Summarizing Diverging String Sequences, with Applications to Chain-Letter Petitions

Patty Commins
Department of Computer Science, Carleton College, Northfield, MN, USA
Department of Mathematics, University of Minnesota, Minneapolis, MN, USA
commins.patty@gmail.com

David Liben-Nowell
Department of Computer Science, Carleton College, Northfield, MN, USA
dln@carleton.edu

Tina Liu
Department of Computer Science, Carleton College, Northfield, MN, USA
Surescripts, Minneapolis, MN, USA
tina.jxy.liu@gmail.com

Kiran Tomlinson
Department of Computer Science, Carleton College, Northfield, MN, USA
Department of Computer Science, Cornell University, Ithaca, NY, USA
kt@cs.cornell.edu

Abstract

Algorithms to find optimal alignments among strings, or to find a parsimonious summary of a collection of strings, are well studied in a variety of contexts, addressing a wide range of interesting applications. In this paper, we consider *chain letters,* which contain a growing sequence of signatories added as the letter propagates. The unusual constellation of features exhibited by chain letters (one-ended growth, divergence, and mutation) make their propagation, and thus the corresponding reconstruction problem, both distinctive and rich. Here, inspired by these chain letters, we formally define the problem of computing an optimal summary of a set of diverging string sequences. From a collection of these sequences of names, with each sequence noisily corresponding to a branch of the unknown tree T representing the letter's true dissemination, can we efficiently and accurately reconstruct a tree $T' \approx T$? In this paper, we give efficient exact algorithms for this summarization problem when the number of sequences is small; for larger sets of sequences, we prove hardness and provide an efficient heuristic algorithm. We evaluate this heuristic on synthetic data sets chosen to emulate real chain letters, showing that our algorithm is competitive with or better than previous approaches, and that it also comes close to finding the true trees in these synthetic datasets.

2012 ACM Subject Classification Mathematics of computing → Combinatorial algorithms; Applied computing → Law, social and behavioral sciences

Keywords and phrases edit distance, tree reconstruction, information propagation, chain letters

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.11

Related Version A full version of the paper including omitted proofs is available at `https://arxiv.org/abs/2004.08993`.

Supplementary Material Related research data and source code hosted at `https://github.com/tomlinsonk/diverging-string-seqs`.

Acknowledgements We thank Jon Kleinberg for extensive discussions, and Anna Johnson, Hailey Jones, Dave Musicant, Layla Oesper, Anna Rafferty, and Ethan Somes for helpful discussions during preliminary or late stages of this project.

1 Introduction

In a range of computational settings, we are given a collection of strings and asked to construct some kind of parsimonious representation of the given set. The task becomes more interesting if the strings result from a generative process, especially if new strings arise from a mechanism involving both replication and mutation of old strings. Now the parsimonious representation might be a *tree* describing the generative history of this population, with nodes corresponding to the strings and the branching structure representing the evolutionary events that produced that population. At this level of description, a host of applications fall under this rubric: reconstructing a phylogeny from a set of genes, tracing the spread of a textual meme in a social network, inferring the version history of a document from its many copies. These domains differ in the way that replication and mutation occur – sometimes randomly ("by nature") and sometimes intentionally by humans embedded in a social structure – and, perhaps, whether some kind of selective pressure affects which strings survive or replicate.

Here, we consider a specific – and surprisingly rich – social setting in which a population of strings is generated: *chain letters.* Chain letters often feature an outlandish claim ("send a copy of this letter to ten friends, or you will have bad luck forever!"), but, more crucially, recipients are instructed to add their names to the document's end, make a copy, and send those copies to multiple friends. Importantly, every subsequent recipient may modify any part of the document, or copy it imprecisely; thus each document contains a list of signatures, representing an ordered (if noisy) trace of its particular path through the social network. In the present work, we study the problem of accurately reconstructing the underlying propagation tree from a set of signature lists. If the lists of signatures contained no errors, the problem would be trivial, but errors abound, including both point mutations and structural variants. (In real email-based chain-letter data, some signatories retyped names, often incorrectly, and both block deletions and duplications appear [29]. Worse, some copies of the emails are only available as low-quality scanned images, introducing further errors.)

The propagation of chain letters. There are three crucial properties in chain-letter–like contexts that, together, make this data intriguingly different from other settings:

- **(i)** *chain letters grow (at one end).* A document has an "active end," and a document typically changes via the deposition of additional text (another name) at its active end.
- **(ii)** *chain letters diverge.* A document can *split* to create multiple "children" documents, which share a prefix up to the split but have differing suffixes below. The split is at the active end; two documents that diverge grow independently after the branching point.
- **(iii)** *chain letters mutate.* Actors introduce noise: an individual sending a chain letter to a friend makes a (potentially imperfect) *copy* of that document, possibly introducing errors – and those errors are "inherited" by subsequent copies of the letter.

Given a collection of many copies of "the same" chain letter, each with its own sequence of names, one can seek to reconstruct the underlying true record of the propagation – both the structure of the propagation tree *and* the strings representing the true names of the signatories. Together, the above properties make this chain-letter reconstruction problem a tantalizing domain for parsimonious reconstruction: as the rate of noise in document copying increases, naturally the reconstruction problem becomes difficult, but there is a great deal of repetition in the input data, particularly near the root of the propagation tree.

The present work. We formally introduce the *Diverging String Sequence Summarization Problem (DSSSP):* given a collection X of sequences of strings (strings correspond to names, and each sequence is a noisy list of names in an instantiation of a chain letter), we seek a tree T that optimally summarizes X. (Rather than using the language of chain letters, we will abstract away the particular application and discuss diverging string sequences in general.)[1] What counts as an "optimal" summary depends on a tradeoff between two competing goods: the *accuracy* of T in representing the strings in the given sequences, and the *efficiency* of T in representing the given string sequences without too much redundancy. Our formal definition of the problem is parameterized to reflect the tradeoff between these two competing goods.

Our main theoretical results on DSSSP are (1) an efficient optimal algorithm for the case of $m = 2$ sequences, based on an approach we call *edit distance with give-up* (Theorem 4.1); (2) a proof of hardness for large m (Theorem 5.1); and (3) an exact polynomial-time algorithm for any fixed value of m (Theorem 5.2). We also give a much more efficient heuristic algorithm for large m – using a combination of divergence-aware pairwise alignment and iterative merging, inspired by progressive alignment algorithms [13] – and show empirically that it does a good job of reconstructing synthetically generated trees.

2 Related Work

Chain-letter data. In joint work with Jon Kleinberg, the second author studied the propagation of a widespread email-based anti-war petition [29]. This work focused on the topological structure of the underlying propagation tree; subsequent research sought to explain the shape of the tree through stochastic branching processes [17] or the rarity of sampled email copies [8]. The present work differs in that here we study the problem of accurately reconstructing the propagation tree from signature lists, rather than seeking to understand the structure of that tree. Still, examining the structure of the propagation tree presupposes a reconstructed tree, which in [29] was done using a hard edit distance cutoff to decide whether two signatures belong to the same signatory. (See Section 7.2.) This specific aspect of our problem – do multiple signatures belong to the same signatory? – has been considered in other forms in the past, including error-tolerant recognition of strings with various error models [4,34], error correction of strings of regular languages [41], and block edit models for approximate string matching [30], all of which use various versions of edit distance.

Chain letters in paper form have also been investigated in the context of constructing a phylogeny based on variations in the text of the document itself (rather than a list of signatories) [3], or the propagation of stories as a network [21].

Other forms of propagation. In rare cases, a situation matching all three key features of chain letters has been studied – including a (controversial) model of the origin of life, based on layered clay accreting over time and even diverging and mutating [5,6]. More common settings share two of the three features. For example, absent any errors, our reconstruction task is solved by a trie [10,15] summarizing a set of diverging strings. Online conversations [24] (e.g., comment threads or especially email threads) have an active end at which new contributions appear, and threads can diverge, but there is no obvious notion of mutation.

[1] There is another layer of complication, literally: rather than viewing a document as a sequence of *characters,* we instead view it as a sequence of *signatures* (each of which is a string that consists of a sequence of characters). Thus there is a "two-level" view of edits, in which either an individual character can be corrupted (a single character-level edit within a particular signature) or an individual signature can be corrupted (an entire signature is deleted, inserted, or replaced by a different signature).

There are also applications in which the objects of interest are strings that grow at one end, with noise but without meaningful divergence. In dendrochronology (the science of dating wood), approaches based on edit distance can be used to study sequences of growth rings in trees, which accumulate on one end, adjacent to the bark [44].

By far, though, the best-studied settings that match two of our three features have strings that mutate and replicate, but have no "active end" at which growth occurs. This is the classical setting of phylogenetic reconstruction, but it also appears in many other contexts. Most prominent is the spread of news, memes, and rumors that evolve as versions are created and shared (e.g., [1, 16, 20, 25, 39]). Mutations in these cases differ from ours, though, in that the content of the information being spread can affect the type of mutations that occur, thereby affecting the likelihood of further propagation (and therefore the structure of the tree). Much of this work seeks to understand various types of dissemination and what factors may impact the propagation structure – different from the goal of reconstructing the underlying tree. Reconstruction of the evolutionary history of a collection of divergent objects is also well studied in a bafflingly wide variety of contexts, from version histories of code snippets in Stack Overflow [2], to variations of the story "Little Red Riding Hood" [40], to diverging cultural histories using textile data [31].

Reconstruction algorithms. In addition to the algorithmic approaches to these various other forms of data, there is a voluminous literature on string alignment in the computational biology literature. The multiple sequence alignment problem is closely related to DSSSP, and many algorithms target a variety of challenges related to it (see [7, 22, 27, 36, 37, 42], among many others). There is also work involving the alignment of amino acid sequences to reconstruct the history of proteins, including mutations and divergence events [12].

3 Summarizing Diverging String Sequences

Before we formally define our abstract problem, we begin with some intuition, with terminology drawn from chain letters. Informally, a *name* is a string over a finite alphabet, and a *petition* is a sequence of names. We are given a *set* of petitions $X = \{x_1, \ldots, x_m\}$, and we seek the tree T that best summarizes the set X. But the "best" tree depends on a tradeoff between two competing goods: (i) the efficiency of T (its number of nodes), and (ii) the accuracy of T in representing the petitions in X.

Consider petitions $x_1 =$ `Aaa Bbb Ccc Ddd Eee` and $x_2 =$ `Aaa Bbx Ccc Dxx Fff`, with spaces separating names, as an example. Depending on the relative importance of efficiency and accuracy, there are *four* distinct "best" trees (see Figure 1): a trivial tree that never diverges (if efficiency matters much more than accuracy); a tree that diverges upon any textual discrepancy (if accuracy matters much more); or two intermediate trees that diverge after the `Ccc`s or the `D??`s (depending on the cost–benefit of adding one node vs. paying for two textual errors).

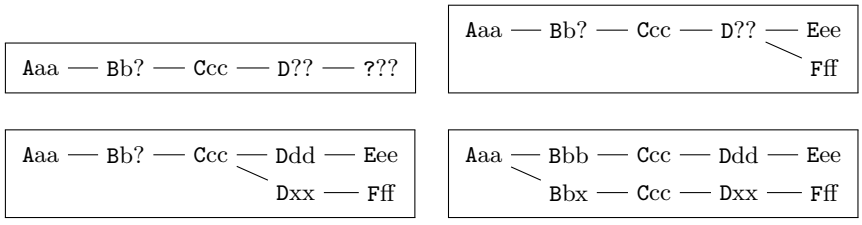


Figure 1 Four "best" trees for $x_1 =$ `Aaa Bbb Ccc Ddd Eee` and $x_2 =$ `Aaa Bbx Ccc Dxx Fff`.

3.1 Distance between a Summary Tree and a String Sequence

To begin, we need to quantify how accurately a set X of string sequences is represented by a summary structure – and, more fundamentally, what it means to summarize X.

▶ **Definition 3.1** (Labeled Summary Tree). *Let X be a set of string sequences. A* labeled summary tree *of X is a pair $\langle T, f\rangle$, where T is a tree with each node labeled with a string, and f is a function mapping each $x \in X$ to a node v_x in T.*

Let $\mathsf{labelseq}_T(v_x)$ denote the sequence of node labels on the path from the root of T to v_x.

That is, a summary of X consists of a labeled tree T, with a node of T designated to correspond to each sequence in X. To assess how accurately a string sequence $x \in X$ is represented, we will compare x with $\mathsf{labelseq}_T(v_x)$. Our metric will be a variation on the classical Levenshtein edit distance [26], with two adjustments:

1. Edit distance is usually defined between two strings, but we wish to compare two *sequences* of strings. We cannot simply concatenate the strings within each sequence and then use standard edit distance, as the edits would no longer respect string boundaries. As such, we need to define an edit distance with two levels of granularity.
2. We insist that every string in each sequence $x \in X$ be represented (perhaps with some error) in the tree. Thus, when we align x to its path in the tree, we do not allow deletion of strings in the sequence. (We forbid deletions from $x \in X$ to preserve the intuition that the optimal summary tree for a singleton sequence $X = \{x\}$ is a nonbranching path successively labeled by the strings in x *even when the cost of nodes is very high.*)

Let x be a string sequence, and let $y = \mathsf{labelseq}_T(v_x)$. Our metric, then, is an asymmetric two-level variant of the edit distance between x and y, allowing deletions from y but not x:

▶ **Definition 3.2** (Asymmetric Edit Distance). *Let x and y be string sequences. The* asymmetric edit distance *of x with respect to y, denoted $\mathsf{AED}(x, y)$, is the cost of the cheapest sequence of operations transforming x into y, where the allowable operations are inserting a string into x and substituting a string. (No string can be deleted from x.)*

These operations' costs are given by (classical) edit distance ED*: substituting w' for w costs $\mathsf{ED}(w, w')$; inserting a string w into x costs $\mathsf{ED}(w, \varepsilon)$, where ε is the empty string. (Unless otherwise specified, all* ED *edits have unit cost, but we allow arbitrary cost matrices.)*

We compute $\mathsf{AED}(x, y)$ with a variation on the classical dynamic program for edit distance, forbidding deletions and using ED to compute the cost of inserting or substituting a string.

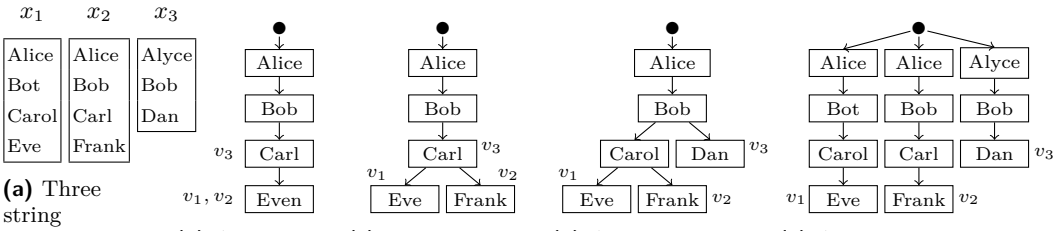
The *distance* between a string sequence x and summary tree $\langle T, f\rangle$, then, is given by $\mathsf{AED}(x, \mathsf{labelseq}_T(f(x)))$ – i.e., the asymmetric edit distance between x and the label sequence on the path from root to the node corresponding to x in T.

3.2 The Problem: Summary Trees for a Set of String Sequences

We can now formally define our problem, where our objective function is – in the style of regularization in machine learning [18] – a weighted sum of the accuracy of the summary tree (as measured by AED) and the simplicity of the tree (as measured by its number of nodes):

▶ **Definition 3.3** (Diverging String Sequence Summarization Problem [DSSSP]).
Input: *A set $X = \{x_1, x_2, ..., x_m\}$ of string sequences and a nonnegative* node cost λ.
Output: *A labeled summary tree $\langle T, f\rangle$ (i.e., a tree T and a function f mapping each x_i to a node v_i in T) minimizing the following, where $|T|$ denotes the number of nodes in T:*

$$\mathrm{err}_\lambda(T) := \left[\sum_{i=1}^{m} \mathsf{AED}(x_i, \mathsf{labelseq}_T(v_i))\right] + \lambda \cdot |T|. \tag{1}$$



(a) Three string sequences to summarize. **(b)** An optimal tree for $\lambda \geq 5$. **(c)** The optimal tree for $3 < \lambda < 5$. **(d)** An optimal tree for $1 \leq \lambda \leq 3$. **(e)** An optimal tree for $\lambda = 0$.

Figure 2 A set of string sequences and their optimal summary trees, for different ranges of λ. The $\bullet$ root node denotes the sentinel string starting every sequence; nodes marked by $\{v_1, v_2, v_3\}$ correspond to the sequences $\{x_1, x_2, x_3\}$. The choice about whether to split Eve and Frank into two nodes (Figure 2b vs. 2c) is a function of their edit distance, $\mathsf{ED}(\texttt{Eve}, \texttt{Frank}) = 5$. When $\lambda > 5$, then it is cheaper to accept the cost of aligning both to a single label than to pay for an extra node.

To ensure that T is a tree with a single root, we place sentinel values at the start of each sequence x_i. (Denote by $|T|$ the number of *non-sentinel* nodes in T.) Note that $\mathsf{AED}(x_i, \mathsf{labelseq}_T(v_i))$ is defined only if $|x_i| \leq |\mathsf{labelseq}_T(v_i)|$ – that is, the depth of the node v_i in T is at least the number of strings in the sequence x_i – as it would otherwise be impossible to convert x_i into $\mathsf{labelseq}_T(v_i)$ without deleting strings.

The parameter λ controls the tradeoff between trees that represent the input sequences accurately and trees that provide more concise summaries of the input set. When $\lambda = 0$, a trivial branch-immediately tree T is optimal; when $\lambda = \infty$, a trivial never-branching tree of depth $\max_i |x_i|$ is. Intermediate values of λ give more interesting structures. See Figure 2.

4 Solving DSSSP for Two Sequences: Edit Distance with Give-up

Consider first the case of just two input sequences, $m = |X| = 2$. (Even with $m = 2$, the problem has interesting subtleties.) We can compute an optimal tree through an alignment algorithm we call *edit distance with give-up*. The resulting tree has exactly one leaf or two leaves; in the latter case, we call the tree a *bifurcation*.

As with AED, the idea is similar to the classical dynamic program for edit distance, but with one additional operation permitted: *give up* entirely on aligning the remaining portions of the sequences, and declare a split at this point. We also modify the costs in the edit distance dynamic program to reflect the λ per-node cost of each operation, corresponding to the node-cost term in $\mathrm{err}_\lambda(T)$. Writing $\mathsf{EDG}(i, j, \lambda)$ to denote the cost of the best alignment of $x_{i,\ldots,|x|}$ and $y_{j,\ldots,|y|}$ under node cost λ, and writing $\mathsf{EDG}(x, y, \lambda) = \mathsf{EDG}(1, 1, \lambda)$, we have

$$\begin{aligned} \mathsf{EDG}(|x|+1, |y|+1, \lambda) &= 0 \\ \mathsf{EDG}(i, |y|+1, \lambda) &= \lambda(|x| - i + 1) \qquad \text{for any } 0 \leq i \leq |x| \\ \mathsf{EDG}(|x|+1, j, \lambda) &= \lambda(|y| - j + 1) \qquad \text{for any } 0 \leq j \leq |y| \end{aligned} \tag{2}$$

and, for any $0 \leq i \leq |x|$ and any $0 \leq j \leq |y|$,

$$\mathsf{EDG}(i, j, \lambda) = \min \begin{cases} \mathsf{EDG}(i+1, j+1, \lambda) + \lambda + \mathsf{ED}(x_i, y_j) & \text{(substitution)} \\ \mathsf{EDG}(i, j+1, \lambda) + \lambda + \mathsf{ED}(\varepsilon, y_j) & \text{(insertion)} \\ \mathsf{EDG}(i+1, j, \lambda) + \lambda + \mathsf{ED}(x_i, \varepsilon) & \text{(deletion)} \\ \lambda(|x| - i + 1) + \lambda(|y| - j + 1) & \text{(give up)} \end{cases}$$

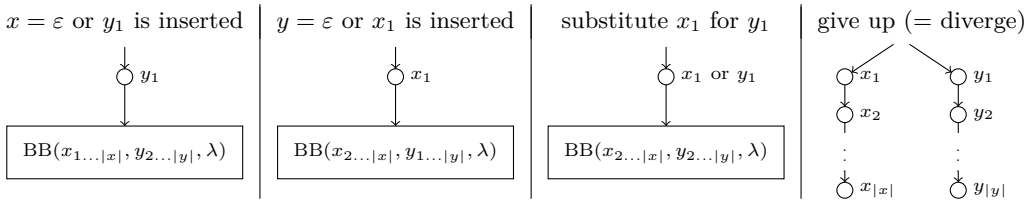


Figure 3 Constructing the tree in BUILDBIFURCATION(x, y, λ). Whenever a node corresponding to $x_{|x|}$ or $y_{|y|}$ is placed, we map the corresponding input sequence to that node of the bifurcation. We write "BB" to abbreviate BUILDBIFURCATION, and we output an empty tree when $x = y = \varepsilon$.

For example, consider the insertion case of the minimum. Here we match the string y_j with no corresponding entry in x, and recursively align $y_{j+1,\ldots,|y|}$ with $x_{i,\ldots,|x|}$, with a total cost of

$$\underbrace{\mathsf{EDG}(i, j+1, \lambda)}_{\text{cost of alignment of remaining strings}} + \underbrace{\lambda}_{\text{cost of creating the root node}} + \underbrace{\mathsf{ED}(\varepsilon, y_j).}_{\text{cost of inserting } y_j \text{ into } \mathsf{labelseq}_T(x)}$$

The cost of "giving up" – i.e., declaring a split in the alignment – is large, requiring $(|x|-i+1)$ nodes on the x branch and $(|y|-j+1)$ on the y branch, each of which incurs cost λ.

Denote by BUILDBIFURCATION the natural dynamic programming algorithm that computes EDG using (2). (We abuse notation: BUILDBIFURCATION(x, y, λ) denotes either the resulting bifurcation or its cost. We can construct this bifurcation simultaneously with the construction of the alignment; see Figure 3.)

BUILDBIFURCATION optimally solves DSSSP for $m = 2$ sequences – i.e., the tree T built by BUILDBIFURCATION(x, y, λ) minimizes $\mathrm{err}_\lambda(T)$. (Due to space constraints, the proof is omitted here, but is available in the full version of the paper linked on the title page.)

▶ **Theorem 4.1.** *The tree T^* := BUILDBIFURCATION(x, y, λ) is an optimal summary tree for the string sequences $\{x, y\}$ with node cost λ. Specifically, $\mathrm{err}_\lambda(T^*) = \mathsf{EDG}(x, y, \lambda)$.*

Writing $n = \max(|x|, |y|)$ to denote the length of the longer of the two sequences, and $k = \max(\max_i |x_i|, \max_j |y_j|)$ to denote the length of the longest string in either sequence, then the running time of BUILDBIFURCATION(x, y, λ) is $O(n^2k^2)$.

5 Optimal Summaries of Larger Sets of String Sequences

BUILDBIFURCATION efficiently finds the optimal summary tree for $m = 2$ sequences, but DSSSP with large m is computationally intractable.

▶ **Theorem 5.1.** *DSSSP (for an arbitrary number m of string sequences) is NP-hard.*

Proof (idea). For large λ, hardness follows from a reduction from String Median, which we will encounter shortly.

While the reduction from Median String is simpler, we can also give an alternative reduction from Shortest Common Supersequence (SCS) [35], which applies for smaller λ as well. (Note that SCS is hard in general, but for a small number of strings it is efficiently solvable [14].) The details of the latter proof are available in the full version. ◀

On the other hand, if we are willing to tolerate running times that are exponential in m (but polynomial in the other measures of input size), we can solve DSSSP in polynomial time:

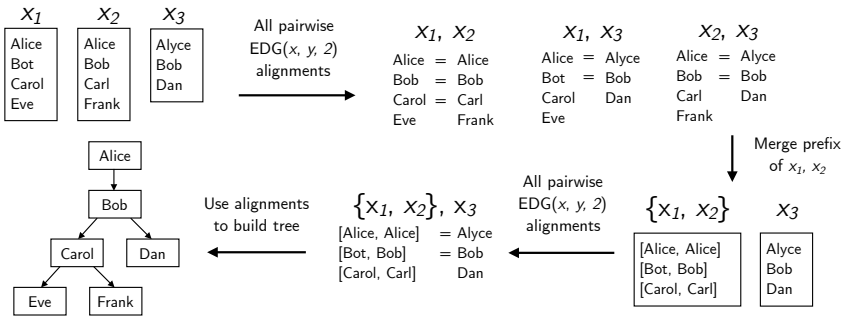


Figure 4 An example run of BUILDTREE(X, λ) for $\lambda = 2$ and the sequences from Figure 2.

▶ **Theorem 5.2.** *Let X be a set of string sequences, where $m = |X|$ is the number of sequences, $n = \max_i |x_i|$ is the length of the longest sequence, and $k = \max_j \max_i |x_{i,j}|$ is the length of the longest string in any of the sequences. Then there is an algorithm solving DSSSP on X (for any λ) that runs in time $O(n^{m^2} \cdot 2^m \cdot k^m \cdot \text{poly}(k, m, n))$.*

Proof. The approach is brute force: we look at every possible tree topology τ, which specifies, for any $x, x' \in X$, the indices i and i' into x and x' at which they diverge. Every τ defines a set of nonbranching segments between divergences; the summary tree problem is then a collection of summary "path" problems, one per segment.

The path problem can be seen as multiple sequence alignment (MSA) [37], solvable via dynamic programming [7] (see also [22, 36, 42]). To implement the MSA dynamic program, we must compute the cost of assigning a set of strings $S = \{s_1, s_2, \ldots, s_\ell\}$ to a single node u. If we label u with the string z, then the alignment cost of this node is $\lambda + \sum_i \mathsf{ED}(s_i, z)$; thus the best label is the *string median* of S – that is, the string z minimizing the summed edit distance to the strings in S. While string median is NP-hard [11], a dynamic programming algorithm solves string median for a fixed number of strings [37] (see also [23, 33, 38]).

There are $O(n^{m \cdot (m-1)})$ tree topologies. Each defines a tree with $\leq m$ leaves and thus $\leq 2m$ nonbranching segments. Each segment contains $\leq m$ subsequences, each of length $\leq n$; thus each multiple sequence alignment requires $O(n^m)$ time [7]. Whenever we compute the string median of a candidate node, we have $\leq m$ strings each of length $\leq k$; computing these medians takes $O((2k)^m)$ time [37]. Finally, it takes $\text{poly}(k, m, n)$ time to compute $\text{err}_\lambda(T)$ for each tree. Thus the overall running time is $O(n^{m \cdot (m-1)} \cdot n^m \cdot (2k)^m \cdot \text{poly}(n, m, k))$. ◀

6 An Efficient Heuristic for Larger Sets of Sequences

Given DSSSP's hardness (Theorem 5.1) and the abominable running time of our exact algorithm (Theorem 5.2), we turn here to an efficient heuristic for DSSSP with larger m. Our algorithm is greedy, and seeks to repeatedly identify the pair of sequences in X with the longest shared prefix, and then merge that shared prefix into a single sequence (as in BUILDBIFURCATION). See Figure 4. There are several issues that we must resolve:

Measuring and merging the shared prefix of x_i and x_j. To calculate how well x_i and x_j match, we compute the $\mathsf{EDG}(x_i, x_j, \lambda)$ alignment. Define the *number of substitutions* (ignoring insertions and deletions) *in the pre-divergence section $p_{i,j}$ of this alignment* as their overlap. Then, for the pair $\{x_i, x_j\}$ with the largest overlap, replace $\{x_i, x_j\}$ with $p_{i,j}$ in X. Note that $p_{i,j}$ is a sequence of *lists of strings,* not a sequence of *strings;* thus we need to generalize EDG to sequences of lists of strings, not just individual strings.

Reconciling labels in the final resulting tree. Repeating this merging process will define a tree, except that each node is labeled by a *list* of strings, not just one. To produce the final labels, we use the *medoid* string. For a list of strings A, the medoid of A is the string in A whose sum of edit distances to strings in A is minimized.[2]

Generalizing EDG to lists of strings. Define the edit distance between a string x and a set of strings X as $\mathsf{ED}(x, X) := \mathsf{ED}(x, \mathsf{medoid}(X))$ – using the medoid of X as its representative string. Then, letting $\mathcal{C}(A, B)$ denote the cost of merging lists A and B, we define

$$\mathcal{C}(A,B) := \sum_{x \in A \cup B} \mathsf{ED}(x, A \cup B) - \sum_{x \in A} \mathsf{ED}(x, A) - \sum_{y \in B} \mathsf{ED}(y, B). \tag{3}$$

This cost quantifies the amount of additional disagreement incurred by merging the lists, relative to leaving them separate. Insertion and deletion costs are found using (3) with a list of empty strings of appropriate length in place of x_i or y_j. We thus define the EDG recurrence (cf. Equation (2)) for sequences of lists of strings x and y as follows:

$$\mathsf{EDG}(i,j,\lambda) = \min \begin{cases} \mathsf{EDG}(i+1,j+1,\lambda) + \lambda + \mathcal{C}(x_i, y_j) & \text{(substitution)} \\ \mathsf{EDG}(i,j+1,\lambda) + \lambda + \mathcal{C}(\{|x_i| \text{ copies of } \varepsilon\}, y_j) & \text{(insertion)} \\ \mathsf{EDG}(i+1,j,\lambda) + \lambda + \mathcal{C}(x_i, \{|y_j| \text{ copies of } \varepsilon\}) & \text{(deletion)} \\ \lambda(|x| - i + 1) + \lambda(|y| - j + 1) & \text{(give up)} \end{cases}$$

Define BuildTree(X, λ) as the greedy iterative algorithm suggested above: until there is only one sequence left in X, find the pair of sequences $x_i, x_j \in X$ with the largest number of substitutions in $\mathsf{EDG}(x_i, x_j, \lambda)$, and replace $\{x_i, x_j\}$ by their merged prefix $p_{i,j}$. (Save the post-divergence branches of the bifurcation; we will reattach those branches at the bottom of $p_{i,j}$ in the final tree.) When there is only one sequence left, reattach all of the saved branches, and replace each node's list-of-strings label by the medoid of that label list. (See Figure 4.)

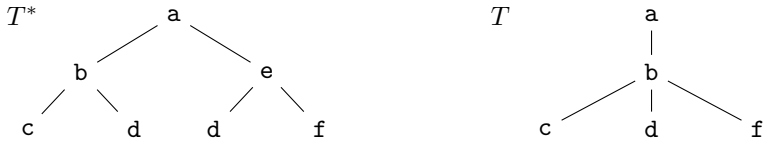
7 Evaluation and Parameter Selection

BuildTree is suboptimal both because greedy merging can yield a poor topology and because medoids can be poor node labels; see Examples 7.1 and 7.2. Still, we will show that it nonetheless performs well on simulated data, suggesting that it is a good heuristic.

▶ **Example 7.1** (A bad example for greedy merging)**.** Consider the instance

$x_1 = \texttt{a b c}$ $x_2 = \texttt{a b d}$ $x_3 = \texttt{a e d}$ $x_4 = \texttt{a e f}$

with $0.5 < \lambda < 1$. The optimal tree T^* is shown below, with $\mathrm{err}_\lambda(T^*) = 7\lambda$. However, BuildTree will choose to merge sequences with the most closely aligned pair of sequences according to the EDG alignment. In this case, it would choose to merge x_2 and x_3 first. The tree T returned by BuildTree has $\mathrm{err}_\lambda(T) = 5\lambda + 2$, making it suboptimal.



[2] When we say "the" medoid of A, we mean the lexicographically first medoid of A. (Which medoid we choose never affects the sum of the distances at hand – e.g., in the sums in (3) – but we need to identify one in particular for the summands to be well-defined.) Note that the medoid, unlike the median, must be an element of A; we use it because it is efficiently computable (unlike median) and is a $(2 - o(1))$-approximation to the median (an implication of the triangle inequality).

▶ **Example 7.2** (A bad example for medoids)**.** Consider the set $S = \{\texttt{XABC}, \texttt{AXBC}, \texttt{ABXC}, \texttt{ABCX}\}$, where each string sequence contains just one string. For large λ, the optimal tree is just a single node. Here the optimal label is the median `ABC`, which has total edit distance 4 to the set S; the medoid label `ABCX` has total edit distance 6 to S.

7.1 Generating Synthetic Data

We will generate synthetic data based on several parameters: the number m of string sequences, the string length k, a string substitution probability σ_s, a string deletion probability δ_s, a character substitution probability σ_c, and a character deletion probability δ_c.

We generate trees using branching processes (i.e., *Galton–Watson trees* [43]): each node chooses to have exactly i children with probability p_i; we fix $p_0 = 0.03$, $p_1 = 0.94$, and $p_2 = 0.03$ to approximate real email petition data [17,29]. We generate a Galton–Watson tree T until it has m leaves, restarting if the branching process terminates early. We label each node $u \in T$ with a random alphabetic string $\ell(u)$ of length k. Let x_i denote $\mathsf{labelseq}_T(u_i)$ for the ith leaf u_i. Call T the *true tree,* ℓ the *true labels,* and x_i the *true sequences.*

Now, we simulate noisy propagation. To mirror petition data derived from low-quality scans of printed emails, we introduce string- and character-level errors in separate phases:

(1) *string-level errors (which are inherited).* Each node u inherits from its parent p the noisy history h_p of its ancestral labels. (The root "inherits" an empty sequence.) The node u (further) corrupts h_p: for each string in h_p, substitute it with a random alphabetic string of length k with probability σ_s, and delete it with probability δ_s. Finally, node u appends its true label $\ell(u)$ to h_p; call the resulting sequence h_u. Now each leaf u stores h_u, a noisy version of $\mathsf{labelseq}_T(u)$. Let $X' = \{x'_1, \ldots, x'_m\}$ be the set of histories at the leaves.

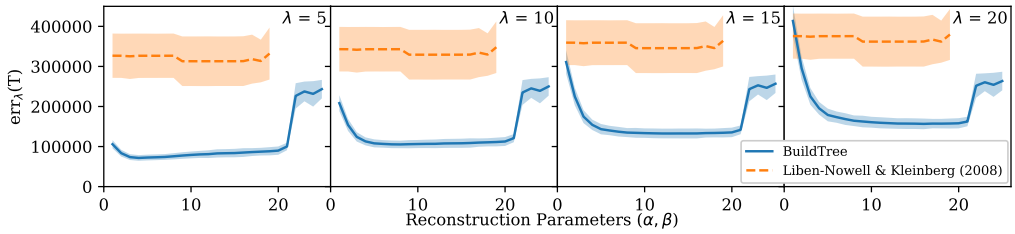
(2) *character-level errors (which appear independently).* For each $x'_i \in X'$, substitute each character in each string in x'_i with a random character with probability σ_c, and delete the character with probability δ_c. Let $X'' = \{x''_1, \ldots, x''_m\}$ be the resulting sequences.

Our experiments use string length $k = 25$, string error rates $\sigma_s = \delta_s = 0.001$, character error rates $\sigma_c = \delta_c = 0.1$, and $m \in \{15, 100\}$. Because string-level errors compound, 0.001 is a nontrivial error rate (roughly comparable to the 10% character-level error rate).

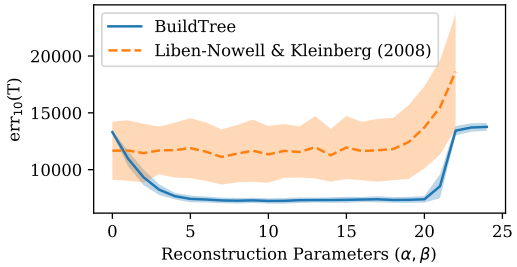
7.2 Comparing Reconstruction to Synthetic Ground Truth

We generate a true tree T, with true sequences $X = \{x_1, \ldots, x_m\}$ and corrupted sequences $X'' = \{x''_1, \ldots, x''_m\}$. We then use our heuristic algorithm to build a reconstructed tree $T' = \textsc{BuildTree}(X'', \alpha)$, for some choice of a node-cost parameter α to use in the reconstruction algorithm. (See Section 7.3 regarding how to choose α.) Note the distinction between the reconstruction parameter α and the evaluation parameter λ: BuildTree seeks to optimize $\mathrm{err}_\alpha(T')$ for some value of α, but we can assess the quality of T' using $\mathrm{err}_\lambda(T')$ whether or not $\alpha = \lambda$, to evaluate sensitivity to α.

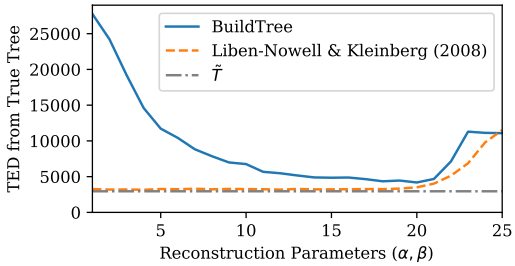
We will compare the quality of BuildTree to the threshold-based reconstruction algorithm from [29], which we briefly describe here. (1) Construct a weighted directed graph G with nodes labeled by all strings in all $x \in X$, and with an $a \to b$ edge if string a immediately precedes string b in any sequence $x \in X$. The weight of this edge is the *number* of sequences $x \in X$ containing a and b successively. (2) To handle minor signature errors, treat two signatures as equivalent if they follow equivalent signatories and have an edit distance below a fixed threshold β. (3) Compute the tree as a max-weight spanning arborescence of G, with extraneous nodes pruned away. (Note that β is on the same scale as α; it defines a cutoff of $\mathsf{ED}(a, a')$ indicating when a and a' should be assigned to two nodes versus one.)



(a) Error under $\text{err}_\lambda(T)$ with $|X| = 100$, for several λ values, averaged across 8 trials.



(b) Error under $\text{err}_{10}(T)$ with $|X| = 15$, averaged across 500 trials. Smaller X makes the error smaller than in (a), but the trend is similar.



(c) Error under ordered tree edit distance (TED) on a $|X| = 15$ dataset. The dash-dotted line shows TED between $\tilde{T}$ and the real tree.

Figure 5 Comparing BuildTree(X, α) to the algorithm from [29] with edit-distance threshold β. Shaded regions in (a) and (b) show standard deviations across the stated number of trials. Observe the flat-bottomed basin shape of the error curve for BuildTree(X, α) as α varies: high error rates when $\alpha < 5$ and when $\alpha > 20$, and roughly constant low error for all α in between.

Measures of performance. We assess the quality of our reconstructed trees T' in two ways.

First, we use the $\text{err}_\lambda(T')$ measure from (1), the sum of $\lambda \cdot |T'|$ and all label distances. Second, we use *tree edit distance* (TED) to compare the structure of T' to the true tree T. Edit distance between unordered trees is NP-hard [46], so we estimate the distance by ordering our trees and computing *ordered* TED using Zhang–Shasha [45], as implemented by Henderson [19]. (Specifically, we recursively order sibling nodes in T' to minimize the number of leaf order inversions with respect to T. As the number of siblings is typically small, this order is not expensive to compute.) To better interpret our results, we also compute TED to a tree with the correct structure but with label corruptions, denoted $\tilde{T}$; this represents the best reconstruction we could hope for (without computing medians). We construct $\tilde{T}$ by labeling the ancestors of each leaf u_i using the noisy history h_{u_i} (picking the medoid for nodes assigned multiple labels), deleting nodes with empty labels.

Evaluation results. We generated a tree T with $m = 100$ sequences, and corrupted sequences X'' (with 8 independent trials generating different X'' from T). Figure 5a compares $\text{err}_\lambda(\cdot)$ of our reconstruction with the method from [29], showing that BuildTree performs significantly better over a range of λ values. These trees were too big to compute tree edit distance, which is computationally prohibitive.

In addition, we generated a tree with $m = 15$ sequences and introduced error in 500 trials. Figure 5 shows both $\text{err}_{10}(T)$ (Figure 5b) and TED (Figure 5c) between the reconstructed tree $\widehat{T}$ and the true tree T. For many values of α, BuildTree(X'', α) produces significantly better solutions than the method from [29], as shown by the $\text{err}_{10}(T)$ measure. It is also competitive at $\alpha = 20$ according to (ordered) tree edit distance, a metric BuildTree was not designed to optimize.

7.3 How Should the Node Cost be Selected?

To reconstruct a tree from real sequence data, we must select a value for the reconstruction parameter α. If α is too low, nodes are too cheap and trees diverge too early; if α is too high, nodes are too costly and trees branch too rarely. But what value should we choose?

Intuitively, we wish to map two (corrupted) strings to the same node if they are (corruptions of) the same true string. Imagine two strings u and $v \neq u$, and let u', u'', and v' be the result of independently introducing errors to u, u, and v, respectively. We desire a value of α so that u' and u'' would probably be mapped to the same node, but u' and v' probably diverge. That is, the cost of creating a new node should be greater than the cost of aligning two corrupted versions of the same string, but the cost of diverging should be less than the cost of deleting/inserting all remaining strings. Thus we want $E[\mathsf{ED}(u', u'')] < \alpha < E[\mathsf{ED}(u', v')]$.

$E[\mathsf{ED}(u', u'')]$ depends on the error rate of the corruption process and must be estimated from data. But estimates of $E[\mathsf{ED}(u', v')]$ appear in the literature if the true strings u and v are uniformly random and have equal length. If the cost of substitution is twice that of insertion/deletion, then we can calculate $E[\mathsf{ED}(u', v')]$ using Chvátal–Sankoff numbers [9] (the expected longest common subsequence length for two equal-length random strings); for unit edit costs, it is conjectured that $E[\mathsf{ED}(u', v')] \approx |u'|(1 - \frac{1}{|\Sigma|})$ for random equal-length strings over Σ [32]. If strings are not uniformly random or have different lengths, then we can estimate $E[\mathsf{ED}(u', v')]$ from data and subsequently select an α between the upper and lower bounds. (In fact, there is some evidence that many values of α in this range perform well; see the flat-bottomed basin shape of Figure 5b.)

8 Discussion and Future Work

We described an efficient, practical heuristic to reconstruct a propagation tree from a noisy set of diverging string sequences – and in Section 7 we showed that BuildTree performs well on synthetic data. But, after all, the motivation for introducing this particular theoretical problem was for its application to real data, particularly chain-letter petitions. Rigorously testing BuildTree on real data, then, is perhaps the most natural direction for future work. (Testing on more realistic synthetic data is also an interesting future direction. Our data-generation process in Section 7.1 is unrealistic in a number of ways, perhaps most strikingly in its assumption that a name is a length-k alphabetic string chosen uniformly at random. More realistic randomized name-generation processes would make the synthetic task more similar to the real one.)

That said, there are several potentially interesting theoretical avenues for further exploration of DSSSP, too. The problem is (at least theoretically) tractable for any fixed number m of string sequences – but the dependence on m in our brute-force algorithm is brutal. Is there a more efficient algorithm for small m? Or are there efficient algorithms with provable approximation guarantees for general m? We can approximate the best labels for a fixed tree topology using medoids or, even better, using a PTAS for the string median problem [28]. Identifying the best tree topology seems more challenging, but perhaps this topological source of error in BuildTree (or some other heuristic) can be bounded.

There is another set of interesting open questions related to efficient algorithms for DSSSP when λ is small. (At the other extreme, the problem remains intractable when λ is very large: even if the optimal tree's topology is the trivial non-branching one, as in Figure 2b, choosing the labels requires repeatedly solving instances of the NP-hard string median problem.) DSSSP is trivial when $\lambda = 0$; the diverge-at-the-root tree (as in Figure 2e) is optimal, with total cost 0. Even for strictly positive but small values of λ, there is an easy solution: if

$\lambda \leq \frac{1}{nm}$, it optimal to diverge upon encountering even a one-character difference between strings (i.e., the optimal summary tree is precisely the trie representation of the set X). Do related approaches make the problem tractable for bigger values of λ – e.g., if λ is small enough that we can only afford a bounded budget of edits in node labels? For how large a value of λ are there exact polynomial-time algorithms?

References

1. Lada Adamic, Thomas Lento, Eytan Adar, and Pauline Ng. Information evolution in social networks. In *International Conference on Web Search and Data Mining (WSDM'16)*, 2016.
2. Sebastian Baltes, Christoph Treude, and Stephan Diehl. Sotorrent: Studying the origin, evolution, and usage of Stack Overflow code snippets. In *International Conference on Mining Software Repositories (MSR'19)*, pages 191–194, 2019.
3. Charles Bennett, Ming Li, and Bin Ma. Chain letters and evolutionary histories. *Scientific American*, 288(6):76–81, 2003.
4. Eric Brill and Robert Moore. An improved error model for noisy channel spelling correction. In *Proc. Association for Computational Linguistics (ACL'00)*, pages 286–293, 2000.
5. Theresa Bullard, John Freudenthal, Serine Avagyan, and Bart Kahr. Test of Cairns-Smith's 'crystals-as-genes' hypothesis. *Faraday Discussions*, 136:231–245, 2007.
6. Alexander Graham Cairns-Smith. *Seven clues to the origin of life: a scientific detective story*. Cambridge University Press, 1990.
7. Humberto Carrillo and David Lipman. The multiple sequence alignment problem in biology. *SIAM Journal on Applied Mathematics*, pages 1073–1082, 1988.
8. Flavio Chierichetti, David Liben-Nowell, and Jon Kleinberg. Reconstructing patterns of information diffusion from incomplete observations. In *Advances in Neural Information Processing Systems (NeurIPS'11)*, pages 792–800, 2011.
9. Vacláv Chvátal and David Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12(2):306–315, 1975.
10. Rene De La Briandais. File searching using variable length keys. In *Western Joint Computer Conference*, pages 295–298, 1959.
11. Colin de la Higuera and Francisco Casacuberta. Topology of strings: Median string is NP-complete. *Theoretical Computer Science*, 230(1-2):39–48, 2000.
12. Russell Doolittle. Reconstructing history with amino acid sequences. *Protein Science*, 1(2):191–200, 1992.
13. Da-Fei Feng and Russell Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360, 1987.
14. Campbell Bryce Fraser. *Subsequences and supersequences of strings*. PhD thesis, University of Glasgow, 1995.
15. Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.
16. Adrien Friggeri, Lada Adamic, Dean Eckles, and Justin Cheng. Rumor cascades. In *Eighth International AAAI Conference on Weblogs and Social Media (ICWSM'14)*, 2014.
17. Benjamin Golub and Matthew Jackson. Using selection bias to explain the observed structure of Internet diffusions. *Proceedings of the National Academy of Sciences*, 107(23):10833–10836, 2010.
18. Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. Springer, 2009.
19. Tim Henderson. Zhang-Shasha: Tree edit distance in Python. `https://github.com/timtadh/zhang-shasha`, 2019.
20. Manoel Horta Ribeiro, Kristina Gligoric, and Robert West. Message distortion in information cascades. In *The World Wide Web Conference (WWW'19)*, pages 681–692, 2019.
21. Folgert Karsdorp and Antal Van den Bosch. The structure and evolution of story networks. *Royal Society Open Science*, 3(6):160071, 2016.

22 John Kececioglu. The maximum weight trace problem in multiple sequence alignment. In *Symposium on Combinatorial Pattern Matching (CPM'93)*, pages 106–119, 1993.

23 Joseph Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review*, 25(2):201–237, 1983.

24 Ravi Kumar, Mohammad Mahdian, and Mary McGlohon. Dynamics of conversations. In *Intl. Conference on Knowledge Discovery and Data Mining (KDD'10)*, pages 553–562, 2010.

25 Jure Leskovec, Lars Backstrom, and Jon Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Intl. Conference on Knowledge Discovery and Data Mining (KDD'09)*, pages 497–506, 2009.

26 Vladimir Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

27 Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in Bioinformatics*, 11(5):473–483, 2010.

28 Ming Li, Bin Ma, and Lusheng Wang. Finding similar regions in many sequences. *Journal of Computer and System Sciences*, 65(1):73–96, 2002.

29 David Liben-Nowell and Jon Kleinberg. Tracing information flow on a global scale using Internet chain-letter data. *Proceedings of the National Academy of Sciences*, 105(12):4633–4638, 2008.

30 Daniel Lopresti. Block edit models for approximate string matching. *Theoretical Computer Science*, 181(1):159–179, 1997.

31 Luke Matthews, Jamie Tehrani, Fiona Jordan, Mark Collard, and Charles Nunn. Testing for divergent transmission histories among cultural characters: A study using Bayesian phylogenetic methods and Iranian tribal textile data. *PLoS ONE*, 6(4), 2011.

32 Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

33 François Nicolas and Eric Rivals. Hardness results for the center and median string problems under the weighted and unweighted edit distances. *Journal of Discrete Algorithms*, 3(2-4):390–415, 2005. Previously in CPM'03.

34 Kemal Oflazer. Error-tolerant finite-state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics*, 22(1):73–89, 1996.

35 Kari-Jouko Räihä and Esko Ukkonen. The shortest common supersequence problem over binary alphabet is NP-complete. *Theoretical Computer Science*, 16(2):187–198, 1981.

36 Benjamin Raphael, Degui Zhi, Haixu Tang, and Pavel Pevzner. A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Research*, 14(11):2336–2346, 2004.

37 David Sankoff. Minimal mutation trees of sequences. *SIAM Journal on Applied Mathematics*, 28(1):35–42, 1975.

38 David Sankoff, Robert Cedergren, and Guy Lapalme. Frequency of insertion-deletion, transversion, and transition in the evolution of 5S ribosomal RNA. *Journal of Molecular Evolution*, 7(2):133–149, 1976.

39 Matthew Simmons, Lada Adamic, and Eytan Adar. Memes online: Extracted, subtracted, injected, and recollected. In *International Conference on Web and Social Media (ICWSM'11)*, 2011.

40 Jamshid Tehrani. The phylogeny of Little Red Riding Hood. *PLOS One*, 8(11):e78871, 2013.

41 Robert Wagner. Order-n correction for regular languages. *Communications of the ACM*, 17(5):265–268, May 1974.

42 Michael Waterman, Temple Smith, and William Beyer. Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387, 1976.

43 Henry William Watson and Francis Galton. On the probability of the extinction of families. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 4:138–144, 1875.

44 Carola Wenk. Applying an edit distance to the matching of tree ring sequences in dendrochronology. In *Symposium on Combinatorial Pattern Matching (CPM'99)*, pages 223–242, 1999.

45 Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing*, 18(6):1245–1262, 1989.

46 Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133–139, 1992.