained in the reduction above, we would have $\frac{m+2}{APPR} \leq (l+2)^\epsilon$, and the size of the corresponding approximate independent set would be $APPR - 2$. For $APPR - 2 \geq 1$, this would imply that the problem of finding a maximum independent set in a graph could be approximated within a factor of $\frac{m}{APPR-2} \leq 3 \cdot \frac{m+2}{APPR} \leq 3(l+2)^{\epsilon} = l^{\epsilon+o(1)}$. However, Håstad proved in [66] that this problem is not approximable within:

(1) $l^{1/2-\delta}$ for any constant $\delta > 0$, unless P=NP; and

(2) $l^{1-\delta}$ for any constant $\delta > 0$, unless ZPP=NP

If $\epsilon = \frac{1}{2} - \tau$ for some constant $\tau > 0$ then $l^{\epsilon+o(1)} = l^{1/2-\tau+o(1)} = l^{1/2-\delta}$, where $\delta = \tau - o(1)$. For large enough l, δ is strictly greater than 0 since $\tau - o(1) \to \tau$ as $l \to \infty$. Thus, from (1) it follows that if P $\neq$ NP then no polynomial-time algorithm can approximate arbitrary instances of MAST restricted to trees of height 2 within a factor of n^{ϵ} for any constant ϵ with $0 \leq \epsilon < \frac{1}{2}$.

Similarly, by (2), MAST restricted to trees of height 2 cannot be approximated in polynomial time within a factor of n^{ϵ} for any constant ϵ with $0 \leq \epsilon < 1$, unless ZPP=NP. □

Engebretsen and Holmerin [36] obtained an even tighter inapproximability result for the maximum independent set problem than the one cited above, under the slightly stronger assumption that NP $\not\subseteq$ ZPTIME$[2^{O(\log n (\log\log n)^{3/2})}]$ (note that if this assumption is true, then NP $\neq$ ZPP automatically follows). More precisely, they showed that the maximum independent set problem cannot be approximated in polynomial time within a factor of $n^{1-O(1/\sqrt{\log\log n})}$, where n is the number of vertices in the input graph, unless NP $\subseteq$ ZPTIME$[2^{O(\log n (\log\log n)^{3/2})}]$. This result can be used to extend Theorem 4.2 accordingly.

4.4 Approximations of MAST with $O(1)$ Trees of $O(1)$ Height

We know that MAST remains hard to approximate even if we restrict the number of input trees to any constant greater than or equal to three [60], or if we restrict the heights of the trees to be bounded by a constant greater than or equal to two (Theorem 4.2). The natural question arises whether or not MAST for instances with a constant number of trees, each one of height bounded by a constant, can be tightly approximated in polynomial time. In this section, we prove the following theorem, which together with Theorem 4.2 yields a characterization of the approximability of MAST restricted to instances with trees of $O(1)$ height (in fact, the theorem only requires that *at least one* of the trees' heights is bounded by a constant). Recall that k denotes the number of input trees and that $h = \min\{height(T_i) \mid 1 \leq i \leq k\}$ and $H = \max\{height(T_i) \mid 1 \leq i \leq k\}$.

Theorem 4.3 *MAST restricted to instances with $k = O(1)$ and $h = O(1)$ can be approximated within a constant factor in polynomial time.*

Before proving Theorem 4.3, we introduce some notation. For any tree T, $V(T)$ stands for the set of nodes of T and $\Lambda(T)$ for the set of labels of the leaves in T. Let v be a node of a rooted tree T. The subtree of T rooted at v (i.e., the minimal subgraph of T which includes v and all of its descendants) is denoted by $T[v]$. The set of children of v in T is denoted by $C(v)$. Furthermore, by a *k-partite hypergraph* $\mathcal{H}$ we mean a pair $(V_1 \cup ... \cup V_k,\ E)$, where V_1 through V_k are pairwise disjoint sets and E is a subset of $V_1 \times ... \times V_k$. The elements of $V_1 \cup ... \cup V_k$ are called *vertices of* $\mathcal{H}$ whereas the elements of E are called *edges of* $\mathcal{H}$. A *matching of* $\mathcal{H}$ is a subset of E in which no two edges include a common vertex.

Given an instance of MAST, for every $(v_1, ..., v_k) \in V(T_1) \times ... \times V(T_k)$, define $Mast(v_1, ..., v_k)$ as the number of leaves in a maximum agreement subtree of the trees $T_1[v_1]$, ..., $T_k[v_k]$. Next, define $Diag(v_1, ..., v_k) = \max\{Mast(w_1, ..., w_k) \mid (w_1, ..., w_k) \in (\{v_1\} \cup C(v_1)) \times ... \times (\{v_k\} \cup C(v_k)) - \{(v_1, ..., v_k)\}\}$. Finally, let $\mathcal{H}(v_1, ..., v_k)$ denote the k-partite hypergraph $(C(v_1) \cup ... \cup C(v_k),\ C(v_1) \times ... \times C(v_k))$ in which each edge $(w_1, ..., w_k)$ has weight equal to $Mast(w_1, ..., w_k)$, and let $Match(v_1, ..., v_k)$ be the maximum weight of a matching in the hypergraph $\mathcal{H}(v_1, ..., v_k)$.

The next lemma is a generalization of the main lemma behind the dynamic programming approach to MAST for the case $k = 2$ presented in [41] and [117].

Lemma 4.4 *For every* $(v_1, ..., v_k) \in V(T_1) \times ... \times V(T_k)$,

$$Mast(v_1, ..., v_k) = \begin{cases} |\Lambda(T_1[v_1]) \cap ... \cap \Lambda(T_k[v_k])|, & \text{if at least one of} \\ & v_1, ..., v_k \text{ is a leaf} \\ \max\{Diag(v_1, ..., v_k),\ Match(v_1, ..., v_k)\}, & \text{otherwise} \end{cases}$$

Proof: If at least one of $v_1, ..., v_k$ is a leaf ℓ then $Mast(v_1, ..., v_k)$ equals 0 or 1 depending on whether ℓ occurs in all of the trees $T_1[v_1], ..., T_k[v_k]$, i.e., $Mast(v_1, ..., v_k) = |\Lambda(T_1[v_1]) \cap ... \cap \Lambda(T_k[v_k])|$.

Next, if none of $v_1, ..., v_k$ is a leaf then let T be a maximum agreement subtree of $T_1[v_1], ..., T_k[v_k]$ and write $L = \Lambda(T)$ so that $|L| = Mast(v_1, ..., v_k)$. There are two possibilities:

1. (The *Diag* case.)
   In at least one tree T_i, the lowest common ancestor of L lies below v_i.

2. (The *Match* case.)
   In every tree T_i, v_i is the lowest common ancestor of L.

In the first case, T is also a maximum agreement subtree of any set of trees $T_1[x_1], ..., T_k[x_k]$, where each x_i belongs to the set of nodes on the path from v_i to the lowest common ancestor of L in T_i. Thus, we have $Mast(v_1, ..., v_k) =$

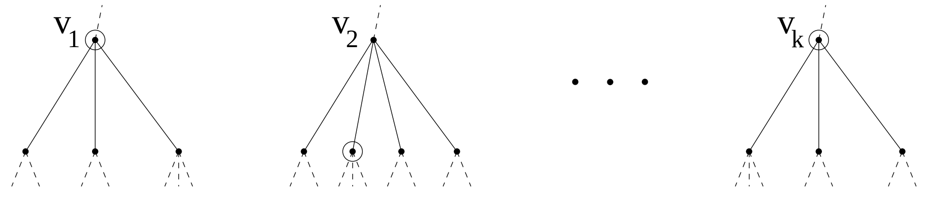

Figure 4.7: (The *Diag* case.) Here, T is also a maximum agreement subtree of $T_1[v_1]$, $T_2[z]$, ..., $T_k[v_k]$, where z is a child of v_2. Hence, $Mast(v_1, v_2, ..., v_k) = Mast(v_1, z, ..., v_k)$.

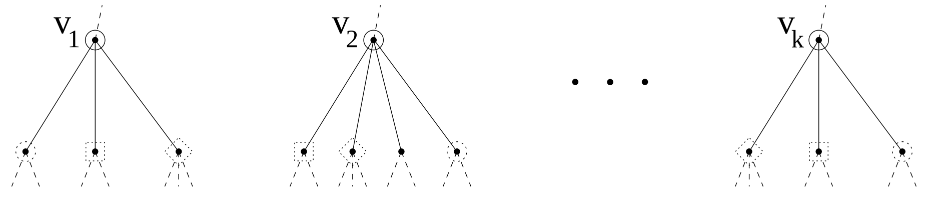

Figure 4.8: (The *Match* case.) T has three children, each of which is the root of a maximum agreement subtree for a set of trees $T_1[w_1], ..., T_k[w_k]$, where w_i is a child of v_i for all $1 \leq i \leq k$. $Mast(v_1, ..., v_k)$ is given by the maximum weight of a matching in $\mathcal{H}(v_1, ..., v_k)$.

$Mast(w_1, ..., w_k)$ for some $(w_1, ..., w_k) \in (\{v_1\} \cup C(v_1)) \times ... \times (\{v_k\} \cup C(v_k)) - \{(v_1, ..., v_k)\}$. This case is illustrated in Figure 4.7.

In the second case, illustrated in Figure 4.8, for every $i \in \{1, ..., k\}$ the elements in L are descendants of at least two of $T_i[v_i]$'s children. Since T is an agreement subtree of $T_1[v_1], ..., T_k[v_k]$, the partition of L into disjoint, nonempty sets each consisting of descendants of one of v_i's children is identical for all trees $T_i[v_i]$. Hence, the elements of L can be partitioned into $L_1, ..., L_q$ with $q \geq 2$ such that for each $j \in \{1, ..., q\}$, $T|L_j$ is a maximum agreement subtree of a set of trees $T_1[w_1], ..., T_k[w_k]$, where $w_i \in C(v_i)$ for all $1 \leq i \leq k$. Denote the k-tuple $(w_1, ..., w_k) \in C(v_1) \times ... \times C(v_k)$ of ancestors of L_j by Γ_j. Observe that $Mast(\Gamma_j) = |L_j|$ for every $j \in \{1, ..., q\}$ and that because $L_1, ..., L_q$ are disjoint, $\Gamma_1, ..., \Gamma_q$ are also disjoint. Thus, $\{\Gamma_1, ..., \Gamma_q\}$ is one of the matchings in the k-partite hypergraph $\mathcal{H}(v_1, ..., v_k)$. By the definition of $\mathcal{H}(v_1, ..., v_k)$, the weight of this matching equals $Mast(\Gamma_1) + ... + Mast(\Gamma_q) = |L_1| + ... + |L_q| = |L|$. Moreover, it is a maximum weight matching since otherwise there would exist an agreement subtree of $T_1[v_1], ..., T_k[v_k]$ with more than $|L|$ leaves, contradicting the maximality of T. Therefore, $Mast(v_1, ..., v_k) = Match(v_1, ..., v_k)$.

Finally, we note that in the *Diag* case, $Match(v_1, ..., v_k) \leq |L|$, and similarly, in the *Match* case, $Diag(v_1, ..., v_k) < |L|$. Thus, the lemma follows. $\square$

Lemma 4.4 implies that we could compute $Mast(v_1, ..., v_k)$ exactly for any k-tuple $(v_1, ..., v_k)$ in $V(T_1) \times ... \times V(T_k)$ if we knew the values of $Mast(w_1, ..., w_k)$ for all $(w_1, ..., w_k) \in (\{v_1\} \cup C(v_1)) \times ... \times (\{v_k\} \cup C(v_k)) - \{(v_1, ..., v_k)\}$. Hence, to compute the number of leaves in a maximum agreement subtree of $T_1, ..., T_k$, we could employ dynamic programming to calculate all values of $Mast$ in a bottom-up manner, e.g., by evaluating the k-tuples in $V(T_1) \times ... \times V(T_k)$ in increasing order in the lexicographic ordering $\mathcal{O}$ of $V(T_1) \times ... \times V(T_k)$ where the nodes in each $V(T_i)$ are postordered (fix an arbitrary left-to-right ordering of the children of each node to obtain a well-defined postordering). The resulting algorithm (Algorithm *Compute Mast*) is listed in Figure 4.9.

```
Algorithm   Compute Mast
Input:   An instance of MAST.
Output:  The number of leaves in a maximum agreement subtree of $\mathcal{T}$.

1  Let $\mathcal{O}$ be the lexicographic ordering of $V(T_1) \times ... \times V(T_k)$, where the nodes
   in each $V(T_i)$ are ordered according to postorder.
2  for each $(v_1, ..., v_k) \in V(T_1) \times ... \times V(T_k)$ in increasing order in $\mathcal{O}$ do
       Compute $Mast(v_1, ..., v_k)$ by using the expression in Lemma 4.4.
   endfor
3  return $Mast(r_1, ..., r_k)$, where $r_i$ is the root of $T_i$ for $1 \leq i \leq k$.
End  Compute Mast
```

Figure 4.9: A dynamic programming algorithm for computing all values of *Mast*.

However, the running time of Algorithm *Compute Mast* may be very large for two reasons.

First of all, there are $O(n^k)$ k-tuples in $V(T_1) \times ... \times V(T_k)$. For most of these, $Mast$ equals zero because there is no leaf which is contained in all of the subtrees rooted at that k-tuple. Therefore, most k-tuples do not contribute to the $Mast$ values of other k-tuples. In the proof of Lemma 4.6 below, we will show that the number k-tuples with $Mast$ strictly greater than 0 is bounded by $n(H+1)^k$, allowing the running time to be reduced correspondingly by only considering these k-tuples.

Secondly, and more problematically, we cannot expect to be able to compute the exact value of $Match(v_1, ..., v_k)$ in the expression for $Mast(v_1, ..., v_k)$ in Lemma 4.4 in polynomial time since finding a maximum weight matching in a k-partite hypergraph is NP-hard already for the special case with $k = 3$ and all weights set to 1 [49, 105]. For this reason, we rely upon a greedy, polynomial-time algorithm for approximating $Match(v_1, ..., v_k)$, which in turn yields an approximation of $Mast(v_1, ..., v_k)$. The performance and running time of the greedy algorithm are given by the next lemma.

Lemma 4.5 *Let $\mathcal{H} = (V, E)$ be a k-partite hypergraph with positive integer weights. A matching of $\mathcal{H}$ with total weight within a factor k of the maximum can be constructed in $O(|V| + k|E| + |E| \log |E|)$ time.*

Proof: Compute a maximal matching M of $\mathcal{H}$ with the following greedy algorithm. Initially, let M be the empty set. Repeat until E is empty: find the heaviest edge e in E, remove e from E, add e to M, and delete from E all edges which overlap e. Finally, return M.

To analyze the greedy algorithm's performance, let M^* be a maximum weight matching of $\mathcal{H}$. For any edge $f \in M^*$, f is eventually removed from E by the greedy algorithm due to some overlapping edge e being selected for inclusion in M; we say that f is *accounted for by* e and write $f \in A(e)$ (if f itself belongs to M then f is accounted for by f). Note that f must be accounted for by some edge in M since at least one vertex in f is in an edge of M. Now, whenever an edge e is added to M by the algorithm, every edge that remains in E has weight less than or equal to $weight(e)$. Furthermore, e can overlap at most k edges in M^*, so the total weight of the edges in M^* that remain in E at that point and overlap e is less than or equal to $k \cdot weight(e)$, i.e.,

$$\sum_{e^* \in A(e)} weight(e^*) \quad \leq \quad k \cdot weight(e)$$

Summing over all edges in M gives us

$$\sum_{e \in M} \sum_{e^* \in A(e)} weight(e^*) \quad \leq \quad k \cdot \sum_{e \in M} weight(e)$$

The left-hand side equals $\sum_{e^* \in M^*} weight(e^*)$, so $weight(M) \geq \frac{1}{k} \cdot weight(M^*)$.

To implement the greedy algorithm, scan the edges in E once to produce a list $L(v)$ for every vertex $v \in V$ of all edges in E incident to v. This takes $O(|V| + k|E|)$ time. Next, sort the edges in E according to nonincreasing weights in $O(|E| \log |E|)$ time and store them in a doubly linked list. Then, when an edge is to be added to M, select the first element e in the sorted list and locate all edges which overlap e using the $L(v)$-lists with $v \in e$. For each such edge, checking if it still belongs to the sorted list and in that case deleting it takes $O(1)$ time, so this step takes a total of $O(k|E|)$ time. Therefore, the running time of the greedy algorithm is $O(|V| + k|E| + |E| \log |E|)$. □

By modifying Algorithm *Compute Mast* and applying Lemma 4.5, we obtain the following:

Lemma 4.6 *$Mast(r_1, ..., r_k)$, where r_i is the root of T_i for $1 \leq i \leq k$, can be approximated within a factor of k^h in $O(2^k \cdot n(H+1)^k \cdot (\log n + k \log H))$ time, where $h = \min\{height(T_i) \mid 1 \leq i \leq k\}$ and $H = \max\{height(T_i) \mid 1 \leq i \leq k\}$.*

Proof: For any $(v_1, ..., v_k) \in V(T_1) \times ... \times V(T_k)$, denote by $s(v_1, ..., v_k)$ the size of the set $\Lambda(T_1[v_1]) \cap ... \cap \Lambda(T_k[v_k])$. Clearly, $Mast(v_1, ..., v_k) \leq s(v_1, ..., v_k)$, and in particular, if at least one of the v_i's is a leaf then $Mast(v_1, ..., v_k) = s(v_1, ..., v_k)$. For every leaf j, we can determine all k-tuples $(v_1, ..., v_k)$ for which $j \in \Lambda(T_1[v_1]) \cap ... \cap \Lambda(T_k[v_k])$ by finding the nodes on the path of length $\leq H$ from the leaf j to the root in each T_i, $i = 1, ..., k$. It follows that the number of such k-tuples is $\leq (H+1)^k$. Consequently, the set L of all k-tuples for which $s(v_1, ..., v_k) > 0$ has size not exceeding $n(H+1)^k$. To generate L, we sort the pointers to the leaves by their labels in $O(n \log n)$ time for each of the k trees and then list the elements of L (including repetitions) by following appropriate tree paths, using a total of $O(kn \log n + |L| \cdot k)$ time.

For every k-tuple $(v_1, ..., v_k)$ not in the set L, we have $Mast(v_1, ..., v_k) = 0$ since $Mast(v_1, ..., v_k) \leq s(v_1, ..., v_k)$. For every $(v_1, ..., v_k) \in L$ where at least one of $v_1, ..., v_k$ is a leaf, $s(v_1, ..., v_k) = 1$ and $Mast(v_1, ..., v_k) = 1$. To efficiently compute approximations of $Mast(v_1, ..., v_k)$ for the rest of the k-tuples in L, we first build a balanced search tree S_L for L (without repetitions of k-tuples) with respect to the lexicographic ordering $\mathcal{O}$. Then, we follow the scheme of Algorithm *Compute Mast* but only evaluate k-tuples which belong to L (we can traverse S_L to enumerate them in the correct order). For every k-tuple $(v_1, ..., v_k)$ in L, we apply the greedy algorithm described in Lemma 4.5 to approximate $Match(v_1, ..., v_k)$ in the hypergraph $\mathcal{H}_L(v_1, ..., v_k)$, defined as the hypergraph $\mathcal{H}(v_1, ..., v_k)$ restricted to edges in L.

To construct the $\mathcal{H}_L$-hypergraphs, we augment each k-tuple $(v_1, ..., v_k)$ in L with a list $E(v_1, ..., v_k)$. Any $(w_1, ..., w_k) \in L$ may occur as an edge in the $\mathcal{H}_L$-hypergraphs only for $\mathcal{H}_L(v_1, ..., v_k)$ where v_i is the parent of w_i for $i = 1, ..., k$, i.e., at most once. Hence, when $(w_1, ..., w_k)$ has been evaluated, if none of the nodes $w_1, ..., w_k$ is a root, we find $(v_1, ..., v_k)$ and then store the k-tuple $(w_1, ..., w_k)$ and the approximate value of $Mast(w_1, ..., w_k)$ in $E(v_1, ..., v_k)$ using S_L. Because of the ordering $\mathcal{O}$, when the algorithm later on needs to approximate $Match(v_1, ..., v_k)$, $E(v_1, ..., v_k)$ contains all edges in $\mathcal{H}_L(v_1, ..., v_k)$ and their approximate weights.

We can employ a similar technique to obtain the *Diag*-values. Equip each k-tuple in L with a list D, and whenever some k-tuple $(w_1, ..., w_k)$ has been evaluated, store the approximate value of $Mast(w_1, ..., w_k)$ in the D-lists of the at most $2^k - 1$ different k-tuples in $\{v_1, w_1\} \times ... \times \{v_k, w_k\} - \{(w_1, ..., w_k)\}$ where for $i = 1, ..., k$, v_i is the parent of w_i (let v_i equal w_i if w_i is the root of T_i), using S_L. Then, when the algorithm has to evaluate a k-tuple $(v_1, ..., v_k)$, its D-list contains approximate *Mast*-values of all k-tuples in $(\{v_1\} \cup C(v_1)) \times ... \times (\{v_k\} \cup C(v_k)) - \{(v_1, ..., v_k)\}$, and by the definition of *Diag*, $Diag(v_1, ..., v_k)$ can be obtained by taking the maximum value in its D-list.

We now prove by induction on h that the approximation factor of this method is k^h. First observe that all non-optimal values obtained in the $Mast$-computations are due to approximations of $Match$. Thus, the returned solution differs from the optimum only if it uses one or more approximate $Match$-values. This immediately implies that if $h = 0$ then $Mast(r_1, ..., r_k)$ is computed exactly. Next, for $h > 0$, assume inductively that k^{h-1}-approximations of every $Mast(w_1, ..., w_k)$, where $w_i \in V(T_i) - \{r_i\}$ for $1 \leq i \leq k$, are available. Let T_j be a tree in $\{T_1, ..., T_k\}$ with height h. By the induction hypothesis, we have k^{h-1}-approximations of the weights of all edges in the hypergraph $\mathcal{H}_L(v_1, ..., r_j, ..., v_k)$ for any $(v_1, ..., r_j, ..., v_k) \in V(T_1) \times ... \times \{r_j\} \times ... \times V(T_k)$. We subsequently obtain an approximation of $Match(v_1, ..., r_j, ..., v_k)$ which is within a factor of k^h of the optimum when using Lemma 4.5. It follows that $Diag(v_1, ..., r_j, ..., v_k)$ for any $(v_1, ..., r_j, ..., v_k) \in V(T_1) \times ... \times \{r_j\} \times ... \times V(T_k)$, and hence $Mast(r_1, ..., r_k)$, is approximated within a factor of k^h.

Finally, we analyze the running time. To generate L takes $O(kn \log n + |L| \cdot k)$ time, where $|L| \leq n(H+1)^k$ as shown above, and to build S_L takes $O(|L| \log |L|)$ time. Denote the number of edges in $\mathcal{H}_L(v_1, ..., v_k)$ by $e(v_1, ..., v_k)$. Then $\mathcal{H}_L(v_1, ..., v_k)$ has at most $k \cdot e(v_1, ..., v_k)$ vertices. Since the $\mathcal{H}_L$-hypergraphs have no more than $|L|$ edges in total (recall that each k-tuple in L appears as an edge in at most one $\mathcal{H}_L$-hypergraph), we have $\sum_{(v_1, ..., v_k) \in L} e(v_1, ..., v_k) \leq |L|$. By Lemma 4.5, the time required to construct all $\mathcal{H}_L$ and approximate all $Match$-values is therefore bounded by

$$O(k + \log |L|) \cdot |L| \\ + \sum_{(v_1, ..., v_k) \in L} O\big(k \cdot e(v_1, ..., v_k) + k \cdot e(v_1, ..., v_k) + e(v_1, ..., v_k) \log e(v_1, ..., v_k)\big)$$

which is $O((k + \log |L|) \cdot |L| + k|L| + k|L| + |L| \cdot \log n) = O(|L| \log |L|)$. Creating the D-lists takes a total of $O((k + 2^k \cdot \log |L|) \cdot |L|)$ time, and the time needed to scan all the D-lists (to find maximum values) is proportional to the sum of their lengths, which is $O(2^k \cdot |L|)$; thus, the $Diag$ computations take a total of $O(2^k \cdot |L| \log |L|)$ time. Adding everything together, we see that the total running time is $O(2^k \cdot |L| \log |L|) = O(2^k \cdot n(H+1)^k \cdot (\log n + k \log H))$. $\Box$

To compute an actual approximate maximum agreement subtree and not just the number of leaves it contains, extend the modified Algorithm *Compute* $Mast$ in the proof of Lemma 4.6 in the following way. Associate an initially empty list $M(v_1, ..., v_k)$ to each k-tuple $(v_1, ..., v_k)$ in L. When $Mast(v_1, ..., v_k)$ is computed, insert pointers to the k-tuples which contribute to the value of $Mast(v_1, ..., v_k)$ into $M(v_1, ..., v_k)$. (If at least one of $v_1, ..., v_k$ is a leaf then $M(v_1, ..., v_k)$ is left empty. Otherwise, if the approximate value of $Diag(v_1, ..., v_k)$ is greater than or equal to the approximate value of $Match(v_1, ..., v_k)$ then insert a pointer to a k-tuple in the expression for $Diag$ yielding the maximum value of $Mast(v_1, ..., v_k)$;

if the approximate value of $Diag(v_1, ..., v_k)$ is less than the approximate value of $Match(v_1, ..., v_k)$ then insert pointers to all k-tuples which are edges in the selected matching of $\mathcal{H}_L(v_1, ..., v_k)$.) After $Mast(r_1, ..., r_k)$ has been calculated, the set S' of elements in an approximate maximum agreement subtree can be reconstructed by following pointers in the M-lists, starting at $M(r_1, ..., r_k)$. Finally, return the tree $T_1|S'$ as the solution.

The additional time needed to build all of the M-lists and to follow pointers to obtain S' is $O(|L| \log |L|)$, and the time required to construct $T_1|S'$ is $O(n)$, so the total running time is still $O(2^k \cdot n(H+1)^k \cdot (\log n + k \log H))$. Since $k = O(1)$, $h = O(1)$, and $H = O(n)$ imply that this expression is a polynomial in n and that k^h is a constant, we have just proved Theorem 4.3.

If we also require the height of every input tree to be bounded by a constant (i.e., $H = O(1)$) then the asymptotic running time is only $O(n \log n)$ since $(H+1)^k$ is $O(1)$.

Corollary 4.7 *MAST restricted to instances with $k = O(1)$ and $H = O(1)$ can be approximated within a constant factor in $O(n \log n)$ time.*

We end this section with the observation that an algorithm of Berman [17] based on local search can be used to approximate maximum weight matchings in k-partite edge-weighted hypergraphs in polynomial time[7]. The resulting approximation factor is $\frac{k+1}{2}$, which beats the approximation factor of k of the greedy algorithm described in Lemma 4.5; however, its running time can be much slower than that of the greedy algorithm. Thus, we can improve the approximation factor in Lemma 4.6 to $(\frac{k+1}{2})^h$ if we are willing to sacrifice some additional running time.

4.5 Concluding Remarks

Below, we summarize how restricting the parameters h, H, and k affects the computational complexity of MAST. The first table lists hardness results, and the second one shows how well we can approximate MAST in polynomial time.

Arrows indicate when a result follows directly from another by generalization (for example, MAST restricted to instances with $H = 2$ and $k = 3$ is NP-hard,

[7]Given a d-claw free graph (i.e., a graph in which no vertex has d neighbors which form an independent set) $G = (V, E)$ and a function $w : V \to \mathbb{R}_+$, the algorithm in [17] finds an independent set I such that $w(I^*)/w(I) \leq d/2$, where I^* is an independent set which maximizes $w(I^*)$. Consider the vertex-weighted graph $G_{\mathcal{H}}$ formed from a k-partite edge-weighted hypergraph $\mathcal{H}$ by representing each hyperedge in $\mathcal{H}$ by a vertex with the same weight, and including an edge between two vertices in $G_{\mathcal{H}}$ if and only if the corresponding hyperedges in $\mathcal{H}$ intersect. Any independent set in $G_{\mathcal{H}}$ corresponds to a matching of $\mathcal{H}$. Furthermore, if a hyperedge e in $\mathcal{H}$ overlaps $k+1$ hyperedges then at least two of them must overlap e in the same vertex in $\mathcal{H}$ and therefore be neighbors in $G_{\mathcal{H}}$, so no set of $k+1$ neighbors of e in $G_{\mathcal{H}}$ can constitute an independent set, i.e., $G_{\mathcal{H}}$ is $(k+1)$-claw free.

so the more general case with $H = 2$ and $k = O(1)$ cannot be any easier) or by specialization (e.g., the $O(1)$-approximation algorithm for the case with $h = O(1)$ and $k = O(1)$ still works for the more restricted case with $h = 2$ and $k = O(1)$). Note that since $h \leq H$, any upper bound imposed on H implies an upper bound on h. Thus, although the case with $H = O(1)$ and $k = 3$ is NP-hard, it admits a polynomial-time $O(1)$-approximation algorithm.

Negative results	$k = 3$	$k = O(1)$	k unrestricted
$H = 2$	NP-hard (Amir and Keselman [6] [8])	NP-hard ($\leftarrow$)	Hard to approximate (Theorem 4.2)
$H = O(1)$	NP-hard ($\uparrow$)	NP-hard ($\nwarrow$)	Hard to approximate ($\uparrow$)
H unrestricted	Hard to approximate (Hein *et al.* [60])	Hard to approximate ($\leftarrow$)	Hard to approximate ($\leftarrow$ or $\uparrow$)

Positive results	$k = 3$	$k = O(1)$	k unrestricted
$h = 2$	$O(1)$-approx. ($\searrow$)	$O(1)$-approx. ($\downarrow$)	$n/\log n$-approx. ($\downarrow$)
$h = O(1)$	$O(1)$-approx. ($\rightarrow$)	$O(1)$-approx. (Theorem 4.3)	$n/\log n$-approx. ($\downarrow$)
h unrestricted	$n/\log n$-approx. ($\rightarrow$)	$n/\log n$-approx. ($\rightarrow$)	$n/\log n$-approx. (Theorem 4.1)

We conclude that only restricting the heights of the input trees *or* the number of input trees is not enough to render efficient approximation algorithms with small approximation factors possible. However, MAST restricted to instances where the height of at least one of the input trees *and* the number of input trees are known to be upper bounded by constants can be approximated within a constant factor in polynomial time.

We also remark that other techniques for approximating MAST might be useful for instances in which the maximum agreement subtree is known to contain many leaves. For example, consider the problem of finding a subset of S of minimum cardinality whose removal leaves a set $\tilde{S}$ such that $T_1|\tilde{S} = T_2|\tilde{S} = \ldots = T_k|\tilde{S}$. This problem is supplementary to MAST in the sense that an exact algorithm for one of the two problems automatically yields an exact algorithm for

[8] A closer inspection of the proof of Amir and Keselman in [6] reveals that the NP-hardness holds even if the problem is further restricted to $H = 2$ since all trees constructed in the reduction have height 2.

the other, but approximation factors are not preserved[9]. In [6], Amir and Keselman gave a polynomial-time, factor 4 approximation algorithm for supplementary UMAST; we note that given an instance $(S, \mathcal{T})$ of UMAST, removing a 4-approximate solution to the supplementary problem from S provides an approximate solution to the original problem which has at least $n-4(n-m) = 4m-3n$ leaves, where m is the number of leaves in a maximum agreement subtree. Therefore, this method yields a good approximation for UMAST if m is known to be large: e.g., if $m \geq 0.95n$ then $4m-3n \geq 0.8n$. We further note that the approximation algorithm for supplementary UMAST can be turned into a polynomial-time, factor 3 approximation algorithm for supplementary MAST[10], and that given an instance $(S, \mathcal{T})$ of MAST, removing a 3-approximate solution to supplementary MAST from S provides an approximate solution to MAST with at least $n-3(n-m) = 3m-2n$ leaves. As above, this results in a good approximation factor for MAST if m is large: e.g., if $m \geq 0.95n$ then $3m-2n \geq 0.85n$; indeed, this method gives a constant approximation factor for MAST whenever $m \geq 0.67n$.

Finally, we list some open problems related to the approximability of MAST suitable for further research.

1. Our results in this chapter show that MAST restricted to instances with $h = 2$ and $k = 3$ can be approximated in polynomial time within a factor of $3^2 = 9$. For this special case, is it possible to construct a polynomial-time approximation scheme (PTAS), or at least a polynomial-time approximation algorithm with better approximation factor than 9? If so, can the same techniques be applied to obtain tighter approximations for other cases of MAST as well?

2. MAST restricted to instances with $H = 1$ can be solved exactly in polynomial time with a trivial algorithm. On the other hand, only requiring that $h = 1$ does not help to make the problem easier to solve since the inapproximability results of Theorem 4.2 can be extended to cover this case[11]. The NP-hardness proof of Amir and Keselman [6] can be modified

[9]The same situation occurs for the maximum independent set problem (MIS) and its "supplement", the minimum vertex cover problem (MVC). MIS cannot be approximated within a factor of $l^{1/2-\delta}$, where l is the number of vertices in the input graph, for any constant $\delta > 0$ in polynomial time unless P=NP [66], whereas MVC can be trivially approximated within a factor of 2 by finding a maximal matching in the graph and outputting the set of matched vertices [49, 105, 128].

[10]Instead of constructing the set $S4$ of all 4-element subsets of S which do not induce identical topologies in all of the input trees and then greedily computing an approximate minimum cover of $S4$ (see [6]), construct the set $S3$ of rooted triples on S which are not consistent with all the input trees and return a 3-approximate minimum cover of $S3$.

[11]In the proof, also construct a rooted tree T_{k+1} consisting of a root node attached to $l+2$ leaves labeled by $w_1, ..., w_l, w_{l+1}, w_{l+2}$. Clearly, T_{k+1} has height 1. Furthermore, any maximum agreement subtree of $T_1, ..., T_k$ is also a maximum agreement subtree of $T_1, ..., T_{k+1}$, and vice versa. Thus, the inapproximability results hold even if $h = 1$ and $H = 2$.

in a similar way to show that MAST restricted to $h = 1$ and $k = 4$ is NP-hard. But what happens to the computational complexity of MAST when $h = 1$ and $k = 3$? Is the problem still NP-hard, or does it become possible to solve exactly in polynomial time?

3. What is the computational complexity of the maximum agreement subtree problem for rooted, *ordered* trees? Sung [119] showed that MAST for two rooted, ordered trees can be solved in $O(n \log^2 n)$ time, which is less than the running times of the currently fastest algorithms for two rooted, unordered trees and for two unrooted trees (see Section 4.1.2). Moreover, as shown in Chapters 2 and 3, the maximum 3-leaf constraints consistency problem is NP-hard for unordered trees but becomes solvable in polynomial time for ordered trees. Also, the problem of computing an optimal alignment between two node-labeled trees which we study in Chapter 5 is NP-hard for unordered trees, yet polynomial-time solvable for ordered trees. We conjecture that MAST for ordered trees can also be solved in polynomial time even though MAST for unordered trees is NP-hard. If, however, MAST for ordered trees turns out to be NP-hard, can it be efficiently approximated even if the trees are allowed to have unbounded degrees and the parameters h, H, and k remain unrestricted?

Chapter 5

Fast Optimal Alignment between Two Labeled, Ordered Trees

Jiang, Wang, and Zhang [70] proposed the concept of an alignment between two node-labeled, rooted trees as a way to measure their similarity and to identify parts of the trees which are alike. They presented an algorithm for computing an optimal alignment between two node-labeled, ordered trees, but left it as an open question to determine whether its running time can be improved. We show that the answer is "yes" for problem instances where the two input trees are similar, i.e., where the score of an optimal alignment between them is high, under some natural assumptions on the scoring scheme.

5.1 Introduction

In this chapter, a tree is said to be *labeled* if each node in the tree is labeled by a symbol from a fixed finite set Σ or by a special blank symbol '$-$' which we assume does not belong to Σ. An *ordered* tree is a rooted tree in which the left-to-right order of the children of each node is significant.

An *insert operation* on a labeled, rooted tree adds a new node u, labeled by the blank symbol '$-$'. The operation either:

(1) turns the current root of the tree into a child of u and lets u become the new root;

or

(2) makes u the parent of a subset of (if the tree is unordered) or consecutive subsequence of (if the tree is ordered) children[1] of an existing node v, and u a child of v.

Figure 5.1 shows an example of an insert operation.

Let S and T be two rooted trees, labeled by Σ. An *alignment between S and T* is a tree obtained by first performing insert operations on S and T so that the two resulting trees S' and T' are isomorphic when labels are ignored and then overlaying S' on T'. In addition, it is required that no node of the alignment corresponds to two nodes $s' \in S'$ and $t' \in T'$ which are both labeled by blank symbols. The *score* of the alignment is the sum of the scores of all pairs of aligned nodes, where the score of a pair of nodes is determined by a given function μ defined on $\Sigma' \times \Sigma' \setminus \{(-,-)\}$ with $\Sigma' = \Sigma \cup \{-\}$. See Figure 5.2 for an example.

An *optimal alignment* between a pair of labeled, rooted trees is an alignment between them achieving the highest possible score[2].

From here on, we assume that Σ and μ have been fixed beforehand so that μ is not part of the actual input. Also, we consider ordered trees only; thus, we focus on the following problem[3]:

The alignment between ordered trees problem

Instance: Two labeled, ordered trees S and T.

Output: An optimal alignment between S and T under the scoring function μ.

For any tree S, $|S|$ represents the number of nodes in S. The *degree of a node u* in a rooted tree is the number of children of u and is denoted by $\deg(u)$. The *degree of a rooted tree S* is the maximum degree of all nodes in S and is denoted by $\deg(S)$. Given an instance (S, T) of the alignment between ordered trees problem, we write $m = \min\{|S|, |T|\}$ and $n = \max\{|S|, |T|\}$, and let $\Delta = \max\{\deg(S), \deg(T)\}$.

[1]Observe that subsets and consecutive subsequences can consist of zero elements.

[2]In [70], Jiang, Wang, and Zhang defined an optimal alignment as one with the *lowest* possible score.

[3]We would have preferred to call it *the tree alignment problem* (in analogy to *the string alignment problem*), but this name is already in use for another optimization problem, defined in, e.g., [10, 58, 111]. See also Section 6.1.1.

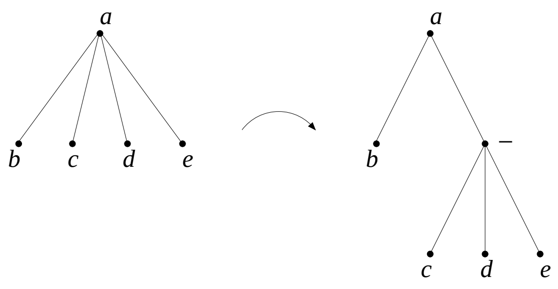

Figure 5.1: An insert operation of type (2). The new node becomes the parent of a consecutive subsequence of children of the node labeled by a, and then becomes a child of that node.

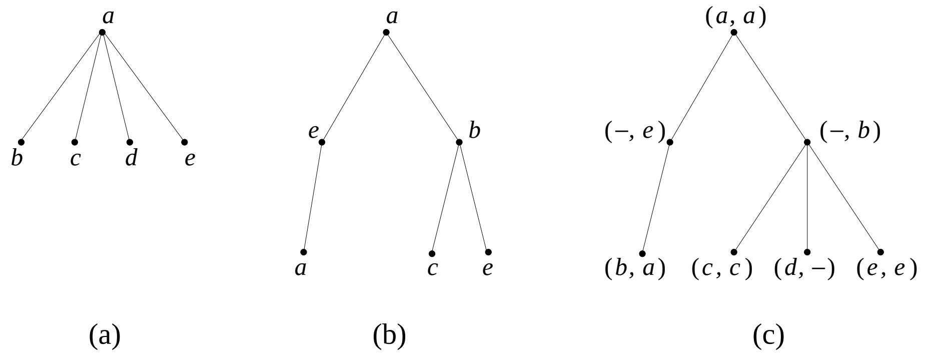

Figure 5.2: Let $\Sigma = \{a, b, c, d, e\}$ and define the scoring function μ as follows: for every $x, y \in \Sigma$ with $x \neq y$, let $\mu(x, x) = 3$, $\mu(x, y) = -1$, and $\mu(x, -) = \mu(-, x) = -2$. Then the score of the alignment in (c) between the two labeled trees shown in (a) and (b) is equal to 2.

5.1.1 Motivation

Many areas of computer science use labeled, ordered trees to represent hierarchically structured information. It is sometimes necessary to measure the similarity between two or more such trees or to find parts of the trees which are similar:

- In computational molecular biology, labeled ordered trees can be used to describe RNA molecules' secondary structures [112], allowing researchers investigating, e.g., evolutionary relationships between RNA molecules to obtain additional clues by measuring and comparing the similarities of their secondary structure trees [29][4]. Also, when trying to determine the

[4]This seems especially helpful when the strings representing the primary structures of the molecules cannot be reliably aligned, as in the case of pRNA and mrpRNA studied in [29]. In

secondary structure of a given RNA molecule, it is useful to look for frequently recurring patterns among a set of candidate secondary structures obtained by simulating the folding process of the molecule [89]. Furthermore, likely locations of certain regulatory sequences in a DNA molecule can be revealed by finding occurrences of particular patterns in the secondary structure of the corresponding mRNA. For example, [38] employed a method based on energy-scoring functions and a simple mRNA secondary structure matching criterion to predict where in twelve different bacterial genomes that rho-independent transcription terminators occur; if the secondary structure of the RNA is known, algorithms for identifying more complex patterns in the secondary structure trees may facilitate the search for other important regions.

- The ability to detect changes in electronic documents and structured data is crucial for information management and data archiving applications. Often, the entire history of modifications made to a data file is unavailable but snapshots of previous versions of the file can be obtained and then analyzed and compared to the current version [25]. Hence, algorithms for comparing labeled, ordered trees in order to find changes between different versions of hierarchically structured documents such as LaTeX- and SGML-files have been developed [25, 129].

- In software construction, the source code of computer programs (or different versions of one program) can be represented by labeled, ordered trees and then compared in order to identify syntactic differences between them [132]. Algorithms for finding correspondencies between labeled, ordered trees thus provide a useful tool for programmers and software maintainers who need to know where in the source code structural changes have been made between different versions, or for a team of programmers who want to merge their work[5].

  A related application is described in [14]. Suppose a computer program controlling an industrial process has been running for some time and that the program has to be updated. However, the current values of counters and certain other variables need to be preserved, e.g., to monitor when various hardware components require maintenance. Rather than restarting the program from scratch and manually assigning the old values to the relevant variables during execution or modifying their initial values directly in the source code by hand before recompiling the program (indeed a tedious task if there are thousands of counters, many with similar names,

general, if the RNA molecules to be compared have evolved for a long time, methods that also take into account secondary structure information are potentially more accurate than those that only rely upon the primary structure [100].

[5]Instead of *computing* the differences between versions, it may be more practical (when possible) to let the programming environment keep track of the modifications which have been made, for example by using techniques such as the ones described in [9]. (This can save a lot of time if the trees are very large and correspondencies have to be reported many times.)

and perhaps even some with identical names, occurring in different blocks of the program), it would be desirable to automatize this step by identifying parts of the program which remain unchanged so that information can be transferred from the old version of the program to the new version easily. For this purpose, the algorithms given in [14] attempt to find pairs of matching nodes among two labeled, rooted trees representing the two versions being compared.

- By comparing various subtrees or subforests of a labeled, ordered tree representing a computer program, one can locate fragments of the source code which are identical to each other. Detecting and replacing such "clones" by, e.g., subroutines or macros may improve the structure of (and thus decrease the maintenance costs of) software [11].
- Compilers need mechanisms for automatic error recovery to be able to report more than just one error per run when analyzing the source code of a computer program. To select the best way to recover from an encountered syntactic error so that the parsing process can continue, a compiler can measure the similarity between the erroneous part of the program and various valid replacements by using comparison algorithms for strings as well as for the corresponding labeled, ordered parse trees [124].

Practical uses of comparing labeled, *unordered* trees can be found in such apparently unrelated disciplines as chemical structure analysis, image recognition, and information retrieval in next-generation database systems (see the references in [70, 113]). Also, identifying structural similarities in pairs of labeled, unordered trees representing sentences expressed in two different languages (e.g., English and Japanese) to build an example base or to extract translation patterns is useful in the preparation of automated natural language translation systems [95].

The applications listed above employ a variety of techniques and heuristic matching rules; it might be possible to improve the performance of some of these methods by incorporating alignments between trees. In any case, it is advantageous to have many methods to choose from since alternative ways of measuring the similarity between labeled trees or alternative criteria for matching nodes may be suitable in different contexts. Hence, algorithms for computing optimal alignments between trees are valuable because of the generality and flexibility provided by the freedom of the programmer to select the scoring function μ as appropriate.

Moreover, in practical applications, it is preferable to have algorithms which execute efficiently. The fastest known algorithms for computing optimal alignments between trees have lower time complexity than the fastest known algorithms for another measure of similarity called the *tree edit distance* [124], both for unordered trees whose degrees are bounded by a constant [70, 133] and for ordered trees whose degrees are much smaller than their depths [70, 134].

Further motivation for studying the alignment between ordered trees problem comes from the fact that it generalizes some well-known combinatorial problems. For instance, the special case in which all internal nodes of the trees have exactly one child[6] is known as *the string alignment problem*, and the special case of the string alignment problem where the scoring function satisfies $\mu(x, x) = 1$ and $\mu(x, -) = \mu(-, x) = \mu(x, y) = 0$ for every $x, y \in \Sigma$ with $x \neq y$ is *the longest common subsequence problem* (LCS) [58]. These two problems have been studied extensively because of their numerous applications to computer science, molecular biology, abstract algebra, speech recognition, dendrochronology, and many other fields [58, 106, 111, 115, 130, 131]. As an example, the indispensable UNIX utility `diff` for comparing two text files interprets each line of the input files as one symbol and then applies an LCS-based algorithm [115]. Given the extreme usefulness of algorithms for solving the string alignment problem and LCS, efficient algorithms for the more general problem should also be useful. *The maximum (ordered) refinement subtree problem* [60], *the smallest common (ordered) supertree problem* [99], and *the (ordered) tree inclusion problem* [83] are other noteworthy problems which can be cast as special cases of the alignment between (ordered) trees problem (see [60] and [70]).

The algorithm presented in this chapter is designed to efficiently solve instances of the alignment between ordered trees problem where S and T are similar, meaning that an optimal alignment between S and T has a high score. The motivation for this is that in many applications, the two input trees being compared can be assumed to be closely related and therefore do not differ greatly (for example, if only a few changes have been made between two versions of an electronic document). Then, an algorithm that just considers alignments without a lot of blank symbols and mismatches can compute an optimal alignment more efficiently than an algorithm which does not make this assumption. For the special case of string alignments, Section 12.2 of [58] lists several examples where two or more very similar strings need to be compared and where the speedups obtained by exploiting bounded difference methods are of great practical importance.

5.1.2 Previous Results

Jiang, Wang, and Zhang [70] generalized string alignments to alignments between labeled trees and gave an algorithm for the alignment between ordered trees problem with $O(|S| \cdot |T| \cdot \Delta^2)$ running time.

Jiang *et al.* [70] also observed that although the score of an optimal alignment between two strings and their edit distance are equivalent notions (see, e.g., [58, 106, 111, 130]), the score of an optimal alignment between two trees (ordered or unordered) and their tree edit distance [124] are not. In fact, they provided

[6]More formally, the restriction of the alignment between ordered trees problem to instances with $\Delta = 1$ and where no insert operation may result in a node with degree > 1.

a polynomial-time algorithm for computing an optimal alignment between two labeled, unordered trees when $\Delta = O(1)$ (and showed the latter problem to be MAX SNP-hard if at least one of the input trees is permitted to have an arbitrary degree), whereas computing the tree edit distance for two labeled, unordered trees is MAX SNP-hard even if $\Delta = O(1)$ [133].

The standard algorithm for the special case of string alignments runs in $O(mn)$ time, where m and n are the lengths of the two input strings [58, 106, 111, 130]. By modifying the algorithm as described in Section 3.3.4 of [111] (see also Section 12.2 of [58]), an optimal alignment between two strings of length n can in fact be computed in $O(nf)$ time under certain assumptions on the scoring scheme, where f is the difference between the score of two identical strings of length n and the optimal score of the two input strings.

5.1.3 Our Contributions

Inspired by the fast method for computing an optimal alignment between two similar strings described in Section 3.3.4 of [111], we present an algorithm for the alignment between ordered trees problem which is faster than the algorithm of Jiang *et al.* when the score of an optimal alignment between the two input trees is high and the scoring scheme satisfies some natural assumptions.

We first give an algorithm called Algorithm *Fast Score* which computes the score of an optimal alignment between S and T in $O(n \cdot (\log n + \Delta^3) \cdot d^2)$ time if an optimal alignment with at most d blank symbols exists and d is specified in advance. The main idea behind Algorithm *Fast Score* is to speed up the algorithm of Jiang *et al.* by only considering what we call d-relevant pairs of subtrees and subforests. Next, we present a more general algorithm called Algorithm *Unspecified d* for when no upper bound on d is provided which computes the score of an optimal alignment in $O(n \cdot (\log n + \Delta^3) \cdot f^2)$ time, where (assuming that the scoring scheme satisfies certain properties) f is the difference between the highest possible score for any alignment between two trees having a total of $|S| + |T|$ nodes and the score of an optimal alignment between S and T. Furthermore, if there exists an optimal alignment between S and T with b blank symbols and $O(b)$ node pairs of the form (x, y), where $x \neq y$, then (under some slightly stronger assumptions on the scoring scheme) Algorithm *Unspecified d* runs in $O(n \cdot (\log n + \Delta^3) \cdot b^2)$ time, even if b is not known in advance. In particular, if the degrees of both input trees are bounded by a constant, the running times stated above reduce to $O(n \log n \cdot d^2)$, $O(n \log n \cdot f^2)$, and $O(n \log n \cdot b^2)$, respectively.

In Section 5.2, we describe the algorithm of Jiang *et al.* from [70]. Then, in Section 5.3, we define the new concept we call d-relevance. In Section 5.4, we show how to test whether a given pair of subtrees or subforests is d-relevant, and in Section 5.5, we describe an efficient method for enumerating all d-relevant pairs of subtrees. Next, we present and analyze Algorithm *Fast Score* and Algorithm *Unspecified d* in Sections 5.6 and 5.7. In Section 5.8, we summarize our results and discuss some open problems.

5.2 The Algorithm of Jiang, Wang, and Zhang

The algorithm of Jiang, Wang, and Zhang [70] for aligning two labeled, ordered trees is based on the standard dynamic programming algorithm for the string alignment problem which calculates the scores of optimal alignments between pairs of *prefixes* (or symmetrically, *suffixes*) of the two input strings in bottom-up order by using a two-dimensional table to store the computed scores, and then, when the table is complete, performs a traceback to obtain an optimal alignment (see, e.g., [58, 106, 111, 130]). The algorithm of Jiang *et al.* computes and stores the scores of optimal alignments between pairs of *ordered subtrees* of S and T and between pairs of *ordered subforests* of S and T in a bottom-up fashion. After the algorithm is finished, an actual optimal alignment between S and T can also be recovered by doing a traceback.

Some notation is necessary to describe the algorithm in more detail.

Definition 5.1 For an ordered tree S and a node u of S, let $S[u]$ denote the ordered subtree of S rooted at u (i.e., the minimal subgraph of S which includes u and all of its descendants). Let $\deg(u)$ be the degree of u, and denote the children of u by $u_1, ..., u_{\deg(u)}$ according to their left-to-right order. $S(u, i, j)$ refers to the ordered subforest $S[u_i], ..., S[u_j]$, and $S(u)$ is short for $S(u, 1, \deg(u))$. The number of nodes in a subtree or subforest S_* is denoted by $|S_*|$. Finally, $\deg(S)$ is defined as the maximum degree of all nodes in S.

Thus, $S(u)$ is the *complete ordered forest* obtained by removing u and all edges incident to u from $S[u]$. Also observe that $S(u, i, i) = S[u_i]$.

Definition 5.2 The score of an optimal alignment between two subtrees or two subforests S_* and T_* is denoted by $D(S_*, T_*)$.

To obtain a bottom-up ordering of the subtrees and subforests suitable for dynamic programming, the nodes in an ordered tree with n nodes are numbered 1 through n according to postorder so that $D(S[|S|], T[|T|])$ will contain the score of an optimal alignment between S and T. Henceforth, Θ represents the empty tree and $g_S(u)$ is the label of node number u in the labeled tree S.

The next lemma forms the basis of the algorithm of Jiang *et al.*

Lemma 5.3 *Let S and T be two labeled ordered trees with $u \in S$ and $v \in T$. Then:*

1. $D(\Theta, \Theta) = 0$

$$D(S[u], \Theta) = D(S(u), \Theta) + \mu(g_S(u), -), \quad D(S(u), \Theta) = \sum_{q=1}^{\deg(u)} D(S[u_q], \Theta)$$

$$D(\Theta, T[v]) = D(\Theta, T(v)) + \mu(-, g_T(v)), \quad D(\Theta, T(v)) = \sum_{q=1}^{\deg(v)} D(\Theta, T[v_q])$$

2. $D(S[u], T[v]) =$

$$\max \begin{cases} D(S(u), T(v)) + \mu(g_S(u), g_T(v)) \\ D(S[u], \Theta) + \max\limits_{1 \le q \le \deg(u)} \{D(S[u_q], T[v]) - D(S[u_q], \Theta)\} \\ D(\Theta, T[v]) + \max\limits_{1 \le q \le \deg(v)} \{D(S[u], T[v_q]) - D(\Theta, T[v_q])\} \end{cases}$$

3. *For any* j *and* l *such that* $1 \le j \le \deg(u)$ *and* $1 \le l \le \deg(v)$,
$D(S(u, 1, j), T(v, 1, l)) =$

$$\max \begin{cases} D(S(u, 1, j-1), T(v, 1, l)) + D(S[u_j], \Theta) \\ D(S(u, 1, j), T(v, 1, l-1)) + D(\Theta, T[v_l]) \\ D(S(u, 1, j-1), T(v, 1, l-1)) + D(S[u_j], T[v_l]) \\ \mu(g_S(u_j), -) + \max\limits_{1 \le q < \deg(l)} \{D(S(u, 1, j-1), T(v, 1, q-1)) + \\ \qquad D(S(u_j), T(v, q, l))\} \\ \mu(-, g_T(v_l)) + \max\limits_{1 \le q < \deg(j)} \{D(S(u, 1, q-1), T(v, 1, l-1)) + \\ \qquad D(S(u, q, j), T(v_l))\} \end{cases}$$

Proof: See [70]. □

The algorithm of Jiang *et al.* (Algorithm *Score*) is displayed in Figure 5.3. As the various values of $D(S_*, T_*)$ are computed using the recurrences in Lemma 5.3, they are stored in a data structure which allows them to be retrieved in $O(1)$ time from then on.

Algorithm *Score* employs an auxiliary procedure called Procedure 1 (not shown here) that takes as input two subforests of the form $S(u, i, \deg(u))$ and $T(v, k, \deg(v))$, where at least one of i and k is equal to 1, and then computes $D(S(u, i, j), T(v, k, l))$ for all j and l such that $i \le j \le \deg(u)$ and $k \le l \le \deg(v)$ by repeatedly applying Lemma 5.3.3. Note that for every pair of subtrees $S[u]$ and $T[v]$, although the algorithm computes $D(S(u, i, j), T(v))$ for all $1 \le i \le$

$j \leq \deg(u)$ and $D(S(u), T(v,k,l))$ for all $1 \leq k \leq l \leq \deg(v)$, it does *not* need to compute the values of $D(S(u,i,j), T(v,k,l))$ for all $1 \leq i \leq j \leq \deg(u)$ and $1 \leq k \leq l \leq \deg(v)$.

Each call to Procedure 1 is proved in [70] to take $O((\deg(u) + \deg(v)) \cdot \deg(u) \cdot \deg(v))$ time, and the total running time of the algorithm is shown to be $O(|S| \cdot |T| \cdot \Delta^2)$.

```
Algorithm   Score
Input:   Two labeled ordered trees $S$ and $T$.
Output: The score of an optimal alignment between $S$ and $T$.

  $D(\Theta, \Theta) := 0$
  for $u := 1$ to $|S|$ do
      Initialize $D(S[u], \Theta)$ and $D(S(u), \Theta)$ according to Lemma 5.3.1.
  endfor
  for $v := 1$ to $|T|$ do
      Initialize $D(\Theta, T[v])$ and $D(\Theta, T(v))$ according to Lemma 5.3.1.
  endfor
  for $u := 1$ to $|S|$ do
      for $v := 1$ to $|T|$ do
          for $i := 1$ to $\deg(u)$ do
              Call Procedure 1 on $S(u, i, \deg(u))$ and $T(v)$.
          endfor
          for $k := 1$ to $\deg(v)$ do
              Call Procedure 1 on $S(u)$ and $T(v, k, \deg(v))$.
          endfor
          Compute $D(S[u], T[v])$ as in Lemma 5.3.2.
      endfor
  endfor
  return $D(S[|S|], T[|T|])$
End Score
```

Figure 5.3: The algorithm of Jiang, Wang, and Zhang.

By adding a traceback step at the end, the algorithm can be extended to return an alignment corresponding to the optimal score without increasing the asymptotic running time[7]. Hence, Jiang *et al.* proved the following result.

Theorem 5.4 *The alignment between ordered trees problem can be solved in $O(|S| \cdot |T| \cdot \Delta^2)$ time.*

[7]An optimal alignment can be recovered by recalculating the terms on the right-hand side of Lemma 5.3 for each pair of subtrees or subforests encountered during the traceback to determine which of the possibilities that resulted in the highest score; alternatively, one can modify the algorithm to also record information about how each value $D(S_*, T_*)$ is obtained as it is computed, e.g., by saving pointers.

5.3 d-Relevance

5.3.1 d-Relevant Pairs of Subtrees

Before defining the new concept of d-relevance, we need to introduce some notation.

Definition 5.5 Let S be a labeled ordered tree and u a node of S. $\overline{S[u]}$ stands for the ordered subtree of S obtained when removing from S the subtree $S[u]$ and (if u is not the root of S) the edge between u and the parent of u. Next, $L(S[u])$ denotes the set of leaves in S that are to the left of the leaves of $S[u]$.

Recall that the number of nodes in S is denoted by $|S|$. The cardinality of the set $L(S[u])$ is denoted by $|L(S[u])|$.

We are now ready to define what we mean by a d-relevant pair of subtrees, a d-descendant, and a d-ancestor.

Definition 5.6 Let d be a positive integer. For two ordered trees S and T containing two nodes u and v respectively, the pair of subtrees $(S[u], T[v])$ is called *d-relevant* if and only if both of the following conditions hold:

- $||S[u]| - |T[v]|| \leq d$
- $||L(S[u])| - |L(T[v])|| \leq d$

Definition 5.7 Let d be a positive integer, and let T be an ordered tree containing two nodes v and w. $T[w]$ is called a *d-descendant* of $T[v]$ if w is a descendant of v and $|T[v]| - |T[w]| \leq d$. Symmetrically, $T[w]$ is called a *d-ancestor* of $T[v]$ if w is an ancestor of v and $|T[w]| - |T[v]| \leq d$.

Definitions 5.6 and 5.7 are illustrated in Figure 5.4.

The following important lemma implies that if a pair of subtrees $(S[u], T[v])$ is not d-relevant then any alignment between S and T which consists of an alignment between $S[u]$ and $T[v]$ and an alignment between $\overline{S[u]}$ and $\overline{T[v]}$ must contain more than d blank symbols. Thus, if only alignments with at most d blank symbols for some specified value of d are of interest (as in the case of Algorithm *Fast Score* given in Section 5.6), we can limit our attention to d-relevant pairs.

Lemma 5.8 *Let S and T be labeled ordered trees, let u and v be two nodes belonging to S and T respectively, and let A be an alignment between S and T consisting of an alignment between $S[u]$ and $T[v]$ and an alignment between $\overline{S[u]}$ and $\overline{T[v]}$. If A uses at most d blank symbols then $(S[u], T[v])$ is $\left(\frac{d+||S|-|T||}{2}\right)$-relevant for S and T.*

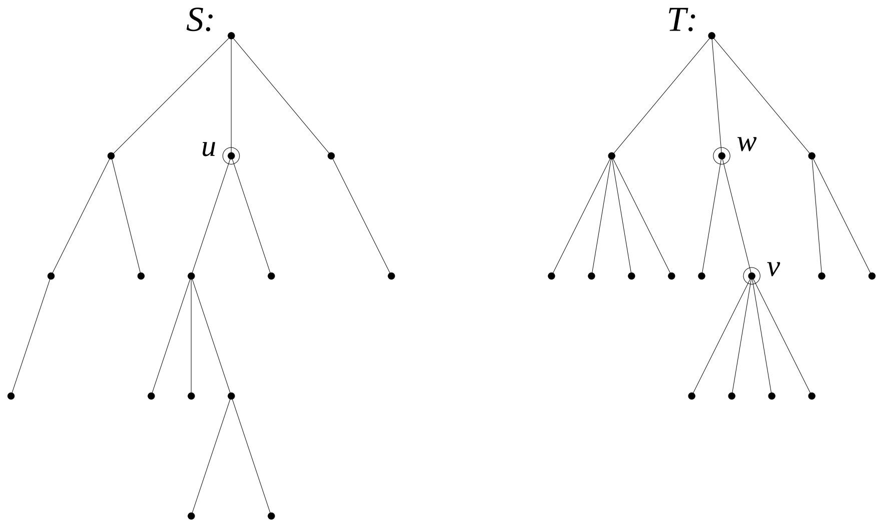

Figure 5.4: In this example, $(S[u], T[v])$ is 3-relevant, $(S[u], T[w])$ is 2-relevant, and $T[w]$ is a 2-ancestor of $T[v]$.

Proof: Let d_1 be the total number of insert operations performed on S and d_2 the total number of insert operations performed on T to obtain A. Denote by d'_1 and d'_2 the number of insert operations performed on $S[u]$ and $T[v]$ respectively to construct the alignment between $S[u]$ and $T[v]$. Clearly, $d'_1 \leq d_1$ and $d'_2 \leq d_2$.

First, consider the alignment between $S[u]$ and $T[v]$. By the above, d'_1 insert operations on $S[u]$ suffice to obtain a tree isomorphic to $T[v]$ with d'_2 nodes inserted, so $|S[u]| + d'_1 = |T[v]| + d'_2$, i.e., $|S[u]| - |T[v]| = d'_2 - d'_1 \leq d'_2 \leq d_2 \leq \max\{d_1, d_2\}$. In the same way, $|T[v]| - |S[u]| \leq d_1 \leq \max\{d_1, d_2\}$. Thus, we have $||S[u]| - |T[v]|| \leq \max\{d_1, d_2\}$.

Next, because $\overline{S[u]}$ and $\overline{T[v]}$ are aligned, $|L(S[u])| + d''_1 = |L(T[v])| + d''_2$ for some $d''_1 \leq d_1$ and $d''_2 \leq d_2$. This yields $|L(S[u])| - |L(T[v])| = d''_2 - d''_1 \leq d''_2 \leq d_2 \leq \max\{d_1, d_2\}$ and $|L(T[v])| - |L(S[u])| \leq \max\{d_1, d_2\}$. Thus, $||L(S[u])| - |L(T[v])|| \leq \max\{d_1, d_2\}$.

Now, since $d_1 + d_2 \leq d$ and $|S| + d_1 = |T| + d_2$, it follows that $d_1 \leq \frac{d+(|T|-|S|)}{2}$ and $d_2 \leq \frac{d+(|S|-|T|)}{2}$. Hence, $\max\{d_1, d_2\} \leq \frac{d+||S|-|T||}{2}$. □

For ease of presentation, we will use the somewhat weaker result implied by Lemma 5.8 that if A uses at most d blank symbols then $(S[u], T[v])$ is d-relevant for S and T. (Note that here $\frac{d+||S|-|T||}{2} \leq d$.)

The next three lemmas are used to derive an upper bound on the number of d-relevant pairs.

Lemma 5.9 *For any positive integer d, if the two pairs of subtrees $(S[u], T[v])$ and $(S[u], T[w])$ are d-relevant for two ordered trees S and T, and w is an ancestor (or, descendant) of v in T, then $T[w]$ is a $2d$-ancestor (or, $2d$-descendant) of $T[v]$.*

Proof: Let w be an ancestor of v in T. Since $(S[u], T[v])$ and $(S[u], T[w])$ are d-relevant, we have $||S[u]| - |T[v]|| \leq d$ and $||S[u]| - |T[w]|| \leq d$, and hence $|S[u]| - |T[v]| \leq d$ and $|T[w]| - |S[u]| \leq d$. Then it follows that $|T[w]| - |T[v]| = (|T[w]| - |S[u]|) + (|S[u]| - |T[v]|) \leq d + d = 2d$.

The proof for the case where w is a descendant of v is analogous. □

Lemma 5.10 *Let d be a positive integer. For any node u of an ordered tree S, the number of d-ancestors of $S[u]$ is at most d.*

Proof: Assume that the number of d-ancestors of $S[u]$ is greater than d. Then there exists a d-ancestor $S[u']$ whose root u' is located at distance greater than d from u. But this would imply $|S[u']| - |S[u]| > d$, which is a contradiction. □

Lemma 5.11 *Let d be a positive integer and let $\{(S[u], T[v_i])\}_{i=0}^{l}$ be a sequence of distinct d-relevant pairs in two ordered trees S and T such that v_i is not a descendant of v_j for any $0 \leq i, j \leq l$. Then $l \leq 2d$.*

Proof: We may assume without loss of generality that the sequence of nodes $\{v_i\}_{i=0}^{l}$ is ordered in accordance with the left-to-right ordering in T. Since $(S[u], T[v_l])$ is d-relevant, it holds that $||L(S[u])| - |L(T[v_l])|| \leq d$, giving us $|L(T[v_l])| - |L(S[u])| \leq d$ and thus $-|L(T[v_l])| + |L(S[u])| \geq -d$. On the other hand, $|L(T[v_l])| - |L(T[v_0])| \geq l$. Hence, if $l > 2d$ then $|L(S[u])| - |L(T[v_0])| = (|L(S[u])| - |L(T[v_l])|) + (|L(T[v_l])| - |L(T[v_0])|) > (-d) + 2d = d$, which contradicts the d-relevance of $(S[u], T[v_0])$. □

By combining Lemmas 5.9–5.11, we obtain an upper bound on the number of d-relevant pairs of subtrees.

Theorem 5.12 *Let d be a positive integer. For any two ordered trees S and T and a node u of S, the number of distinct d-relevant pairs of subtrees in which u participates is $O(d^2)$.*

Proof: Let $\{(S[u], T[v_i])\}_{i=0}^{l}$ be a maximal sequence of distinct d-relevant pairs of subtrees for two ordered trees S and T such that for each $0 \leq i \leq l$ there is no d-relevant pair $(S[u], T[v])$, where v is a descendant of v_i. It follows from Lemma 5.9 that for each d-relevant pair $(S[u], T[w])$, it either belongs to the sequence or $T[w]$ is a $2d$-ancestor of a member in the sequence. Hence, the number of d-relevant pairs in which u participates is at most $(2d + 1) \cdot (l + 1)$ by Lemma 5.10. Finally, it is sufficient to observe that l cannot exceed $2d$ by Lemma 5.11. □

Corollary 5.13 *For any positive integer d and two ordered trees S and T, there are $O(m \cdot d^2)$ d-relevant pairs of subtrees for S and T, where $m = \min\{|S|, |T|\}$.*

5.3.2 d-Relevant Pairs of Subforests

The algorithm of Jiang *et al.* computes scores not only between pairs of subtrees of the input trees, but also between certain pairs of subforests. Therefore, we need to generalize the concepts of d-relevance, d-descendants, and d-ancestors for pairs of nodes inducing full subtrees to include pairs of subforests of the form $(S(u,i,j), T(v,k,l))$.

Definition 5.14 Let $S(u,i,j)$ be an ordered subforest in a labeled ordered tree S. $\overline{S(u,i,j)}$ stands for the ordered subtree of S obtained when removing from S the forest $S(u,i,j)$ and all edges incident to $S(u,i,j)$. $L(S(u,i,j))$ denotes the set of leaves in S that are to the left of the leaves of $S(u,i,j)$.

The number of nodes in $S(u,i,j)$ is denoted by $|S(u,i,j)|$ and the cardinality of $L(S(u,i,j))$ by $|L(S(u,i,j))|$.

Definition 5.15 Let d be a positive integer. For two ordered trees S and T containing nodes u and v respectively, the pair of ordered subforests $(S(u,i,j)$, $T(v,k,l))$ is called *d-relevant* if and only if both of the following conditions hold:

- $||S(u,i,j)| - |T(v,k,l)|| \leq d$
- $||L(S(u,i,j))| - |L(T(v,k,l))|| \leq d$

Definition 5.16 Let d be a positive integer, and let T be an ordered tree containing two nodes v and w. $T(w,k',l')$ is called a *d-descendant* of $T(v,k,l)$ if w is a descendant of v, $T(w,k',l')$ is contained in $T(v,k,l)$, and $|T(v,k,l)| - |T(w,k',l')| \leq d$. Symmetrically, $T(w,k',l')$ is called a *d-ancestor* of $T(v,k,l)$ if w is an ancestor of v, $T(v,k,l)$ is contained in $T(w,k',l')$, and $|T(w,k',l')| - |T(v,k,l)| \leq d$.

The definition of d-relevance for pairs of subforests yields the next lemma, analogous to Lemma 5.8.

Lemma 5.17 *Let S and T be labeled ordered trees, let $S(u,i,j)$ and $T(v,k,l)$ be two ordered subforests in S and T respectively, and let A be an alignment between S and T consisting of an alignment between $S(u,i,j)$ and $T(v,k,l)$ and an alignment between $\overline{S(u,i,j)}$ and $\overline{T(v,k,l)}$. If A uses at most d blank symbols then $(S(u,i,j), T(v,k,l))$ is $\left(\frac{d+||S|-|T||}{2}\right)$-relevant for S and T.*

The proofs of the next three lemmas are analogous to the corresponding proofs of Lemmas 5.9–5.11.

Lemma 5.18 *For any positive integer d, if the two pairs of subforests $(S(u,i,j)$, $T(v))$ and $(S(u,i,j), T(w))$ are d-relevant for two ordered trees S and T, and w is an ancestor (or, descendant) of v in T, then $T(w)$ is a $2d$-ancestor (or, $2d$-descendant) of $T(v)$.*

Lemma 5.19 *Let d be a positive integer. For any node u of an ordered tree S, the number of d-ancestors of the form $S(w)$ of the forest $S(u)$ is at most d.*

Lemma 5.20 *Let d be a positive integer and let $\{(S(u,i,j), T(v_q)\}_{q=0}^{l}$ be a sequence of distinct d-relevant pairs in two ordered trees S and T such that $v_{q'}$ is not a descendant of $v_{q''}$ for any $0 \le q', q'' \le l$. Then $l \le 2d$.*

By combining Lemmas 5.18–5.20, we obtain an upper bound on the number of d-relevant pairs $(S(u), T(v,k,l))$ and $(S(u,i,j), T(v))$ like in Theorem 5.12 and Corollary 5.13.

Theorem 5.21 *Let d be a positive integer. For any two ordered trees S and T and a node u of S, the number of distinct d-relevant pairs of the form $(S(u,i,j), T(v))$ is $O(d^2 \cdot (\deg(S))^2)$. Symmetrically, for any node v of T, the number of distinct d-relevant pairs of the form $(S(u), T(v,k,l))$ is $O(d^2 \cdot (\deg(T))^2)$.*

Corollary 5.22 *For any positive integer d and two ordered trees S and T, there are $O(n \cdot d^2 \cdot \Delta^2)$ d-relevant pairs of subforests of the form $(S(u), T(v,k,l))$ and $(S(u,i,j), T(v))$ for S and T, where $n = \max\{|S|, |T|\}$ and $\Delta = \max\{\deg(S), \deg(T)\}$.*

5.4 Testing for d-Relevance

In this section, we show how to preprocess S and T in linear time so that any pair of subtrees or subforests of S and T can be tested for d-relevance in constant time.

First, compute $|S[u]|$ and $|L(S[u])|$ for all $u \in S$. Figure 5.5 demonstrates how this can be done recursively in $O(|S|)$ time by using the Euler tour technique [125]. The algorithm is started by calling *Euler Tour* $(root, 0)$. As the values of $|S[u]|$ and $|L(S[u])|$ for various nodes u are computed, store them in a tree $\hat{S}$ which is isomorphic to S and equipped with auxiliary data fields.

Next, augment each node u of $\hat{S}$ with an integer array s of size $\deg(u) + 1$ for storing the cumulative sums (from left to right) of the sizes of the subtrees rooted at the children of u. To assign values to the entries of s, set $s[0] := 0$ and for $q := 1$ to $\deg(u)$ let $s[q] := s[q-1] + |S[u_q]|$ so that $s[i] = \sum_{q=1}^{i} |S[u_q]|$ for any $i \in \{1, ..., \deg(u)\}$. The total time needed to fill in the s-arrays for all nodes in S is $\sum_{u \in S} O(\deg(u)) = O(|S|)$.

Then, compute $|T[v]|$ and $|L(T[v])|$ for all $v \in T$ in $O(|T|)$ time in the same way as for S and store them in a tree $\hat{T}$. Augment each node of $\hat{T}$ with an integer array t defined analogously as the s-arrays and assign values to them in $O(|T|)$ time.

```
Algorithm  Euler Tour
Input:  Node $u$, integer $left$.
Output: Integer $sumNodes$, integer $sumLeaves$.

  $|L(S[u])| := left$
  $sumNodes := 1$
  if $u$ is a leaf then
    $sumLeaves := 1$
  else
    $sumLeaves := 0$
    for all children $w$ of $u$ in left-to-right order do
      $sN, sL := Euler\ Tour\,(w,\ left + sumLeaves)$
      $sumLeaves := sumLeaves + sL$
      $sumNodes := sumNodes + sN$
    endfor
  endif
  $|S[u]| := sumNodes$
  return $sumNodes, sumLeaves$
End Euler Tour
```

Figure 5.5: The Euler tour algorithm for computing $|S[u]|$ and $|L(S[u]|)$ for all $u \in S$.

After having constructed $\hat{S}$ and $\hat{T}$ as above, all the information required to determine whether a given pair of subtrees or subforests is d-relevant is immediately available. To test a pair $(S[u], T[v])$ or $(S(u,i,j), T(v,k,l))$ for d-relevance in constant time, we simply check if the conditions in Definition 5.6 or Definition 5.15 are satisfied by using the values of $|S[u]|$, $|T[v]|$, $|L(S[u])|$, and $|L(T[v])|$, or $|S(u,i,j)|$, $|T(v,k,l)|$, $|L(S(u,i,j))|$, and $|L(T(v,k,l))|$:

- For a subtree $S[u]$, the values of $|S[u]|$ and $|L(S[u])$ can be obtained directly from $\hat{S}$.
- For a subforest $S(u,i,j)$, $|L(S(u,i,j))|$ equals $|L(S[u_i])|$ and we can obtain $|S(u,i,j)|$ from node u's s-array in $\hat{S}$ by using the formula $s[j] - s[i-1]$ since for any $i, j \in \{1, ..., \deg(u)\}$ with $i < j$, we have

$$s[j] - s[i-1] \;=\; \sum_{q=1}^{j} |S[u_q]| - \sum_{q=1}^{i-1} |S[u_q]| \;=\; \sum_{q=i}^{j} |S[u_q]| \;=\; |S(u,i,j)|$$

We summarize the above in the next theorem.

Theorem 5.23 *Let S and T be two ordered trees. After $O(|S| + |T|)$ time preprocessing, any given pair of subtrees $(S[u], T[v])$ or pair of subforests $(S(u,i,j), T(v,k,l))$ can be tested for d-relevance in $O(1)$ time.*

5.5 Enumerating the d-Relevant Pairs of Subtrees

In order to improve on the quadratic running time of the algorithm of Jiang *et al.*, we need an efficient method to enumerate all d-relevant pairs of subtrees. We cannot afford to test all of the $O(|S| \cdot |T|)$ possible pairs for d-relevance individually; instead, we proceed as follows.

First, we compute $\hat{S}$ and $\hat{T}$ in $O(|S| + |T|)$ time as described in Section 5.4 so that the values of $|S[u]|$ and $|L(S[u])|$ for any $u \in S$ and the values of $|T[v]|$ and $|L(T[v])|$ for any $v \in T$ are accessible in $O(1)$ time. We then traverse $\hat{T}$. At each node v, we fetch the values of $|T[v]|$ and $|L(T[v])|$, and insert the point $(|T[v]|, |L(T[v])|)$ into a standard data structure for two-dimensional range search, e.g., a layered range tree [33, 107]. The construction of the data structure takes $O(|T| \cdot \log |T|)$ time. Then, for each u in S, we query the range search data structure with the square centered at $(|S[u]|, |L(S[u])|)$ having side length $2d$ (note that a point (x, y) lies inside this square if and only if $||S[u]| - x| \leq d$ and $||L(S[u])| - y| \leq d$). Each such query takes $O(\log |T| + r)$ time, where r is the number of reported points. Since each of the reported points is in one-to-one correspondence with a node v such that the pair $(S[u], T[v])$ is d-relevant, $r = O(d^2)$ holds by Theorem 5.12.

Next, we build and lexicographically sort the list of all $O(|S| \cdot d^2)$ d-relevant pairs of subtrees in $O(|S| \cdot d^2 \cdot \log |S|)$ time.

Putting everything together, we obtain the next theorem.

Theorem 5.24 *Given two ordered trees with at most n nodes each and a positive integer d, a lexicographically sorted list of all d-relevant pairs of subtrees can be constructed in $O(n \log n \cdot d^2)$ time.*

5.6 Algorithm *Fast Score*

Our Algorithm *Fast Score* for computing the score of an optimal alignment between two labeled, ordered trees S and T is displayed in Figure 5.6. It works under the assumption that there exists an optimal alignment which uses at most d blank symbols, for some specified positive integer d.

First, Algorithm *Fast Score* constructs $\hat{S}$ and $\hat{T}$ and a list of all d-relevant pairs of subtrees of S and T. According to Theorem 5.23 and Theorem 5.24, this preprocessing takes $O(n \log n \cdot d^2)$ time. The scores for all pairs containing an empty subtree or subforest are also precomputed, which takes $O(|S| + |T|) = O(n)$ time.

We then modify the algorithm of Jiang *et al.* to only evaluate scores for d-relevant pairs of subtrees and d-relevant pairs of subforests. (By Lemmas 5.8 and 5.17, the other pairs correspond to alignments using more than d blank symbols and can therefore be ignored.) Whenever one of the formulas in Lemma 5.3.2

```
Algorithm   Fast Score
Input:   Two labeled ordered trees $S$ and $T$, positive integer $d$.
Output:  The score of an optimal alignment between $S$ and $T$ (assuming there
         exists an optimal alignment with at most $d$ blank symbols).

  Construct $\hat{S}$ and $\hat{T}$ as described in Section 5.4 and construct a lexicographically
  sorted list $L$ of all $d$-relevant pairs of subtrees as described in Section 5.5.

  $D(\Theta, \Theta) := 0$
  for $u := 1$ to $|S|$ do
      Initialize $D(S[u], \Theta)$ and $D(S(u), \Theta)$ according to Lemma 5.3.1.
  endfor
  for $v := 1$ to $|T|$ do
      Initialize $D(\Theta, T[v])$ and $D(\Theta, T(v))$ according to Lemma 5.3.1.
  endfor
  for all $d$-relevant pairs of subtrees $(S[u], T[v])$, determined by traversing $L$,
  do
      for $i := 1$ to $\deg(u)$ do
          if $(S(u, i, \deg(u)), T(v))$ is $d$-relevant then
              Call Procedure $1'$ on $S(u, i, \deg(u))$ and $T(v)$.
          endif
      endfor
      for $k := 1$ to $\deg(v)$ do
          if $(S(u), T(v, k, \deg(v)))$ is $d$-relevant then
              Call Procedure $1'$ on $S(u)$ and $T(v, k, \deg(v))$.
          endif
      endfor

      Compute $D(S[u], T[v])$ as in Lemma 5.3.2, only considering $d$-relevant
      pairs on the right-hand side of the expression.

  endfor
  return $D(S[|S|], T[|T|])$
End Fast Score
```

Figure 5.6: The fast algorithm for computing the score of an optimal alignment between two ordered trees which uses at most d blank symbols.

or Lemma 5.3.3 is to be applied, we test each of the components on the right-hand side for d-relevance. If the test is positive, we fetch the score for that pair (by the bottom-up ordering, it has been evaluated by this time); otherwise, we set the score to minus infinity. Procedure $1'$ referred to in Figure 5.6 is the same as Procedure 1 with such tests for d-relevance included. Now, any given pair of subtrees or subforests can be tested for d-relevance in $O(1)$ time by using $\hat{S}$ and $\hat{T}$ as explained in Section 5.4. We conclude that the cost of determining the score of an optimal alignment for a d-relevant pair on the left-hand side in

Lemma 5.3 by using the scores of optimal alignments for d-relevant pairs occuring on the right-hand side increases by at most a factor of $O(1)$. Hence, each call to Procedure $1'$ still takes $O((\deg(u)+\deg(v))\cdot\deg(u)\cdot\deg(v))$ time. In the following, we denote the running time of one call to Procedure $1'$ by $\mathrm{P1}'$.

For each d-relevant pair of subtrees $(S[u], T[v])$, the algorithm tests $\deg(u)$ and then $\deg(v)$ pairs of subforests for d-relevance and makes at most this many calls to Procedure $1'$. Next, it evaluates $D(S[u], T[v])$ by testing $\deg(u)+\deg(v)$ pairs of subtrees and one pair of subforests on the right-hand side of the relation in Lemma 5.3.2 for d-relevance. Thus, each d-relevant pair of subtrees contributes $O(\deg(u)\cdot(O(1)+\mathrm{P1}')+\deg(v)\cdot(O(1)+\mathrm{P1}')+(\deg(u)+\deg(v)+1)\cdot O(1)) = O((\deg(u)+\deg(v))^2\cdot\deg(u)\cdot\deg(v))$ to the total running time. Summing over all d-relevant pairs of subtrees, we see that the entire main loop takes

$$\begin{aligned}
&\sum_{\substack{d\text{-relevant pairs of}\\ \text{subtrees } (S[u],T[v])}} O((\deg(u)+\deg(v))^2\cdot\deg(u)\cdot\deg(v))\\
&= O(\Delta^3 \cdot \sum_{\substack{d\text{-relevant pairs of}\\ \text{subtrees } (S[u],T[v])}} \deg(u))\\
&= O(\Delta^3 \cdot \sum_{u\in S} \sum_{\substack{v\in T \text{ and}\\ (S[u],T[v])\\ \text{is } d\text{-relevant}}} \deg(u))\\
&= O(\Delta^3 \cdot \sum_{u\in S} d^2\cdot\deg(u))\\
&= O(\Delta^3\cdot d^2\cdot n)
\end{aligned}$$

time by using Theorem 5.12 and the fact that $\sum\limits_{u\in S}\deg(u) = n$.

Including the preprocessing, the total running time is $O(n\log n\cdot d^2 + n + \Delta^3\cdot d^2\cdot n) = O(n\cdot(\log n+\Delta^3)\cdot d^2)$, which gives us the main theorem of this section.

Theorem 5.25 *If there exists an optimal alignment between S and T which uses at most d blank symbols and d is given, we can compute its score in $O(n\cdot(\log n+\Delta^3)\cdot d^2)$ time.*

We remark that Algorithm *Fast Score* can be modified to return an optimal alignment without increasing the asymptotic running time by adding a traceback step just like for the algorithm of Jiang *et al.* (see Section 5.2). Thus, we can solve the alignment between ordered trees problem in $O(n\cdot(\log n+\Delta^3)\cdot d^2)$ time if there exists an optimal alignment between S and T which uses at most d blank symbols and d is known in advance.

Also note that if $\Delta = O(1)$ then the running time of Algorithm *Fast Score* becomes $O(n\log n\cdot d^2)$.

5.7 Algorithm *Unspecified* d

Here, we extend our Algorithm *Fast Score* from Section 5.6 to compute the score of an optimal alignment between the two input trees even if no upper bound on the number of blank symbols in an optimal alignment is given. We show that under some natural assumptions on the scoring scheme, the resulting method is faster than the algorithm of Jiang *et al.* for problem instances consisting of similar trees (i.e., instances in which the score of an optimal alignment is high). The technique we employ stems from Section 3.3.4 in [111], where it is applied to compute the score of an optimal alignment between two strings of equal length by using an algorithm which only evaluates a band of specified width around the main diagonal of the dynamic programming matrix.

As before, write $m = \min\{|S|, |T|\}$ and $n = \max\{|S|, |T|\}$. The algorithm of Jiang *et al.* runs in $O(m \cdot n \cdot \Delta^2)$ time, regardless of the number of insertions required by an optimal solution (see Section 5.2). On the other hand, by Theorem 5.25, Algorithm *Fast Score* runs in $O(n \cdot (\log n + \Delta^3) \cdot d^2)$ time, where d is the maximum number of insertions allowed. Thus, Algorithm *Fast Score* is asymptotically faster than the algorithm of Jiang *et al.* if d is small[8]. The drawback is that Algorithm *Fast Score* needs a value of d to be specified beforehand; the running time may be much worse than that of the algorithm of Jiang *et al.* if no sufficiently strong upper bound on d is known[9]. One way to overcome this difficulty is by running Algorithm *Fast Score* with successively larger values of d until a certain stop condition is satisfied, as explained below.

Let M be maximum value of $\mu(s,t)$ over all pairs of symbols (s,t) belonging to $\Sigma \times \Sigma$, and let B be maximum value of $\mu(s,t)$ over all pairs (s,t) in $(\Sigma \times \{-\}) \cup (\{-\} \times \Sigma)$, i.e., all pairs where precisely one of s and t is equal to the blank symbol. Assume that $M > 0$ and $B \leq 0$.

Lemma 5.26 *For any positive integer d, if an alignment between S and T uses at least $d+1$ blank symbols then its score is at most* $(d+1) \cdot B + \frac{m+n-(d+1)}{2} \cdot M$.

Proof: Let A be an alignment between S and T with at least $d+1$ blank symbols. Then the total number of nodes in S and T which can be paired off with each other is at most $|S| + |T| - (d+1)$. The maximum possible score of A is achieved when all such pairs of nodes have score M; thus, the score of A is at most $(d+1) \cdot B + \frac{|S|+|T|-(d+1)}{2} \cdot M$. □

For any positive integer d, let D_d be the value returned by Algorithm *Fast Score* on input (S, T, d). As d increases, D_d increases or remains the same while

[8]More precisely, if $d = o\left(\sqrt{\frac{m \cdot \Delta^2}{\log n + \Delta^3}}\right)$.

[9]For example, just plugging in the trivial upper bound $d = |S| + |T| \leq 2n$ does not help here.

the value of $(d+1)\cdot B + \frac{m+n-(d+1)}{2}\cdot M$ decreases because $B \leq 0$ and $M > 0$. Thus, by gradually increasing d, D_d eventually becomes larger than or equal to $(d+1)\cdot B + \frac{m+n-(d+1)}{2}\cdot M$. This yields a useful stop condition because when it occurs, Lemma 5.26 ensures that all alignments containing more blank symbols than the current value of d will have scores which are lower than or equal to D_d and therefore do not need to be considered.

The algorithm is called Algorithm *Unspecified d* and is listed in Figure 5.7. Initially, it sets d to $(n-m)+1$ since all alignments between S and T use at least $n-m$ blank symbols. It then finds the score of an optimal alignment by doubling d until the stop condition is satisfied.

```
Algorithm   Unspecified d
Input:   Two labeled ordered trees $S$ and $T$.
Output:  The score of an optimal alignment between $S$ and $T$.

   $d := n - m + 1$
   $D_d :=$ Fast Score$(S, T, d)$
   while $D_d < (d+1)\cdot B + \frac{m+n-(d+1)}{2}\cdot M$ do
         $d := d \cdot 2$
         $D_d :=$ Fast Score$(S, T, d)$
   endwhile
   return $D_d$
End  Unspecified d
```

Figure 5.7: An algorithm for computing the score of an optimal alignment between two ordered trees when no upper bound on the number of blank symbols is provided.

We now analyze the running time of Algorithm *Unspecified d*. Denote the algorithm's final value of d by $\tilde{d}$. The first call to Algorithm *Fast Score* takes $O(n\cdot(\log n+\Delta^3)\cdot(n-m+1)^2)$ time, the second one $O(n\cdot(\log n+\Delta^3)\cdot(2(n-m+1))^2)$ time, etc., and the last one $O(n\cdot(\log n+\Delta^3)\cdot\tilde{d}^2)$ time. Since

$$x^2 + (2x)^2 + (4x)^2 + (8x)^2 + \ldots + \tilde{d}^2 \;=\; x^2 \cdot \sum_{i=0}^{\log_2(\frac{\tilde{d}}{x})} (2^i)^2 \;=\; \frac{4\tilde{d}^2 - x^2}{3},$$

the running time is $O(n\cdot(\log n+\Delta^3)\cdot(\tilde{d}^2-(n-m+1)^2))$.

We then proceed as in [111] to obtain a nontrivial upper bound on $\tilde{d}$ in terms of m, n, M, B, and s, where s is the score of an optimal alignment between S and T. When the algorithm stops, there are two possibilities:

- If $D_{\tilde{d}} = D_{\tilde{d}/2}$ then $s = D_{\tilde{d}/2}$. The inequality $D_{\tilde{d}/2} < (\frac{\tilde{d}}{2}+1)\cdot B + \frac{m+n-(\frac{\tilde{d}}{2}+1)}{2}\cdot M$ (due to the algorithm not finishing in the previous iteration) then implies that $\tilde{d} < \frac{2(m+n)M-4s}{M-2B} - 2$.

- If $D_{\tilde{d}} > D_{\tilde{d}/2}$ then any optimal alignment contains $> \frac{\tilde{d}}{2}$ blank symbols so that by Lemma 5.26, $s \leq (\frac{\tilde{d}}{2}+1)\cdot B + \frac{m+n-(\frac{\tilde{d}}{2}+1)}{2}\cdot M$. Rearranging gives us $\tilde{d} \leq \frac{2(m+n)M-4s}{M-2B} - 2$.

Thus, in both cases we have the upper bound

$$\tilde{d} \leq 2\left(\frac{(m+n)M-2s}{M-2B} - 1\right) \tag{5.1}$$

The score of an optimal alignment between S and T is at most $m \cdot M$. Therefore, $s \leq \frac{m+n}{2}\cdot M$. By inequality (5.1), if the score of an optimal alignment between S and T is high (so that s is close to $\frac{m+n}{2}\cdot M$) then $\tilde{d}$ is small. Assuming that $M - 2B$ is a constant, we can express the running time of Algorithm *Unspecified d* as follows.

Theorem 5.27 *If $M-2B$ is a constant and $B \leq 0$, $M > 0$ then Algorithm* Unspecified d *computes the score of an optimal alignment between S and T in $O(n \cdot (\log n + \Delta^3) \cdot f^2)$ time, where $f = \frac{m+n}{2}\cdot M - s$ and s is the score of an optimal alignment between S and T.*

We also note the following:

Corollary 5.28 *If there exist constants α, β, and γ such that $\alpha > 0$, $\beta \leq 0$, $\gamma \leq \alpha$ and for every $x, y \in \Sigma$ with $x \neq y$ it holds that $\mu(x,x) = \alpha$, $\mu(x,-) = \mu(-,x) = \beta$, and $\mu(x,y) = \gamma$, and if there exists an optimal alignment between S and T with b blank symbols and $O(b)$ node pairs of the form (x,y) with $x \neq y$, then Algorithm* Unspecified d *runs in $O(n \cdot (\log n + \Delta^3) \cdot b^2)$ time.*

Proof: Write $s = b \cdot \beta + q \cdot \gamma + \frac{m+n-b-2q}{2} \cdot \alpha$, where q is the number of node pairs (x,y) with $x \neq y$ and $x, y \in \Sigma$. Combining this with inequality (5.1) yields $\tilde{d} \leq 2\left(\frac{b(\alpha-2\beta)+2q(\alpha-\gamma)}{\alpha-2\beta} - 1\right)$. Now, $q = O(b)$ implies that $\tilde{d} = O(b)$. □

In particular, if $\Delta = O(1)$ then the running times given in Theorem 5.27 and Corollary 5.28 reduce to $O(n \log n \cdot f^2)$ and $O(n \log n \cdot b^2)$, respectively.

Finally, as mentioned at the end of Section 5.6, it is possible to modify Algorithm *Fast Score* (and hence also Algorithm *Unspecified d*) to return an optimal alignment by performing a traceback with no increase in the asymptotic running time.

5.8 Concluding Remarks

We have introduced the concept of d-relevance in order to speed up the algorithm of Jiang *et al.* for instances of the alignment between ordered trees problem where the two input trees are similar.

The next table summarizes the running times of the algorithms described in this chapter; sufficient conditions for the respective asymptotic upper bounds on the running times to hold are also listed.

Algorithm	Condition	Running time	Reference
Jiang *et al.*	—	$O(m \cdot n \cdot \Delta^2)$	[70]; see also Section 5.2.
Fast Score	There exists an optimal alignment with $\leq d$ blank symbols and d is given.	$O(n \cdot (\log n + \Delta^3) \cdot d^2)$	Theorem 5.25
Unspecified d	$M - 2B$ is a constant and $B \leq 0$, $M > 0$.	$O(n \cdot (\log n + \Delta^3) \cdot f^2)$, where $f = \frac{m+n}{2} \cdot M - s$ and s is the score of an optimal alignment.	Theorem 5.27
Unspecified d	μ satisfies $\mu(x,x) = \alpha$, $\mu(x,-) = \beta$, $\mu(-,x) = \beta$, and $\mu(x,y) = \gamma$ for all $x, y \in \Sigma$ with $x \neq y$, where α, β, γ are constants such that $\alpha > 0$, $\beta \leq 0$, and $\gamma \leq \alpha$.	$O(n \cdot (\log n + \Delta^3) \cdot b^2)$, where b is the smallest number such that there exists an optimal alignment with b blank symbols and $O(b)$ node pairs of the form (x,y).	Corollary 5.28

We note that if the conditions in Theorem 5.27 are satisfied, then Algorithm *Unspecified d* is faster than the algorithm of Jiang *et al.* if, for example, $\Delta = o(\frac{m}{\log^3 n})$ and $\frac{m+n}{2} \cdot M = s + O(\log n)$. In certain other cases, the algorithm of Jiang *et al.* is faster. By running both algorithms in parallel, executing them one step at a time and alternating between them until one of them is finished, we can calculate the score of an optimal alignment between S and T in $\min\{O(m \cdot n \cdot \Delta^2),\ O(n \cdot (\log n + \Delta^3) \cdot f^2)\}$ time. Similarly, if the conditions in Corollary 5.28 are satisfied then this technique yields a running time of

$\min\{O(m \cdot n \cdot \Delta^2),\ O(n \cdot (\log n + \Delta^3) \cdot b^2)\}$.

We also note that the value of $n \cdot (\log n + \Delta^3) \cdot d^2$ is much larger than $m \cdot n \cdot \Delta^2$ when d is close to its upper bound $m + n$. Thus, we would like to know: Is it possible to lower the time complexity of Algorithm *Fast Score*, especially the exponent 2 of d?

It remains to demonstrate the practical usefulness of alignments between trees. We believe that some of the applications listed in Section 5.1.1 can benefit greatly from using alignments, something which should be investigated by implementing the algorithms described in this chapter, evaluating the quality and relevance of the solutions they produce, and comparing them to existing methods.

Our method does not seem immediately adaptable to unordered trees. Indeed, the proof of Lemma 5.11 relies on having a fixed left-to-right ordering on the nodes. It is an interesting open problem whether a substantial speedup in the construction of optimal alignments between two similar labeled, unordered trees whose degrees are bounded by a constant is achievable.

Another issue worth exploring is if there exist any nontrivial polynomial-time approximation algorithms for the alignment between unordered trees problem when at least one of the two input trees can have arbitrary degree. This question was posed by Jiang *et al.* in [70] (where the authors proved the problem to be MAX SNP-hard) but has not yet been answered.

Finally, a few comments on generalizations. Many of the extensions of the string alignment problem discussed in [58, 106, 111, 130] can be carried over directly to the alignment between trees problem. As an example, letting the input contain more than two trees results in the *multiple* alignment between trees problem[10]. Other extensions include computing optimal *local* alignments (for finding substructures of S and T with high similarity), using scoring schemes with *non-constant gap weights*, allowing *non-fixed alphabets* (e.g., where Σ is allowed to grow with the size of the input), and computing *suboptimal* alignments[11]. However, although the alignment between trees problem is easy to extend by examining famous variants of the string alignment problem, it is much harder to extend the algorithm of Jiang *et al.* with the various refinements of the standard dynamic programming algorithm for the string alignment problem

[10]A potential difficulty here is how to define the score of a multiple alignment between trees in a good way. The sum-of-pairs (SP) scoring function for multiple string alignments (defined as the sum of the scores of all induced pairwise string alignments) is a popular scoring scheme which can be generalized to alignments between trees in a straightforward manner and which may be practical because of its simplicity.

[11]As pointed out in [58], an "optimal" string alignment is only optimal with respect to a given objective function, and is not necessarily the most biologically relevant one. Therefore, it is sometimes useful to generate a candidate set of nearly optimal alignments which is then evaluated by some additional criteria.

which have been proposed. For example, we do not know if any divide-and-conquer technique similar to the one developed by Hirschberg for reducing the space complexity (see [58, 106, 111, 130]) can be applied to the algorithm of Jiang *et al.*

Part III

Clustering under the Hamming Metric

Chapter 6

Approximation Algorithms for the Hamming Center Problem

The Hamming center problem (HCP) for a set S of binary strings, each of length n, is to find a binary string of length n (not necessarily in S) that is close to every one of the strings in S, where distances between strings are measured using the Hamming metric. HCP is known to be NP-hard [46]. In this chapter, we describe some exact polynomial-time algorithms for special cases of HCP as well as some approximation algorithms for the general case.

6.1 Introduction

We start by introducing some notation and defining the problem studied in this chapter. Let $\{0,1\}^n$ be the set of all strings of length n over the alphabet $\{0,1\}$. For any $\alpha \in \{0,1\}^n$, we use the notation $\alpha[m]$ to refer to the symbol placed at the mth position of α, where $m \in \{1,...,n\}$, and we let $\alpha[i..j]$ represent the substring of α starting at position i and ending at position j, where $i, j \in \{1,...,n\}$ and $i < j$. The *Hamming distance* between two strings $\alpha_1, \alpha_2 \in \{0,1\}^n$ is the number of positions in which the strings differ, and is denoted by $d_H(\alpha_1, \alpha_2)$.

The Hamming center problem (HCP) is:

The Hamming center problem (HCP)

Instance: Finite set $S = \{\alpha_1, ..., \alpha_k\}$ with $S \subseteq \{0,1\}^n$ for some positive integer n.

Output: A string $\beta \in \{0,1\}^n$ which minimizes the value of $\max\limits_{\alpha_i \in S} d_H(\alpha_i, \beta)$.

HCP is referred to as *the minimum radius problem* in [46], *the closest string problem* in [57, 88, 91], and *the Hamming p-radius clustering problem* (HRC) with $p = 1$ in Chapter 7 of this thesis.

Given an instance S of HCP, the smallest possible value of $\max_{\alpha_i \in S} d_H(\alpha_i, \beta)$ over all strings $\beta \in \{0,1\}^n$ is called the *radius of* S, and is denoted by r. Any $\beta \in \{0,1\}^n$ which attains this optimal value is called a *center of* S, or a *1-center of* S. It follows from the definitions above that $k \leq 2^n$ and $r \leq n$ for any instance of HCP.

Observe that although some instances of HCP only have one 1-center (for example, if S consists of the n different strings of length n with exactly $n-1$ zeros and 1 one then the only optimal solution is the string of n zeros), some instances may have exponentially many alternative 1-centers (for example, if S consists of the two length n strings 000...0 and 111...1 where n is a positive even integer, there are $\binom{n}{n/2}$ 1-centers, which is exponential since $\binom{n}{n/2} > \frac{2^n}{n+1}$).

An algorithm $\mathcal{A}$ is said to approximate HCP within a factor of f if for any instance of the problem, $\mathcal{A}$ outputs a $\beta' \in \{0,1\}^n$ such that $\max_{\alpha_i \in S} d_H(\alpha_i, \beta') \leq f \cdot r$. In this case, $\mathcal{A}$ is also called a factor f approximation algorithm (or just an f-approximation algorithm) for HCP. From here on, optimal solutions to instances of HCP will be denoted by β, and approximate centers computed by our approximation algorithms by β'.

The Hamming distance function d_H satisfies the relations listed in Fact 6.1 below (see, e.g., p. 274 in [22]). Hence, d_H is a metric on $\{0,1\}^n$ and d_H is therefore also known as the *Hamming metric*.

Fact 6.1 [22] *Let n be any positive integer. Then, for all $x, y, z \in \{0,1\}^n$,*

- $d_H(x,y) \geq 0$, *with equality if and only if* $x = y$.
- $d_H(x,y) = d_H(y,x)$.
- $d_H(x,y) \leq d_H(x,z) + d_H(z,y)$ *("the triangle inequality").*

6.1.1 Motivation

A fundamental concept in coding theory is *the covering radius of a code* [27]. Given a set $S \subseteq \{0,1\}^n$, called the set of *code words*, the covering radius of S is defined as the smallest integer ϱ such that all strings in $\{0,1\}^n$ are within Hamming distance ϱ of some code word belonging to S. The covering radius is a basic geometric parameter of a code which measures its quality. For example, if a code is used for data compression, the covering radius is a measure of the maximum distortion [16, 27], and if the code is used for error correction, the covering radius gives the maximum weight of a correctable random error [27]. A related concept is the covering radius of a lattice in Euclidean space, which has applications to quantization and to coding for the Gaussian channel [27, 30, 31].

For any $S \in \{0,1\}^n$, it holds that $r + \varrho = n$ [27, 46, 80]. Hence, the decision problem version of the Hamming center problem[1] is computationally equivalent to the problem of deciding whether the covering radius of a given code is less than a given integer.

The Hamming center problem (and in particular, its generalization to larger constant-size alphabets than $\{0,1\}$) also has applications to computational molecular biology and data mining.

When classifying biomolecular sequences, consensus representatives are useful. For example, the around 100000 different proteins in humans can be divided into 1000 (or less) protein families, which makes it easier for researchers to understand their structures and biological functions [58]. A lot of information about a newly discovered protein may be deduced by establishing which family it belongs to. Here, it is more efficient to compare the sequence of the new protein (where the sequence of a protein is a string over the 20-letter amino acid alphabet [58, 106, 111, 130]) with representatives for various families than with individual family members. As another example, given a set S of k related sequences, one way to find other similar sequences is by computing a representative for S and then using the representative to probe a genome database. The representative should resemble[2] all sequences in the given set S, and must be chosen carefully. For instance, the sequence s that minimizes the sum of all pairwise distances between s and elements in S is biased towards similar sequences that occur frequently in S, so if the experiments used to obtain S lead to certain sequences being overrepresented then s will not reflect the true diversity of S. Using a 1-center as representative can help avoid this problem [13].

Another connection between the Hamming center problem and computational molecular biology is the following. In a problem termed *the phylogenetic alignment problem* or *the tree alignment problem* [10, 58, 111] (not to be confused with the problem studied in Chapter 5!), we are given an unrooted tree T distinctly leaf-labeled by a set L of strings, and the object is to determine a labeling of T's internal nodes so that the induced edge weights are small. (Next, one can derive a multiple alignment that is consistent with the fully labeled tree, and then remove all strings corresponding to internal nodes to obtain a good multiple alignment for L [58].) HCP is the special case of the phylogenetic

[1]The decision problem version of HCP is defined in the same way as HCP, except that it also takes as input a positive integer D, and the output is the answer to the question "Is the radius of S less than or equal to D?".

[2]Depending on the application, the difference between strings is sometimes measured in terms of edit distance, which also takes insertions and deletions into account, rather than Hamming distance, which just considers substitutions.

alignment problem in which T is a star graph with k leaves, each leaf is labeled by a string from S, the function used to determine the weight of an edge is the Hamming metric, and the optimization criterion is that the maximum weight of all edges in T should be minimized.

A classical problem in operations research and computational geometry is *the smallest enclosing circle problem*[3]: Given k points in the plane, find the smallest circle that encloses them. In other words, the objective is to find a point β which minimizes the maximum of all Euclidean distances between β and the given points. See Figure 6.1 for an example. This problem may arise when deciding where to build an emergency facility (e.g., a fire station or a hospital) so that the worst-case response time to some specified points on a map is minimized [107]. The Hamming center problem can be regarded as the analog of the smallest enclosing circle problem in which the number of dimensions is unrestricted, each coordinate of the input points and the returned solution is required to be 0 or 1, and distances are measured using the L_1 metric. (This also motivates the use of the terms *center* and *radius* for HCP.)

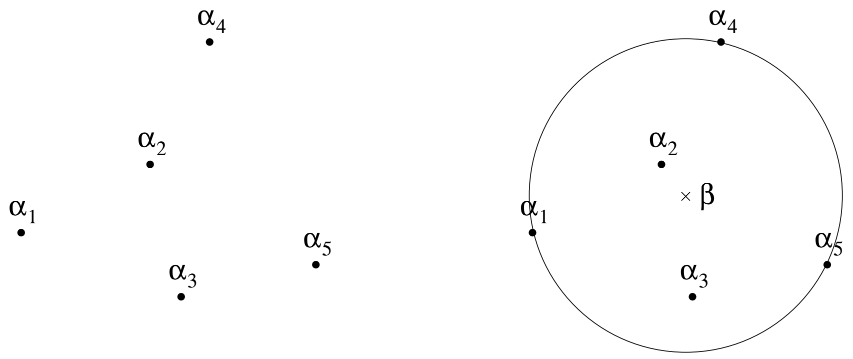

Figure 6.1: An instance of the smallest enclosing circle problem (left) and its optimal solution (right).

6.1.2 Previous Results

Frances and Litman [46] proved that the decision problem version of the Hamming center problem is NP-complete via a reduction from 3SAT.

As for polynomial-time approximations, there exists a trivial 2-approximation algorithm for HCP (described in Section 6.2) which is essentially a special case

[3]The smallest enclosing circle problem is sometimes called *the minimum spanning circle problem* or *the 2D-Euclidean center problem*.

of Gonzalez' farthest-point clustering algorithm [56] (see Section 7.5.1). Ben-Dor, Lancia, Perone, and Ravi [13] showed that randomized rounding can be used to obtain approximate solutions to HCP which are close to the optimal with high probability for instances where the radius is large compared to the input size (for instances where the radius is small, their method may yield poor approximations). Then, two groups of authors independently gave randomized $(\frac{4}{3} + \varepsilon)$-approximation algorithms for HCP, where ε can be selected to be any constant > 0; the one by Gąsieniec, Jansson, and Lingas [50] (to be described in this chapter) is guaranteed to run in polynomial time if r is at least superlogarithmic in k or if $r = O(1)$ (for other instances, this method still achieves an approximation factor of $\frac{4}{3} + \varepsilon$ with high probability but the running time may be exponential in the size of the input), whereas the algorithm by Lanctot, Li, Ma, Wang, and Zhang [88] runs in polynomial time for all r. This was followed by a polynomial-time approximation scheme (PTAS) for HCP by Li, Ma, and Wang [91].

Gramm, Niedermeier, and Rossmanith [57] studied the parameterized complexity of HCP and proved that if an upper bound R on r is provided then HCP can be solved in $O(kn + kR^{R+1})$ time. Hence, HCP can be solved in $O(kn)$ time if r is known in advance to be less than or equal to a given constant. In [57], Gramm *et al.* also showed that HCP restricted to instances with $k = 3$ can be solved in $O(n)$ time.

Below, we briefly describe some known results for other, related problems.

The closest substring problem is: Given a set $\{\alpha_1, ..., \alpha_k\}$ of binary strings of length n and a positive integer L with $L \leq n$, output a string $\beta \in \{0,1\}^L$ minimizing r such that for every input string α_i, there exists a length L substring γ_i of α_i with $d_H(\gamma_i, \beta) \leq r$. Li, Ma, and Wang [91] gave a PTAS for the closest substring problem based on their PTAS for HCP.

The R-mismatch problem is: Given a set of strings $\{\alpha_1, ..., \alpha_k\}$ of length n and positive integers L and R, (if possible) output a string β of length L and an integer m such that $d_H(\alpha_i[m..(m + L - 1)], \beta) \leq R$ for all $1 \leq i \leq k$. Gramm, Niedermeier, and Rossmanith [57] showed that the R-mismatch problem is solvable in $O(kL + (n - L)kR^{R+1})$ time which in linear in the size of the input if $R = O(1)$.

The distinguishing string selection problem is: Given a set G of "good" strings of length n, a set B of "bad" strings of length at least n, and two integers r_G, r_B, output a string β (if one exists) such that $\min_{\alpha_i \in G} d_H(\alpha_i, \beta) \geq r_G$ and such that for every $\alpha_j \in B$, there exists a length n substring γ_j of α_j with $d_H(\gamma_j, \beta) \leq r_B$. Deng, Li, Li, Ma, and Wang [34] gave a PTAS for the problem which, for any given constant $\varepsilon > 0$, finds a string β' of length n such that for every $\alpha_i \in G$, $d_H(\alpha_i, \beta') \geq (1 - \varepsilon) \cdot r_G$ and such that for every $\alpha_j \in B$, there exists a length n substring γ_j of α_j with $d_H(\gamma_j, \beta') \leq (1 + \varepsilon) \cdot r_B$. Gramm, Niedermeier, and Rossmanith [57] showed how to solve the special case where all strings in B have length n and $r_B = O(1)$ exactly in polynomial time.

The generalization of HCP to the problem of finding p centers is called *the Hamming p-radius clustering problem* (HRC). It is treated in depth in Chapter 7, where several new results are presented.

Finally, we comment on the smallest enclosing circle problem mentioned above. According to [93], it was posed by Sylvester [123] in 1857. After the first algorithm for the problem had been suggested, a great number of people tried to find increasingly efficient algorithms (see [93] for references) until Megiddo [93] finally settled the issue in 1982 by showing how to solve it in $O(k)$ time, matching the lower bound of $\Omega(k)$. Unfortunately, the methods used to solve the smallest enclosing circle problem, its generalizations to higher dimensions, and other variants listed in, e.g., the survey by Agarwal and Sharir [2] do not appear to work directly for HCP due to the discreteness of $\{0,1\}^n$. Therefore, the techniques developed for HCP are quite different.

6.1.3 Our Contributions

The main result of this chapter is a randomized $(\frac{4}{3}+\varepsilon)$-approximation algorithm for HCP with success probability at least $\frac{1}{2}$, where ε can be selected to be any constant greater than 0. Its running time is guaranteed to be polynomial if $r \geq \frac{12.7\ln(4k)}{\varepsilon^3}$ or if $r = O(1)$. It was originally published in [50], and although its performance has subsequently been surpassed by that of the PTAS of Li, Ma, and Wang [91] (which is deterministic and can approximate HCP within a factor of $1+\varepsilon$ for any constant $\varepsilon > 0$ in polynomial time), we include the original version of our algorithm here.

The rest of the chapter is organized as follows. First, we describe the trivial factor 2 approximation algorithm for HCP in Section 6.2. Then, in Section 6.3, we provide an integer programming formulation of HCP which can be used to obtain exact solutions in $n^{O(k)}$ time, which is polynomial in the input size if $k = O(1)$. We also note that if $n = O(\log k)$ or if $r = O(1)$ then exhaustive search finds exact solutions in polynomial time. Next, in Section 6.4, we analyze the method of randomized rounding applied to the linear programming relaxation of our integer programming formulation of HCP. We show that it yields approximate solutions which are close to the optimum with high probability if $r \gg \sqrt{4n\ln n}$ and $k \ll n^2$, or if the minimum generalized distance between the optimal solution of the relaxed version of the instance and a string in the instance is large. Finally, in Section 6.5, we present our randomized $(\frac{4}{3}+\varepsilon)$-approximation algorithm for HCP.

6.2 A Very Simple 2-Approximation Algorithm

Consider the following approximation algorithm:

> Given an instance $\{\alpha_1, ..., \alpha_k\}$ of HCP, set the approximate solution β' to α_l, where l is chosen arbitrarily from $\{1, ..., k\}$.

It can immediately be shown to yield a constant approximation factor:

Theorem 6.2 *The above algorithm approximates HCP within a factor of 2.*

Proof: Let β be an optimal solution to the given instance. For every i in $\{1, ..., k\}$, the inequality $d_H(\alpha_i, \beta') \leq d_H(\alpha_i, \beta) + d_H(\beta, \beta')$ holds due to Fact 6.1 and $d_H(\alpha_i, \beta) + d_H(\beta, \beta') = d_H(\alpha_i, \beta) + d_H(\beta, \alpha_l) \leq r + r = 2r$ holds by the definition of r. □

The algorithm has been discovered independently by many researchers. It can be regarded as a special case of Gonzalez' farthest-point clustering algorithm [56] with the parameter p set to 1 (see Section 7.5.1).

6.3 Integer Programming Formulation and Optimal Solutions in Polynomial Time for Restricted Cases

The Hamming center problem is equivalent to a special case of the integer programming problem. Any given instance $\{\alpha_1, ..., \alpha_k\}$ of HCP, where $\alpha_i \in \{0,1\}^n$ for $1 \leq i \leq k$, can directly be expressed as a system of k linear inequalities as follows.

Let $x_1, ..., x_n$ be $0-1$-variables representing the n consecutive positions of a center β of $\{\alpha_1, ..., \alpha_k\}$ and let y be an integer variable corresponding to the (unknown) radius of the instance. For $i = 1, ..., k$, let the ith inequality be

$$\sum_{\substack{\alpha_i[m]=0 \\ 1 \leq m \leq n}} x_m + \sum_{\substack{\alpha_i[m]=1 \\ 1 \leq m \leq n}} (1 - x_m) \ \leq \ y$$

The left-hand side of inequality i equals the Hamming distance between α_i and β. (For each position m, if $\alpha_i[m] = 0$ then the sum is incremented by one if and only if $x_m = 1$, and conversely, if $\alpha_i[m] = 1$ then the sum is incremented by one if and only if $x_m = 0$.) The constraint "$\leq y$" ensures that $d_H(\alpha_i, \beta)$ is smaller than or equal to the radius.

Next, the above system of inequalities can be transformed into the form $Ax \leq b$, where A is a $(k \times n)$-matrix with every entry belonging to the set $\{-1, 1\}$, x is the $(n \times 1)$-vector $(x_1, ..., x_n)$ of $0-1$-variables, and b is a $(k \times 1)$-vector of expressions involving y. The scalar product of any prefix of any row in A with a $0-1$-vector of the same length is neither less than $-n$ nor greater than n. Therefore, we can solve the transformed system of k inequalities by a dynamic programming procedure, proceeding in stages [104]. In stage l, we compute the set W_l of all $(k \times 1)$-vectors which can be expressed as $\sum_{m=1}^{l} c_m z_m$, where c_m is the mth column of A and $z_m \in \{0,1\}$. Since the cardinality of W_l cannot be larger than $(2n+1)^k$ and there are n stages, this procedure takes a total of $O((2n+1)^k \cdot k \cdot n)$ time. Next, for each $v \in W_n$, solve the inequality $v \leq b$ in $O(k)$ time to identify a v^* which yields the smallest possible value of y (i.e., equal to the given instance's r). A center β for the given instance is then obtained by setting $\beta[m] = z_m^*$ for $1 \leq m \leq n$, where $v^* = \sum_{m=1}^{n} c_m z_m^*$. The whole algorithm uses $O((2n+1)^k \cdot k \cdot n + (2n+1)^k \cdot k + n) = n^{O(k)}$ time.

We have just proved the next theorem.

Theorem 6.3 *HCP is solvable in $n^{O(k)}$ time.*

On the other hand, if $n = O(\log k)$ then exhaustive search can be employed to find a center in polynomial time. Each candidate center can be evaluated in $O(k \cdot n)$ time, so generating and testing all of the 2^n binary strings of length n takes $O(2^n \cdot k \cdot n) = k^{O(1)}$ time.

Theorem 6.4 *HCP restricted to instances with $n = O(\log k)$ is solvable in $k^{O(1)}$ time.*

Alternatively, an optimal solution can be obtained by exhaustive search as follows.

Theorem 6.5 *HCP is solvable in $O(r \cdot n^{r+1} \cdot k)$ time.*

Proof: Choose an l arbitrarily from $\{1, ..., k\}$. Then, with R initially set to zero and increasing by one after each iteration, evaluate the $\sum_{j=0}^{R} \binom{n}{j} = O(n^R)$ strings in $\{0,1\}^n$ within Hamming distance R of α_l until a string with Hamming distance at most R to every string in S has been discovered (some optimal center β will be evaluated when R reaches r since $d_H(\alpha_l, \beta) \leq r$). The running time for this method is $\sum_{R=0}^{r} O(n^R) \cdot O(k \cdot n) = (r+1) \cdot O(n^r) \cdot O(k \cdot n) = O(r \cdot n^{r+1} \cdot k)$. □

We summarize the above in a corollary.

Corollary 6.6 *The following are solvable in polynomial time:*

- *HCP restricted to instances with $k = O(1)$.*
- *HCP restricted to instances with $n = O(\log k)$.*
- *HCP restricted to instances with $r = O(1)$.*

6.4 Randomized Rounding

By relaxing the integer constraints on the $0-1$-variables $x_1, ..., x_n$ in the integer programming formulation of HCP described in Section 6.3, thus allowing each variable x_m to assume a real number in the interval $[0,1]$, we get a linear programming problem that can be solved in polynomial time by standard methods [109]. Randomization can then be applied to the solution of the relaxed problem to obtain an approximate $0-1$-solution β' to the original problem.

We use the following randomized rounding scheme:

> For each m, where $1 \leq m \leq n$, set $\beta'[m]$ to 1 with probability $\hat{x}_m$ and to 0 with probability $1 - \hat{x}_m$, where $\hat{x}_m$ is the value assigned to x_m in the optimal solution to the relaxed linear program.

Chernoff bound techniques described in, e.g., Chapter 11 in [63] or Chapter 4 in [96] can be applied to analyze this method. We proceed as in the analysis of the randomized rounding scheme for the lattice approximation problem given on pp. 449–450 in [63]. Denote the probability of an event A by $\mathbf{Pr}[A]$ and the expectation of a random variable X by $\mathbf{E}[X]$. We will use the following two variants of the Chernoff bound.

Fact 6.7 [63] *Let $X_1, ..., X_n$ be a sequence of independent $0-1$ random variables such that $\mathbf{Pr}[X_m = 1] = p_m$ and $\mathbf{Pr}[X_m = 0] = 1 - p_m$ for every $m \in \{1, ..., n\}$. Define $Y = \sum X_m$ so that $\mathbf{E}[Y] = \sum p_m$. Then:*

1. $\mathbf{Pr}\left[|Y - \mathbf{E}[Y]| > \sqrt{4n \ln n}\right] \leq \frac{1}{n^2}$, *and*
2. *For any $\varepsilon \in [0,1]$,* $\mathbf{Pr}\left[|Y - \mathbf{E}[Y]| > \varepsilon \cdot \mathbf{E}[Y]\right] \leq 2\exp(-0.375 \cdot \varepsilon^2 \cdot \mathbf{E}[Y])$.

To simplify things later on, we introduce the following terminology.

Definition 6.8 For any two $z_1, z_2 \in [0,1]^n$, the *generalized distance between z_1 and z_2* is

$$d_G(z_1, z_2) = \sum_{m=1}^{n} |z_1[m] - z_2[m]|$$

Also, define $\hat{\beta} \in [0,1]^n$ to be the vector $(\hat{x}_1, ..., \hat{x}_n)$ obtained from the optimal solution to the relaxed version of the instance. Note that $\max_{\alpha_i \in S} d_G(\alpha_i, \hat{\beta}) \leq r$.

Lemma 6.9 *Given an instance S of HCP, let β' be the approximate solution computed by our randomized rounding scheme. Then the maximum Hamming distance between β' and any string in S is at most:*

1. *$r + \sqrt{4n \ln n}$ with probability $\geq 1 - \frac{k}{n^2}$, and*
2. *$r(1+\varepsilon)$ for any $\varepsilon \in [0,1]$ with probability $\geq 1 - k \cdot 2\exp(-0.375 \cdot \varepsilon^2 \cdot w)$, where w is the minimum generalized distance between a string in S and $\hat{\beta}$.*

Proof: Select an $i \in \{1, ..., k\}$. For each m, where $1 \leq m \leq n$, let X_m be the random variable that has the value 1 if $\alpha_i[m] = 0$ and the randomized rounding scheme sets $\beta'[m]$ to 1, or if $\alpha_i[m] = 1$ and the randomized rounding scheme sets $\beta'[m]$ to 0; otherwise, $X_m = 0$. Then

$$\mathbf{E}[X_m] = \begin{cases} \hat{x}_m, & \text{if } \alpha_i[m] = 0 \\ 1 - \hat{x}_m, & \text{if } \alpha_i[m] = 1 \end{cases}$$

Hence, $\mathbf{E}[X_m] = d_G(\alpha_i[m], \hat{\beta}[m])$. Now, $Y = \sum X_m$ is a random variable representing the Hamming distance between α_i and β'. By linearity of expectation, $\mathbf{E}[Y] = \mathbf{E}\big[\sum X_m\big] = \sum \mathbf{E}[X_m]$, which is equal to the generalized distance between α_i and $\hat{\beta}$. This implies that $\mathbf{E}[Y] \leq r$, so

$$\begin{aligned} \mathbf{Pr}\left[d_H(\alpha_i, \beta') > r + \sqrt{4n \ln n}\right] &= \mathbf{Pr}\left[Y > r + \sqrt{4n \ln n}\right] \\ &\leq \mathbf{Pr}\left[Y > \mathbf{E}[Y] + \sqrt{4n \ln n}\right] \\ &\leq \mathbf{Pr}\left[|Y - \mathbf{E}[Y]| > \sqrt{4n \ln n}\right] \\ &\leq \tfrac{1}{n^2} \end{aligned}$$

by Fact 6.7.1 and

$$\begin{aligned} \mathbf{Pr}\big[d_H(\alpha_i, \beta') > r(1+\varepsilon)\big] &= \mathbf{Pr}\big[Y > r(1+\varepsilon)\big] \\ &\leq \mathbf{Pr}\big[Y > \mathbf{E}[Y] \cdot (1+\varepsilon)\big] \\ &\leq \mathbf{Pr}\big[|Y - \mathbf{E}[Y]| > \varepsilon \cdot \mathbf{E}[Y]\big] \\ &\leq 2\exp(-0.375 \cdot \varepsilon^2 \cdot \mathbf{E}[Y]) \\ &\leq 2\exp(-0.375 \cdot \varepsilon^2 \cdot w) \end{aligned}$$

by Fact 6.7.2 together with $w = \min_{\alpha_j \in S} d_G(\alpha_j, \hat{\beta}) \leq d_G(\alpha_i, \hat{\beta}) = \mathbf{E}[Y]$.

Since the previous argument can be repeated for all $i \in \{1, ..., k\}$,

$$\begin{aligned} \mathbf{Pr}\left[\max_{\alpha_i \in S} d_H(\alpha_i, \beta') > r + \sqrt{4n \ln n}\right] &= \mathbf{Pr}\left[\bigcup_{\alpha_i \in S} \left(d_H(\alpha_i, \beta') > r + \sqrt{4n \ln n}\right)\right] \\ &\leq \sum_{\alpha_i \in S} \mathbf{Pr}\big[d_H(\alpha_i, \beta') > r + \sqrt{4n \ln n}\big] \\ &\leq k \cdot \tfrac{1}{n^2} \end{aligned}$$

and we have $\mathbf{Pr}\left[\max_{\alpha_i \in S} d_H(\alpha_i, \beta') \le r + \sqrt{4n \ln n}\right] \ge 1 - \frac{k}{n^2}$.

In the same way, $\mathbf{Pr}\left[\max_{\alpha_i \in S} d_H(\alpha_i, \beta') \le r(1+\varepsilon)\right] \ge 1 - 2\exp(-0.375 \cdot \varepsilon^2 \cdot w)$. □

Corollary 6.10 *Given an instance S of HCP, let β' be the approximate solution computed by our randomized rounding scheme. Then the following holds:*

1. *For any positive q, if $r \ge q \cdot \sqrt{4n \ln n}$ then the maximum Hamming distance between β' and any string in S is at most $r(1+\frac{1}{q})$ with probability $\ge 1 - \frac{k}{n^2}$.*
2. *If the minimum generalized distance w between a string in S and the optimal relaxed solution is at least $\frac{\ln(4k)}{0.375\varepsilon^2}$ then the maximum Hamming distance between β' and any string in S is at most $(1+\varepsilon)r$ with probability $\ge \frac{1}{2}$.*

In other words, the method of randomized rounding is likely to yield nearly optimal solutions if the radius r is substantially larger than $\sqrt{4n \ln n}$ and k is substantially smaller than n^2, or if the minimum generalized distance w is sufficiently large.

6.5 A Randomized $(\frac{4}{3}+\varepsilon)$-Approximation Algorithm

In this section, we present a randomized algorithm that for any instance of HCP and any constant $\varepsilon > 0$ returns a solution which with probability at least $\frac{1}{2}$ is within a factor of $(\frac{4}{3}+\varepsilon)$ of the optimum. The running time depends exponentially on $1/\varepsilon$, but if ε is fixed then the running time is polynomial in n and k as long as r is superlogarithmic in k or $r = O(1)$. Thus, we will assume that ε is a constant which has been specified in advance.

Recall that r is defined as

$$r = \min_{\beta \in \{0,1\}^n} (\max_{\alpha_i \in S} d_H(\alpha_i, \beta))$$

where β refers to an optimal solution to the given instance. The approximate solution found by our algorithm is called β', and we denote the value of $\max_{\alpha_i \in S} d_H(\alpha_i, \beta')$ by r'.

We first describe the algorithm and then analyze its approximation factor.

Description of the algorithm

First of all, calculate the *diameter* d of the instance, defined as the maximum over all Hamming distances between any two of the input strings, i.e.,

$$d = \max_{\alpha_j \in S} (\max_{\alpha_i \in S} d_H(\alpha_i, \alpha_j)).$$

Next, compute $\frac{12.7 \ln(4k)}{\varepsilon^3}$ using the specified value of ε. If $d < \frac{12.7 \ln(4k)}{\varepsilon^3}$ then let the algorithm branch to Case 1 below to obtain β'; if $d \geq \frac{12.7 \ln(4k)}{\varepsilon^3}$ then branch to Case 2 to obtain β'.

Case 1: $d < \frac{12.7 \ln(4k)}{\varepsilon^3}$
Note that $r \leq d$ (since $r \leq \min_{\alpha_j \in S} (\max_{\alpha_i \in S} d_H(\alpha_i, \alpha_j)) \leq d$). Find an exact solution by exhaustive search, using the method described in the proof of Theorem 6.5. Let β' be the solution found.

This takes $O(r \cdot n^{r+1} \cdot k) = O(n^{d+2} \cdot k)$ time.

Case 2: $d \geq \frac{12.7 \ln(4k)}{\varepsilon^3}$
The second case is divided into two subcases: $d \leq \frac{4}{3}r$ and $d > \frac{4}{3}r$.

At this stage, the algorithm cannot know which subcase holds for the given instance since r is still unknown. To get around this, the algorithm runs both procedures described below, evaluates the two approximate solutions obtained, and chooses the better one as the final approximate solution β'.

Subcase 2a: $d \leq \frac{4}{3}r$
Set β' to α_l, where l is chosen arbitrarily from $\{1, ..., k\}$.

Subcase 2b: $d > \frac{4}{3}r$
Rearrange the α_i's so that $d_H(\alpha_1, \alpha_k) = d$. Then, normalize the strings as follows:

```
old_$\alpha_1$ := $\alpha_1$
**for** $m := 1$ **to** $n$ **do**
    **if** $\alpha_1[m] = 1$ **then**
        **for** $i := 1$ **to** $k$ **do** $\alpha_i[m] := 1 - \alpha_i[m]$
```

In this way, the new α_1 will be the string 0^n, where 0^m for any positive integer m denotes the string consisting of exactly m 0s. The transformation does not change any of the pairwise Hamming distances because whenever some position in a string is changed, the corresponding position in every other string is changed as well. Next, let some permutation $\sigma : \{1, ..., n\} \rightarrow \{1, ..., n\}$ act on the columns of the strings so that the d positions of α_k that contain 1s end up at $\alpha_k[1..d]$

(see Figure 6.2). This operation does not affect the pairwise Hamming distances either.

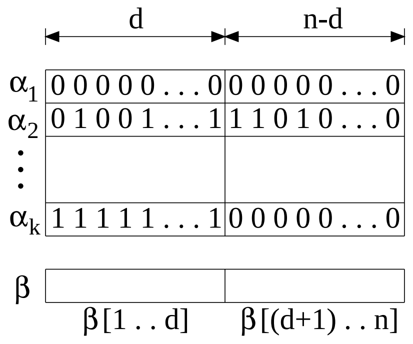
d
n-d
α1 0 0 0 0 0 . . . 0 0 0 0 0 0 . . . 0
α2 0 1 0 0 1 . . . 1 1 1 0 1 0 . . . 0
αk 1 1 1 1 1 . . . 1 0 0 0 0 0 . . . 0
β
β[1 . . d]
β[(d+1) . . n]

Figure 6.2: After normalization, α_1 and α_k are two strings that are the farthest apart. They differ at precisely positions $1..d$.

Now consider the $0-1$-integer programming problem corresponding to the normalized instance with the integer constraints relaxed as described in Section 6.4. Add constraints which force the last $n-d$ positions of any valid solution to be 000..0. Let ρ be an optimal solution to this problem, and set γ to that $\alpha_i[1..d]\,0^{n-d}$, $i=1,...,k$, which minimizes the generalized distance between γ and ρ. Next, apply the randomized rounding scheme from Section 6.4, and call the obtained solution μ. Let β' be the one of the two strings γ,μ with the smallest maximum distance to the strings α_i, $i=1,...,k$. At this point, β' always contains 0s on its last $n-d$ positions (see Figure 6.3).

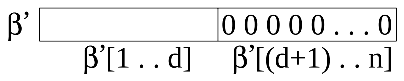

Figure 6.3: For Subcase 2b, the last $n-d$ positions of β' are 0s.

Finally, β' needs to be transformed back to the original instance. Apply σ^{-1} to β' and then perform:

```
for $m := 1$ to $n$ do
    if old_$\alpha_1[m] = 1$ then
        $\beta'[m] := 1 - \beta'[m]$
```

and Subcase 2b is done.

Algorithm analysis

The only operation that might take more than polynomial time to perform is the exhaustive search in Case 1. Hence, for instances with $d \geq \frac{12.7\ln(4k)}{\varepsilon^3}$, the algorithm always runs in polynomial time.

Next, we prove that in all cases, $r' \leq (\frac{4}{3} + \varepsilon)\, r$ with probability $\geq \frac{1}{2}$.

Case 1: $d < \frac{12.7\ln(4k)}{\varepsilon^3}$
The algorithm finds an exact solution, so $r' = r$.

Subcase 2a: $d \geq \frac{12.7\ln(4k)}{\varepsilon^3}$ and $d \leq \frac{4}{3}$
Since $r' \leq d \leq \frac{4}{3}r$, we are done.

Subcase 2b: $d \geq \frac{12.7\ln(4k)}{\varepsilon^3}$ and $d > \frac{4}{3}$
First, we observe that an optimal solution β can not have a lot of ones in the last $n - d$ positions.

Lemma 6.11 *$\beta[(d{+}1)..n]$ contains less than $\frac{1}{3}r$ ones.*

Proof: Suppose that $\beta[(d{+}1)..n]$ contains $\geq \frac{1}{3}r$ ones. Since $d_H(\alpha_1, \beta) \leq r$, $\beta[1..d]$ must contain $\leq \frac{2}{3}r$ ones. Similarly, $d_H(\alpha_k, \beta) \leq r$ implies that $\beta[1..d]$ contains $\leq \frac{2}{3}r$ zeros. But this would mean that $d \leq \frac{2}{3}r + \frac{2}{3}r = \frac{4}{3}r$, which is a contradiction since $d > \frac{4}{3}r$. □

Next, define ζ to be a string in $\{0,1\}^n$ with zeros on the last $n - d$ positions that minimizes the value of $\max\limits_{\alpha_i \in S} d_H(\alpha_i, \zeta)$.

Lemma 6.12 $\max\limits_{\alpha_i \in S} d_H(\alpha_i, \zeta) \leq \frac{4r}{3}$.

Proof: Let ξ be the concatenation of the first d symbols of β and the string consisting of $n{-}d$ zeros, i.e., $\xi = \beta[1..d]\, 0^{n-d}$. From the definitions and Lemma 6.11, we see that

$$\max_{\alpha_i \in S} d_H(\alpha_i, \zeta) \leq \max_{\alpha_i \in S} d_H(\alpha_i, \xi) \leq \max_{\alpha_i \in S}(d_H(\alpha_i, \beta) + d_H(\beta, \xi)) \leq r + \frac{1}{3}r = \frac{4r}{3}.$$

□

Lemma 6.13 *If $d \geq \frac{12.7\ln(4k)}{\varepsilon^3}$ then $r' \leq (1{+}\frac{3\varepsilon}{4}) \cdot \max\limits_{\alpha_i \in S} d_H(\alpha_i, \zeta)$ with probability at least $\frac{1}{2}$.*

Proof: We have

$$r' = \max_{\alpha_i \in S} d_H(\alpha_i, \beta') \le \max_{\alpha_i \in S}(d_G(\alpha_i, \rho) + d_G(\rho, \beta')) \le (\max_{\alpha_i \in S} d_H(\alpha_i, \zeta)) + d_G(\rho, \beta').$$

Let w be the minimum generalized distance between ρ and $\alpha_i[1..d]\, 0^{n-d}$, where $i = 1, ..., k$. If w is at most $\frac{3\varepsilon}{8}d$, then

$$d_G(\rho, \beta') \le d_G(\rho, \gamma) \le \frac{3\varepsilon}{8}d \le \frac{3\varepsilon}{4}r = \frac{3\varepsilon}{4} \cdot \max_{\alpha_i \in S} d_H(\alpha_i, \beta) \le \frac{3\varepsilon}{4} \cdot \max_{\alpha_i \in S} d_H(\alpha_i, \zeta)$$

since $d = d_H(\alpha_1, \alpha_k) \le d_H(\alpha_1, \beta) + d_H(\beta, \alpha_k) \le 2r$.

Otherwise, w satisfies $w > \frac{3\varepsilon}{8}d > \frac{\ln(4k)}{0.375(\frac{3\varepsilon}{4})^2}$, and the lemma follows from the relation $\max\limits_{\alpha_i \in S} d_H(\alpha_i, \beta') \le \max\limits_{\alpha_i \in S} d_H(\alpha_i, \mu)$ and the second part of Corollary 6.10 along with a slight modification to account for the constraint requiring zeros on the last $n-d$ positions of the solution and with ε set to $\frac{3\varepsilon}{4}$. □

Now, it is easy to verify that in Subcase 2b, we achieve the approximation factor stated earlier. Because of $d \ge \frac{12.7\ln(4k)}{\varepsilon^3}$, it follows from Lemma 6.12 and Lemma 6.13 that

$$r' \le (1 + \frac{3\varepsilon}{4}) \cdot \frac{4r}{3} = (\frac{4}{3} + \varepsilon)\, r$$

holds with probability $\ge \frac{1}{2}$.

This concludes Subcase 2b. We have thus proved the main theorem of this section.

Theorem 6.14 *The approximate center β' returned by the algorithm is within distance $(\frac{4}{3}+\varepsilon)\, r$ of all strings in S with probability $\ge \frac{1}{2}$. If the diameter d of the input instance satisfies $d \ge \frac{12.7\ln(4k)}{\varepsilon^3}$ then the algorithm runs in polynomial time. If $d < \frac{12.7\ln(4k)}{\varepsilon^3}$ then β' is actually an optimal solution, and the algorithm runs in $O(r \cdot n^{r+1} \cdot k) = O(n^{\frac{12.7\ln(4k)}{\varepsilon^3}+2} \cdot k)$ time.*

Note that since $d \ge r$, if $r \ge \frac{12.7\ln(4k)}{\varepsilon^3}$ then the algorithm is guaranteed to run in polynomial time. On the other hand, if $r = O(1)$ then the running time is also polynomial.

Finally, we remark that the success probability can be amplified from $\ge \frac{1}{2}$ to $\ge 1 - (\frac{1}{2})^c$ for any positive integer c by running the algorithm independently c times and selecting the best one of the obtained approximate solutions. (The probability that the algorithm fails to produce any approximate solution which is good enough is $\le (1-\frac{1}{2})^c = (\frac{1}{2})^c$.)

6.6 Concluding Remarks

We have shown that HCP can be solved exactly in polynomial time if restricted to instances with $k = O(1)$ (Theorem 6.3), or $n = O(\log k)$ (Theorem 6.4), or $r = O(1)$ (Theorem 6.5). In the general case, HCP can be approximated within a factor of two in polynomial time (Theorem 6.2). The method of randomized rounding yields good approximations for instances of HCP in which r is large compared to $\sqrt{4n \ln n}$, or k is small compared to n^2, or the minimum generalized distance between the optimal solution of the relaxed version of the instance and a string in the instance is large (Corollary 6.10). Our main result was a randomized approximation algorithm for HCP that for any given constant $\varepsilon > 0$ returns a solution which with probability at least one half is within a factor of $(\frac{4}{3} + \varepsilon)$ of the optimum, and whose running time is polynomial if $r \geq \frac{12.7 \ln(4k)}{\varepsilon^3}$ or $r = O(1)$ (Theorem 6.14).

After reading about the randomized, polynomial-time $(\frac{4}{3}+\varepsilon)$-approximation algorithm of Lanctot *et al.* [88], we realized that a minor modification to our algorithm will actually make it run in polynomial time for *all* instances of HCP. The idea is to change Case 1 ($d < \frac{12.7 \ln(4k)}{\varepsilon^3}$) so that instead of finding an exact solution (which might take exponential time), we let the algorithm look for an approximate solution within the $(\frac{4}{3} + \varepsilon)$-bound using only polynomial time. More precisely, let the approximate solution for Case 1 be the best of the two approximate solutions obtained from the following subcases:

- Return α_l, where l is chosen arbitrarily from $\{1, ..., k\}$ (i.e., the same method as for Subcase 2a). If $d \leq \frac{4}{3}r$ then this gives $r' \leq d \leq \frac{4}{3}r$.
- Transform the instance as for Subcase 2b and let ζ be a string in $\{0,1\}^n$ with zeros on the last $n-d$ positions that minimizes $\max_{\alpha_i \in S} d_H(\alpha_i, \zeta)$. Find ζ by testing all of the $2^d < (4k)^{\frac{12.7 \cdot \ln(2)}{\varepsilon^3}}$ strings in $\{0,1\}^n$ with zeros on the last $n - d$ positions. The number of strings which have to be tested is polynomial in k since ε is constant, and each such string can be evaluated in $O(kn)$ time; therefore, the time required is polynomial in n and k. Transform ζ back to the original instance and return ζ. If $d > \frac{4}{3}r$ then $r' = \max_{\alpha_i \in S} d_H(\alpha_i, \zeta) \leq \frac{4r}{3}$ by Lemma 6.12.

As for further improvements on the polynomial-time approximability of HCP, there is not much left to be done. The PTAS of Li, Ma, and Wang [91] can approximate HCP within a factor of $1 + \varepsilon$ for any constant $\varepsilon > 0$ in polynomial time; furthermore, it is deterministic and works for strings over any constant-size alphabet. However, it has a high time complexity and may be difficult to use in practice (see p. 159 in [91]). Therefore, the main open problem concerning HCP is to find a simpler PTAS whose running time is upper bounded by a polynomial of smaller degree than the one in [91].

Chapter 7

Hamming p-Radius and p-Diameter Clustering

Here, we consider the Hamming versions of two classical clustering problems. The first one, *the Hamming p-radius clustering problem* (HRC), is the natural generalization of HCP (the problem we studied in detail in Chapter 6) where instead of finding *one* binary string β of length n which is close to *all* of the input strings, we want to find p binary strings $\{\beta_1, ..., \beta_p\}$ of length n such that every input string is close to *at least one* string in the set $\{\beta_1, ..., \beta_p\}$. The second problem we consider is called *the Hamming p-diameter clustering problem* (HDC). It is the related problem in which the objective is to partition the input strings into p groups so that the maximum of the group diameters is minimized.

We extend our investigation of the computational complexity of HCP to HRC and HDC, leading us to new inapproximability results, exact polynomial-time algorithms for certain restrictions of the problems, and various types of approximation algorithms.

7.1 Introduction

As in Chapter 6, let $\{0,1\}^n$ be the set of all strings of length n over the alphabet $\{0,1\}$. For any $\alpha \in \{0,1\}^n$, we use the notation $\alpha[m]$ to refer to the symbol placed at the mth position of α, where $m \in \{1, ..., n\}$. The *Hamming distance* between two strings $\alpha_1, \alpha_2 \in \{0,1\}^n$ is the number of positions in which the strings differ, and is denoted by $d_H(\alpha_1, \alpha_2)$.

The Hamming p-radius clustering problem (HRC) is defined as:

The Hamming p-radius clustering problem (HRC)

Instance: Finite set $S = \{\alpha_1, ..., \alpha_k\}$ such that $S \subseteq \{0,1\}^n$ for some positive integer n, positive integer p with $p \leq k$.

Output: A set $\{\beta_1, ..., \beta_p\} \subseteq \{0,1\}^n$ which minimizes the value of

$$\max_{\alpha_i \in S} \min_{1 \leq q \leq p} d_H(\alpha_i, \beta_q) \tag{7.1}$$

Such an optimal set $\{\beta_1, ..., \beta_p\}$ of strings is called a *p-center set of S* or a *p-center of S*. The corresponding value of (7.1) is called the *p-radius of S*, and is denoted by r. The definitions imply that $p \leq k \leq 2^n$ and $r \leq n$. Note that HRC restricted to instances with p set to 1 is the Hamming center problem (HCP), the problem which was studied in detail in the previous chapter.

Radius clustering is also called *minmax radius clustering* or *central clustering* in the literature [63]. The corresponding problem for graphs is often termed *the p-center problem* [63, 103].

The Hamming p-diameter clustering problem (HDC) is defined on the same set of instances as HRC. However, the goal is defined differently:

The Hamming p-diameter clustering problem (HDC)

Instance: Finite set $S = \{\alpha_1, ..., \alpha_k\}$ such that $S \subseteq \{0,1\}^n$ for some positive integer n, positive integer p with $p \leq k$.

Output: A partition of S into p disjoint subsets $S_1, ..., S_p$ which minimizes the value of

$$\max_{1 \leq q \leq p} \max_{\alpha_i, \alpha_j \in S_q} d_H(\alpha_i, \alpha_j) \tag{7.2}$$

The minimum value of (7.2) is called the *p-diameter of S*, and is referred to by d. A partition of S into p disjoint subsets which achieves this minimum value is called a *p-cluster set of S*, and each element of a p-cluster set of S is a *p-cluster of S*. Again, from the problem definitions it follows that $p \leq k \leq 2^n$ and $d \leq n$. For any subset S' of S, the 1-*diameter of S'* is the value of $\max_{\alpha_i, \alpha_j \in S'} d_H(\alpha_i, \alpha_j)$.

Diameter clustering is sometimes called *minmax diameter clustering* or *pairwise clustering* [63].

An algorithm $\mathcal{A}$ is said to approximate HRC within a factor of f if for any instance of the problem, $\mathcal{A}$ outputs a set $B' = \{\beta'_1, ..., \beta'_p\} \subseteq \{0,1\}^n$ such that $\max_{\alpha_i \in S} \min_{\beta'_q \in B'} d_H(\alpha_i, \beta'_q) \leq f \cdot r$. Similarly, an algorithm $\mathcal{A}$ is said to approximate HDC within a factor of f if for any instance of the problem, $\mathcal{A}$ outputs a partition $\{S'_1, ..., S'_p\}$ of S such that $\max_{1 \leq q \leq p} \max_{\alpha_i, \alpha_j \in S'_q} d_H(\alpha_i, \alpha_j) \leq f \cdot d$. An algorithm which approximates HRC/HDC within a factor of f is also called a factor f approximation algorithm (or just an f-approximation algorithm) for HRC/HDC. We say that a problem is NP-hard to approximate within a factor of f if it cannot be approximated within a factor of f by any polynomial-time algorithm unless P=NP.

7.1.1 Motivation

A clustering problem is a computational problem in which the elements of a given set have to be divided into groups so that all elements within a group are similar to each other. Important applications of algorithms for clustering problems have turned up in operations research, pattern recognition, data mining, concept learning, statistical data analysis, astrophysics, and data compression, and more recently, in the automatic classification of web pages and in the study of gene expression data in computational molecular biology (see below and [1, 2, 7, 101] for references). In each such application, a given practical problem is modeled as a specific, well-defined clustering problem which is then solved exactly, or approximately if an exact solution cannot be obtained. The most appropriate measure of similarity and objective function to use depend on the application at hand and the nature of the elements that are being analyzed. For instance, the Hamming metric has been used as the measure of similarity in applications involving elements represented as binary strings of equal length; examples include compressing correlated bitmaps [20], automatic script identification from scanned images (e.g., distinguishing between Arabic, Armenian, Burmese, etc.) [64], reconstructing unknown Boolean functions from incomplete sets of samples [97], and gene expression analysis [114].

There is a vast literature on algorithms and computational complexity results for clustering problems in the graph theoretic and fixed-dimensional geometric settings (see, e.g., [7, 10, 24, 35, 56, 59, 63, 103, 128] and [1, 2, 7, 10, 43, 56, 63, 101], respectively), but not as much is known about the polynomial-time solvability of geometric clustering problems where the dimension is unrestricted [101]. Our goal in this chapter is to determine the computational complexities of the two unrestricted dimensional clustering problems that use the Hamming metric and the general-purpose criteria corresponding to minimizing expression (7.1) or (7.2) on p. 118 as their objective function, i.e., HRC and HDC.

In many applications, the number of clusters that need to be produced is relatively small [101]. On the other hand, p might be large in certain pattern matching applications; a system for Chinese character recognition, for example, would need to be able to discriminate between thousands of characters. We are therefore interested in the computational complexities of HRC and HDC both when p is small and when p is unrestricted.

HRC generalizes the Hamming center problem from Chapter 6. Hence, another potential use for HRC (and HDC) is the application described in Section 6.1.1 in which an unbiased representative (i.e., an unbiased consensus string) for a given set S of k related biomolecular sequences has to be computed, e.g., to categorize other sequences later on, or to probe a database in order to discover similar sequences. It can be extended directly to the problem of computing $p > 1$ representatives for S, where $p \ll k$. Here, the p representatives can be the members of a p-center set of S, or simply p sequences from different p-clusters of S.

HRC can be viewed as a facility location problem where p facilities have to be assigned to locations in $\{0,1\}^n$. However, since any p-center of a given set S induces a partition of S (as explained in Section 7.2), we can think of HRC as a clustering problem as well. Indeed, in this chapter, we use the name *the Hamming p-radius clustering problem* rather than *the Hamming p-center problem* to emphasize the close relationship to *the Hamming p-diameter clustering problem.*

7.1.2 Previous Results

Many computational complexity results related to p-radius and p-diameter clustering in the graph theoretic and geometric settings have been published previously; see [1, 2, 10, 35, 43, 56, 63, 101, 103, 128] and the numerous references therein. Below, we mention the ones that are the most relevant for this chapter.

Both the p-radius and the p-diameter clustering problems on edge-weighted, complete, undirected graphs are NP-hard to approximate within a factor of $2-\varepsilon$ for any $\varepsilon > 0$ even if the edge weights satisfy the triangle inequality (see [10, 63, 128]). Feder and Greene [43] proved that the p-radius and p-diameter clustering problems in the plane under the L_2 metric are NP-hard to approximate within a factor of 1.822 and 1.969, respectively; moreover, they proved that under the L_1 metric and the L_∞ metric, the problems are NP-hard to approximate within a factor of $2-\varepsilon$ for any $\varepsilon > 0$.

Gonzalez' farthest-point clustering algorithm [56] can be used to obtain an approximation factor of 2 in polynomial time for p-radius and p-diameter clustering problems whenever the used distance function satisfies the triangle inequality; see Theorem 8.14 in [63] for a short proof. Fact 6.1 immediately implies that HRC and HDC are polynomial-time approximable within a factor of 2 (details are given in Section 7.5.1).

Some of the few known clustering results which are specifically tied to the Hamming metric are listed next.

Frances and Litman [46] proved that HRC is NP-hard already for $p = 1$ (incidentally, this suggests that HRC is at least as hard as the corresponding p-radius clustering problem on an edge-weighted, complete, undirected graph since the latter can be solved exactly in polynomial time by exhaustive search when restricted to any constant p). Li, Ma, and Wang [91] gave a polynomial-time approximation scheme (PTAS) for HRC restricted to $p = 1$. See Chapter 6 for other known results on HRC with $p = 1$.

The Hamming p-median clustering problem (HMC) is defined on the same set of instances as HRC and HDC, but its objective is to find a set $\{\beta_1, ..., \beta_p\} \subseteq \{0,1\}^n$ which minimizes the value of $\sum_{i=1}^{k} \min_{1 \leq q \leq p} d_H(\alpha_i, \beta_q)$. It is solvable in $O(kn)$ time for $p = 1$ by setting position m of the solution to majority$(\alpha_i[m])_{i=1}^{k}$ for every $1 \leq m \leq n$, but NP-hard for every fixed $p \geq 2$ [84]. Ostrovsky and Rabani [101] provided a randomized PTAS for HMC restricted to $p = O(1)$, and showed how it can be used to obtain randomized polynomial-time approximation schemes for p-median clustering problems with $p = O(1)$ in certain other metric spaces.

7.1.3 Our Contributions

In this chapter, we derive several new results concerning the polynomial-time solvability and approximability/inapproximability of HRC and HDC. We outline our results below.

Section 7.2 demonstrates that while the p-diameter of a set of binary strings is not necessarily equal to its p-radius, it is always within a factor of two.

In Section 7.3, we prove not only that HRC and HDC are NP-hard in the general case[1], but that both HRC and HDC are in fact NP-hard to approximate within a factor of $2 - \varepsilon$ for any constant $\varepsilon > 0$. We also consider another kind of approximation of HDC obtained by relaxing the constraint on the number of produced clusters (but still requiring that none of their 1-diameters exceed the p-diameter of the instance) and show that it is NP-hard to approximate the number of clusters within a factor of $pk^{1/7-\varepsilon}$ for any constant $\varepsilon > 0$. As a corollary, we obtain that HDC is an NP-hard problem already for $p = 3$.

Restricted cases of HRC and HDC are studied in Section 7.4. We prove that HRC can be solved exactly in polynomial time if $k = O(1)$, or if $p = O(1)$ and $n = O(\log k)$, or if $p = O(1)$ and $r = O(1)$. We prove that HDC is solvable in polynomial time if $p = O(1)$ and $k = O(\log n)$, or if $p = 2$. The techniques we use are based on integer programming, exhaustive search, and breadth-first search.

In Section 7.5, we first observe that an approximation factor of two for the general case of HRC and HDC can be achieved in $O(pkn)$ time by using Gonzalez' farthest-point clustering algorithm [56]. It follows from our inapproximability result mentioned above that this is the best possible polynomial-time constant approximation factor for the unrestricted versions of HRC and HDC, unless P=NP. We then provide a (deterministic) approximation scheme which approximates HRC within a factor of $(1 + \varepsilon)$ for any given constant ε, where $0 < \varepsilon < 1$, in $k^{O(p/\varepsilon)} \cdot 2^{O(rp/\varepsilon)} \cdot n$ time, which is polynomial for problem instances with $p = O(1)$ and $r = O(\log(k + n))$. Next, we combine the randomized PTAS

[1]As pointed out earlier, HRC was first proved to be NP-hard even if restricted to $p = 1$ in [46].

of Ostrovsky and Rabani [101] for the Hamming p-median clustering problem with the PTAS of Li, Ma, and Wang [91] for HRC restricted to $p = 1$ to obtain a randomized PTAS for HRC restricted to $p = O(1)$ that has a high success probability.

Finally, in Section 7.6, we give an approximation algorithm for HRC which approximates the p-radius within a factor of $(1+\varepsilon)$ for any constant $0 < \varepsilon < 1$ at the expense of a slight increase in the number of output strings: it produces at most $(1 + \ln k) \cdot p$ strings that together approximate a p-center set. This twofold approximation algorithm runs in $O((k \cdot 2^r)^{2/\varepsilon} \cdot (n + k) \cdot k \cdot \log n)$ time, which is polynomial whenever $r = O(\log(k + n))$ even if p is unbounded.

Our conclusions and some open problems related to HRC and HDC are presented in Section 7.7.

7.2 Preliminaries

HRC and HRC are defined for the same set of instances, but the p-radius r and the p-diameter d of a set of binary strings are in general different, as the following example illustrates.

Example 7.1

Consider the instance $S = \{00010000, 00100000, 01000000, 10000000, 11110000, 11111111\}$ with $p = 2$.

An optimal solution to HRC is $\{\beta_1 = 00000000, \beta_2 = 11110101\}$ with $r = 2$.

On the other hand, an optimal solution to HDC is $\big\{S_1 = \{00010000, 00100000, 01000000, 10000000, 11110000\}, S_2 = \{11111111\}\big\}$ with d equal to 3. □

Let (S, p) be an instance of HRC/HDC. A p-center set $\{\beta_1, ..., \beta_p\}$ of S with p-radius r induces an approximate p-cluster set $\{\tilde{S}_1, ..., \tilde{S}_p\}$ of S with diameter $\tilde{d}$ (for $i = 1, ..., k$, if β_q is the string in the p-center closest to α_i with the lowest index then let $\alpha_i \in \tilde{S}_q$). In the same way, a p-cluster set $\{S_1, ..., S_p\}$ of S with p-diameter d induces an approximate p-center set $\{\tilde{\beta}_1, ..., \tilde{\beta}_p\}$ of S with radius $\tilde{r}$ (for $q = 1, ..., p$, let $\{\tilde{\beta}_q\}$ be a 1-center set for the set of strings belonging to S_q).

Example 7.2

Let S be the instance in Example 7.1.

The approximate 2-cluster set induced by $\{\beta_1, \beta_2\}$ is $\big\{\tilde{S}_1 = \{00010000, 00100000, 01000000, 10000000\}, \tilde{S}_2 = \{11110000, 11111111\}\big\}$, so the corresponding value of $\tilde{d}$ is 4.

An approximate 2-center set induced by $\{S_1, S_2\}$ is $\{\tilde{\beta}_1 = 01010000, \tilde{\beta}_2 = 11111111\}$, which implies $\tilde{r} = 3$. □

The next lemma shows that an approximate solution to HDC induced by an optimal solution to HRC is within a factor of two of optimum, and vice versa. Moreover, it shows that the p-diameter of a set of binary strings is always less than or equal to twice its p-radius.

Lemma 7.3 *Given an instance of HRC/HDC, define $r, \tilde{r}, d$, and $\tilde{d}$ as above. Then:*
(a) $\tilde{r} \le 2r$;   (b) $\tilde{d} \le 2d$;   (c) $r \le d \le 2r$

Proof: By definition, we have (1) $r \le \tilde{r}$ and (2) $d \le \tilde{d}$. Also, (3) $\tilde{r} \le d$ because setting $\tilde{\beta}_q$ to an arbitrary string in S_q for each $q \in \{1, ..., p\}$ gives an approximate p-center set with radius less than or equal to d. Next, since the Hamming distance function obeys the triangle inequality (see Fact 6.1), the distance between two strings α_i, α_j that end up in the same $\tilde{S}_q$ must be less than or equal to $d_H(\alpha_i, \beta_q) + d_H(\beta_q, \alpha_j) \le 2r$, so it holds that (4) $\tilde{d} \le 2r$.

Now, (a) follows from (3), (2), and (4); (b) follows from (4), (1), and (3). Finally, (c) follows from (1), (3), (2), and (4). □

7.3 HRC and HDC are NP-Hard to Approximate

In this section, we prove that both HRC and HDC are NP-hard to approximate within any constant factor smaller than two. We also prove that for any constant $\varepsilon > 0$, it is NP-hard to split S into at most $pk^{1/7-\varepsilon}$ disjoint clusters whose 1-diameters do not exceed the p-diameter of S. It follows from the reduction we use to prove the latter result that solving HDC exactly is NP-hard for every fixed $p \ge 3$.

7.3.1 NP-Hardness of Approximating the p-Radius and the p-Diameter

The starting point for proving the hardness results in this subsection is the reduction in [43] from vertex cover for planar graphs of degree at most three to the p-radius and p-diameter clustering problems in the plane under the L_1 metric[2]. We first show that all points in the resulting instance of the corresponding p-clustering problem as well as the points in an approximate p-center can be required to lie on an integer grid whose size is polynomial in the size of the input planar graph, giving us the following technical strengthening of Theorem 2.1 in [43].

[2] *The p-radius clustering problem in the plane under the L_1 metric* is the following: Given a finite set S of points in the plane, find a set P of p points in the plane that minimizes $\max_{s \in S} \min_{u \in P} d_1(s, u)$, where d_1 is the L_1 distance. *The p-diameter clustering problem in the plane under the L_1 metric* is defined analogously.

Lemma 7.4 *The p-radius and p-diameter clustering problems in the plane under the L_1 metric for a finite set S of points, where the points in S lie on an integer square grid whose size is polynomial in $|S|$ and where the approximate solution to the radius version is required to lie on the grid, are NP-hard to approximate within a factor of $2 - \varepsilon$ for any constant $\varepsilon > 0$.*

Proof: The reduction in [43] embeds an instance of vertex cover for planar graphs of degree at most three in the plane by replacing each edge with a path composed of an odd number of unit-length edges. The midpoints of these unit-length edges form an instance $\mathcal{I}$ of p-radius or p-diameter clustering in the plane which admits a solution with p-radius 0.5 or p-diameter 1, respectively, if and only if the embedded graph has a vertex cover of size p. The key observation is that the distance between the midpoints of any two nonadjacent edges is at least 2 in case of the L_1 metric (see Figure 7.1). It follows that *finding an approximate solution to $\mathcal{I}$ within any factor smaller than 2 is as hard as finding an exact solution*, yielding the NP-hardness of approximating the p-radius and p-diameter clustering problems in the plane under the L_1 metric within any factor smaller than 2. For further details concerning the reduction, see [43] or [63].

Consider the smallest square box B with sides parallel to the x- and y-axes which contains the embedded graph constructed in the reduction. Since the graph can be assumed to be connected, the length of a side of the box is $O(l)$, where l is the number of points in the instance of the radius or diameter clustering problem in the plane. Note that l has to be polynomial in the size n of the original vertex cover instance [43]. We conclude that the size of the box is polynomial in n.

Form a uniform point grid within B such that the distance between nearest neighbors in the grid is δ, where $0 < \delta \leq 0.01$. Move each of the midpoints in $\mathcal{I}$ to its nearest grid point. Such a movement changes the relative distance between two midpoints by at most 2δ. Adding the requirement that an approximate p-center must also lie on the grid can further increase the radius by at most δ. It follows that $\mathcal{I}$ admits a clustering with p-radius 0.5 or p-diameter 1, respectively, if and only if the resulting instance $\mathcal{I}'$ of clustering on the grid admits a solution with p-radius $\leq (0.5 + \delta) + \delta = 0.5 + 2\delta$ or p-diameter $\leq 1 + 2\delta$. It also follows that $\mathcal{I}$ has p-radius at least 1 or p-diameter at least 2 if and only if $\mathcal{I}'$ has p-radius $\geq 1 - 2\delta$ or p-diameter $\geq 2 - 2\delta$. Now, if the p-radius of $\mathcal{I}'$ could be approximated within $2 - 12\delta$ then the p-radius of $\mathcal{I}$ could be computed exactly since $(0.5 + 2\delta) \cdot (2 - 12\delta) < 1 - 2\delta$. Similarly, if the p-diameter of $\mathcal{I}'$ could be approximated within $2 - 6\delta$ then the p-diameter of $\mathcal{I}$ could be computed exactly since $(1 + 2\delta) \cdot (2 - 6\delta) < 2 - 2\delta$.

Since δ can be selected arbitrarily close to 0 and $\mathcal{I}'$ can be constructed in time which is polynomial in n for any fixed δ, it is sufficient to transform the grid to an integer grid by rescaling by $1/\delta$ in order to obtain the theorem in both cases. $\square$

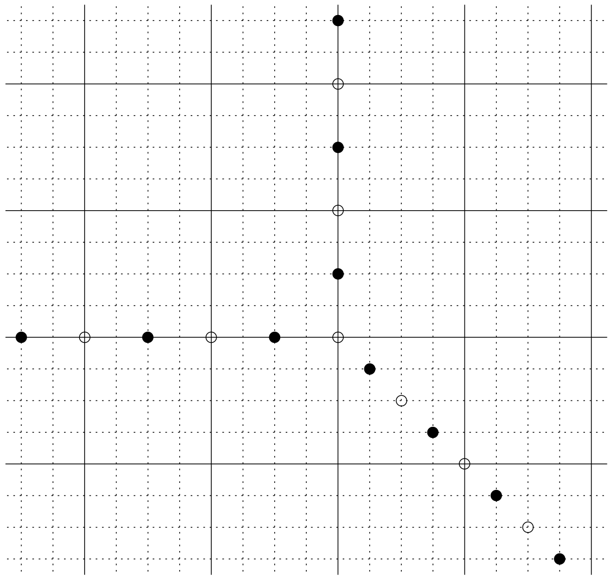

Figure 7.1: The L_1 distance between two edge midpoints (shown as filled circles) is 1 if the edges are adjacent, and ≥ 2 otherwise.

By embedding the L_1 metric on an integer square grid into the Hamming metric, we obtain the main result of this subsection.

Theorem 7.5 *HRC and HDC are NP-hard to approximate within a factor of* $2 - \varepsilon$ *for any constant* $\varepsilon > 0$.

Proof: Let S be a set of points on an integer square grid of size $W \times W$, where W is bounded by some polynomial in $|S|$. For each $s \in S$, denote the x- and y-coordinates of s by s_x and s_y, respectively. Encode each $s \in S$ by the binary string $e(s)$ of length $2W$ composed of s_x consecutive 1's followed by $W - s_x$ consecutive 0's, then s_y consecutive 1's, and finally, $W - s_y$ consecutive 0's. Then, for any two points s' and s'' in S, their L_1 distance is the absolute difference in x-coordinates plus the absolute difference in y-coordinates, which is equal to the Hamming distance between their encodings $e(s')$ and $e(s'')$ (see Figure 7.2 for an example). This observation, together with Lemma 7.4, yields the theorem for HDC.

Next, consider an approximate solution $\{a_1, ..., a_p\}$ to HRC for the strings $e(s)$, $s \in S$. For $q = 1, ..., p$, transform a_q to a'_q having the form $1^l 0^{W-l} 1^m 0^{W-m}$ for some $l, m \leq W$ by moving all the 1's contained in the left half of a_q to the beginning of the left half, and all the 1's in the right half of a_q to the beginning of the right half. The resulting set of strings $\{a'_1, ..., a'_p\}$ is a solution which is at least as good as $\{a_1, ..., a_p\}$. Also, it can be directly decoded into a set of grid

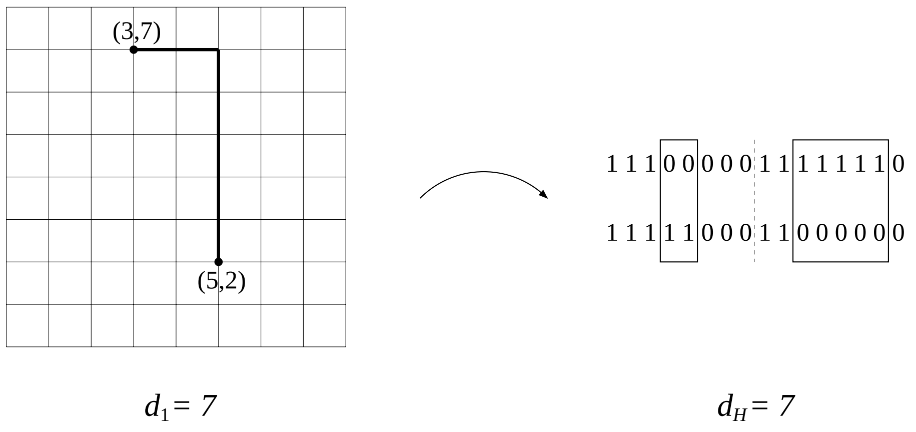

Figure 7.2: The L_1 distance between any two points in S equals the Hamming distance between the two encoding strings.

points $\{g_1, ..., g_p\}$ such that $a'_q = e(g_q)$ for $q = 1, ..., p$. Lemma 7.4 now gives us the theorem for HRC. □

7.3.2 NP-Hardness of Approximating HDC in Terms of the Number of Clusters

Let G be an undirected graph. A *partition of G into q cliques* is a partition of the set of vertices of G into disjoint subsets $V_1, ..., V_q$ such that for $j = 1, ..., q$, the subgraph of G induced by V_j is a complete graph. *The minimum clique partition problem* is: Given an undirected graph G, find a partition of G into as few cliques as possible. A partition of G into q cliques induces a coloring with q colors of the complement graph $\overline{G}$ of G and vice versa since two vertices in $\overline{G}$ are nonadjacent (i.e., can be assigned the same color) if and only if they are adjacent in G. Thus, known inapproximability results for minimum graph coloring [12] imply that for any constant $\varepsilon > 0$, the problem of finding an approximate solution to the minimum clique partition problem consisting of at most $p \cdot |V|^{1/7-\varepsilon}$ cliques, where $|V|$ is the number of vertices in the input graph G and p is the number of cliques in an optimal solution, is NP-hard.

By a reduction from the minimum clique partition problem to HDC, we obtain:

Theorem 7.6 *For any constant $\varepsilon > 0$, the problem of finding a partition of a set of k binary strings of length $O(k^2)$ into at most $pk^{1/7-\varepsilon}$ disjoint clusters such that each cluster has 1-diameter not exceeding the p-diameter is NP-hard.*

Proof: Given an instance G of the minimum clique partition problem, let $|V|$ denote the number of vertices in G and construct an undirected graph G' with $2|V|$ vertices by augmenting G with $|V|$ new vertices and then, for every vertex v appearing in G, adding edges between v and new vertices until v gets degree $|V|$ in G'. Enumerate the edges of G' from 1 to m, where $m = O(|V|^2)$. For every vertex v in G, form a binary string $s(v)$ of length m such that there is a 1 on the ith position of $s(v)$ if and only if the ith edge of G' is incident to v. Note that for any pair of vertices v_1, v_2 in G, the Hamming distance between $s(v_1)$ and $s(v_2)$ is $2|V| - 2$ if they are adjacent, otherwise it is $2|V|$. Therefore, any partition of G into p cliques yields a p-cluster set of the resulting strings whose maximum 1-diameter is less than or equal to $2|V| - 2$, and conversely, any q-cluster set of the resulting strings whose maximum 1-diameter is less than or equal to $2|V| - 2$ trivially yields a partition of G into q cliques. The theorem follows from the inapproximability result cited above, with $k = |V|$. □

As for the corresponding approximation problem for HRC (i.e., producing a larger set of approximate centers such that each input string is within the p-radius of at least one of them), we doubt whether it is as hard to approximate. Indeed, if we weaken the requirement of being within the p-radius by a multiplicative factor of $1 + \varepsilon$, then the problem admits a logarithmic approximation in polynomial time if $r = O(\log(k + n))$, as shown in Section 7.6.

We also note the following. *The partition into cliques problem* is: Given an undirected graph G and a positive integer p, is it possible to partition G into p cliques? The partition into cliques problem is NP-complete for every fixed $p \geq 3$ (see [49]), so the reduction in the proof of Theorem 7.6 also implies:

Corollary 7.7 *HDC is NP-hard for every fixed $p \geq 3$.*

7.4 Polynomial-Time Optimal Solutions for Restricted Cases

Although HRC and HDC are NP-hard in general by the result of Frances and Litman [46] and Corollary 7.7 above, certain special cases of HRC and HDC (e.g., HRC restricted to instances with $p = k$ and HDC restricted to instances with $p = 1$ or $p = k$) can be solved by trivial algorithms. Here, we investigate some other, more interesting cases of HRC and HDC which can be solved in polynomial time.

Recall from Section 6.3 that HRC with $p = 1$ can be expressed as a special case of the integer programming problem which is solvable in $n^{O(k)}$ time (Theorem 6.3). We first show that HRC can be solved in $(pn)^{O(k)}$ time for $p > 1$.

Theorem 7.8 *HRC is solvable in $(pn)^{O(k)}$ time.*

Proof: Let $(\alpha_1, ..., \alpha_k, p)$ be a given instance of HRC, where $\alpha_i \in \{0,1\}^n$ for $1 \leq i \leq k$ and $p \in \{1, ..., k\}$. For each of the $O(p^k)$ ways to partition the k strings into p subsets $\{S_1, ..., S_p\}$, construct p instances of HRC with p set to 1 such that for $1 \leq q \leq p$, instance q consists of the strings in subset S_q. Then use the method of Theorem 6.3 to solve each instance optimally, and let the value of the current partition equal the maximum of the p resulting 1-radii. As the final solution, return the set of 1-centers of a partition that yields the smallest value.

To prove the correctness of this method, consider an optimal p-center set $\{\beta_1, ..., \beta_p\}$. It induces a partition $\{\tilde{S}_1, ..., \tilde{S}_p\}$ of $\{\alpha_1, ..., \alpha_k\}$ where for $1 \leq i \leq k$, $\alpha_i \in \tilde{S}_q$ if β_q is the string in the p-center closest to α_i with the lowest index. Let r be the p-radius. By the definition of a p-center set, $d_H(\alpha_i, \beta_q) \leq r$ for all $\alpha_i \in \tilde{S}_q$. Thus, the distance between an optimal 1-center of $\tilde{S}_q$ and a string in $\tilde{S}_q$ cannot be greater than r. All partitions of the input strings, including $\{\tilde{S}_1, ..., \tilde{S}_p\}$, are tested, so an optimal solution will be found.

The method takes a total of $O(p^k) \cdot O(p) \cdot n^{O(k)} = (pn)^{O(k)}$ time. □

We conclude that HRC with $k = O(1)$ can be solved exactly in polynomial time.

Exhaustive search over all possible p-centers results in a $k^{O(p)}$-time algorithm when $n = O(\log k)$ because there are $(2^n)^p$ candidate solutions, each of which can be evaluated in $O(kpn)$ time. We have:

Theorem 7.9 *HRC restricted to instances with $n = O(\log k)$ is solvable in $k^{O(p)}$ time.*

Exhaustive search over sets of strings of increasing distance to the input strings can also be used to obtain an optimal solution, as the next theorem shows.

Theorem 7.10 *HRC is solvable in $O(r \cdot k^{p+1} \cdot n^{pr+1} \cdot p)$ time.*

Proof: Initialize a counter R to zero. Repeat the following until a solution B has been found, incrementing R by one after each iteration: For every cardinality p subset $\{\gamma_1, ..., \gamma_p\}$ of the set of input strings S, evaluate all different sets of the form $\{y_1, ..., y_p\}$ where for $1 \leq q \leq p$, $y_q \in \{0,1\}^n$ and y_q is within Hamming distance R of γ_q; if there exists such a set $\{y_1, ..., y_p\}$ satisfying $\max_{\alpha_i \in S} \min_{1 \leq q \leq p} d_H(\alpha_i, y_q) = R$ then let $B := \{y_1, ..., y_p\}$.

To see that this method always finds an optimal solution, let $\{\beta_1, ..., \beta_p\}$ be any p-center set of S. As in the proof of Theorem 7.8, it induces a partition $\{\tilde{S}_1, ..., \tilde{S}_p\}$ of $\{\alpha_1, ..., \alpha_k\}$ where for $1 \leq i \leq k$, $\alpha_i \in \tilde{S}_q$ if β_q is the string in the p-center closest to α_i with the lowest index. Now, for $1 \leq q \leq p$, if $\tilde{S}_q$ is nonempty then define $\hat{\gamma}_q$ to be the string in $\tilde{S}_q$ with lowest index; note that $d_H(\hat{\gamma}_q, \beta_q) \leq r$, and furthermore, $\hat{\gamma}_q \notin \tilde{S}_j$ for all $j \neq q$. Next, for $1 \leq q \leq p$, if $\tilde{S}_q$ is empty then define $\hat{\gamma}_q$ to be any element in S which is different from all $\hat{\gamma}_j$

defined so far and set β_q to $\hat{\gamma}_q$ (if $\tilde{S}_q$ is empty then β_q can be replaced by any string in $\{0,1\}^n$ without affecting the value of (7.1) in the definition of HRC on p. 118). The resulting set $\{\beta_1, ..., \beta_p\}$ is a p-center set of S which will be evaluated at some point when R has reached r and the subset $\{\hat{\gamma}_1, ..., \hat{\gamma}_p\}$ is considered.

In every iteration, there are $\binom{k}{p} = O(k^p)$ subsets of S to consider. There are $\sum_{j=0}^{R} \binom{n}{j} = O(n^R)$ strings in $\{0,1\}^n$ within Hamming distance R of any string in S, and each candidate solution can be evaluated in $O(kpn)$ time, so the total running time is $\sum_{R=0}^{r} O(k^{p+1} \cdot n^{pR+1} \cdot p) = O(r \cdot k^{p+1} \cdot n^{pr+1} \cdot p)$. □

One of the main differences between HDC and HRC is that the former does not involve strings outside the input set S. For this reason, it seems simpler to solve exactly than HRC does[3]. It can be solved in $O(k^2 n + p^k k^2)$ time by precomputing all Hamming distances between pairs of strings in S and then doing an exhaustive search, which immediately yields the following result.

Theorem 7.11 *HDC restricted to instances with $k = O(\log n)$ is solvable in $n^{O(\log p)}$ time.*

Finally, we observe that the Hamming 2-diameter clustering problem admits a rather straightforward polynomial-time algorithm.

Theorem 7.12 *HDC restricted to instances with $p = 2$ is solvable in $O(k^2 n)$ time.*

Proof: Let d be a candidate value for the 2-diameter of S. Form a graph G whose vertices are in one-to-one correspondence with the input strings, and connect a pair of vertices by an edge whenever the Hamming distance between the corresponding strings is less than or equal to d. The problem of partitioning S into two subsets whose 1-diameters are at most d is now equivalent to the problem of partitioning G into two cliques; the latter problem in turn reduces to 2-coloring the complement graph $\overline{G}$ of G (see Section 7.3.2). A 2-coloring of $\overline{G}$ (if one exists) can be found in $O(k^2)$ time by breadth-first search. To determine the smallest possible d for which a 2-coloring of $\overline{G}$ exists, use the procedure just described for different values of d, generated by a binary search. Calculating all pairwise Hamming distances requires $O(k^2 n)$ time, but this can be done before starting the search for d. The total running time is $O(k^2 n + k^2 \cdot \log n) = O(k^2 n)$. □

The results of this section are summarized in the next corollary. It generalizes Corollary 6.6.

[3]However, as for approximations in terms of the number of clusters, it might be more difficult, as indicated by the results in Sections 7.3.2 and 7.6.

Corollary 7.13 *The following are solvable in polynomial time:*

- *HRC restricted to instances with* $k = O(1)$.
- *HRC restricted to instances with* $p = O(1)$ *and* $n = O(\log k)$.
- *HRC restricted to instances with* $p = O(1)$ *and* $r = O(1)$.
- *HDC restricted to instances with* $p = O(1)$ *and* $k = O(\log n)$.
- *HDC restricted to instances with* $p = 2$.

7.5 Approximation Algorithms for HRC & HDC

In this section, we describe three approximation algorithms. The first one works for both HRC and HDC, whereas the second and third are more specialized and work only for HRC.

7.5.1 A Polynomial-Time 2-Approximation Algorithm for HRC and HDC

We apply Gonzalez' farthest-point clustering algorithm [56] to HRC and HDC, respectively, as shown in Figure 7.3 to obtain an approximation factor of two for both problems. This algorithm can be viewed as an extension of the 2-approximation algorithm for HRC restricted to $p = 1$ given in Section 6.2.

```
Algorithm   Farthest-Point Clustering
Input:   An instance of HRC or HDC.
Output:  An approximate solution which is within a factor of 2 of the optimum.

1  Set $B$ to $\{\alpha_l\}$, where $\alpha_l$ is an arbitrary string in $S$.
2  for $q := 2$ to $p$ do
       Augment $B$ by a string $\alpha \in S$ that maximizes $\min_{\beta' \in B} d_H(\alpha, \beta')$, i.e.,
       a string that is as far away as possible from the strings already in $B$.
   endfor
3  (HRC) return $B$.
   (HDC) Assign each string in $S$ to a closest member in $B$ and return the
   resulting clusters.
End  Farthest-Point Clustering
```

Figure 7.3: Gonzalez' farthest-point clustering algorithm applied to HRC and HDC.

By Fact 6.1, the Hamming distance function satisfies the triangle inequality. Therefore, by the proof of Theorem 8.14 in [63], Algorithm *Farthest-Point Clustering* yields an approximate solution to HRC or HDC that is always within a factor of two of the optimum.

We can implement the algorithm by updating the Hamming distance from each string outside B to the nearest string in B after each augmentation of B. To update and then compute a string in S farthest from B takes $O(kn)$ time in each iteration; hence, we obtain the next theorem.

Theorem 7.14 *HRC and HDC can be approximated within a factor of two in $O(pkn)$ time.*

Theorem 7.5 implies that if $\mathrm{P} \neq \mathrm{NP}$ then this is the best constant approximation factor for the unrestricted versions of HRC and HDC achievable by any algorithm running in polynomial time. Interestingly, Algorithm *Farthest-Point Clustering* does not even consider any strings outside the input set S.

7.5.2 An Approximation Scheme for HRC

Here, we present an algorithm that for any given constant ε, where $0 < \varepsilon < 1$, approximates HRC within a factor of $(1+\varepsilon)$, and show that its running time is polynomial if $p = O(1)$ and $r = O(\log(k+n))$. The algorithm is partly based on the idea used in the PTAS for HRC restricted to $p = 1$ by Li, Ma, and Wang [91].

Our algorithm is called Algorithm *HRC Approximation Scheme* and is shown in Figure 7.4.

```
Algorithm   HRC Approximation Scheme
Input:   An instance of HRC, positive number $\varepsilon < 1$.
Output:  An approximate solution which is within a factor of $(1+\varepsilon)$ of the
         optimum.

1  Let $R := \min\{\lceil \frac{1+\varepsilon}{2\varepsilon} \rceil, k\}$ and set $\mathcal{C}$ to the empty set.
2  for each subset $S'$ of $S$ with exactly $R$ strings do
         Compute the set $Q$ consisting of all positions $m$, $1 \leq m \leq n$, on which
         all strings in $S'$ contain the same symbol. Set $P$ to $\{1, ..., n\} \setminus Q$.
         For every possible $f : P \to \{0,1\}$, let $q_f$ be the string in $\{0,1\}^n$ which
         agrees with the strings in $S'$ on all positions in $Q$ and contains $f(j)$ on
         each position $j \in P$. Augment $\mathcal{C}$ by $q_f$.
   endfor
3  Let $\mathcal{C}^p$ be the set of all cardinality $p$ subsets of $\mathcal{C}$.
4  Test every set in $\mathcal{C}^p$ and return a $B \in \mathcal{C}^p$ that minimizes $\max_{1 \leq i \leq k} \min_{c \in B} d_H(\alpha_i, c)$.
End HRC Approximation Scheme
```

Figure 7.4: An approximation scheme for HRC.

For instances of HRC with $k \leq \lceil \frac{1+\varepsilon}{2\varepsilon} \rceil$, Algorithm *HRC Approximation Scheme* will find an exact solution since $\mathcal{C}$ contains all possible strings which can belong to a p-center. To prove that the algorithm attains the specified approximation factor for the remaining cases (i.e., when $k > \lceil \frac{1+\varepsilon}{2\varepsilon} \rceil$), we need the next lemma. It follows from Lemma 2.1 in [91] together with the part of the proof of Lemma 2.5 in [91] which uses enumeration to find an optimal completion of the string obtained from Lemma 2.1.

Lemma 7.15 *For any subset U of S, there is a c in $\mathcal{C}$ such that*

$$\max_{\alpha \in U} d_H(\alpha, c) \leq (1 + \frac{1}{2R-1}) \cdot \min_{\beta \in \{0,1\}^n} \max_{\alpha \in U} d_H(\alpha, \beta)$$

Theorem 7.16 *For any given ε, where $0 < \varepsilon < 1$, Algorithm* HRC Approximation Scheme *approximates HRC within a factor of $(1+\varepsilon)$ in $k^{O(p/\varepsilon)} \cdot 2^{O(rp/\varepsilon)} \cdot n$ time.*

Proof: To prove the correctness and the claimed approximation factor of Algorithm *HRC Approximation Scheme*, consider an optimal p-center $\{\beta_1, ..., \beta_p\}$ of S. Partition S into disjoint subsets U_1 through U_p such that for all $1 \leq q \leq p$ and any $\alpha \in U_q$, β_q has minimum Hamming distance to α among $\beta_1, ..., \beta_p$. By Lemma 7.15, the set $\mathcal{C}^p$ constructed in Step 3 contains a set $\{\beta'_1, ..., \beta'_p\}$ such that for all $1 \leq q \leq p$ and any $\alpha \in U_q$, the Hamming distance between α and β'_q is at most $1 + \frac{1}{2R-1}$ times the radius of U_q. Thus, Algorithm *HRC Approximation Scheme* yields a solution within $1 + \frac{1}{2R-1}$ of the optimum. Now, for all instances of HRC with $k > \lceil \frac{1+\varepsilon}{2\varepsilon} \rceil$, we have $R = \lceil \frac{1+\varepsilon}{2\varepsilon} \rceil \geq \frac{1+\varepsilon}{2\varepsilon}$ and $1 + \frac{1}{2R-1} \leq 1 + \varepsilon$. (If $k \leq \lceil \frac{1+\varepsilon}{2\varepsilon} \rceil$ then the algorithm returns an exact solution.)

To derive an upper bound on the running time of Algorithm *HRC Approximation Scheme*, first observe that each of the sets P in Step 2 has size at most $r \cdot R$ so that for each subset S' considered, at most 2^{rR} strings of the form q_f are added to $\mathcal{C}$, taking $O(Rn + 2^{rR}n) = O(2^{rR}n)$ time. Hence, $|\mathcal{C}| \leq k^R 2^{rR}$, and $\mathcal{C}$ can be constructed in $O(k^R 2^{rR} n)$ time. Consequently, $\mathcal{C}^p$ is of size at most $k^{Rp} 2^{rRp}$ and its construction from $\mathcal{C}$ takes $O(k^{Rp} 2^{rRp} n)$ time. Each set in $\mathcal{C}^p$ can be tested in $O(kpn)$ time. The total running time is therefore $O(k^{Rp+1} \cdot 2^{rRp} \cdot p \cdot n)$, which is bounded by $k^{O(p/\varepsilon)} \cdot 2^{O(rp/\varepsilon)} \cdot n$ since $R \leq \lceil \frac{1+\varepsilon}{2\varepsilon} \rceil < \frac{1+\varepsilon}{2\varepsilon} + 1 = \frac{1+3\varepsilon}{2\varepsilon} < \frac{2}{\varepsilon}$ and $p \leq k$. □

Corollary 7.17 *Algorithm* HRC Approximation Scheme *yields a polynomial-time approximation scheme for HRC restricted to instances with $p = O(1)$ and $r = O(\log(k+n))$.*

7.5.3 A Randomized PTAS for HRC with $p = O(1)$

Ostrovsky and Rabani [101] provided a randomized polynomial-time approximation scheme for the Hamming p-median clustering problem (see Section 7.1.2) restricted to $p = O(1)$. In this subsection, we show that a modification to the evaluation phase of their algorithm makes the algorithm work for HRC, too. To be more precise, we apply the PTAS of Li, Ma, and Wang [91] for HRC restricted to $p = 1$ to obtain a randomized PTAS for HRC restricted to $p = O(1)$ whose success probability increases with increasing k.

Preliminaries

Let $\mathcal{A}_{n,m}(q)$ be the probability distribution on the $(m \times n)$-matrices over $\{0,1\}$ in which the entries are independent, identically distributed random $0-1$-variables with $\mathbf{Pr}[1] = q$. For any $(m \times n)$-matrix A over $\{0,1\}$ (henceforth also referred to as a *linear transformation from* $\{0,1\}^n$ *to* $\{0,1\}^m$) and vector $x \in \{0,1\}^n$, Ax is the vector in $\{0,1\}^m$ obtained by multiplying A and x (modulo 2). Ostrovsky and Rabani [101] proved the following.

Lemma 7.18 [101] *For every $\gamma > 0$ there exists a $\lambda > 0$ such that for every ε, $0 < \varepsilon \leq \frac{1}{8}$, and all integers k, n, and l with $l \in \{1, ..., n\}$, the following holds:*

> *Let $S \subseteq \{0,1\}^n$ with $|S| = k$. Let $m = \lambda \ln k/\varepsilon^4$, and let A be a random matrix drawn from $\mathcal{A}_{n,m}(\varepsilon^2/l)$. Then with probability at least $1 - k^{-\gamma}$, for all $x, y, z \in S$ with $l \leq d_H(y,z) \leq 2l$, if $d_H(Ax, Ay) \leq d_H(Ax, Az)$ then $d_H(x,y) \leq (1 + 8\varepsilon) \cdot d_H(x,z)$.*

Definition 7.19 A *tournament* is a directed graph in which there is exactly one directed edge between each pair of vertices. An *apex* of a tournament is a vertex of maximum outdegree.

The next lemma was stated without proof in [101].

Lemma 7.20 *Let a be an apex of a tournament T, and let c be any vertex in T. Then there is a directed path of length at most 2 from a to c.*

Proof: Denote the maximum outdegree of T by M. Let B be the set of vertices of T which are reachable by following one directed edge from a, i.e., $|B| = M$.

Assume that there is no directed path of length ≤ 2 from a to c. Then each directed edge between c and a vertex in $\{a\} \cup B$ must originate from c, which means that the outdegree of c is at least $M + 1$. This contradicts the maximality of a. □

The algorithm

The algorithm is called Algorithm *HRC Randomized PTAS* and is listed in Figure 7.5. It is substantially based on the approach used in [101] for approximating the Hamming p-median clustering problem. The only difference between the PTAS of [101] and Algorithm *HRC Randomized PTAS* is the evaluation phase;

```
Algorithm    HRC Randomized PTAS
Input:   An instance of HRC with $p = O(1)$, positive constants $\gamma$, $\varepsilon$, and $f$.
Output:  An approximate solution which is within a factor of $(1+8\varepsilon)^2 \cdot (1+f)$
         of the optimum with probability at least $1 - 2 \cdot k^{-\gamma}$.

  Set $\lambda$ to the constant in Lemma 7.18 (depends on $\gamma$).
  $m := \lambda \cdot \ln(k+p)/\varepsilon^4$
  for each $l \in \{1,...,n\}$ do
      Draw a random matrix from $\mathcal{A}_{n,m}(\varepsilon^2/l)$ and call it $A^{(l)}$.
  endfor
  for all $(l_{s,t})_{1 \le s < t \le p} \in \{1,...,n\}^{\binom{p}{2}}$ do
      for all choices of $(c_{i,j})_{1 \le i,j \le p}^{i \ne j} \in (\{0,1\}^m)^{p(p-1)}$ do
          $(S_1,...,S_p) := (\emptyset,...,\emptyset)$
          for $x \in S$ do
              Construct a tournament $T$ over vertex set $\{1,...,p\}$ as follows:
              for $1 \le i < j \le p$ do
                  if $d_H(A^{(l_{i,j})}x, c_{i,j}) \le d_H(A^{(l_{i,j})}x, c_{j,i})$ then
                      Let $ij$ be an edge of $T$.
                  else
                      Let $ji$ be an edge of $T$.
                  endif
              endfor
              Compute an apex $a$ of $T$.
              $S_a := S_a \cup \{x\}$
          endfor
          for $q := 1$ to $p$ do
              $B'_q := PTAS(S_q, f)$
          endfor
          cost $:= \max_{1 \le q \le p} \{\max_{\alpha \in S_q} d_H(\alpha, B'_q)\}$
      endfor
  endfor
  return The $\{B'_1,...,B'_p\}$ which induces the smallest cost.
End  HRC Randomized PTAS
```

Figure 7.5: The randomized PTAS of Ostrovsky and Rabani, modified to approximate HRC restricted to $p = O(1)$.

whereas it is easy to compute the exact cost of each candidate clustering for the Hamming p-median problem, it is NP-hard for HRC (see below).

To understand the general idea behind the algorithm, first consider the case $p = 2$. The initial part of the algorithm generates a set of n linear transformations from $\{0,1\}^n$ to $\{0,1\}^m$, where $m = O(\log k)$, which are applied to the input strings later on to reduce the number of dimensions. (Note that the elements of $\{0,1\}^m$ can be enumerated in polynomial time.) In the main loop, for every possible projection $(c_{1,2}, c_{2,1})$ in $(\{0,1\}^m)^2$ of a 2-center in $(\{0,1\}^n)^2$, the original strings are partitioned into two sets (S_1, S_2); a string is placed in S_1 or S_2 depending on if its projection is closer to the string $c_{1,2}$ or the string $c_{2,1}$, with ties broken arbitrarily. By enumerating over all elements in $(\{0,1\}^m)^2$, some $(\tilde{c}_{1,2}, \tilde{c}_{2,1})$ in the reduced space which is the image of an optimal solution (β_1, β_2) in the original space will be encountered. Denote the partition of S induced by $(\tilde{c}_{1,2}, \tilde{c}_{2,1})$ by $(\tilde{S}_1, \tilde{S}_2)$. Now, if the linear transformation would preserve all relations between pairwise distances between strings, every string in $\tilde{S}_1$ would be closer to β_1 than to β_2. This is not true in general though, but Lemma 7.18 guarantees that we can select a linear transformation from $\mathcal{A}_{n,m}(\varepsilon^2/l_{1,2})$ at random, where $l_{1,2}$ is the distance between β_1 and β_2, so that for any string x which is placed in $\tilde{S}_1$, the distance between x and β_1 is at most $(1+8\varepsilon)$ times the distance between x and β_2 with high probability. Thus, even if x is placed in the "wrong" set, this does not worsen the quality of the approximation too much. Since the actual distance between β_1 and β_2 is unknown, all possible values for $l_{1,2}$ are tried.

For $p > 2$, the algorithm enumerates over all possible pairwise distances in the original space between cluster centers as well as over all elements in $(\{0,1\}^m)^{p(p-1)}$. For some choice of distances and some element $(\tilde{c}_{1,2}, \tilde{c}_{1,3}, ..., \tilde{c}_{p,p-2}, \tilde{c}_{p,p-1})$ in $(\{0,1\}^m)^{p(p-1)}$, each string $\tilde{c}_{i,j}$ corresponds to the projection in $\{0,1\}^m$ of β_i using the linear transformation associated with the length $d_H(\beta_i, \beta_j)$, where $\{\beta_1, ..., \beta_p\}$ is an optimal p-center in $\{0,1\}^n$. (For a linear transformation $A^{(l)}$, Lemma 7.18 can be applied to pairs of cluster centers (β_i, β_j) which satisfy $l \leq d_H(\beta_i, \beta_j) \leq 2l$. Thus, one linear transformation is employed for each pair of cluster centers, i.e., $\binom{p}{2}$ linear transformations at a time.) To decide in which clusters to place the input strings, a tournament T among cluster centers is constructed for every $x \in S$; for each pair of centers (i,j), the edge between vertices i and j in T is directed away from the center whose projection is closer to the projection of x. Then, x is assigned to a cluster S_q only if no vertex in T has more outgoing edges than vertex q. Again, for the correct choice of distances between cluster centers, Lemma 7.18 ensures that with high probability, pairwise distances in the reduced spaces are not greatly distorted. Assume that x is placed in $\tilde{S}_a$ but that the cluster center which is closest to x is β_c. By Lemma 7.20, there is a path of length at most two from a to c in T, so at most two applications of Lemma 7.18 are needed to obtain an upper bound on $d_H(x, \beta_a)$ in terms of $d_H(x, \beta_c)$.

Since the algorithm enumerates over many possible solutions in order to find the best one, a method for evaluating the quality of a proposed clustering is also required. Unfortunately, it is NP-hard to calculate the 1-radius (in the original space) of each cluster $\tilde{S}_q$ [46]. Furthermore, the algorithm should output p binary strings that approximate the p-center of the instance. Therefore, for each computed partition of S into $(S_q)_{q=1}^{p}$, we run the PTAS for HRC with $p = 1$ by Li, Ma, and Wang [91] p times. For any constant $f > 0$ and $S' \subseteq S$,

$PTAS(S', f)$ returns an approximate 1-center in polynomial time whose distance to every string in S' is less than or equal to the 1-radius of S' multiplied by $(1 + f)$.

Algorithm analysis

If p, f, γ, and ε are constant, the total running time of Algorithm *HRC Randomized PTAS* is polynomial since there are $O(n^{p(p-1)/2})$ sets of pairwise distances to try, the number of choices for $(c_{i,j})_{1 \le i,j \le p}^{i \neq j}$ is bounded by $O(2^{m \cdot p(p-1)}) = O(k^C)$ where C is a constant less than $p(p-1)\lambda/\varepsilon^4$, each tournament can be constructed in polynomial time, and each call to $PTAS(S, f)$ takes polynomial time when f is constant.

Next, we show that every solution returned by the algorithm is close to the optimum with high probability, despite the inexactness introduced by the projection from $\{0,1\}^n$ into $\{0,1\}^m$ and the approximations of 1-centers.

Theorem 7.21 *Let f be a constant, $0 < f < 1$. For every $\gamma > 0$ and ε, where $0 < \varepsilon \le \frac{1}{8}$, Algorithm* HRC Randomized PTAS *approximates HRC restricted to $p = O(1)$ within a factor of $(1 + 8\varepsilon)^2 \cdot (1 + f)$ with probability at least $1 - 2 \cdot k^{-\gamma}$ in polynomial time.*

Proof: Let λ be the constant in Lemma 7.18, let $\{\beta_1, ..., \beta_p\}$ be an optimal p-center set for the given instance, and consider the iteration of the algorithm in which $l_{s,t} = d_H(\beta_s, \beta_t)$ for all $1 \le s < t \le p$. One of the choices of $(c_{i,j})_{1 \le i,j \le p}^{i \neq j}$ tried by the algorithm is when each $c_{i,j}$ is precisely $A^{(l_{i,j})}\beta_i$ (where $l_{i,j}$ is set to equal $l_{j,i}$ for $j < i$); denote the partition of S obtained for this $(c_{i,j})_{1 \le i,j \le p}^{i \neq j}$ by $(\tilde{S}_1, ..., \tilde{S}_p)$.

By Lemma 7.18, it holds that for any $x \in S$, if $d_H(A^{(l_{i,j})}x,\, A^{(l_{i,j})}\beta_i) \le d_H(A^{(l_{i,j})}x,\, A^{(l_{i,j})}\beta_j)$ then $d_H(x, \beta_i) \le (1 + 8\varepsilon) \cdot d_H(x, \beta_j)$ with probability at least $1 - k^{-\gamma}$. Suppose that the algorithm places x in some cluster $\tilde{S}_a$, and that β_c is closest to x among $\{\beta_1, ..., \beta_p\}$. Since a is an apex of T, Lemma 7.20 implies that there is a path in T from a to c of length at most 2. If the path has length 2, let b be its middle vertex. By the construction of T, $d_H(A^{(l_{a,b})}x,\, A^{(l_{a,b})}\beta_a) \le d_H(A^{(l_{a,b})}x,\, A^{(l_{a,b})}\beta_b)$ and $d_H(A^{(l_{b,c})}x,\, A^{(l_{b,c})}\beta_b) \le d_H(A^{(l_{b,c})}x,\, A^{(l_{b,c})}\beta_c)$. Otherwise, if the path has length 1, set b equal to c; if the path has length 0, then let $a = b = c$. Thus, with probability at least $1 - (k^{-\gamma} + k^{-\gamma})$ we have that

$$d_H(x, \beta_a) \;\le\; (1 + 8\varepsilon) \cdot d_H(x, \beta_b) \;\le\; (1 + 8\varepsilon)^2 \cdot d_H(x, \beta_c).$$

For each $1 \le q \le p$, let $\tilde{B}_q$ be an optimal 1-center for $\tilde{S}_q$ and let $\tilde{B}'_q$ be the approximate 1-center computed by $PTAS(S_q, f)$. Because

$$\max_{x \in \tilde{S}_q} d_H(x, \tilde{B}'_q) \;\le\; (1 + f) \cdot \max_{x \in \tilde{S}_q} d_H(x, \tilde{B}_q) \;\le\; (1 + f) \cdot \max_{x \in \tilde{S}_q} d_H(x, \beta_q),$$

the solution returned by Algorithm *HRC Randomized PTAS* has a cost which is less than or equal to

$$\begin{aligned}\max_{1\le q\le p}\max_{x\in\tilde{S}_q} d_H(x,\tilde{B}'_q) &\le (1+f)\cdot\max_{1\le q\le p}\max_{x\in\tilde{S}_q} d_H(x,\beta_q)\\ &\le (1+f)\cdot(1+8\varepsilon)^2\cdot\max_{x\in S}\min_{1\le q\le p} d_H(x,\beta_q)\end{aligned}$$

with probability at least $1-2\cdot k^{-\gamma}$. □

7.6 A Relaxed Type of Approximation of HRC

We now show how to find, for any given constant ε with $0<\varepsilon<1$, a set L of at most $O(p\log k)$ strings of length n such that for each string in S there is at least one string in L within distance $(1+\varepsilon)\cdot r$. This yields a twofold approximation algorithm for HRC with $O((k\cdot 2^r)^{2/\varepsilon}\cdot(n+k)\cdot k\cdot\log n)$ running time.

Let ε be a given constant, $0<\varepsilon<1$. Given an instance of HRC, compute R and the set $\mathcal{C}$ as in Algorithm *HRC Approximation Scheme* in Section 7.5.2. If $k\le\lceil\frac{1+\varepsilon}{2\varepsilon}\rceil$ then for every c in $\mathcal{C}$, define $S(c)$ to be the set of all strings in S within Hamming distance r of c; there exists a set of p such sets that covers all of S because $\mathcal{C}$ contains an exact p-center. If $k>\lceil\frac{1+\varepsilon}{2\varepsilon}\rceil$ then for every c in $\mathcal{C}$, define $S(c)$ to be the set of all strings in S within Hamming distance $(1+\frac{1}{2R-1})\cdot r$ of c; by Lemma 7.15, there exists a set consisting of p such sets, covering all of S. Also note that by the proof of Theorem 7.16, if $k>\lceil\frac{1+\varepsilon}{2\varepsilon}\rceil$ then $\frac{1}{2R-1}\le\varepsilon$.

Now, if r is known, construct $S(c)$ for all $c\in\mathcal{C}$ and run the classical greedy approximation algorithm for minimum set cover (see [63] or [128]) on the instance $\big(S,\ \{S(c)\,|\,c\in\mathcal{C}\}\big)$ to find a set of at most $(1+\ln k)\cdot p$ sets of the form $S(c)$ covering S, and return the corresponding elements of $\mathcal{C}$ as the solution. Otherwise, perform a binary search to find the smallest possible r; for each candidate value of r, construct a new instance of minimum set cover by recomputing the sets $S(c)$ using this value of r, run the greedy minimum set cover algorithm, and test whether the size of the resulting cover is $\le(1+\ln k)\cdot p$.

Recall from the proof of Theorem 7.16 that $|\mathcal{C}|\le k^R 2^{rR}$ and that $\mathcal{C}$ can be constructed in $O(k^R 2^{rR} n)$ time. The instance of minimum set cover for a given value of r can be constructed in $O(|\mathcal{C}|kn)$ time and the greedy minimum set cover algorithm can be implemented to run in $O(|\mathcal{C}|k^2)$ time. Since the binary search for the optimal value of r takes $O(\log n)$ iterations and $R<\frac{2}{\varepsilon}$, we obtain the following theorem.

Theorem 7.22 *For any constant* $0<\varepsilon<1$, *in* $O((k\cdot 2^r)^{2/\varepsilon}\cdot(n+k)\cdot k\cdot\log n)$ *time we can find a set* L *of at most* $(1+\ln k)\cdot p$ *strings such that each string in* S *is within Hamming distance* $(1+\varepsilon)\cdot r$ *of at least one string in* L.

The time bound in Theorem 7.22 is polynomial in n and k for every fixed ε as long as $r=O(\log(k+n))$.

7.7 Concluding Remarks

In this chapter, we have proved that the unrestricted versions of HRC and HDC cannot be approximated within any constant factor less than two in polynomial time, unless P=NP (Theorem 7.5). On the other hand, an approximation factor of two in polynomial time is always achievable for both problems (Theorem 7.14).

We have also proved that it is possible to compute exact solutions or approximate solutions with better approximation factors than two in polynomial time for several restrictions of HRC and HDC, as summarized below. Arrows show when a more general algorithm can be used on a restricted case (for example, HDC with $p = 2$ and $k = O(\log n)$ can be solved exactly in polynomial time either according to Theorem 7.11 or Theorem 7.12).

HRC with $p = O(1)$	$r = O(1)$	$r = O(\log(k + n))$	r unrestricted
$n = O(\log k)$	Exact solution ($\downarrow$ or $\rightarrow$)	Exact solution ($\rightarrow$)	Exact solution (Theorem 7.9)
n unrestricted	Exact solution (Theorem 7.10)	PTAS (Corollary 7.17)	Randomized PTAS (Theorem 7.21)

HDC	$p = 2$	$p = O(1)$	p unrestricted
$k = O(\log n)$	Exact solution ($\downarrow$ or $\rightarrow$)	Exact solution (Theorem 7.11)	2-approximation ($\downarrow$)
k unrestricted	Exact solution (Theorem 7.12)	2-approximation ($\rightarrow$)	2-approximation (Theorem 7.14)

In addition to the above, HRC with $k = O(1)$ and unrestricted p is solvable exactly in polynomial time (Theorem 7.8).

Furthermore, we have proved that HRC with $r = O(\log(k + n))$ and unrestricted p can be approximated within a factor of $(1+\varepsilon)$ for any constant $\varepsilon > 0$ in polynomial time if the number of output strings is allowed to increase by a factor of at most $(1 + \ln k)$ (Theorem 7.22). Relaxing the number of allowed clusters in an approximate solution to HDC seems less likely to help, though, since it is NP-hard for any constant $\varepsilon > 0$ to split the input strings into $\leq pk^{1/7-\varepsilon}$ disjoint clusters whose 1-diameters do not exceed the p-diameter of the given instance (Theorem 7.6).

The reduction we employed to prove Theorem 7.6 and Corollary 7.7 is general enough to be of use for proving hardness results for other types of Hamming clustering problems as well. For example, consider *the Hamming p-sum of diameters clustering problem* (HSC) which is defined on the same set of instances as HRC and HDC and where the objective is to partition S into p disjoint subsets

$S_1, ..., S_p$ so that the value of $\sum_{q=1}^{p} \max_{\alpha_i,\alpha_j \in S_q} d_H(\alpha_i, \alpha_j)$ is minimized. We can prove that HSC is NP-hard by reducing from the clique problem (given an undireced graph G and a positive integer C, does G have a clique of size C?) as follows[4]: Given $G = (V, E)$ and C, construct G' and a set S of $|V|$ binary strings of length $O(|V|^2)$ as in the proof of Theorem 7.6. Set $p = |V| - C + 1$. Recall that for any pair of vertices v_1, v_2 in G, the Hamming distance between their encoding strings $s(v_1)$ and $s(v_2)$ is $2|V| - 2$ if v_1 and v_2 are adjacent, otherwise it is $2|V|$. Therefore, if G has a clique of size C then S can be partitioned into p subsets whose sum of diameters is $2|V| - 2$ (1 subset with diameter $2|V| - 2$ and $|V| - C$ subsets with diameter 0), and if G has no clique of size C then for any partition of S into p subsets, the sum of diameters must be strictly greater than $2|V| - 2$.

We remark that HSC can be approximated within a constant factor in polynomial time with the algorithm of Charikar and Panigrahy [24] and that HSC restricted to $p = 2$ can be solved exactly in polynomial time with the algorithm of Hansen and Jaumard [59]. The computational complexity status of HSC restricted to $p = O(1)$ is still open; we conjecture that HSC is NP-hard for every fixed $p \geq 3$.

We now discuss some other open questions.

The PTAS of Li, Ma, and Wang [91] for HRC restricted to $p = 1$ does not seem readily adaptable to $p = O(1)$. By Theorem 7.21, there exists a randomized PTAS for this case, but it is an open question whether a *deterministic* PTAS can be constructed. If not, is there any (deterministic) polynomial-time approximation algorithm for HRC restricted to $p = O(1)$ with a better approximation factor than two at all? Also, since the smallest value of p which makes HDC NP-hard is 3 (Corollary 7.7 and Theorem 7.12), we would like to know if there exists any polynomial-time approximation algorithm for HDC restricted to $p = 3$ with a better approximation factor than two.

Is it possible to design more efficient approximation algorithms for HRC and HDC by taking into account the specific distribution of the input? Such algorithms might be useful in practical applications related to computing unbiased representatives, e.g., for protein data (see Sections 6.1.1 and 7.1.1).

Another problem worthy of closer examination is the following simultaneous generalization of HRC and the closest substring problem defined in Section 6.1.2: Given a set $S = \{\alpha_1, ..., \alpha_k\}$ of binary strings of length n and a positive integer L with $L \leq n$, output a set of strings $\{\beta_1, ..., \beta_p\} \subseteq \{0, 1\}^L$

[4]Compare this reduction to the reduction used in [35] to prove that the p-sum of diameters clustering problem on a complete, undirected graph whose edge weights satisfy the triangle inequality is NP-hard to approximate within $2 - \varepsilon$ for any $\varepsilon > 0$.

minimizing r such that for every $\alpha_i \in S$, there exists a length L substring γ_i of α_i with $\min_{1 \le q \le p} d_H(\gamma_i, \beta_q) \le r$. All we currently know is that this problem has to be at least as hard as both HRC and the closest substring problem.

Bibliography

[1] P. K. Agarwal and C. M. Procopiuc. Exact and approximation algorithms for clustering. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'98), pages 658–667, 1998. (Cited on pp. 119, 120.)

[2] P. K. Agarwal and M. Sharir. Efficient algorithms for geometric optimization. *ACM Computing Surveys*, 30(4):412–458, 1998. (Cited on pp. 106, 119, 120.)

[3] 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(3):405–421, 1981. (Cited on pp. 10, 11, 13, 15, 16, 28, 47.)

[4] T. Akutsu and M. M. Halldórsson. On the approximation of largest common subtrees and largest common point sets. *Theoretical Computer Science*, 233(1–2):33–50, 2000. (Cited on pp. 56, 57.)

[5] J. Alber, J. Gramm, and R. Niedermeier. Faster exact algorithms for hard problems: A parameterized point of view. *Discrete Mathematics*, 229:3–27, 2001. (Cited on p. 55.)

[6] 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. A preliminary version appeared in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science* (FOCS'94), pages 758–769, 1994. (Cited on pp. 54, 55, 69, 70.)

[7] P. Arabie, L. J. Hubert, and G. De Soete, editors. *Clustering and Classification.* World Scientific Publishing Co. Pte. Ltd., 1996. (Cited on p. 119.)

[8] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science* (FOCS'92), pages 14–23, 1992. (Cited on p. 29.)

[9] U. Asklund. *Configuration Management for Distributed Development in an Integrated Environment.* PhD thesis, Lund University, 2002. ISSN 1404-1219, Dissertation 14, LU-CS-DISS:2002-1. (Cited on p. 76.)

[10] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer-Verlag, Berlin, 1999. (Cited on pp. 2, 3, 74, 103, 119, 120.)

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the IEEE International Conference on Software Maintenance* (ICSM 1998), pages 368–377, 1998. (Cited on p. 77.)

[12] M. Bellare, O. Goldreich, and M. Sudan. Free bits, PCPs, and nonapproximability – towards tight results. *SIAM Journal on Computing*, 27(3):804–915, 1998. (Cited on p. 126.)

[13] A. Ben-Dor, G. Lancia, J. Perone, and R. Ravi. Banishing bias from consensus sequences. In *Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching* (CPM'97), volume 1264 of *Lecture Notes in Computer Science*, pages 247–261. Springer-Verlag Berlin Heidelberg, 1997. (Cited on pp. 103, 105.)

[14] K. Benchemsi. Delgrafsisomorfi i en DAG med namngivna noder (in Swedish). Master's thesis, Lund University, 2000. CODEN: LUNDFD6/NFCS-5176/1–29/2000. (Cited on pp. 76, 77.)

[15] C. Bennett, M. Li, and B. Ma. Linking chain letters. *Scientific American*, to appear. (Cited on p. 12.)

[16] T. Berger. *Rate Distortion Theory.* Prentice-Hall, New Jersey, 1971. (Cited on p. 103.)

[17] P. Berman. A $d/2$ approximation for maximum weight independent set in d-claw free graphs. *Nordic Journal of Computing*, 7(3):178–184, 2000. (Cited on p. 68.)

[18] M. Bonet, C. Phillips, T. Warnow, and S. Yooseph. Constructing evolutionary trees in the presence of polymorphic characters. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing* (STOC'96), pages 220–229, 1996. (Cited on p. 12.)

[19] P. Bonizzoni, G. Della Vedova, and G. Mauri. Approximating the maximum isomorphic agreement subtree is hard. *International Journal of Foundations of Computer Science*, 11(4):579–590, 2000. (Cited on p. 56.)

[20] A. Bookstein and S. T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991. (Cited on p. 119.)

[21] D. Bryant. *Building Trees, Hunting for Trees, and Comparing Trees: Theory and Methods in Phylogenetic Analysis.* PhD thesis, University of Canterbury, Christchurch, New Zealand, 1997. (Cited on pp. 4, 13, 14, 19, 47, 55.)

[22] P. Cameron. *Combinatorics: Topics, Techniques, Algorithms.* Cambridge University Press, 1994. (Cited on p. 102.)

[23] The Canterbury Tales Project. De Montfort University, University of Oxford, Harvard University, University of Leeds, University of Sheffield, Brigham Young University, Virginia Polytechnic Institute & State University (Virginia Tech), and University of Münster. Website: `http://www.shef.ac.uk/uni/projects/ctp/` (Cited on p. 12.)

[24] M. Charikar and R. Panigrahy. Clustering to minimize the sum of cluster diameters. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing* (STOC'01), pages 1–10, 2001. (Cited on pp. 119, 139.)

[25] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD 1996), pages 493–504, 1996. (Cited on p. 76.)

[26] S.-W. Cheng and R. Janardan. Efficient maintenance of the union of intervals on a line, with applications. *Journal of Algorithms*, 12(1):57–74, 1991. (Cited on p. 33.)

[27] G. Cohen, M. Karpovsky, H. F. Mattson, Jr., and J. Schatz. Covering radius – survey and recent results. *IEEE Transactions on Information Theory*, IT-31(3):328–343, 1985. (Cited on pp. 102, 103.)

[28] 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. (Cited on pp. 54, 55.)

[29] L. J. Collins, V. Moulton, and D. Penny. Use of RNA secondary structure for studying the evolution of RNase P and RNase MRP. *Journal of Molecular Evolution*, 51(3):194–204, 2000. (Cited on p. 75.)

[30] J. H. Conway and N. J. A. Sloane. Fast quantizing and decoding and algorithms for lattice quantizers and codes. *IEEE Transactions on Information Theory*, IT-28(2):227–231, 1982. (Cited on p. 103.)

[31] J. H. Conway and N. J. A. Sloane. Voronoi regions of lattices, second moments of polytopes, and quantization. *IEEE Transactions on Information Theory*, IT-28(2):211–226, 1982. (Cited on p. 103.)

[32] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* The MIT Press, Massachusetts, 1990. (Cited on pp. 2, 3, 43.)

[33] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications (Second Edition).* Springer-Verlag, Berlin, 2000. (Cited on pp. 35, 89.)

[34] X. Deng, G. Li, Z. Li, B. Ma, and L. Wang. A PTAS for distinguishing (sub)string selection. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming* (ICALP 2002), volume 2380 of *Lecture Notes in Computer Science*, pages 740–751. Springer-Verlag Berlin Heidelberg, 2002. (Cited on p. 105.)

[35] S. Doddi, M. V. Marathe, S. S. Ravi, D. S. Taylor, and P. Widmayer. Approximation algorithms for sum clustering to minimize the sum of diameters. *Nordic Journal of Computing*, 7(3):185–203, 2000. (Cited on pp. 119, 120, 139.)

[36] L. Engebretsen and J. Holmerin. Clique is hard to approximate within $n^{1-o(1)}$. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming* (ICALP 2000), volume 1853 of *Lecture Notes in Computer Science*, pages 2–12. Springer-Verlag Berlin Heidelberg, 2000. (Cited on p. 61.)

[37] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *Journal of the ACM*, 44(5):669–696, 1997. (Cited on p. 18.)

[38] M. D. Ermolaeva, H. G. Khalak, O. White, H. O. Smith, and S. L. Salzberg. Prediction of transcription terminators in bacterial genomes. *Journal of Molecular Biology*, 301(1):27–33, 2000. (Cited on p. 76.)

[39] S. Even and Y. Shiloach. An on-line edge-deletion problem. *Journal of the ACM*, 28(1):1–4, 1981. (Cited on p. 18.)

[40] M. Farach, T. Przytycka, and M. Thorup. On the agreement of many trees. *Information Processing Letters*, 55:297–301, 1995. A preliminary version appeared in *Proceedings of the 3rd Annual European Symposium on Algorithms* (ESA'95), volume 979 of *Lecture Notes in Computer Science*, pages 381–393. Springer-Verlag Berlin Heidelberg, 1995. (Cited on pp. 52, 54, 55.)

[41] M. Farach and M. Thorup. Fast comparison of evolutionary trees. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'94), pages 481–488, 1994. (Cited on pp. 55, 62, 149.)

[42] M. Farach and M. Thorup. Sparse dynamic programming for evolutionary-tree comparison. *SIAM Journal on Computing*, 26(1):210–230, 1997. (Cited on pp. 54, 55.)

[43] T. Feder and D. H. Greene. Optimal algorithms for approximate clustering. In *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing* (STOC'88), pages 434–444, 1988. (Cited on pp. 119, 120, 123, 124.)

[44] J. Felsenstein. PHYLIP – Phylogeny Inference Package (version 3.2). *Cladistics*, 5:164–166, 1989. See also `http://evolution.genetics.washington.edu/phylip.html` (Cited on p. 13.)

[45] C. R. Finden and A. D. Gordon. Obtaining common pruned trees. *Journal of Classification*, 2:255–276, 1985. (Cited on pp. 54, 55.)

[46] M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997. (Cited on pp. 101, 102, 103, 104, 120, 121, 127, 135.)

[47] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM Journal on Computing*, 14(4):781–798, 1985. (Cited on p. 18.)

[48] Z. Galil and N. Megiddo. Cyclic ordering is NP-complete. *Theoretical Computer Science*, 5(2):179–182, 1977. (Cited on p. 20.)

[49] M. Garey and D. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979. (Cited on pp. 2, 20, 64, 70, 127.)

[50] L. Gąsieniec, J. Jansson, and A. Lingas. Efficient approximation algorithms for the Hamming center problem. Technical Report LU-CS-TR:99-211, Lund University, 1999. A short form version of this article was published in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), pages S905–S906, 1999. (Cited on pp. 5, 105, 106.)

[51] L. Gąsieniec, J. Jansson, and A. Lingas. Approximation algorithms for Hamming clustering problems. *Journal of Discrete Algorithms*, to appear. A preliminary version appeared in *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching* (CPM 2000), volume 1848 of *Lecture Notes in Computer Science*, pages 108–118. Springer-Verlag Berlin Heidelberg, 2000. (Cited on pp. 5, 6.)

[52] L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin. Inferring ordered trees from local constraints. In *Proceedings of Computing: the 4th Australasian Theory Symposium* (CATS'98), volume 20(3) of *Australian Computer Science Communications*, pages 67–76. Springer-Verlag Singapore, 1998. (Cited on p. 4.)

[53] L. Gąsieniec, J. Jansson, A. Lingas, and A. Östlin. On the complexity of constructing evolutionary trees. *Journal of Combinatorial Optimization*, 3(2–3):183–197, 1999. A preliminary version appeared in *Proceedings of the 3rd Annual International Computing and Combinatorics Conference* (COCOON'97), volume 1276 of *Lecture Notes in Computer Science*, pages 134–145. Springer-Verlag Berlin Heidelberg, 1997. (Cited on pp. 4, 5, 19.)

[54] W. Goddard, E. Kubicka, G. Kubicki, and F. R. McMorris. The agreement metric for labeled binary trees. *Mathematical Biosciences*, 123:215–226, 1994. (Cited on pp. 54, 55, 149.)

[55] W. Goddard and G. Kubicki. The minimum size of agreement subtrees of two binary trees. *Congressus Numerantium*, 97:131–136, 1993. (Cited on p. 56.)

[56] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985. (Cited on pp. 105, 107, 119, 120, 121, 130.)

[57] J. Gramm, R. Niedermeier, and P. Rossmanith. Exact solutions for Closest String and related problems. In *Proceedings of the 12th Annual International Symposium on Algorithms and Computation* (ISAAC 2001), volume 2223 of *Lecture Notes in Computer Science*, pages 441–453. Springer-Verlag Berlin Heidelberg, 2001. (Cited on pp. 102, 105.)

[58] D. Gusfield. *Algorithms on Strings, Trees, and Sequences : Computer Science and Computational Biology.* Cambridge University Press, 1997. (Cited on pp. 74, 78, 79, 80, 96, 97, 103.)

[59] P. Hansen and B. Jaumard. Minimum sum of diameters clustering. *Journal of Classification*, 4:215–226, 1987. (Cited on pp. 119, 139.)

[60] J. Hein, T. Jiang, L. Wang, and K. Zhang. On the complexity of comparing evolutionary trees. *Discrete Applied Mathematics*, 71:153–169, 1996. (Cited on pp. 55, 56, 57, 61, 69, 78.)

[61] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM*, 46(4):502–516, 1999. (Cited on p. 18.)

[62] M. R. Henzinger, V. King, and T. Warnow. Constructing a tree from homeomorphic subtrees, with applications to computational evolutionary biology. *Algorithmica*, 24(1):1–13, 1999. (Cited on pp. 14, 15, 16, 19, 28, 34, 40, 47.)

[63] D. S. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Company, Boston, 1997. (Cited on pp. 3, 109, 118, 119, 120, 124, 130, 137.)

[64] J. Hochberg, L. Kerns, P. Kelly, and T. Thomas. Automatic script identification from images using cluster-based templates. In *Proceedings of the 3rd International Conference on Document Analysis and Recognition* (ICDAR'95), pages 378–381, 1995. (Cited on p. 119.)

[65] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM*, 48(4):723–760, 2001. (Cited on pp. 19, 28, 47.)

[66] J. Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999. (Cited on pp. 61, 70.)

[67] J. Jansson. On the complexity of inferring rooted evolutionary trees. In *Proceedings of the Brazilian Symposium on Graphs, Algorithms, and Combinatorics* (GRACO 2001), volume 7 of *Electronic Notes in Discrete Mathematics*, pages 121–125. Elsevier, 2001. (Cited on pp. 4, 19.)

[68] J. Jansson and A. Lingas. A fast algorithm for optimal alignment between similar ordered trees. *Fundamenta Informaticae*, to appear. A preliminary version appeared in *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching* (CPM 2001), volume 2089 of *Lecture Notes in Computer Science*, pages 232–240. Springer-Verlag Berlin Heidelberg, 2001. (Cited on p. 5.)

[69] T. Jiang, P. Kearney, and M. Li. A polynomial time approximation scheme for inferring evolutionary trees from quartet topologies and its application. *SIAM Journal on Computing*, 30(6):1942–1961, 2001. (Cited on p. 29.)

[70] T. Jiang, L. Wang, and K. Zhang. Alignment of trees – an alternative to tree edit. *Theoretical Computer Science*, 143(1):137–148, 1995. A preliminary version appeared in *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching* (CPM'94), volume 807 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag Berlin Heidelberg, 1994. (Cited on pp. 5, 73, 74, 77, 78, 79, 80, 81, 82, 95, 96.)

[71] S. Kannan, E. Lawler, and T. Warnow. Determining the evolutionary tree using experiments. *Journal of Algorithms*, 21(1):26–50, 1996. (Cited on p. 13.)

[72] S. Kannan, T. Warnow, and S. Yooseph. Computing the local consensus of trees. *SIAM Journal on Computing*, 27(6):1695–1724, 1998. (Cited on p. 13.)

[73] M.-Y. Kao. Tree contractions and evolutionary trees. *SIAM Journal on Computing*, 27(6):1592–1616, 1998. (Cited on p. 55.)

[74] M.-Y. Kao, T.-W. Lam, T. Przytycka, W.-K. Sung, and H.-F. Ting. General techniques for comparing unrooted evolutionary trees. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing* (STOC'97), pages 54–65, 1997. (Cited on p. 55.)

[75] M.-Y. Kao, T.-W. Lam, W.-K. Sung, and H.-F. Ting. A decomposition theorem for maximum weight bipartite matchings with applications to evolutionary trees. In *Proceedings of the 7th Annual European Symposium on Algorithms* (ESA'99), volume 1643 of *Lecture Notes in Computer Science*, pages 438–449. Springer-Verlag Berlin Heidelberg, 1999. (Cited on p. 55.)

[76] M.-Y. Kao, T.-W. Lam, W.-K. Sung, and H.-F. Ting. A faster and unifying algorithm for comparing trees. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching* (CPM 2000), volume 1848 of *Lecture Notes in Computer Science*, pages 129–142. Springer-Verlag Berlin Heidelberg, 2000. (Cited on p. 55.)

[77] M.-Y. Kao, T.-W. Lam, W.-K. Sung, and H.-F. Ting. Unbalanced and hierarchical bipartite matchings with applications to labeled tree comparison. In *Proceedings of the 11th Annual International Symposium on Algorithms and Computation* (ISAAC 2000), volume 1969 of *Lecture Notes in Computer Science*, pages 479–490. Springer-Verlag Berlin Heidelberg, 2000. (Cited on p. 55.)

[78] M.-Y. Kao, T.-W. Lam, W.-K. Sung, and H.-F. Ting. An even faster and more unifying algorithm for comparing trees via unbalanced bipartite matchings. *Journal of Algorithms*, 40(2):212–233, 2001. (Cited on pp. 54, 55.)

[79] D. R. Karger. Minimum cuts in near-linear time. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing* (STOC'96), pages 56–63, 1996. (Cited on pp. 26, 27, 28.)

[80] M. Karpovsky. Weight distribution of translates, covering radius, and perfect codes correcting errors of given weights. *IEEE Transactions on Information Theory*, IT-27(4):462–472, 1981. (Cited on p. 103.)

[81] P. Kearney. Phylogenetics and the quartet method. In T. Jiang, Y. Xu, and M. Q. Zhang, editors, *Current Topics in Computational Molecular Biology*, pages 111–133. The MIT Press, Massachusetts, 2002. (Cited on pp. 13, 14.)

[82] S. Khanna, R. Motwani, and F. F. Yao. Approximation algorithms for the largest common subtree problem. Technical Report CS-TR-95-1545, Stanford University, 1995. (Cited on p. 56.)

[83] P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing*, 24(2):340–356, 1995. (Cited on p. 78.)

[84] J. Kleinberg, C. Papadimitriou, and P. Raghavan. Segmentation problems. In *Proceedings of the 30th Annual ACM Symposium on the Theory of Computing* (STOC'98), pages 473–482, 1998. (Cited on p. 121.)

[85] E. Kubicka, G. Kubicki, and F. R. McMorris. On agreement subtrees of two binary trees. *Congressus Numerantium*, 88:217–224, 1992. (Cited on p. 56.)

[86] E. Kubicka, G. Kubicki, and F. R. McMorris. An algorithm to find agreement subtrees. *Journal of Classification*, 12:91–99, 1995. *Note:* This article was accepted for journal publication in 1992 but not published until 1995, i.e., after the publication of [41], [54], and [117]. (Cited on p. 55.)

[87] T.-W. Lam, W.-K. Sung, and H.-F. Ting. Computing the unrooted maximum agreement subtree in sub-quadratic time. In *Proceedings of the 5th Scandinavian Workshop on Algorithm Theory* (SWAT'96), volume 1097 of *Lecture Notes in Computer Science*, pages 124–135. Springer-Verlag Berlin Heidelberg, 1996. (Cited on p. 55.)

[88] J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. *Information and Computation*, to appear. A preliminary version appeared in *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA'99), pages 633–642, 1999. (Cited on pp. 102, 105, 116.)

[89] S.-Y. Le, J. Owens, R. Nussinov, J.-H. Chen, B. A. Shapiro, and J. V. Maizel. RNA secondary structures: comparison and determination of frequently recurring substructures by consensus. *Computer Applications in the Biosciences*, 5(3):205–210, 1989. (Cited on p. 76.)

[90] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation (Second Edition).* Prentice-Hall, New Jersey, 1998. (Cited on pp. 2, 43.)

[91] M. Li, B. Ma, and L. Wang. On the closest string and substring problems. *Journal of the ACM*, 49(2):157–171, 2002. (Cited on pp. 6, 102, 105, 106, 116, 120, 122, 131, 132, 133, 135, 139.)

[92] W.-H. Li. *Molecular Evolution.* Sinauer Associates, Inc., Sunderland, 1997. (Cited on pp. 12, 13.)

[93] N. Megiddo. Linear-time algorithms for linear programming in $\mathbb{R}^3$ and related problems. *SIAM Journal on Computing*, 12(4):759–776, 1983. A preliminary version appeared in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science* (FOCS'82), pages 329–338, 1982. (Cited on p. 106.)

[94] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1984. (Cited on pp. 34, 35, 36.)

[95] A. Meyers, R. Yangarber, and R. Grishman. Alignment of shared forests for bilingual corpora. In *Proceedings of the 16th International Conference on Computational Linguistics* (COLING-96), pages 460–465, 1996. (Cited on pp. 54, 77.)

[96] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995. (Cited on p. 109.)

[97] M. Muselli and D. Liberati. Training digital circuits with Hamming clustering. *IEEE Transactions on Circuits and Systems – I: Fundamental Theory and Applications*, 47(4):513–527, 2000. (Cited on p. 119.)

[98] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992. (Cited on pp. 26, 27.)

[99] N. Nishimura, P. Ragde, and D. M. Thilikos. Finding smallest supertrees under minor containment. *International Journal of Foundations of Computer Science*, 11(3):445–465, 2000. (Cited on p. 78.)

[100] C. Notredame, E. A. O'Brien, and D. G. Higgins. RAGA: RNA sequence alignment by genetic algorithm. *Nucleic Acids Research*, 25(22):4570–4580, 1997. (Cited on p. 76.)

[101] R. Ostrovsky and Y. Rabani. Polynomial-time approximation schemes for geometric min-sum median clustering. *Journal of the ACM*, 49(2):139–156, 2002. (Cited on pp. 6, 119, 120, 121, 122, 132, 133.)

[102] M. Overmars. Computational geometry on a grid: an overview. In R. A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, volume F 40 of *NATO ASI Series*, pages 167–184. Springer-Verlag, 1988. (Cited on pp. 34, 36.)

[103] R. Panigrahy and S. Vishwanathan. An $O(\log^* n)$ approximation algorithm for the asymmetric p-center problem. *Journal of Algorithms*, 27(2):259–268, 1998. (Cited on pp. 118, 119, 120.)

[104] C. Papadimitriou. On the complexity of integer programming. *Journal of the ACM*, 28(4):765–768, 1981. (Cited on p. 108.)

[105] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994. (Cited on pp. 2, 3, 64, 70.)

[106] P. A. Pevzner. *Computational Molecular Biology : An Algorithmic Approach.* The MIT Press, Massachusetts, 2000. (Cited on pp. 78, 79, 80, 96, 97, 103.)

[107] F. Preparata and M. I. Shamos. *Computational Geometry.* Springer-Verlag, New York, 1985. (Cited on pp. 35, 89, 104.)

[108] T. Przytycka. Sparse dynamic programming for maximum agreement subtree problem. In B. Mirkin, F. R. McMorris, F. Roberts, and A. Rzhetsky, editors, *Mathematical Hierarchies and Biology*, volume 37 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 249–264. AMS, 1997. (Cited on p. 55.)

[109] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley, New York, 1986. (Cited on p. 109.)

[110] C. Semple and M. Steel. A supertree method for rooted trees. *Discrete Applied Mathematics*, 105(1–3):147–158, 2000. (Cited on p. 29.)

[111] J. C. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, Boston, 1997. (Cited on pp. 12, 13, 74, 78, 79, 80, 92, 93, 96, 97, 103.)

[112] B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Computer Applications in the Biosciences*, 6(4):309–318, 1990. (Cited on p. 75.)

[113] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. In *Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (PODS 2002), pages 39–52, 2002. (Cited on p. 77.)

[114] I. Shmulevich and W. Zhang. Binary analysis and optimization-based normalization of gene expression data. *Bioinformatics*, 18(4):555–565, 2002. (Cited on p. 119.)

[115] I. Simon. Sequence comparison: some theory and some practice. In *Electronic Dictionaries and Automata in Computational Linguistics, LITP Spring School on Theoretical Computer Science, 1987*, volume 377 of *Lecture Notes in Computer Science*, pages 79–93. Springer-Verlag Berlin Heidelberg, 1989. (Cited on p. 78.)

[116] M. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9(1):91–116, 1992. (Cited on pp. 14, 29.)

[117] M. Steel and T. Warnow. Kaikoura tree theorems: Computing the maximum agreement subtree. *Information Processing Letters*, 48:77–82, 1993. (Cited on pp. 55, 62, 149.)

[118] M. Stoer and F. Wagner. A simple min-cut algorithm. *Journal of the ACM*, 44(4):585–591, 1997. (Cited on pp. 26, 27.)

[119] W.-K. Sung. *Fast Labeled Tree Comparison via Better Matching Algorithms.* PhD thesis, University of Hong Kong, 1998. (Cited on pp. 55, 71.)

[120] W.-K. Sung. Personal communication, June 2002. (Cited on p. 55.)

[121] D. L. Swofford. *PAUP∗: Phylogenetic Analysis Using Parsimony (and Other Methods) 4.0 Beta for UNIX or OpenVMS.* Sinauer Associates, Inc., Sunderland, 2002. See also `http://paup.csit.fsu.edu/about.html` (Cited on p. 13.)

[122] D. L. Swofford, G. J. Olsen, P. J. Waddell, and D. M. Hillis. Phylogenetic inference. In D. M. Hillis, C. Moritz, and B. K. Mable, editors, *Molecular Systematics (Second Edition)*, pages 407–514. Sinauer Associates, Inc., Sunderland, 1996. (Cited on p. 13.)

[123] J. J. Sylvester. A question in the geometry of situation. *The Quarterly Journal of Pure and Applied Mathematics*, volume I, page 79, 1857. (Cited on p. 106.)

[124] K.-C. Tai. The tree-to-tree correction problem. *Journal of the ACM*, 26(3):422–433, 1979. (Cited on pp. 77, 78.)

[125] R. E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, 1985. (Cited on p. 87.)

[126] M. Thorup. Decremental dynamic connectivity. *Journal of Algorithms*, 33(2):229–243, 1999. (Cited on p. 19.)

[127] M. Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the 32nd Annual ACM Symposium on the Theory of Computing* (STOC 2000), pages 343–350, 2000. (Cited on p. 19.)

[128] V. Vazirani. *Approximation Algorithms.* Springer-Verlag, Berlin, 2001. (Cited on pp. 3, 70, 119, 120, 137.)

[129] J. T. L. Wang, D. Shasha, G. J. S. Chang, L. Relihan, K. Zhang, and G. Patel. Structural matching and discovery in document databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD 1997), pages 560–563, 1997. (Cited on p. 76.)

[130] M. Waterman. *Introduction to Computational Biology : Maps, Sequences, and Genomes.* Chapman & Hall, London, 1995. (Cited on pp. 13, 78, 79, 80, 96, 97, 103.)

[131] C. Wenk. Applying an edit distance to the matching of tree ring sequences in dendrochronology. In *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching* (CPM'99), volume 1645 of *Lecture Notes in Computer Science*, pages 223–242. Springer-Verlag Berlin Heidelberg, 1999. (Cited on p. 78.)

[132] W. Yang. Identifying syntactic differences between two programs. *Software: Practice & Experience*, 21(7):739–755, 1991. (Cited on p. 76.)

[133] K. Zhang and T. Jiang. Some MAX SNP-hard results concerning unordered labeled trees. *Information Processing Letters*, 49(5):249–254, 1994. (Cited on pp. 77, 79.)

[134] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989. (Cited on p. 77.)

CONGRATULATION

THIS STORY IS HAPPY END.