

On the Uniqueness and Complexity of Reconstructing Repeat-Annotated Phylogenetic Trees

Firas Swidan*

Michal Ziv-Ukelson[†]

Ron Y. Pinter*

Abstract

Phylogenetic inference is an essential problem in evolutionary studies. It is being addressed by a wide class of methods, ranging from those based on phenotypic morphologies to others using various genotypic mutations. Yet, they all lead to results which are potentially both ambiguous and biologically inconclusive. In this paper, a novel approach is presented for tackling the inference problem, based on recent biological findings indicating a strong association between *reversals* (aka *inversions*) and *repeats*. These biological findings are formalized here in a new model, called *repeat-annotated phylogenetic trees* (RAPT). We show that under RAPT, the evolutionary process — including both the tree-topology as well as the ancestral node assignments — is *uniquely* determined, a property that is of major significance in theory and in practice. Furthermore, the repeats are employed to provide linear-time algorithms for reconstructing both the assignments and the phylogeny, which are NP-hard problems under the classical model of *sorting by reversals* (SBR) [32].

1 Introduction

Phylogenetic inference and ancestral genome order reconstruction are important problems in evolutionary, genetic, and bioinformatic studies [10, 40]. Given one-to-one mappings between orthologous segments of a set of organisms, one seeks to reconstruct the organisms' phylogeny as well as the genomic order (*i.e.*, the order of the genomic segments) of their ancestors. An example demonstrating the problem is given in Figures 1a and 1c. Here, a one-to-one mapping of the orthologous segments of the two strains *Xanthomonas campestris* pathovar *campestris* ATCC 33913 (*X. campestris*) [15] and 8004 (*X. campestris* 8004) [34] is presented schematically. These bacteria cause black rot disease in crucifers such as *Brassica* (cabbage) and *Arabidopsis* (mustard), which results in severe losses in agricultural yield world-wide [15]. The goal is to reconstruct the genomic order of the bacteria's cenancestral, *i.e.*, the bacteria's most recent common ancestor. It is obvious from the figure that 3 reversals have affected the two bacteria since their divergence. However, during the speciation of which of the two bacteria have these reversals occurred and what is the ancestral genomic order?

Using current methods, reconstructing ancestral genomic order involves solving a multiple *sorting by reversals* (SBR) problem. In SBR, which has been thoroughly examined over the last two decades [3–8, 13, 14, 18–24, 46], one represents the one-to-one orthologous mappings as permutations and the inversion mutations as reversal operations. The goal of the optimization problem is to find a phylogeny and corresponding reversal scenarios along its branches with the minimum number of reversals. In SBR, however, the ancestral genomic order cannot be implied based on the comparison of a pair of genomes, as is needed for the bacteria pair in Figure 1a. Moreover, adding one more organism to enable deducing the ancestral order (in which case the problem is known as the *median problem* [41]), makes this task NP-hard [12].

*Department of Computer Science, Technion – Israel Institute of Technology, Haifa 32000, Israel. Email: {firas,pinter}@cs.technion.ac.il.

[†]School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. Email: michaluz@post.tau.ac.il.

Nonetheless, exhaustive search techniques [30] as well as heuristics [9] were developed for tackling this problem.

The pairwise ancestral genome order reconstruction problem is usually extended to the reconstruction of large phylogenetic trees over multiple input leaves (including or excluding the recovery of genomic order in internal nodes). Current phylogenetic inference methods, based on different biological evidence ranging from phenotypic morphologies to various genotypic mutations — including point substitutions (distance-based, maximum-likelihood, and parsimony approaches), insertions and deletions (the Dollo parsimony approach), and genome rearrangements (distance-based and parsimony approaches) — are computationally hard. Moreover, current approaches often yield many alternative solutions; choosing the most sound phylogeny among them has very important biological consequences, but is yet a very challenging task [11, 28, 47].

In this paper we investigate a new approach to phylogenetic inference and ancestral genome order reconstruction. The new approach is inspired by a recent biological discovery regarding the role of *repeats*, *i.e.*, short genomic sequences that are highly similar to each other, in inducing reversals (or recombinations in general). Several studies indicate a strong association between repeats and recombination events [2, 35–38]. Either by a mechanism of illegitimate recombination [29], or by a mechanism of homologous recombination (dependent on RecA) [26, 39], inverted repeats cause rearrangements — see [11, 25] for reviews and Figure 2 for an illustration. Moreover, these findings demonstrate that most of the repeats engaged in reversals correspond to *mobile DNA* elements, *i.e.*, regions of DNA that selfishly duplicate and move into new sites [27]. Hence, these repeats are usually found only in the organism affected by the reversals; see, *e.g.*, [31]. This new and important information regarding the repeats became accessible recently with the availability of many sequenced genomes and its automatic generation is made possible by the development of accurate comparative genome mapping methods [34, 43]. Taking repeats into consideration both makes the modeling more realistic and increases its potential for producing biologically relevant insights.

Example. Qian *et al.* [34] demonstrate an initial application of inverted repeats to the task of ancestral genome order reconstruction in the *Xanthomonas campestris* bacteria. They have identified two identical IS1478-related insertion sequences (corresponding to the repeat pair $-b, b$ in Figure 1b) spanning a putative recombination site. Moreover, they predicted a rearrangement scenario for transforming one genome to the other — see <http://www.genome.org/content/vol10/issue2005/images/data/gr.3378705/DC1/SI.Fig.2.gif> for a detailed (and vivid) animation of this contribution. We continue their analysis by applying our approach to the very same data: in Figures 1b and 1d we incorporate the information regarding the repeats into the mapping. In addition to the repeat pair reported by Qian *et al.*, we identify two more pairs spanning two putative recombination sites. According to the repeats, one inversion occurred during the speciation of *X. campestris*, while two inversions occurred during the speciation of *X. campestris* 8004. Given this information, deducing the ancestral genomic order is straightforward, as demonstrated in Figure 1d.

The above example demonstrates how the additional information contributed by the repeats can be utilized to uniquely determine the order of the genomic segments in the cenancestral genome — based solely on pairwise genomic comparisons. Furthermore, the “repeat footprints” can be applied to aid the efficient computation of ancestral genomic orders. To generalize these observations, in Section 2 we formalize the biological assumptions introduced above into a theoretical evolutionary model. For the pairwise case, we present two important results: uniqueness of solutions and simplification of computation (Section 3). In Section 4 we show that these two results scale up to the more general case of multiple genomes. For a formal overview of the combinatorial results of this paper we refer the reader to Section 2.1.

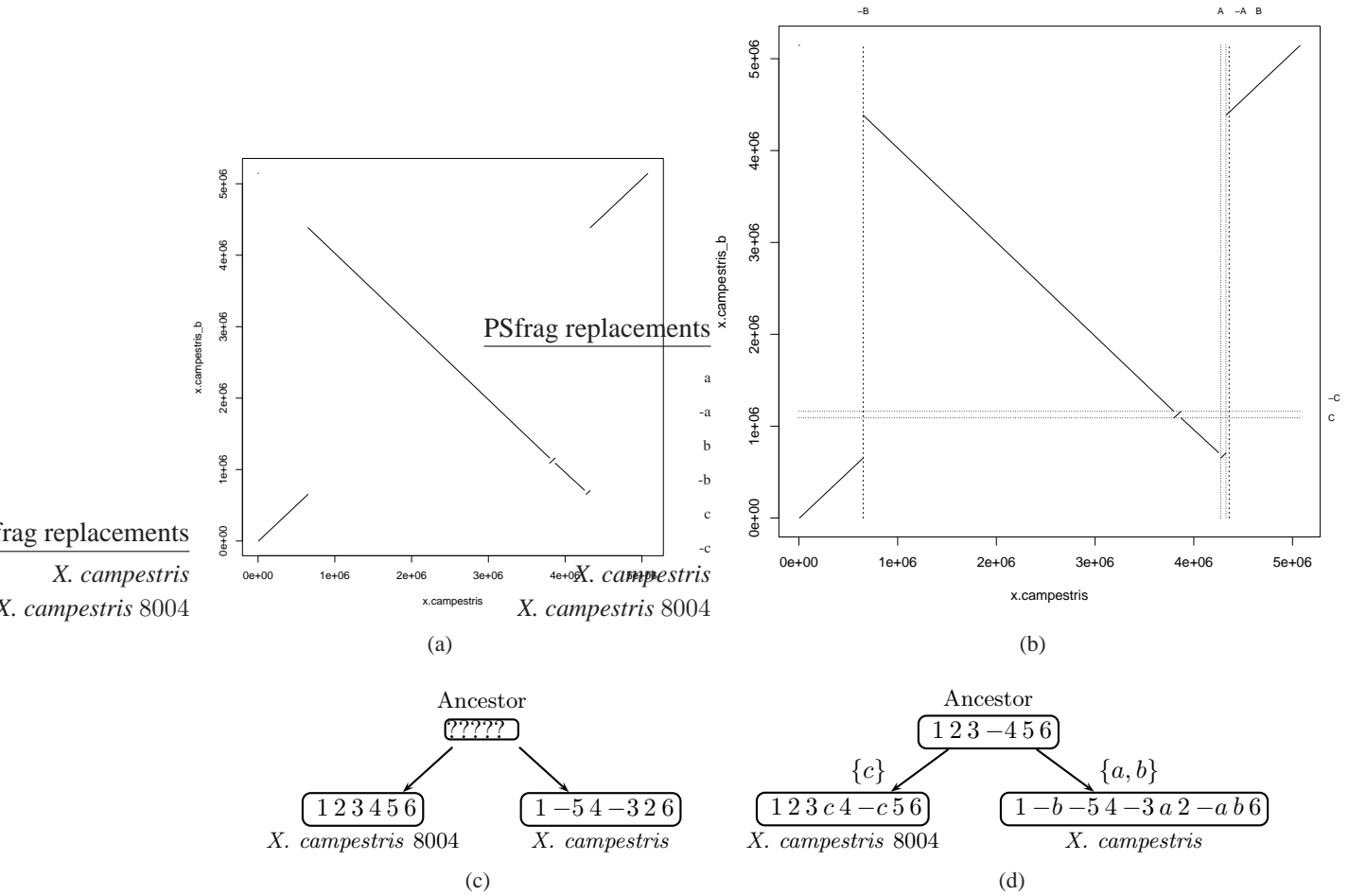


Figure 1: Inferring the order of genomic segments in the ancestor of the bacteria *X. campestris* and *X. campestris* 8004. (a) The comparative mapping is the result of applying MAGIC [43] to the considered organisms. MAGIC was run with its default parameters, except for discarding segments of length smaller than 10000bp. The lines in the figure represent corresponding rearrangement-free (RF) segments in the two bacteria (since the genomes are circular, the square drawing should be wrapped into a torus). The mapping suggests that 3 reversals have occurred since the divergence of the two bacteria, in agreement with the observations made by Qian *et. al.* [34]. (b) Incorporating the repeats into the mapping. The repeats were obtained by applying Repseek [1] to each one of the genomes separately. Repseek was configured to detect inexact repeats having exact seeds of length greater than 25, according to the biological findings in [25]. The repeats are represented by dashed lines and marked by $-a$, a , $-b$, b , and $-c$, c . The repeat pair $-c$, c is found in *X. campestris* 8004 and correspond to the genomic segments (706898, 708420) and (4384023, 4385545), which are 1522bp long and share $> 99\%$ identity. The other two pairs of repeats are found in *X. campestris*. The pair a , $-a$ corresponds to the genomic segments (4271445, 4272983) and (4326214, 4327752), which are 1538bp long and share $> 99\%$ identity. The pair $-b$, b , which has been reported in [34], corresponds to the genomic segments (650675, 652202) and (4326231, 4327758), which are 1527bp long and share $> 99\%$ identity. All the segments $\pm a$, $\pm b$, and $\pm c$ have a high translated sequence similarity (on the amino acid level) to the insertion sequence IS1478. (c) The phylogeny and the permutations corresponding to the mapping in (a). The permutation of *X. campestris* 8004 is chosen to equal the identity permutation. Yet, one cannot infer the ancestral genomic order or, correspondingly, decide during the speciation of which of the bacteria the reversals have occurred. (d) Applying the new approach which consists of including the repeats in the permutations and using them to annotate the edges in the phylogeny: deducing the genomic order of the ancestor by inverting the permutation elements surrounded by the repeats yields a unique solution to the common ancestor gene order recovery problem.

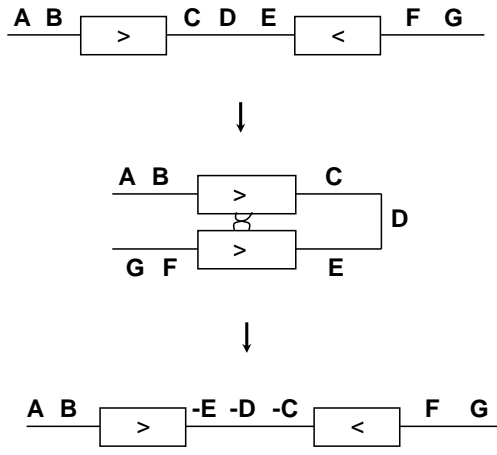


Figure 2: A schematic demonstration of a putative mechanism for a recombination caused by inverted repeats: during DNA replication, a chromosome containing an inverted repeat pair forms looping structures resulting in an inversion.

2 A Formal Model Based on Repeats

The model described in this paper is based on the following biological assumptions:

1. Reversals are usually induced by *inverted* repeat pairs, *i.e.*, repeats having opposite orientation [25].
2. Repeats engaged in reversals — corresponding mostly to mobile DNA elements — are easily identified on the borders of reversed genomic segments, and are present only in the affected organism [25, 31].
3. The information mapping each repeat to its pair-mate is part of the input¹.
4. Each repeat has a very low probability for causing a reversal that remains fixed in the population [43].

Therefore, in our model we assume that each repeat causes up to one reversal.

Note that, though the above assumptions may not capture the great variety found in real biological problems, it is easy to check if a given set of input genomes comprises a reasonable target for our approach. Furthermore, as demonstrated by the theoretical results listed in Section 2.1, the elementary assumptions above offer a solid basis for potential future extensions and enhancements.

Based on Assumption 3, the input to our problem comprises sequences, referred to as *repmaps*, of both permutation elements, belonging to a set N , and paired repeats, belonging to a set R . Each permutation element appears exactly once in the repmap, while each repeat appears exactly twice. In addition to the permutation elements (represented by digits) and based on Assumption 1 the repeats (represented by lowercase characters) are also signed.

Given a repmap s , two repeat elements $s_i, s_j \in R$ are considered a pair if $|s_i| = |s_j|$ (*i.e.*, their absolute values are equal). If they have opposite signs ($s_i = -s_j$) we refer to them as an *inverted repeat pair*; otherwise they are called a *direct repeat pair*. The set of repeats appearing in s is denoted by $R(s) = \{|s_i|, s_i \in R\}$ and referred to as the *repeat set*. We denote the restriction of s to the permutation elements in N by $s|_N$ and refer to it as the *induced permutation*. The restriction of s to the repeat elements is denoted by $s|_R$ and referred to as the *repeat subsequence*.

Example. Consider the repmap $s = 1 \ a \ -4 \ -a \ -b \ 3 \ 2 \ -b \ 5$. Here, $+a, -a$ is an inverted repeat pair, while $-b, -b$ is a direct repeat pair. Moreover, we have $R(s) = \{a, b\}$. The induced permutation is $s|_N = 1 \ -4 \ 3 \ 2 \ 5$, and the repeat subsequence is $s|_R = a \ -a \ -b \ -b$.

The next three definitions are intended to formalize the biological assumptions into a mathematical model of an evolutionary process. The first definition is based on Assumption 1, as follows.

¹This information can be obtained using techniques similar to those standardly used for preparing permutations.

$$\begin{array}{l}
s = g10 \ -a \ -b \ g12 \ b \ -c \ g3 \ c \ -d \ -g5 \ d \ a \\
s' = 1 \ -a \ -8 \ -b \ 7 \ b \ -6 \ -c \ 5 \ c \ -4 \ -d \ 3 \ d \ -2 \ a \ 9 \\
\text{(a)} \\
\\
S' = 1 \ -a \ 2 \ -d \ 3 \ d \ 4 \ -b \ 5 \ b \ 6 \ -b \ 7 \ b \ 8 \ a \ 9 \\
S = g10 \ -a \ -d \ -g5 \ d \ -c \ g3 \ c \ -b \ g12 \ b \ a \\
\text{(b)} \\
\\
\begin{array}{l}
s^* = 1 \ -a \ -b \ [4] \ b \ -c \ 3 \ c \ -d \ 2 \ d \ a \\
1 \ -a \ -b \ -4 \ b \ -c \ [3] \ c \ -d \ 2 \ d \ a \\
1 \ -a \ -b \ -4 \ b \ -c \ -3 \ c \ -d \ [2] \ d \ a \\
1 \ -a \ [-b \ -4 \ b \ -c \ -3 \ c \ -d \ -2 \ d] \ a \\
S^* = 1 \ -a \ -d \ 2 \ d \ -c \ 3 \ c \ -b \ 4 \ b \ a \\
\text{(c)}
\end{array}
\qquad
\begin{array}{l}
s^*|_N = 1 \ [4 \ 3] \ 2 \\
1 \ -3 \ [-4 \ 2] \\
1 \ [-3 \ -2] \ 4 \\
S^*|_N = 1 \ 2 \ 3 \ 4 \\
\text{(d)}
\end{array}
\end{array}$$

Figure 3: Calculating the ancestor assignment (a-b) and comparing a legal scenario to an SBR scenario (c-d). (a) Example of transforming a repmap s to a normalized repmap s' . The correspondence between the permutation elements of s and those of s' is drawn as edges connecting between the respective elements. The permutation elements in s are given over a different alphabet for clarity sake. (b) Determining the ancestor in normalized format S' and in input format S from s' . In both (c) and (d) we assume that the ancestor S is known. We rename s and S to s^* and S^* to enable running SBR on them and we compare a legal scenario (c) to an SBR scenario (d). The reversals are denoted by brackets, $[\]$. Note that the reconstructed scenarios are different, since one of them is guided by fulfilling the constraints imposed by the repeats, while the other is aiming toward minimizing the number of reversals. Lemmas 5 and 14 show that if s^* is a normalized repmap then a scenario is legal iff it is SBR.

Definition 1 (Legal Reversal). Let $s = s_1, \dots, s_n$ be a repmap and let $\rho = \rho(i, j)$ for $1 < i < j < n$ be a reversal affecting the subsequence s_i, \dots, s_j in s . The reversal ρ is called *legal* if it is bordered by an inverted repeat pair, *i.e.*, if $s_{i-1} = -s_{j+1}$ (see Figure 3c). We say then that ρ *fulfills* the repeat pair s_{i-1}, s_{j+1} .

The next two definitions are both based on Assumptions 2 and 4.

Definition 2 (Legal Scenario). Given a reversal sequence $\varrho = \rho_1, \dots, \rho_m$ affecting s , we say that ϱ is a *legal scenario relative to a subset of repeat pairs* $\mathcal{R} \subseteq R(s)$ if $\forall i \in \{1, \dots, m\}$, ρ_i is a legal reversal when acting on $s \cdot \rho_1 \cdots \rho_{i-1}$ and if ϱ fulfills each repeat in \mathcal{R} exactly once (see Figure 3c). If $\mathcal{R} = R(s)$, we refer to ϱ simply as a *legal scenario*. If $\mathcal{R} \neq R(s)$ is obvious from the context, we refer to ϱ as a *partially legal scenario*.

Definition 3 (RAPT). Given a repmap S (ancestor), a *Repeat-Annotated Phylogenetic Tree* originating in S (see Figure 4) is a triplet (T, f, g) , where $T = (V, E)$ is a directed tree with root $v_r \in V$ such that all the inner nodes (except perhaps the root) are of degree ≥ 3 , $f : V \rightarrow (R \cup N)^*$ maps assignments to the nodes, and $g : E \rightarrow 2^{R(S)}$ maps labels to the edges, such that:

1. The edge labels are a partition of $R(S)$, *i.e.*, for every two edges $e, e' \in E : g(e) \cap g(e') = \emptyset$ and $\bigcup_{e \in E} g(e) = R(S)$.
2. The assignments to the nodes fulfill the following two requirements:
 - (a) The assignment to the root v_r equals S , that is $f(v_r) = S$.

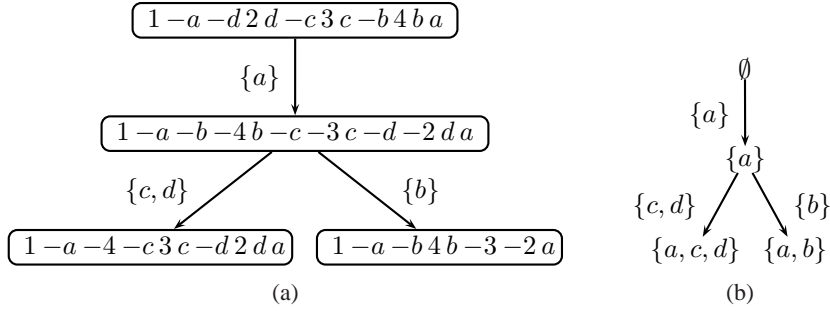


Figure 4: An example of a RAP tree (a) and the corresponding set-trie (b). The set-trie is obtained from the RAP tree by preserving both the tree topology as well as the edge labels of the RAP tree, while discarding the node assignments. The virtual node assignments of the set-trie are determined from the edge labels — see Definition 6.

- (b) Assuming $u \in V$ is the immediate parent of $v \in V$ and that $e \in E$ is the edge connecting them, we require that $g(e) \subset R(f(u))$ and that there exists a legal scenario ρ_1, \dots, ρ_m with respect to $g(e)$ such that $(f(u) \cdot \rho_1 \cdots \rho_m)|_N = f(v)|_N$ (Definition 2).
- 3. The repeat set $R(s)$ of a leaf reperm s contains only repeats that engaged in reversals throughout the history of s , i.e., $R(s) = g(\text{path}(v_r, s))$.

2.1 The Main Results of This Paper

In this paper we study the following problems: Can one reconstruct an unknown RAP tree (T, f, g) given, as input, a set L of the corresponding leaf reperms? More specifically, does L uniquely determine the RAP tree? Are the legal scenarios linking the assignments in the RAP tree nodes unique? Furthermore, can one efficiently reconstruct the unknown RAP tree and the corresponding scenarios? These questions are answered as follows:

First, in Section 3, we consider the basic case in which the tree T of the RAP tree contains a single leaf and a single ancestor. Since in this case both the tree T and the labels g are trivially determined, our results pertain to both the scenario and the ancestral assignment reconstructions, as follows:

Uniqueness: We show that the ancestral assignment is uniquely determined (Section 3.1). This result is surprising given the ambiguity of the scenarios (Section 3.2).

Complexity: We give a linear-time algorithm for reconstructing the ancestor (Section 3.3.1). Contrary to the classical SBR problem, our algorithm utilizes the constraints introduced by the repeats to calculate the unique ancestor. This algorithm is then employed to solve the problem of reconstructing a plausible legal scenario, by a reduction to SBR. (Section 3.3.2).

Next, in Section 4, the multiple leaf RAP tree is studied. Based on the results obtained for the single leaf case, it is straightforward to show that, in the multiple leaf case, the tree topology T , the edge labels g , and the leaf reperms L both uniquely determine the induced permutations in the inner node assignments and also enable their reconstruction in linear-time. Hence, the complexity and uniqueness issues are reduced to the pair (T, g) (see Figures 4 and 6). To investigate the latter, we introduce a new data structure, which is

an abstraction of such (T, g) pairs, called *set-tries*, which are trie-like structures over sets instead of words (Section 4.1).

Uniqueness: We define a new property on sets (*monotonic common subsets*) and assert that the repeat sets of the RAPT leaf assignments follow this property. Then we show that a monotonic leaf set collection uniquely determines the underlying set-trie (Section 4.2).

Complexity: We give a linear-time algorithm to efficiently reconstruct set-tries from monotonic set collections (Section 4.3).

3 The Single Leaf RAPT Case

Throughout this section, we assume without loss of generality that the repmaps are given in an easy to handle format, as follows. Consider a repmap (ancestor) S on which a legal scenario ϱ was applied and denote the result by $s = S \cdot \varrho$. (The notation of S denoting the ancestor, and s denoting the descendant is consistently used throughout the section.) We assume that S (and hence s) starts and ends with a permutation element (otherwise it can be padded). In addition, we assume that S (and hence s) does not contain successive permutation elements (or otherwise they can be united to form a single new permutation element).

Note that, whereas uniting successive permutation elements into a single element is straightforward, dealing with successive repeat elements in the input sequence is more challenging. For instance, having successive repeats implies that the corresponding breakpoints may be reused (the issue of breakpoint reuse has been repeatedly debated in the literature and is currently controversial [9, 33, 42, 45]). From a modeling point of view, however, successive repeats distinguish the RAPT problem from the SBR problem: When they are present in a repmap, its set of legal scenarios (see Definition 2) and the set of SBR scenarios [23] of its induced permutation are not equal, as demonstrated in Figure 3. This is due to the fact that SBR aims to minimize the number of reversals, while RAPT is driven by the objective of fulfilling the constraints imposed by the repeats. Still, for a subset of the repmaps, both sets of legal and SBR scenarios are equal. We refer to these special repmaps as “normalized” and define them below. Because of the above equivalence between legal and SBR scenarios on normalized repmaps, SBR techniques can be employed to the study of the properties associated with legal scenarios on such repmaps. Then, a transformation from general repmaps to their normalized form, which preserves the above studied properties, will be sought. Examples for normalized repmaps are given in Figures 3b and 3a.

Definition 4 (Normalized Repmap). A repmap S is a *normalized repmap* if between every two repeats in it there is a permutation element from N .

3.1 Asserting Uniqueness of Ancestor

In this section we prove that all legal scenarios lead to the same ancestral repmap. This result is surprising given the richness of the set of all legal scenarios (see Section 3.2). We first consider the special subclass of normalized repmaps. In this subclass, the proof of uniqueness involves a breakpoint counting argument, showing that all legal scenarios are optimal sorting scenarios (namely SBR scenarios) and is asserted in a series of assertions (Claim 1–Lemma 7). Next we extend the uniqueness claim to the general repmap case in another series of assertions (Claim 8–Theorem 10). The proof here is achieved by transforming any given repmap to a corresponding normalized one and by asserting that this transformation indeed preserves the uniqueness property.

Without loss of generality, we assume that $S|_N$ is sorted (the elements can always be renamed to accommodate this order). In the following we investigate legal scenarios affecting normalized repmaps, and show that all legal scenarios are SBR scenarios.

Claim 1. *Let S be a repmap, ϱ a legal scenario, and $s = S \cdot \varrho$. Then S is normalized iff s is normalized.*

Proof. Assume S is normalized. We show that s is normalized as well. The other direction is obtained by changing the roles of S and s .

The proof is done by induction on m , the number of reversals in the scenario ϱ .

Base case: for $m = 1$, assume the first reversal ρ_1 fulfills the inverted repeat pair S_i and S_j , where $i < j$. Since S is normalized, we have $S_{i+1}, S_{j-1} \in N$ (notice that $i + 1 \leq j - 1$). Denote $t = S \cdot \rho_1$. Then we get $t_{i-1}, t_i, t_{i+1} = S_{i-1}, S_i, -S_{j-1}$ and $t_{j-1}, t_j, t_{j+1} = -S_{i+1}, S_j, S_{j+1}$, and hence each of the repeats t_i and t_j is still surrounded by permutation elements. The rest of the repmap stays trivially normalized.

The induction step is established similarly. \square

Claim 2. *Let S be a normalized repmap and $\varrho = \rho_1, \dots, \rho_m$ a partially legal scenario. Denote $t = S \cdot \rho_1$, where ρ_1 fulfills the inverted repeat pair S_i and S_j . Then the surroundings t_{i-1}, t_i, t_{i+1} and t_{j-1}, t_j, t_{j+1} of the fulfilled repeat pair t_i and t_j remain consecutive throughout the rest of the reversals in ϱ .*

Proof. By Claim 1, t is normalized. Moreover, note that $t_i = S_i$ and $t_j = S_j$. Thus, the surroundings of t_i , namely t_{i-1}, t_i, t_{i+1} , and those of t_j , namely t_{j-1}, t_j, t_{j+1} , contain no repeats. Since the inverted repeat pair t_i and t_j is already fulfilled, and since it is assumed that each repeat pair is fulfilled exactly once, no reversal can cut through their surroundings. \square

By induction on Claim 2 we get the following corollary.

Corollary 3. *Let S be a normalized repmap and $\varrho = \rho_1, \dots, \rho_m$ a partially legal scenario. Denote $t = S \cdot \rho_1 \cdots \rho_k$ for $k < m$. The surroundings of any fulfilled repeat pair in ρ_1, \dots, ρ_k remain consecutive throughout the rest of the reversals in ϱ .*

Corollary 3 implies that the surroundings of each repeat in s are the same as its surroundings after the reversal fulfilling it was applied during the scenario ϱ . This observation as well as the following definition and theorem, commonly used in SBR, are helpful for the reconstruction step, which we consider next.

Definition 5 (Breakpoint [23]). Given a repmap s , a *breakpoint* in $t = s|_N$ is a pair of successive elements t_i, t_{i+1} , such that $t_{i+1} - t_i \neq 1$. A *breakpoint* in s is a pair of permutation elements that is a breakpoint in $s|_N$. If $s|_N$ contains no breakpoints, we call it *sorted*. We call s *sorted* if $s|_N$ is sorted.

Theorem 4 (Kececioglu and Sankoff 1993 [23]). *Each reversal can create at most two breakpoints.*

Lemma 5. *Let S be a (sorted) normalized repmap and $\varrho = \rho_1, \dots, \rho_m$ be a legal scenario. Denote $s = S \cdot \varrho$. Then, a legal reversal affecting s eliminates two breakpoints.*

Proof. Suppose the reversal fulfills the inverted repeat pair s_p and s_q in s . Let ρ_k for $k < m$ be the reversal in the scenario ϱ fulfilling this repeat pair. Denote $t = S \cdot \rho_1 \cdots \rho_k$ and let t_i and t_j be the corresponding elements to s_p and s_q in t , respectively. Without loss of generality, we assume that $i < j$ (otherwise rename p and q). By Corollary 3 the surroundings of s_p must be either $t_{i-1}, t_i, -t_{j-1}$ or $t_{j-1}, -t_i, -t_{i-1}$. Similarly, the surroundings of s_q must be either $-t_{i+1}, t_j, t_{j+1}$ or $-t_{j+1}, -t_j, t_{i+1}$. However, since t_i and t_j are an inverted pair in t , and s_p and s_q are an inverted pair in s , t_i and t_j (as well as their surroundings by Corollary 3) must have been affected either both by an odd or both by an even number of reversals throughout the rest of the scenario ϱ . Therefore, if the surroundings of s_p equal $t_{i-1}, t_i, -t_{j-1}$, then the surroundings of s_q must equal $-t_{i+1}, t_j, t_{j+1}$. If, on the other hand, the surroundings of s_p equal $t_{j-1}, -t_i, -t_{i-1}$ then the surroundings of s_q must equal $-t_{j+1}, -t_j, t_{i+1}$.

In the first configuration, if $q < p$ we get that after performing the legal reversal the surroundings become $-t_{i+1}, t_j, -t_{i-1}$ and $-t_{j+1}, t_i, -t_{j-1}$. Note that t_{i-1} and t_{i+1} are successive in t , and have not been affected by any reversal throughout ρ_1, \dots, ρ_k . Therefore, they are successive in S as well. A similar claim holds for t_{j-1} and t_{j+1} . Thus, the legal reversal combines two pairs of successive elements and hence reduces the number of breakpoints by 2. The case $p < q$ and the other configuration is dealt with similarly. \square

By induction on Lemma 5 we get the following corollary.

Corollary 6. *Let S be a normalized repmap and $\varrho = \rho_1, \dots, \rho_m$ a legal scenario. Denote $s = S \cdot \varrho$. Then, each reversal in a partially legal scenario ϱ' affecting s eliminates two breakpoints.*

Let $k = |R(S)|$ be the number of different repeats in S . Since each repeat is fulfilled once, the number of reversals in a legal scenario affecting S is k as well. By Theorem 4, the number of breakpoints in s is bounded by $2k$. Corollary 6 implies that a legal scenario eliminates $2k$ breakpoints (and does not create new ones). Thus, all legal scenarios must be SBR scenarios and hence lead to the same unique ancestor S .

Lemma 7. *Let S be a normalized repmap, ϱ a legal scenario, and $s = S \cdot \varrho$. Then, all legal scenarios affecting the (normalized) repmap s result in the same correct ancestor S .*

Now assume that S is a repmap (not necessarily normalized). We need to show that all legal scenarios affecting s result in S . In order to achieve this, we transform S to a normalized repmap and apply Lemma 7. The transformation is done by adding rational numbers between successive repeats. Thus the resulting repmap is no longer a permutation of integers. To distinguish it, we refer to it as an *extended repmap*.

Claim 8. *Let S be a sorted repmap, ϱ a legal scenario, and $s = S \cdot \varrho$. There exists an extended normalized repmap S' such that S is a subsequence of S' , $S|_R = S'|_R$, and S' is sorted.*

Proof. Let $S_j \cdots S_k \in R^*$, for $j < k$, be a block of successive repeats in S surrounded by permutation elements $S_{j-1}, S_{k+1} \in N$. Adding the rational number $S_{j-1} + (i - j + 1)/(k - j + 1)$ after the repeat S_i for $i \in \{j, \dots, k - 1\}$ and repeating the process for all blocks of successive repeats result in a sorted (extended) normalized repmap S' having S as a subsequence, which fulfills $S|_R = S'|_R$ as well. \square

Note that since the repeat sequences in S and S' are the same, the legal scenarios affecting S and S' are also the same. Thus we can simulate ϱ on S' . Denote the result of this simulation by $s' = S' \cdot \varrho$. Note that since the elements of S in S' undergo the same scenario when applying ϱ to either of S or S' , the relative order between these elements in s and s' is the same. This observation establishes the following claim.

Claim 9. *Let S be a repmap, ϱ a legal scenario, and $s = S \cdot \varrho$. Let S' be the normalization of S as in Claim 8 and let $s' = S' \cdot \varrho$. Then s is a subsequence of s' .*

Claim 9 implies that s and s' have the same set of legal scenarios. However, s' is normalized. Let ϱ' be a legal scenario affecting s and s' . By Lemma 7, we know that $S' = s' \cdot \varrho'$. By applying Claim 9 to s, s' and ϱ' (where $S \leftarrow s, S' \leftarrow s', \varrho \leftarrow \varrho'$), we get that $P = s \cdot \varrho'$ is a subsequence of $S' = s' \cdot \varrho'$, and is hence sorted. Since, by definition, S is a subsequence of S' as well, and P and S have exactly the same elements, we get that $S = P$ and S is sorted. Thus, we conclude that all legal scenarios on s must result in the same correct ancestor S .

Theorem 10. *Let S be a repmap, ϱ a legal scenario, and $s = S \cdot \varrho$. Then, all legal scenarios affecting s result in the same correct ancestor S .*

3.2 Are the Reconstructed Scenarios Unique?

Theorem 10 would have been trivial to prove had there been a single legal scenario affecting s , or alternatively, if all the legal scenarios affecting s were “very similar”. In this section we show that such is not the case, and that these scenarios might be significantly different.

Given two disjoint reversals, one can always reorder them to get different scenarios. However, these scenarios, despite being different, are “very similar”, and trivially yield the same ancestral repmap. Formally, we define “very similar” as an equivalence relation over the space of scenarios as follows: Given a reversal

ρ affecting a repmap s , we define the image $\text{Im}(\rho)$ of the reversal ρ to be the set of permutation elements in s that the reversal affects.

Example. Suppose $s = 1 \ a \ -4 \ -a \ [-b \ 3 \ 2 \ b] \ 5$, and $\rho = \rho(5, 8)$ (designated by the brackets $[\]$), then $\text{Im}(\rho) = \{3, 2\}$.

Thus, given a scenario $\varrho = \rho_1, \dots, \rho_m$, we define the image of the scenario to equal the set of images collected from all its reversals: $\text{Im}(\varrho) = \{\text{Im}(\rho_i) : i \in \{1, \dots, m\}\}$. Given two legal scenarios ϱ_1 and ϱ_2 affecting the same repmap s , we say that they are equivalent, and denote $\varrho_1 \equiv \varrho_2$, if they affect the same sets of elements in s , *i.e.*, $\text{Im}(\varrho_1) = \text{Im}(\varrho_2)$. One can easily verify that the above relation is an equivalence relation over the space of scenarios.

We can now check the “complexity” of the set of legal scenarios affecting a repmap s relative to the above defined equivalence relation. More specifically, the question is whether the set of legal scenarios affecting a repmap s is contained in a single equivalence class (or is the reconstructed scenario unique w.r.t. the equivalence relation)? Figure 5 demonstrates that such is not the case.

$$\begin{array}{r}
s = \begin{array}{cccccccccccccccc}
1 & a & -7 & b & 3 & c & [6 & -b & -2 & -a & 8 & d & -4] & -c & -5 & d & 9 \\
1 & a & -7 & b & 3 & c & 4 & -d & [-8 & a & 2 & b & -6 & -c & -5] & d & 9 \\
1 & a & -7 & b & [3 & c & 4 & -d & 5 & c & 6] & -b & -2 & -a & 8 & d & 9 \\
1 & a & [-7 & b & -6 & -c & -5 & d & -4 & -c & -3 & -b & -2] & -a & 8 & d & 9 \\
S = 1 & a & 2 & b & 3 & c & 4 & -d & 5 & c & 6 & -b & 7 & -a & 8 & d & 9
\end{array} \\
\text{(a)}
\end{array}$$

$$\begin{array}{r}
s = \begin{array}{cccccccccccccccc}
1 & a & [-7 & b & 3 & c & 6 & -b & -2] & -a & 8 & d & -4 & -c & -5 & d & 9 \\
1 & a & 2 & b & [-6 & -c & -3] & -b & 7 & -a & 8 & d & -4 & -c & -5 & d & 9 \\
1 & a & 2 & b & 3 & c & [6 & -b & 7 & -a & 8 & d & -4] & -c & -5 & d & 9 \\
1 & a & 2 & b & 3 & c & 4 & -d & [-8 & a & -7 & b & -6 & -c & -5] & d & 9 \\
S = 1 & a & 2 & b & 3 & c & 4 & -d & 5 & c & 6 & -b & 7 & -a & 8 & d & 9
\end{array} \\
\text{(b)}
\end{array}$$

Figure 5: An example of a repmap s having two inequivalent legal scenarios. The reversals affect different sets of permutation elements in the two scenarios. In (a) the sets of permutation elements are $\{\{6, 2, 8, 4\}, \{8, 2, 6, 5\}, \{3, 4, 5, 6\}, \{7, 6, 5, 4, 3, 2\}\}$. But in (b) the sets of permutation elements are $\{\{7, 3, 6, 2\}, \{6, 3\}, \{6, 7, 8, 4\}, \{8, 7, 6, 5\}\}$. Thus, the two scenarios are not equivalent.

3.3 Algorithms for Ancestor and Scenarios Reconstruction

Given a repmap $s = S \cdot \varrho$, where S and ϱ are unknown, in this section we present a linear-time algorithm for reconstructing S and a sub-quadratic algorithm for reconstructing a possible legal scenario ϱ' . The reconstruction of the ancestor in linear-time is made possible by utilizing the constraints introduced by the repeats and the strong connection established between the repeats and their surroundings in normalized repmaps (Corollary 3). In fact, we show first how to transform s to a normalized repmap s' for which the ancestor S' is sorted² based only on the repeat subsequence $s|_R$. Then we simply rename S' to S . Unlike Claim 8 where we transformed a *sorted* repmap to an extended normalized one, here the transformation to a normalized format is done based on the repeats in s and without knowing the ancestor repmap S .

²Note that, unlike the previous section, here we can no longer assume without loss of generality that S is sorted.

After computing the ancestor we solve the problem of finding a legal scenario transforming s to S in sub-quadratic time by a reduction to SBR. In the general case, as exemplified in Figures 3c and 3d, applying SBR to $s|_N$ may yield illegal scenarios. This is due to the fact that SBR aims to minimize the number of reversals, while RAPT is driven by the objective of fulfilling the constraints imposed by the repeats. However, this barrier is overcome here by transforming a repmap to its normalized format, which intuitively uses $O(|s|)$ additional “virtual” permutation elements to simulate the constraints imposed by the repeats (see Figures 3a and 3b). Thus, to reconstruct a legal scenario, we apply SBR algorithms to the permutation elements of s' and S' and show that the resulting scenario is legal on s . Note that whereas reconstructing the ancestor in linear-time is made possible thanks to the constraints introduced separately by each repeat pair, calculating a legal scenario is complicated by the interaction between the constraints introduced by the different repeat pairs.

3.3.1 Reconstructing the Unique Ancestral Repmap S

Reconstructing the ancestral repmap S can be naïvely achieved by applying some of the techniques demonstrated in [7, 22] to the *overlap graph* constructed over the repeat pairs. This approach, however, yields a quadratic-time algorithm for reconstructing the ancestor and finding a legal scenario.

Here, we present a different approach for tackling the problem. Let S be an ancestral repmap, ϱ a legal scenario affecting S , and $s = S \cdot \varrho$. Consider a transformation of S yielding a normalized sorted repmap S' , and let $s' = S' \cdot \varrho$. According to the above and since all the transformations are reversible, calculating s' from s can be done by the following series of transformations:

$$s \longrightarrow S \longrightarrow S' \longrightarrow s'.$$

However, since S is unknown, this path is intractable. Yet, surprisingly, calculating s' from s can alternatively be done based on the repeat sequence $s|_R$ and without knowing S . Intuitively, this can be explained as follows. Since s' is normalized, the locations of the permutation elements are constrained by the locations of the repeats. Moreover, as Corollary 6 shows, each permutation element is constrained by the repeat next to it. Thus, the position of a permutation element can be determined from local information (the position of a single repeat) in constant time, and the whole repmap can be reconstructed in linear-time. Note that the above transformation implies that the diagram having S , S' , s , and s' as its vertices is commutative:

$$\begin{array}{ccc} S & \xrightarrow{\text{normalize}} & S' \\ \varrho \downarrow & & \downarrow \varrho \\ s & \xrightarrow{\text{normalize}} & s' \end{array}$$

Before describing the formal algorithm, we give an example illustrating its strategy.

Example. Consider the sequence s and the corresponding sequence s' in Figure 3a and suppose that we wish to recover s' based on $s|_R$. The first and second elements in s' are easily fixed, since s' must always start with 1 and the second element in s' always equals to the repeat appearing in the second position of s . Fixing the third element in s' is more challenging. For that, we consider the second element in both s and s' , *i.e.*, the repeat $-a$, and its corresponding pair-mate — the repeat a . By Corollary 3 we know that, since S' is sorted and normalized, once the repeat pair $\{-a, a\}$ got fulfilled while transforming S' to s' , the surroundings of both repeats remain consecutive. In particular, the permutation element 2, which appears directly after the repeat $-a$ in S' must appear in the surroundings of the repeat a in s' ; its exact position (*i.e.*, before or after the repeat) is determined based on two factors: whether the repeat pair is inverted or direct, and whether the preceding permutation element 1 appeared before the repeat $-a$ or after it. In this example, since the repeat pair is inverted and since the preceding permutation element 1 appears before the

repeat $-a$, the permutation element 2 must precede the repeat a in s' (this results from a similar argument to the one presented in the proof of Lemma 5). The sign of the permutation element 2 is determined by similar consideration.

The idea demonstrated in the above example is generalized via the following lemma.

Lemma 11. *Let P be a sorted normalized repmap, ϱ a legal scenario affecting it, and $p = P \cdot \varrho$. Consider a permutation element of p , $1 \neq p_i \in N$. Given the index of the preceding permutation element $p_i - 1$, the index j of the successive permutation element $p_i + 1$ is determined by i , p_i , and the repeat subsequence $p|_R$.*

Proof. Consider the repeats p_{i-1} and p_{i+1} surrounding p_i , and let $p_{i'}$ and $p_{i''}$ be their counterpart repeats, respectively. By Corollary 3, the neighboring permutation elements $p_i - 1$ and $p_i + 1$ are neighbors to the repeats $p_{i'}$ and $p_{i''}$. Since the index of $p_i - 1$ is given, we know near which repeat its found. Suppose first that $p_i - 1$ is found near the repeat $p_{i'}$ (the other case is handled similarly). This implies that $p_i + 1$ is found near the repeat $p_{i''}$. Moreover, by Corollary 3, the sign and the relative order between $p_i + 1$ and the repeat $p_{i''}$ is determined by the sign of p_i and whether the repeat pair $p_{i''}$ and p_{i+1} is a direct or inverted pair. \square

As exemplified above, since the permutation element 1 always appears with a positive sign in the first index, Lemma 11 implies that we can determine both the the index and the sign of the permutation element 2. Note that for determining the index of element 2 we need to know the index of the preceding element, its value (in this case 1), as well as the repeat sequence $s'|_R = s|_R$. By induction, one can determine the index of the permutation element k in s' , based on the index of the preceding permutation element $k - 1$ and the repeat sequence $s|_R$. Thus, Lemma 11 implies that the repmap s' can be deduced from the repeat sequence of s . Moreover, calculating the index of the successive permutation element as demonstrated in Lemma 11 can be done in constant time. Therefore, reconstructing s' can be done in linear-time.

Lemma 12. *Given a repmap $s = S \cdot \varrho$, where both the repmap S and the legal scenario ϱ are unknown, let S' be the sorted normalization of S and $s' = S' \cdot \varrho$. The normalized repmap s' can be calculated in linear-time and linear-space based on the repeat sequence $s|_R$.*

After calculating the repmap s' , determining S is a matter of renaming the repmap S' according to the correspondency established between the permutation elements of s and s' . Thus, the calculation follows the path $s \rightarrow s' \rightarrow S' \rightarrow S$. Figures 3a and 3b give an example illustrating this process and Algorithm 1 below implements this idea.

Algorithm 1: getAncestor

Data: A repmap s .
Result: The ancestor S of s .

```

1 begin
2    $i \leftarrow 1$  ;  $s'[1] \leftarrow 1$  ;  $r \leftarrow s|_R$  ;
3    $k \leftarrow |r|$  ;  $s'[2k + 1] \leftarrow k + 1$  ;
4   while  $i \neq 2k + 1$  do
5      $j \leftarrow \text{getNextIndex}(i, s')$  ;
6      $val \leftarrow \text{getNextVal}(i, j, s')$  ;
7      $s'[j] \leftarrow val$  ;
8      $i \leftarrow j$  ;
9   return getS( $s, s'$ )
10 end
```

Theorem 13 (Time and Space Complexity). *Given a repmap s , Algorithm `getAncestor(s)` reconstructs the ancestor S in linear-time, and in linear-space ($O(|s|)$).*

3.3.2 Reconstructing a Legal Scenario

Unlike the ancestor repmap reconstruction, the scenario reconstruction involves a look ahead to guarantee avoiding conflicts between repeat pairs. This conflict is best demonstrated by an example.

Example. Consider the following repmap:

$$1 \quad a \quad -b \quad -2 \quad -a \quad -c \quad -4 \quad b \quad 3 \quad c \quad .$$

Suppose we were to choose first to fulfill the inverted repeat pair b and $-b$. Such a choice would turn the other two repeat pairs (a and c) into direct-repeat pairs. Thus, we reach a deadlock without getting a legal scenario.

As demonstrated in the above example, choosing a legal reversal sequence which avoids deadlocks is a delicate matter. We address this problem by utilizing the fact that we can calculate the normalized repmaps s' and S' in linear-time (Section 3.3.1). When both repmaps are known, we show that an SBR reversal sequence sorting $s'|_N$ (to $S'|_N$) corresponds to a legal scenario transforming s to S . Currently, the best algorithm for solving SBR works in sub-quadratic time [44]. Hence, we get a sub-quadratic algorithm for reconstructing a legal scenario.

Consider a normalized repmap p resulting from a legal scenario affecting a sorted and normalized repmap P . Lemma 5 states that a legal reversal affecting p reduces the breakpoint count by 2. The following lemma claims the opposite, and is proved similarly.

Lemma 14. *Given a sorted normalized repmap P , let ϱ be a legal scenario, and denote $p = P \cdot \varrho$. Let ρ denote a reversal which eliminates two breakpoints, such that ρ affects a subsequence of p that has permutation elements on both its edges; ρ is a legal reversal.*

Proof. Since p is normalized and ρ affects a subsequence of it having permutation elements on both its edges, ρ must be bordered by repeats. By the proof of Lemma 5, these repeats must correspond to an inverted repeat pair. \square

By induction on Lemma 14 we get the following corollary.

Corollary 15. *Let P be a given sorted normalized repmap, let $\varrho = \rho_1, \dots, \rho_m$ be a legal scenario, and denote $p = P \cdot \varrho$. Let ϱ' be a scenario (not necessarily legal) in which each reversal eliminates 2 breakpoints. Then, ϱ' is a partially legal scenario.*

If the scenario eliminates all the breakpoints in p (i.e., it sorts p) then it obviously must fulfill all the repeat pairs and is hence legal.

Corollary 16. *Let P be a given sorted normalized repmap, let $\varrho = \rho_1, \dots, \rho_m$ be a legal scenario, and denote $p = P \cdot \varrho$. Let ϱ' be a scenario (not necessarily legal) in which each reversal eliminates 2 breakpoints such that $P = p \cdot \varrho'$. Then, ϱ' is a legal scenario.*

Since $p = P \cdot \varrho$, Corollary 6 implies that $p|_N$ has an SBR scenario in which each reversal eliminates 2 breakpoints, and hence all SBR scenarios fulfill this property. Thus, by Lemma 14, all SBR scenarios are legal.

Theorem 17. *Let P be a sorted and normalized repmap, ϱ a legal scenario, and denote $p = P \cdot \varrho$. A solution of SBR on $p|_N$ corresponds to a legal scenario on p .*

Theorem 17 and Lemma 12 enable us to find a legal scenario transforming s to S as follows: calculate s' from s (Algorithm `getAncestor`) and use SBR to find an optimal scenario sorting $s'|_N$. By Theorem 17 this scenario is legal on s' . Since s and s' have the same repeat sequence, the scenario is legal on s as

well. Since computing s' from s can be done in linear-time, the complexity of finding a legal scenario for s is determined by the SBR bottleneck. Currently, the most efficient algorithm for solving SBR has a time complexity of $O(n\sqrt{n \log n})$, where $n = |s|$ [44].

Theorem 18 (Time Complexity). *Given a repmap $s = S \cdot \varrho$, where both the repmap S and the legal scenario ϱ are unknown, one can reconstruct a legal scenario transforming s to S in $O(n\sqrt{n \log n})$ time, where $n = |s|$.*

4 The Multiple Leaf RAPT Case and Set-tries

In this section we show that the leaf assignments $L = \{s^1, \dots, s^q\}$ uniquely determine the underlying RAPT (T, f, g) up to (and not including) repeats in the inner nodes, *i.e.*, they dictate the tree topology, the induced permutations in inner node assignments, and the edge labels. We then describe a linear-time algorithm for reconstructing this information from the given input.

The uniqueness results are developed in two stages: first, the RAPT is reduced to a new auxiliary data structure denoted a *set-trie* (see Section 4.1 and Figure 4), which encodes partial information (tree topology and edge labels). Using this reduction, we show that both the tree topology and the edge labels are uniquely determined (Section 4.2) and can be reconstructed in linear-time based on the repeat sets $\{R(s) : s \in L\}$ of the leaf assignments (Section 4.3). Finally, the application of Theorems 10 and 13 to the above findings leads to the conclusion that the induced permutations in the inner node assignments are uniquely determined and can be reconstructed in linear-time based on the tree topology, the edge labels, and the leaf assignments (Section 4.4).

4.1 Set-tries and Monotonic Collections

Word-tries are well-known data structures, commonly used in text compression and database search. They are used to store the information about the contents of each node in the path from the root to the node rather than in the node itself, thus grouping words with a common prefix along similar paths [17]. Here we introduce a new data structure which, similarly to word-tries, is also based on a tree topology and path-encoding, however, the leaves of the new data structure correspond to sets instead of words (or sequences), as defined below.

Definition 6 (Set-tries). Let $\mathcal{A} = \{A_1, \dots, A_\ell\}$ be a collection of finite subsets of \mathbb{N} . A *set-trie* st over \mathcal{A} is a pair $st = (T, g)$, where $T = (V, E)$ is a directed tree with a root v_r such that all the inner nodes (except perhaps the root) are of degree ≥ 3 and $g : E \rightarrow 2^{\mathbb{N}}$ are labels to the edges. In the following discussions we assume that assignments to the nodes $f : V \rightarrow 2^{\mathbb{N}}$ are also given. The labels g and the “virtual” assignments f need to fulfill the following requirements:

1. $f(v_r) = \emptyset$ and f is 1 : 1 from the leafs of T to \mathcal{A} . Given that $u \in V$ is an ancestor of $v \in V$, we require that $f(v) = f(u) \cup g(\text{path}(u, v))$. In particular, this requirement implies $\forall v \in V - \{v_r\} : f(v) = g(\text{path}(v_r, v))$.
2. $\forall e, e' \in E, e \neq e' : g(e) \cap g(e') = \emptyset$. Thus, the node assignments are determined by the edge labels and vice versa.

Figure 4 gives an example of a set-trie and its derivation from a RAPT. We observe the following *monotonicity* property of set collections corresponding to leafs of set-tries,

Definition 7 (Monotonic Set Collection). A collection \mathcal{A} is *monotonic* if, for any three sets $A, B, C \in \mathcal{A}$, either $A \cap B \subseteq A \cap C$ or $A \cap C \subseteq A \cap B$.

4.2 Uniqueness of Set-tries Based on Monotonic Collections

In this section we show that a set-trie is uniquely determined by the monotonic collection assigned to its leafs. However, before addressing the uniqueness issue, we first prove that the leaf assignments in a set-trie indeed correspond to a monotonic collection. Furthermore, we assert that, for any given monotonic collection L , there exists a corresponding set-trie over L . The existence proof is constructive, and serves as the basis for the set-trie reconstruction algorithm presented in Section 4.3.

Lemma 19. *Let \mathcal{A} be a finite collection of finite subsets of \mathbb{N} . There exists a set-trie over \mathcal{A} iff \mathcal{A} is monotonic.*

Proof. Assume first that \mathcal{A} has a set-trie st . Consider three sets $A, B, C \in \mathcal{A}$ and let v_A, v_B, v_C be leafs in T such that $f(v_A) = A, f(v_B) = B, f(v_C) = C$. Assume without loss of generality that the inner node $lca(v_A, v_C)$ (namely the least common ancestor) is lower (*i.e.*, further away from the root) than the inner node $lca(v_A, v_B)$. By Definition 6.1 it is easy to see that $A \cap C = f(lca(v_A, v_C))$ and $A \cap B = f(lca(v_A, v_B))$. Moreover, since $lca(v_A, v_C)$ is lower, we have $f(lca(v_A, v_B)) \subseteq f(lca(v_A, v_C))$, and thus $A \cap B \subseteq A \cap C$ and \mathcal{A} is monotonic.

Assume now that \mathcal{A} is monotonic. We need to prove that there exists a set-trie over \mathcal{A} . We show this by induction on ℓ , the number of sets in \mathcal{A} . Base case: for $\ell = 1$ the claim is trivial. Induction step: assume the claim holds for $\ell \geq 1$, we need to prove it for $\ell + 1$. Consider the first ℓ sets in \mathcal{A} and denote them by $\mathcal{A}' = \{A_1, \dots, A_\ell\}$. By the induction assumption, there exists a set-trie st' over \mathcal{A}' . Note that since \mathcal{A} is monotonic, one can order the sets $A_{\ell+1} \cap A_1, \dots, A_{\ell+1} \cap A_\ell$ in an increasing order (with respect to the containment relation). Let $A_{\ell+1} \cap A_i$ be the maximum set in this collection and let $v \in V$ be the lowest (*i.e.*, furthest away from the root) vertex on the path from v_r to $f^{-1}(A_i)$ such that $f(v) \subseteq A_{\ell+1}$. There exists such a vertex, since v_r fulfills the requirement ($f(v_r) = \emptyset \subseteq A_{\ell+1}$). Note that, by definition, $f(v) \subseteq A_i$, and thus $f(v) \subseteq A_{\ell+1} \cap A_i$. Next, we define the vertex v' , which is the parent of v in the emerging set-trie (see Figure 6).

To do that, we distinguish between the following plausible cases: v' exists already in st' , in which case it can be either a leaf or an inner node, or v' does not exist in st' , and therefore needs to be constructed.

1. If $f(v) = A_{\ell+1} \cap A_i$, let $v' = v$ (v' exists in st'). If v' is a leaf, define a new vertex u and connect it as a son to v' , label the connecting edge with the empty set, and define $f(u) = A_i$.
2. If $f(v) \neq A_{\ell+1} \cap A_i$ (v' does not exist in st'), let $e_{tail} \in E'$ be the edge going out of v to the vertex u such that $(A_{\ell+1} \cap A_i - f(v)) \cap f(u) \neq \emptyset$. By the monotonicity assumption, there is only one such node u . Define a new vertex v' , connect it with an edge e' from v and with an edge e'' to u , and define $f(v') = A_{\ell+1} \cap A_i, g(e') = f(v') - f(v)$, and $g(e'') = f(u) - f(v')$.

In all cases, create a new vertex w and connect it from v' with an edge e''' such that $f(w) = A_{\ell+1}$, and $g(e''') = f(w) - f(v')$. It is easy to see that the above modification is a set-trie over \mathcal{A} . \square

We are now ready to prove the *uniqueness* of set tries over monotonic collections.

Lemma 20. *Let \mathcal{A} be a monotonic collection. There exists a unique set-trie over \mathcal{A} .*

Proof. By Lemma 19, there exists a set-trie over \mathcal{A} . Here we prove that this trie is unique by induction on ℓ , the number of sets in the collection \mathcal{A} . Base case: for $\ell = 1$, the claim is trivial. Induction step: assume the claim holds for $\ell \geq 1$, and consider a collection \mathcal{A} over $\ell + 1$ sets. By contradiction, let st and st' be two different set-tries over \mathcal{A} . Consider the collection of the first ℓ sets in \mathcal{A} and denote it by \mathcal{A}' . Trivially, the restrictions $st|_{\mathcal{A}'}$ and $st'|_{\mathcal{A}'}$ are set-tries over \mathcal{A}' . By the induction assumption $st|_{\mathcal{A}'} = st'|_{\mathcal{A}'}$. In particular, if the tree topologies of st and st' differ, then $\text{path}_{st}(v_r, f^{-1}(A_{\ell+1})) \neq \text{path}_{st'}(v_r, f^{-1}(A_{\ell+1}))$ (the paths in st and st' respectively). Let d and d' be the direct parents of $f^{-1}(A_{\ell+1})$ in T and T' respectively. By Definition 6 and since the paths are not equal, $g(\text{path}_{st}(v_r, d))$ and $g'(\text{path}_{st'}(v_r, d'))$ are not equal. Without loss of generality, assume that $\exists a \in g(\text{path}_{st}(v_r, d))$ and $a \notin g'(\text{path}_{st'}(v_r, d'))$. By Definition 6,

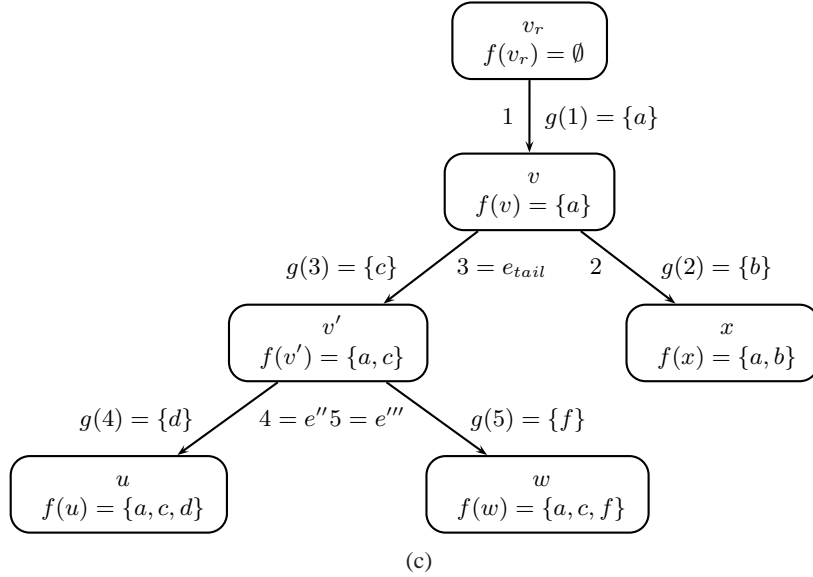
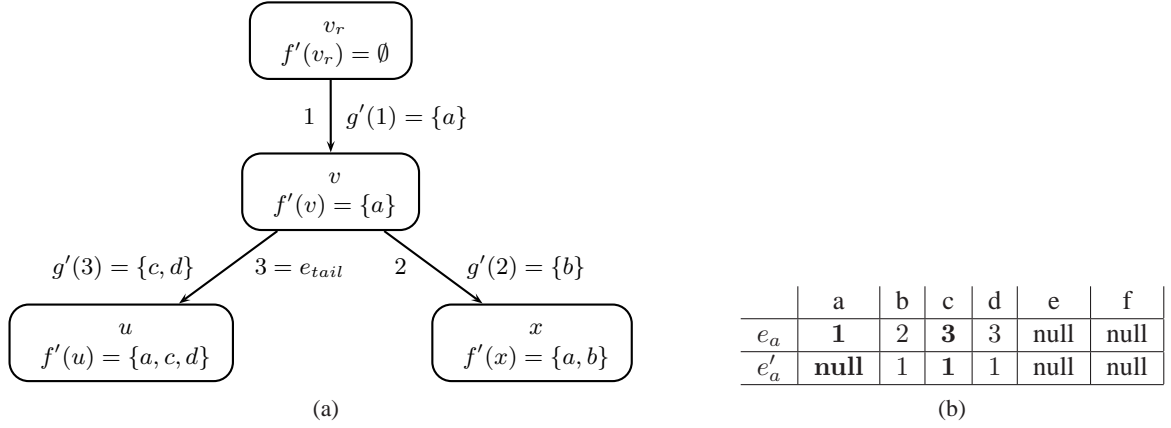


Figure 6: An example of an insertion of a new set $A_k = \{a, c, f\}$ to an existing set-trie $st' = (T', g')$ with virtual node assignments f' over the collection $\mathcal{A} = \{\{a, b\}, \{a, c, d\}\}$. The set-trie before the insertion is given in (a). The correspondences $a \rightarrow e_a$ and $e_a \rightarrow e'_a$ are demonstrated by arrays (representing `aToEdge` and `prevEdge` in Algorithms 2 and 3) in (b). This figure also illustrates the `Find` operation over A_k (see below). Illustrations of the `Split` operation and its result are given in (c). In this example $A_i = \{a, c, d\}$, $A_k \cap A_i = \{a, c\}$, $A_k \cap A_i - f'(v) = \{c\}$. Hence, in the e'_a array edge 3 is not marked while edge 1 is. From this we conclude that $e_{tail} = 3$. This figure continues the example of Figure 4.

$a \in g(\text{path}_{st}(v_r, d)) \subseteq g(\text{path}_{st}(v_r, f^{-1}(A_{\ell+1})))$ and hence $a \in A_{\ell+1}$. Let $A_i \neq A_{\ell+1}$ be a descendant of d (by Definition 6 all internal nodes are of degree ≥ 3 . If d is the root and has only $A_{\ell+1}$ as its child, then the tree is trivial). By similar reasoning, we have that $a \in A_i$. However, since $a \notin g'(\text{path}_{st'}(v_r', d'))$, we must have that $a \in g'(\text{path}_{st'}(d', f^{-1}(A_{\ell+1})))$. Thus a is not in $g'(\text{path}_{st'}(v_r', f^{-1}(A_i)))$. A contradiction.

Note that the assignments to the inner nodes are uniquely determined by the topology, since we have $f(\text{lca}(A_i, A_j)) = A_i \cap A_j$. Furthermore, the edge labels are uniquely determined by the inner node assignments, since for an edge $e \in E$ going from u to v , $u \xrightarrow{e} v$, we have $g(e) = f(u) - f(v)$. \square

4.3 A Linear-time Algorithm for Set-trie Construction

Adding an element to a standard word-trie utilizes the fact that the labels on the path from the root to the leaf representing the word preserve the original order of the characters in the word. Such is not the case when dealing with sets. It is easy to see that adapting the current strategies used in constructing word-tries [17] to the task of set-trie reconstruction results in algorithms with quadratic-time complexity. In this section we present a more efficient method with linear-time complexity ($\Theta(|\mathcal{A}|)$) where $|\mathcal{A}| = \sum_{j=1}^{\ell} |A_j|$. We describe an algorithm to construct the set-trie via incremental leaf insertions, based on the induction described in the proof of Lemma 19 and we use the same notation. At step k of the algorithm, the set-trie st' is updated with the leaf node corresponding to A_k . As elaborated in the proof of Lemma 19 and as demonstrated in Figure 6, the leaf insertion requires two consecutive operations:

1. **Find**: Identify the node $v \in V$ which is the lowest vertex such that $f(v) \subseteq A_k$, and the edge $e_{tail} \in E'$ which is the edge connecting v to the vertex u such that $(A_{\ell+1} \cap A_i - f(v)) \cap f(u) \neq \emptyset$. In addition, compute the sets $B = g(e_{tail}) \cap A_k$ (elements common to the label of e_{tail} and the new set A_k) as well as $C = A_k - \bigcup_{j=1}^{k-1} A_j$ (new elements to be added to the current set-trie T').
2. **Split**: Use the result of the Find operation to update the set-trie $st' = (T', g')$ with the new leaf.

The above Find and Split operations need to be distinguished from the classical disjoint set operations surveyed in [16]. Here, tricks such as path compression cannot be applied since the set-trie is defined by its topology. Further, note that strategies such as employing numbering on the edges to reflect their level in the tree in order to support efficient Find queries are likely to yield inefficient solutions, since the numbering could change dynamically via Split operations. Therefore, the crux of our solution is in utilizing monotonicity to efficiently (in $O(|A_k|)$) maintain information regarding the local neighborhood of each edge, and in using this information to enable efficient implementation both operations in time-complexity which is linear in the size of each set $O(|A_k|)$. Summing up over all leaves in \mathcal{A} then yields a linear-time complexity, $\Theta(|\mathcal{A}|)$. The pseudo-codes for the linear time Find (Algorithm 2) and Split (Algorithm 3) are given below.

Lemma 21. *Given a monotonic set collection $\mathcal{A} = \{A_1, \dots, A_\ell\}$ and a set-trie st' over $\mathcal{A}' = \{A_1, \dots, A_{k-1}\}$, where $2 \leq k \leq \ell$, Algorithm 2 performs Find (A_k) in time complexity linear in the set size $O(|A_k|)$.*

Proof. Explicitly constructing the edge label function $g(\cdot)$ and using it to find v and e_{tail} is likely to yield an inefficient solution because of the need to perform set intersection and subtraction operations. Instead, we construct a quasi-inverse function by mapping each element in $\bigcup_{A \in \mathcal{A}} A$ to the edge whose label contains it (if any). This way, the node v can be efficiently found via two passes over the elements of A_k , as follows. Similarly to the proof of Lemma 19, consider $e_{tail} \in E'$, the lowest (i.e., furthest away from the root) edge whose label intersects with A_k (if none exists then $e_{tail} = \text{null}$ and we consider it as pointing to the root v_r). Clearly, e_{tail} leads into v if $f(v) = A_k \cap A_i$, and otherwise e_{tail} is the edge connecting v with u (see the proof of Lemma 19 for a reminder of the notation). For $a \in A_k$, let $e_a \in E'$ denote the edge with $a \in g'(e_a)$ (i.e., the quasi-inverse function) and let $e'_a \in E'$ denote the edge immediately preceding e_a in T' — see Figure 6 for an example ($e_a = e'_a = \text{null}$ if a does not appear in T'). In the first pass, the edges in $\{e'_a : a \in A_k\}$ are marked (i.e., for each $a \in A_k$ first the edge e_a is queried, and the edge e'_a which

Algorithm 2: Find

Data: A set A_k .**Result:** The lowest level edge e_{tail} whose label intersects with A_k , the non-empty set $B = g(e_{tail}) \cap A_k$, and the set $C = A_k - \bigcup_{j=1}^{k-1} A_j$.

```
1 begin
2    $B \leftarrow \emptyset$  ;
3    $C \leftarrow \emptyset$  ;
4    $e_{tail} \leftarrow 0$ ;
5   for  $a \in A_k$  do
6      $e \leftarrow \text{aToEdge}[a]$  ;
7     mark prevEdge  $[e]$  ;
8   for  $a \in A_k$  do
9     if  $\text{aToEdge}[a] = 0$  then  $C \leftarrow C \cup \{a\}$  ;
10    else if  $\text{aToEdge}[a]$  is unmarked then
11       $B \leftarrow B \cup \{a\}$ ;
12       $e_{tail} \leftarrow \text{aToEdge}[a]$  ;
13  return  $e_{tail}, B, C$ 
14 end
```

precedes e_a is marked). Let $b \in g(e_{tail}) \cap A_k$ (by definition $g(e_{tail}) \cap A_k \neq \emptyset$ if $e_{tail} \neq null$). Note that $e_b = e_{tail}$. Because of the extremity of e_{tail} , we know that it is the only edge among $\{e_a : a \in A_k\}$ that was not marked in the first scan. Therefore, finding $B = \{b \in A_k : e_b \neq null \text{ is unmarked}\} = g(e_{tail}) \cap A_k$, and hence e_{tail} , as well as $C = \{c \in A_k : e_c = null\} = A_k - \bigcup_{j=1}^{k-1} A_j$ can be computed via one additional pass over the elements of A_k (e_{tail} , B , and C are required for the `Split` operation). Clearly, both e_a and e'_a can be queried in constant time, e.g., by maintaining the correspondences $a \rightarrow e_a$ and $e_a \rightarrow e'_a$ in two arrays, namely `aToEdge` and `prevEdge` in the pseudo-code, respectively. The two passes are computed in $O(|A_k|)$, and hence the `Find` has a linear-time complexity. \square

Algorithm 3: Split

Data: The edge e_{tail} , and the two sets B and C .**Result:** Splitting the edge e_{tail} and adding a new edge e' according to the set B , redefining e_{tail} , adding a new edge e''' according to C , and updating `aToEdge`, `prevEdge`, `edgeElementsCount`, and `count`.

```
1 begin
2   if edgeElementsCount  $[e_{tail}] \neq |B|$  then
3     count  $\leftarrow$  count + 1;
4      $e' \leftarrow$  count ;
5     edgeElementsCount  $[e'] \leftarrow |B|$ ;
6     edgeElementsCount  $[e_{tail}] \leftarrow$  edgeElementsCount  $[e_{tail}] - |B|$ ;
7     prevEdge  $[e'] \leftarrow$  prevEdge  $[e_{tail}]$  ;
8     prevEdge  $[e_{tail}] \leftarrow e'$  ;
9     for  $b \in B$  do aToEdge  $[b] \leftarrow e'$ ;
10  else  $e' \leftarrow e_{tail}$ ;
11  count  $\leftarrow$  count + 1 ;
12   $e''' \leftarrow$  count ;
13  for  $c \in C$  do aToEdge  $[c] \leftarrow e'''$  ;
14  prevEdge  $[e'''] \leftarrow e'$  ;
15  edgeElementsCount  $[e'''] \leftarrow |C|$ ;
16 end
```

Lemma 22. Given a monotonic set collection $\mathcal{A} = \{A_1, \dots, A_\ell\}$, a set-trie st' over $\mathcal{A}' = \{A_1, \dots, A_{k-1}\}$, where $2 \leq k \leq \ell$, and e_{tail} , B , and C , the result of the `Find` (A_k) operation (Algorithm 2), Algorithm 3 performs `Split` (A_k) in time complexity linear in set size $O(|A_k|)$.

Proof. Let's first analyze the implementation of the `Split` operation for the more challenging case of adding v' as a new node, and only then move on to analyze the implementation in the other case (see the proof of Lemma 19). Let A_k be the set to be inserted in the set-trie $st' = (T', g')$. Apply the `Find` operation to it, and consider its result, namely the edge to be split e_{tail} , the set B of elements appearing in both A_k as well as in the label of e_{tail} (i.e., $B = A_k \cap g(e_{tail})$), and the set C of elements appearing in A_k but not in the set-trie st' (i.e., $C = A_k - \cup_{j=1}^{k-1} A_j$). The `Split` operation needs to cut the edge e_{tail} according to the set B , so that instead of a single edge

$$v \xrightarrow[e_{tail}]{g(e_{tail})} u$$

we get two new edges

$$v \xrightarrow[B]{e'} v' \xrightarrow[g(e_{tail})-B]{e''} u .$$

Furthermore, we need to add a leaf w corresponding to the set A_k and an edge connecting it to v' with C as the edge's label: $v' \xrightarrow[C]{e'''} w$ (see Figure 6). Let `aToEdge` and `prevEdge` be arrays as introduced in the `Find` implementation, let `count` denote a global variable keeping count of the total number of edges in the set-trie at any given moment, and let `edgeElementsCount` denote an additional array mapping each edge to the number of elements in its label $e \rightarrow |g(e)|$. These arrays and counters are used to implement `Split` in $O(A_k)$ as follows: Create a new vertex v' and a new edge e' pointing from v to v' (using the global counter `count`). Let all the elements in B point to e' , $v \xrightarrow[B]{e'} v'$ (by changing their values in `aToEdge`). This implies that only elements in $g(e_{tail}) - A_k$ now point to e_{tail} . Instead of adding a new edge e'' connecting v' to u (which would require $O(g(e_{tail}) - A_k)$ time), redefine e_{tail} to connect v' to u : $v' \xrightarrow[g(e_{tail})-A_k]{e_{tail}} u$. Update both the number of elements contained in the labels of each of e' and e_{tail} (in the `edgeElementsCount` array) as well as their preceding edges (in the `prevEdge` array). This implicitly establishes the new path

$$v \xrightarrow[B]{e'} v' \xrightarrow[g(e_{tail})-B]{e''} u .$$

Next, construct the edge $v' \xrightarrow[C]{e'''} w$ by adding a new vertex w and a new edge e''' , update the edge's number of elements, its preceding edge, and let the elements in C point to it.

If v' exists in the set-trie st' , then splitting e_{tail} is not required. One needs only to add w (and perhaps one additional vertex if v' is a leaf). \square

By applying the above `Find` and the `Split` operations via incremental additions of sets from \mathcal{A} to the set-trie, it is possible to construct the tree topology T and to implicitly construct the edge labels g in linear-time ($O(|\mathcal{A}|)$). Explicitly constructing g from `aToEdge` can be done in linear-time by a straightforward single pass over the array `aToEdge`.

Theorem 23. Given a monotonic collection \mathcal{A} , a set-trie over \mathcal{A} can be constructed in linear-time ($\Theta(|\mathcal{A}|)$).

4.4 From Set-tries Back to RAPT

Based on Definition 3 it is easy to see that the collection of repeat sets $\{R(s) : s \in L\}$ is monotonic. In particular, a RAPT (T, f, g) with leaf assignments $L = s^1, \dots, s^q$ can be readily mapped to a set-trie (T', g') with node assignments f' as follows: define $T' = T$, $g' = g$, and $\forall s \in L : f'(s) = R(s)$, i.e., the leaf

assignments of the set-trie are the repeat sets of the leaf assignments of the RAPT. Figure 4 gives an example illustrating this mapping. Since, by Lemma 20, a set-trie is uniquely determined by its leaf assignments, we imply that the repeat sets $\{R(s) : s \in L\}$ uniquely determine the tree topology T and the edge labels g of the RAPT. Thus, to prove the uniqueness of the RAPT, it is sufficient to show that the induced permutations in the inner node assignments are uniquely determined by T, g and L . However, this result is immediately established by Theorem 10.

Theorem 24. *The leaf assignments L uniquely determine the underlying RAPT up to (and not including) repeats in the inner nodes.*

Based on the linear-time algorithm for reconstructing set-tries (Theorem 23) and the linear-time algorithm for reconstructing ancestral assignments (Theorem 13), we get a linear-time algorithm for reconstructing a RAPT from its leaf assignments.

Theorem 25. *Given the leaf assignments L of an unknown RAPT, reconstructing the RAPT (up to repeats in the inner nodes) can be done in linear-time ($\Theta(|L|)$).*

References

- [1] G. Achaz, F. Boyer, E. P. C. Rocha, A. Viari, and E. Coissac. Extracting approximate repeats from large dna sequences. 2004.
- [2] G. Achaz, E. Coissac, P. Netter, and E. P. C. Rocha. Associations Between Inverted Repeats and the Structural Evolution of Bacterial Genomes. *Genetics*, 164(4):1279–1289, 2003.
- [3] D. A. Bader, B. M. E. Moret, and M. Yan. A linear-time algorithm for computing inversion distances between signed permutations with an experimental study. *Journal of Computational Biology*, 8:483–491, 1982.
- [4] V. Bafna and P. Pevzner. Sorting permutations by transpositions. In *Proceedings of the 6th Annual Symposium on Discrete Algorithms (SODA)*, pages 614–623, 1995.
- [5] V. Bafna and P. Pevzner. Genome rearrangements and sorting by reversals. *SIAM J. Computing*, 25:272–289, 1996.
- [6] M. Bender, D. Ge, S. He, H. Hu, R. Pinter, S. Skiena, and F. Swidan. Improved bounds on sorting with length-weighted reversals. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 912–921, 2004.
- [7] A. Bergeron. A very elementary presentation of the hannenhalli-pevzner theory. In *Proc. 12th Symp. Combinatorial Pattern Matching (CPM)*, volume 2089, pages 106–117. Springer-Verlag Lecture Notes in Computer Science, 2001.
- [8] P. Berman and S. Hannenhali. Fast sorting by reversals. In *Proc. 7th Symp. on Combinatorial Pattern Matching*, pages 168–185, 1996.
- [9] G. Bourque and P. A. Pevzner. Genome-scale evolution: Reconstructing gene orders in the ancestral species. *Genome Res.*, 12(1):26–36, 2002.
- [10] G. Bourque, E. M. Zdobnov, P. Bork, P. A. Pevzner, and G. Tesler. Comparative architectures of mammalian and chicken genomes reveal highly variable rates of genomic rearrangements across different lineages. *Genome Res.*, 15(1):98–110, 2005.
- [11] L. Brocchieri. Phylogenetic inferences from molecular sequences: Review and critique. *Theor. Popul. Biol.*, 59.
- [12] A. Caprara. Formulations and hardness of multiple sorting by reversals. In *Proc 3th Ann. Int. Conf. on Computational Molecular Biology (RECOMB)*, pages 84–93, New York, NY, USA, 1999. ACM Press.
- [13] T. Chen and S. Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71:79–95, 1996.
- [14] D. A. Christie and R. W. Irving. Sorting strings by reversals and by transpositions. *SIAM Journal on Discrete Mathematics*, 14:193 – 206, 2001.
- [15] A. C. R. da Silva, J. A. Ferro, F. C. Reinach, C. S. Farah, L. R. Furlan, R. B. Quaggio, C. B. Monteiro-Vitorello, M. A. V. Sluys, N. F. Almeida, L. M. C. Alves, A. M. do Amaral, M. C. Bertolini, L. E. A. Camargo, G. Camarotte, F. Cannavan, J. Cardozo, F. Chamberg, L. P. Ciapina, R. M. B. Cicarelli, L. L. Coutinho, J. R. Cursino-Santos, H. El-Dorry, J. B. Faria, A. J. S. Ferreira, R. C. C. Ferreira, M. I. T. Ferro, E. F. Formighieri, M. C. Franco, C. C. Greggio, A. Gruber, A. M. Katsuyama, L. T. Kishi, R. P. Leite, E. G. M. Lemos, M. V. F. Lemos, E. C. Locali, M. A. Machado, A. M. B. N. Madeira, N. M. Martinez-Rossi, E. C. Martins, J. Meidanis, C. F. M. Menck, C. Y. Miyaki, D. H. Moon, L. M. Moreira, M. T. M. Novo, V. K. Okura, M. C. Oliveira, V. R. Oliveira, H. A. Pereira, A. Rossi, J. A. D. Sena, C. Silva, R. F. de Souza, L. A. F. Spinola, M. A. Takita, R. E. Tamura, E. C. Teixeira, R. I. D. Tezza, M. Trindade dos Santos, D. Truffi, S. M. Tsai, F. F. White, J. C. Setubal, and J. P. Kitajima. Comparison of the genomes of two *Xanthomonas* pathogens with differing host specificities. *Nature*, 417:459–463, May 2002. 10.1038/417459a.

- [16] Z. Galil and G. F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23:319–344, 1991.
- [17] G. Gonnet. *Handbook of Algorithms and Data Structures*. International Computer Science Services, 1983.
- [18] Q.-P. Gu, S. Peng, and I. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theor. Comp. Sci.*, 210, 1999.
- [19] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46:1–27, 1999.
- [20] T. Hartman. A simpler 1.5-approximation algorithm for sorting by transpositions. In *Proc. 14th Ann. Symp. on Combinatorial Pattern Matching (CPM)*, pages 156–169, 2003.
- [21] T. Hartman and R. Sharan. A 1.5-approximation algorithm for sorting by transpositions and transreversals. In *Proc. 4th Workshop on Algorithms in Bioinformatics (WABI)*, pages 50–61, 2004.
- [22] H. Kaplan, R. Shamir, and R. E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. In *Proc. 8th Ann. Symp. on Discrete Algorithms (SODA)*, pages 344–351, 1997.
- [23] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for the inversion distance between two permutations. In *Proc. of 4th Ann. Symp. on Combinatorial Pattern Matching*, Lecture Notes in Computer Science 684, pages 87–105. Springer Verlag, 1993.
- [24] J. Kececioglu and D. Sankoff. Efficient bounds for oriented chromosome inversion distance. In *Proc. of 5th Ann. Symp. on Combinatorial Pattern Matching*, pages 307–325. Springer-Verlag LNCS 807, 1994.
- [25] S. C. Kowalczykowski, D. A. Dixon, A. K. Eggleston, S. D. Lauder, and W. M. Rehrauer. Biochemistry of homologous recombination in *Escherichia coli*. *Microbiol. Rev.*, 58:401–65, 1994.
- [26] S. L. Lusetti and M. M. Cox. The bacterial recA protein and the recombinational dna repair of stalled replication forks. *Annual Review of Biochemistry*, 71(1):71–100, 2002.
- [27] J. Mahillon and M. Chandler. Insertion Sequences. *Microbiol. Mol. Biol. Rev.*, 62(3):725–774, 1998.
- [28] W. Martin, T. Rujan, E. Richly, A. Hansen, S. Cornelsen, T. Lins, D. Leister, B. Stoebe, M. Hasegawa, and D. Penny. Evolutionary analysis of arabidopsis, cyanobacterial, and chloroplast genomes reveals plastid phylogeny and thousands of cyanobacterial genes in the nucleus. *Proc. Natl. Acad. Sci. USA.*, 99:12246–12251, 2002.
- [29] B. G. Milligan, J. N. Hampton, and J. D. Palmer. Dispersed repeats and structural reorganization in subclover chloroplast DNA. *Molecular Biology Evolution*, 6:355–368, 1989.
- [30] B. Moret, L. Wang, T. Warnow, and S. Wyman. New approaches for reconstructing phylogenies from gene order data. In *Proc. 9th Int. Conf. Intell. Syst. Mol. Biol. (ISMB)*, 2001.
- [31] J. Parkhill, M. Sebaihia, A. Preston, L. Murphy, N. Thomson, D. Harris, M. Holden, C. Churcher, S. Bentley, K. Mungall, A. Cerdeno-Tarraga, L. Temple, K. James, B. Harris, M. Quail, M. Achtman, R. Atkin, S. Baker, D. Basham, N. Bason, I. Cherevach, T. Chillingworth, M. Collins, A. Cronin, P. Davis, J. Doggett, T. Feltwell, A. Goble, N. Hamlin, H. Hauser, S. Holroyd, K. Jagels, S. Leather, S. Moule, H. Norberczak, S. O’Neil, D. Ormond, C. Price, E. Rabinowitsch, S. Rutter, M. Sanders, D. Saunders, K. Seeger, S. Sharp, M. Simmonds, J. Skelton, R. Squares, S. Squares, K. Stevens, L. Unwin, S. Whitehead, B. Barrell, and D. Maskell. Comparative analysis of the genome sequences of *Bordetella pertussis*, *Bordetella parapertussis* and *Bordetella bronchiseptica*. *Nat. Genet.*, 35:32–40, 2003.
- [32] P. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press, Cambridge, MA, 2000.
- [33] P. A. Pevzner and G. Tesler. Genome rearrangements in mammalian evolution: lessons from human and mouse genomes. *Genome Research*, 13:37–45, 2003.
- [34] W. Qian, Y. Jia, S.-X. Ren, Y.-Q. He, J.-X. Feng, L.-F. Lu, Q. Sun, G. Ying, D.-J. Tang, H. Tang, W. Wu, P. Hao, L. Wang, B.-L. Jiang, S. Zeng, W.-Y. Gu, G. Lu, L. Rong, Y. Tian, Z. Yao, G. Fu, B. Chen, R. Fang, B. Qiang, Z. Chen, G.-P. Zhao, J.-L. Tang, and C. He. Comparative and functional genomic analyses of the pathogenicity of phytopathogen *Xanthomonas campestris* pv. *campestris*. *Genome Res.*, 15(6):757–767, 2005.
- [35] E. P. Rocha. An Appraisal of the Potential for Illegitimate Recombination in Bacterial Genomes and Its Consequences: From Duplications to Genome Reduction. *Genome Res.*, 13(6a):1123–1132, 2003.
- [36] E. P. C. Rocha. DNA repeats lead to the accelerated loss of gene order in bacteria. *TRENDS in Genetics*, 19(11):600–603, 2003.
- [37] E. P. C. Rocha. Order and disorder in bacterial genomes. *Current Opinion in Microbiology*, 7:519–537, 2004.
- [38] E. P. C. Rocha, A. Danchin, and A. Viari. Functional and evolutionary roles of long repeats in prokaryotes. *Res. Microbiol.*, 150:725–733, 1999.

- [39] R. Rothstein, B. Michel, and S. Gangloff. Replication fork pausing and recombination or "gimme a break". *Genes Dev.*, 14(1):1–10, 2000.
- [40] D. Sankoff. Rearrangements and chromosomal evolution. *Curr. Opin. Genet. Dev.*, 13(6):583–7, 2003.
- [41] D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *J. Comp. Biol.*, 5(3):555–570, 1998.
- [42] D. Sankoff and P. Trinh. Chromosomal breakpoint reuse in genome sequence rearrangement. *Journal of Computational Biology*, 12(6):812–821, 2005.
- [43] F. Swidan, E. Rocha, M. Shmoish, and R. Pinter. An integrative method for accurate genome mapping. submitted, 2005.
- [44] E. Tannier and M.-F. Sagot. Sorting by reversals in subquadratic time. In S. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, editors, *Proc. of the 15th Ann. Sym. on Combinatorial Pattern Matching (CPM)*, volume 3109 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, 2004. Springer Verlag.
- [45] P. Trinh, A. McLysaght, and D. Sankoff. Genomic features in the breakpoint regions between syntenic blocks. *Bioinformatics*, 20:i318–i325, 2004.
- [46] G. A. Watterson, W. J. Ewens, , T. E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99:1–7, 1982.
- [47] Y. Wolf, I. Rogozin, N. Grishin, R. Tatusov, and E. Koonin. Genome trees constructed using five different approaches suggest new major bacterial clades. *BMC Evolutionary Biology*, 1(1):8, 2001.