Fast computation of a string duplication history under no-breakpoint-reuse

Broňa Brejová¹, Martin Kravec¹, Gad M. Landau²,³ and Tomáš Vinař¹

¹Faculty of Mathematics, Physics and Informatics, Comenius University, Mlynská dolina, 842 48 Bratislava, Slovakia
²Department of Computer Science, University of Haifa, Haifa 31905, Israel
³Department of Computer Science and Engineering, NYU-Poly, Six MetroTech Center, Brooklyn, NY 11201-3840, USA

In this paper, we provide an $O(n \log^2 n \log \log n \log^* n)$ algorithm to compute a duplication history of a string under no-breakpoint-reuse condition. The motivation of this problem stems from computational biology, in particular, from analysis of complex gene clusters. The problem is also related to computing edit distance with block operations, but, in our scenario, the start of the history is not fixed, but chosen to minimize the distance measure.

1. Introduction

We are given a string $T$ over a finite alphabet $\Sigma$, and we assume that it was created by the following process. We start with some string $S$ which is a permutation of alphabet symbols. In each step, we apply a duplication which takes a substring of the current string (source), and inserts its copy at a different position (target) within the string (the target position cannot be inside the source). For example, if the current string is $uvw$, then the duplication with source $v$ and target between $w$ and $x$ creates string $uvwx$. The last string in this sequence of duplications should be $T$. We call a sequence of duplications leading to the construction of string $T$ from some permutation of the alphabet a duplication history of $T$.

In the general model, starting from a simple string $uv$, string $(uv)^{\ell}$ can be created by $\lceil \log \ell \rceil$ duplication operations. However, we consider a restricted version of
the problem under the no-breakpoint-reuse rule. In this version of the problem, each duplication introduces three breakpoints: at the left boundary of the source, at the right boundary of the source and between the two symbols around the target position. The above-mentioned example does not satisfy the breakpoint-reuse rule, because we repeatedly re-use the breakpoint between $u$ and $v$.

For each symbol $x$, we denote $|x|$ its left boundary and $\frac{x}{x}$ its right boundary. A breakpoint between symbols $x$ and $y$ uses the right boundary of $x$ and the left boundary of $y$. Under the no-breakpoint-reuse rule, each boundary of each symbol of the alphabet can be used at most once in the whole duplication history. Consider, for example, the sequence of duplications:

$$abcdefg \rightarrow abcdefbcdg \rightarrow abccddefbcdeg.$$ 

The first duplication uses boundaries $a\mid, b\mid, e\mid, f\mid$ and $g\mid$, and the second duplication $b\mid, e\mid, c\mid$ and $d\mid$. Thus, no more duplications can be performed on this string without violating the no-breakpoint-reuse rule.

**Definition 1.1 (Viable strings under no-breakpoint-reuse).** We call string $T$ viable if there is a permutation of alphabet symbols $S$ such that $T$ can be created from $S$ by a sequence of duplications under the no-breakpoint-reuse rule.

In this paper, we introduce an efficient quasi-linear algorithm that finds a shortest duplication history of a viable string. Our new algorithm builds on the work of Zhang et al. [1] and is the first efficient algorithm presented for this problem. In the rest of the section, we describe motivation for this problem and introduce related work.

(a) Motivation and related work

A similar problem has been studied in computational biology, in particular, in the context of gene cluster evolution analysis. Briefly, each symbol of our alphabet corresponds to a segment of the DNA sequence, which may appear in several copies in the studied gene cluster. In the simplest scenario, we ignore small local changes (e.g. point mutations and short insertions and deletions), and concentrate only on large-scale evolutionary events, such as duplications. The segmentation of the DNA sequence is created based on the observed local self-alignments of the DNA sequence, so that the breakpoints of duplications always fall between the individual segments.

In this context, Zhang et al. [1] introduced a problem of reconstruction of duplication histories, while considering an extended set of operations, including not only segmental duplications, but also tandem duplications and duplications with reversal; we do not consider these additional types of events in this paper.

Because evolution is considered to be a random process, there is only a small probability that two or more events of the same type would act on the same boundary, assuming that the sequences are long and the number of events is small [2]. While this notion can be easily incorporated into likelihood models, it is more difficult to account for it in parsimony-based models. Instead, parsimony models introduce stricter no-breakpoint-reuse rule (e.g. infinite sites model of [3]). This simplifying rule helps to make the underlying algorithmic problems tractable, as is also the case in this paper.

Zhang et al. [1] developed an elegant solution to the problem of computing the number of duplications under no-breakpoint-reuse. In the input string, they identify a candidate duplication defined by a set of simple rules. They assume this duplication is the last (most recent) duplication in the history and undo it by deleting the string inserted by the duplication event. Then, they continue the process on the resulting shorter string, until they obtain a string in which each symbol appears only once. We describe the notion of candidate duplications in detail in the next section.
Interestingly, although there can be several candidate duplications in a given step, undoing any of them leads to an optimal history. Zhang et al. [1] did not investigate the time complexity of the algorithm. Straightforward implementation takes \(O(n^3)\) time, where \(n\) is the length of the input string, whereas in this paper, we provide a quasi-linear algorithm.

Another model of duplicated strings has been introduced by Kahn et al. [4]. Here, duplications are not performed within the string, but instead we are given a source string \(S\) and a target string \(T\), and our task is to assemble \(T\) by repeatedly inserting substrings of \(S\) into the initially empty target string. The structure of the target string is much simpler in this case, and Kahn et al. [4] show a dynamic programming algorithm to compute the smallest number events needed to create the target string.

Several authors have considered also local mutations in the DNA sequences corresponding to individual segments and combined phylogenetic trees inferred on individual segments with the reconstruction of duplication histories. Optimization of such combined objectives is difficult to formulate as a clean combinatorial problem. Works of Song et al. [5] and Vinar et al. [6] extend the Zhang et al. [1] model, whereas Elemento et al. [7] and Lajoie et al. [8] consider only a single type of segments (e.g. strings of the form \(u^k\)), making the problem similar to the gene tree versus species tree reconciliation problem. Holloway et al. [9] recently introduced the two-species small phylogeny problem with duplications and deletions, where the goal was to infer the smallest cost ancestor of two extant sequences. Compared with our work, they consider a simpler evolutionary model, where duplications and deletions cannot overlap, and they solve the problem by integer linear programming.

In a more combinatorial setting, several authors have considered computing the edit distance with block operations, where in addition to the usual symbol insertions, deletions and substitutions, we allow moving, deleting or copying whole blocks of text to transform one string into another. Computing the edit distance in the presence of such large-scale operations is typically NP-hard, depending on the exact model [10,11], and authors concentrate on providing approximation algorithms [12–14] or algorithms for special cases where editing proceeds from left to right [15]. In general, these problems differ from ours by allowing a richer set of operations, but also by fixing both endpoints of the history, whereas we minimize over all possible initial permutations.

(b) Relationship to breakpoint graphs

Ma et al. [3] give a polynomial-time algorithm for a very general problem of inferring an evolutionary history of multiple species under several large-scale operations, including duplications, insertions, deletions and rearrangements. To achieve this result, they use the no-breakpoint-reuse rule, and additionally they assume that it is possible to reconstruct the phylogeny of each segment type (each symbol in our alphabet). In our work, we do not use this assumption, and indeed it would not be practical in gene cluster analysis, where individual segments are often short and highly similar to each other, making phylogeny reconstruction difficult.

Nonetheless, techniques of Ma et al. [3] can be used to compute the number of events in a duplication history of a viable string in linear time. We create a breakpoint graph that has two vertices for each symbol in the alphabet, representing the boundaries \([a\) and \(a]\). Two vertices are connected by an edge if they are adjacent in the input string. In a string without duplications, all connected components have size one or two. Each duplication replaces three components of size two by one component of size six or two components of size three. Therefore, the number of duplication events in the history is simply \(s + t/2\), where \(s\) is the number of components of size six and \(t\) is the number of components of size three. However, examination of the breakpoint graph of a string is not sufficient to reconstruct the history itself or even to distinguish if a string is viable or not (Figure 1). In the rest of this paper, we provide an algorithm capable of answering both these questions.
2. Candidate duplications

Here, we introduce necessary notation and define candidate duplications which are the crucial ingredient of the algorithm from Zhang et al. [1].

Consider a string \( X = x_1 x_2 \ldots x_n \). Let \( X[i \ldots j] \) denote the substring \( x_i \ldots x_j \). Let \( #_X(u) \) be the number of occurrences of a string \( u \) in a string \( X \). Let \( s_X(a) \) be the set of all symbols that immediately follow one of the occurrences of \( a \) in \( X \), and similarly let \( p_X(a) \) be the list of immediate predecessors of \( a \) in \( X \). We say that two symbols \( a \) and \( b \) are linked in a string \( X \) if \( s_X(a) = \{ b \} \) and \( p_X(b) = \{ a \} \). We say that two symbols \( a \) and \( b \) form a unique pair in \( X \) if \( #_X(ab) = 1 \). For the sake of simplicity, we implicitly assume two sentinel symbols at the beginning and at the end of the string which are never duplicated.

The algorithm of Zhang et al. [1] chooses in each step a candidate duplication consisting of two regions: source \( S = X[i \ldots j] \) and destination \( D = X[k \ldots ℓ] \). As we see below, the definition of candidate duplication ensures that there must be a history in which the last event was indeed duplication which created \( D \) as a copy of \( S \). The algorithm then deletes \( D \) from \( X \) and continues by finding the second most recent duplication in the same manner.

In this paper, we are focused on efficient implementation of this algorithm. To this end, we use a different definition of a candidate duplication, which is equivalent to the original definition by Zhang et al. [1] (as we show in §5), yet allows us to detect candidate duplications faster. In particular, we concentrate on finding the destination of a candidate duplication. Note that a single destination can be paired with multiple sources, but we never list all these sources explicitly.

**Definition 2.1 (Candidate destination).** A non-empty substring \( D = X[k \ldots ℓ] \) of a string \( X \) is a candidate destination if the following conditions hold:

1. \( x_{k-1} \) and \( x_k \) are a unique pair in \( X \), and so are \( x_ℓ \) and \( x_{ℓ+1} \).
2. String \( X[k \ldots ℓ] \) has at least one other copy in \( X \), i.e. \( #_X(X[k \ldots ℓ]) \geq 2 \).
3. \( x_ℓ \neq x_{k-1} \) and \( x_k \neq x_{ℓ+1} \).
4. The two symbols surrounding \( D \) occur paired with each other everywhere else in \( X \), i.e. \( s_X(x_{k-1}) \subseteq \{ x_k, x_{ℓ+1} \} \), \( p_X(x_{ℓ+1}) \subseteq \{ x_ℓ, x_{k-1} \} \).
5. \( p_X(x_k) = \{ x_{k-1}, c \} \) for some \( c \neq x_{k-1} \), and similarly \( s_X(x_ℓ) = \{ x_{ℓ+1}, d \} \) for some \( d \neq x_{ℓ+1} \).
6. Symbols \( c \) and \( d \) from the previous condition have only one neighbour, more precisely \( s_X(c) = \{ x_k \} \) and \( p_X(d) = \{ x_ℓ \} \).

See figure 1 for an example of candidate destinations in several viable strings. Note that every copy of \( D \) in \( X \) (other than \( D \) itself) is a possible source. Owing to conditions D1 and D5, these copies are flanked by the same characters (characters \( c \) and \( d \) in the definition). Therefore, the choice of the source does not influence breakpoint-reuse in the history.

Theorem 2.2 forms the basis of the algorithm.

![Figure 1](http://rsta.royalsocietypublishing.org/figures/40564g1g833.png)

**Figure 1.** (a) Two strings with the same breakpoint graph have different candidate duplications. (b) Two strings have the same breakpoint graph, but one of them is viable and the other is not. In viable strings, destinations of candidate duplications are underlined; any copy of these strings can serve as a source of the duplication.
Theorem 2.2 (Zhang et al. [1]). Let us consider all duplication histories with no-breakpoint-reuse for a viable string $X$. Then, the following holds.

1. The most recent event in each history inserts one of the candidate destinations in $X$.
2. For every candidate destination $D$ in $X$, there is a history in which $D$ was inserted by the most recent event.
3. All histories of $X$ have the same number of events.

Note that Zhang et al. [1] allow a richer set of events, yet their proof of correctness can be modified and even simplified for our scenario. The proof of the theorem is in §5.

According to theorem 2.2, we obtain the shortest duplication history by repeatedly deleting an arbitrary candidate destination, until we arrive at a string in which each symbol occurs exactly once. Regardless of the choice of candidate destinations, we always obtain a history of equal length, and the history does not contain breakpoint reuse. (Note that if two symbols are linked, in the algorithm, then they never become a breakpoint of a candidate duplication.)

3. Finding a candidate duplication in linear time

Here, we introduce a simple linear-time algorithm for finding a candidate duplication in a string $X$. We can simply repeat this algorithm in each iteration, until we undo all duplications. Because there are $O(|\Sigma|)$ duplications in any valid history under the no-breakpoint-reuse rule, this yields $O(n|\Sigma|)$ algorithm for duplication history reconstruction. We later introduce additional data structures to achieve $\tilde{O}(n)$ over all time.

According to condition D1, each candidate destination is bordered by a unique pair on each side. Clearly, a unique pair cannot occur inside any destination, otherwise it would not have a source copy (D2). Therefore, we can simply split string $X$ into possible destinations bordered by successive unique pairs and to check for each such possible destination whether it indeed satisfies the conditions from definition 2.1.

In order to quickly find all unique pairs and check conditions D4–D6, we create a data structure that lists for each symbol $a \in \Sigma$ its successors $s_X(a)$, and for each successor $b \in s_X(a)$, the number of occurrences of the pair $ab$ in $X$. Similarly, we keep a list of predecessors and the number of their occurrences. Both tables can be easily created by a single linear-time pass through the string, because each symbol has at most two distinct successors, as implied by lemma 3.1.

**Lemma 3.1.** Consider a valid duplication history under no-breakpoint-reuse, that is, a sequence of strings $X_1 \ldots X_m$ such that $X_1$ contains each symbol at most once and $X_{i+1}$ was obtained from $X_i$ by a duplication. Then, for every symbol $c$, the set of its successors $s_X(c)$ changes at most once in the history and only the following three cases can occur:

1. The set $s_X(c)$ has size 1 and does not change throughout the history.
2. The set $s_X(c)$ starts with size 1 at $i = 1$, and at some point, another element is added, creating a set of size 2.
3. The set $s_X(c)$ starts with size 1 at $i = 1$, and at some point, it changes to a different singleton set.

**Proof.** The set of successors of $c$ can change at most once, in the duplication step that uses the right boundary of $c$ in a breakpoint. Other duplications can increase the number of occurrences of individual successors of $c$, but do not add new successors.

Consider the duplication which uses the right boundary of $c$ in the duplication history. Let us assume it copies source $X[i \ldots j]$ right after character $x_i$ in a string $X$, which is one of $X_1, \ldots, X_{m-1}$. If $c = x_{i-1}$, the set of successors of $c$ does not change. If $c = x_i$, symbol $c$ gains a new successor, $x_{k+1}$, resulting in a set of successor of size two. Finally, if $c = x_k$, the number of occurrences of pair $cx_{k+1}$ decreases by one and $c$ gains a new successor, $x_i$. This can result in case (2) or (3) of the lemma, depending on the number of copies of $cx_{k+1}$ before the duplication. ■
In addition to the successor and predecessor tables, we construct a suffix array of string \( X \) and the longest common prefix (lcp) table, both in \( O(n) \) time \([16,17]\). With these data structures, we can easily enumerate all possible destinations and check conditions from definition 2.1 as follows.

First of all, we scan through the string and for each pair of adjacent symbols \( ab \) use the successor table to verify if \( b \) follows \( a \) only once. This gives us a list of unique pairs, and we process each two successive unique pairs \( x_{k-1}x_k \) and \( x_kx_{\ell+1} \), in turn.

Condition D1 is already satisfied, and condition D3 can be checked trivially. Condition D2 can be checked in \( O(1) \) time by testing if the lcp value of the suffix \( X[k \ldots n] \) with one of its neighbours in the suffix array is at least \( \ell - k + 1 \). Conditions D4–D6 can be checked easily using the successor and predecessor tables. The overall running time for finding all destinations of candidate duplications is \( O(n) \), because we need \( O(n) \) time for preprocessing, and we check \( O(n) \) potential destinations, each in \( O(1) \) time.

4. Quasi-linear algorithm for history reconstruction

In order to obtain a faster algorithm, we note that it is not necessary to build the data structures from scratch after undoing each duplication. Instead, we show how to update the data structures in each iteration, so that fast evaluation of all conditions D1–D6 can be achieved.

(a) Testing the existence of the source

The main hurdle is testing of D2, i.e. evaluating whether a substring has more than one copy in the current string. In §3, a suffix array was sufficient to facilitate this query, but it would be difficult to update the array after deleting the substring corresponding to a candidate destination. Instead, we use a data structure of Alstrup et al. \([18]\) capable of supporting the following operations over a dynamic collection of texts:

- STRING\((a)\), where \( a \in \Sigma \) creates a new string of length one in the collection,
- CONCATENATE\((s_1,s_2)\) concatenates two existing strings in the collection,
- SPLIT\((s,i)\) splits string \( s \) into two strings at position \( i \), and
- FIND\((s)\) finds all occurrences of string \( s \) from the collection in the remaining collection strings.

Operations CONCATENATE and SPLIT work in \( O(\log^2 n \log \log n \log^* n) \) time, STRING works in \( O(\log n \log^* n) \) time and FIND works in \( O(\text{occ} \log n + \log n \log \log n) \) time, where \( \text{occ} \) is the number of occurrences of the string.

In our scenario, the number of occurrences can be high, but we need only two of them (arbitrarily chosen). The underlying data structure for the FIND operation is a dynamic range tree \([19]\), which can be easily modified to retrieve two arbitrary occurrences (instead of all of them), and the resulting running time is \( O(\log n \log \log n) \).

With this data structure, we support operations needed in our scenario as follows. At the beginning, we use \( n \) times STRING and \( n - 1 \) times CONCATENATE to create a representation of \( X \) in the data structure, in the overall time \( O(n \log^2 n \log \log n \log^* n) \). To delete a destination \( D = X[k \ldots \ell] \), we use SPLIT twice to split \( X \) into \( X[1 \ldots k-1] \), \( X[k \ldots \ell] \), and \( X[\ell+1 \ldots n] \). The middle part is split into individual singletons by additional \( \ell - k \) splits, and the two remaining parts of \( X \) are concatenated together. The overall time for a deletion of a substring of length \( p \) from \( X \) is thus \( O((p+1) \log^2 n \log \log n \log^* n) \). Because every symbol in \( X \) is deleted at most once over the course of the whole algorithm, we spent a total of \( O(n \log^2 n \log \log n \log^* n) \) time on all deletions together.

Finally, to check if a possible destination \( X[k \ldots \ell] \) has at least one other copy in \( X \), we split \( X \) into \( X[1 \ldots k-1] \), \( X[k \ldots \ell] \) and \( X[\ell+1 \ldots n] \) and use operation FIND on the middle string. Regardless of the total number of occurrences of \( X[k \ldots \ell] \) in \( X \), we need only to find out if there is at least one additional copy, and if yes, at which position. Finally, we concatenate the two strings.
back together. The whole operation can be achieved in $O(\log^2 n \log \log n \log^* n)$ time. (Note that singletons from deleted substrings technically remain in the collection, and therefore we need to test for occurrences of strings of length one by a different method. This can be easily achieved by other data structures maintained in our algorithm.)

(b) Efficient maintenance of possible destinations

Even though we have a data structure that can efficiently answer queries about the existence of a source for a particular destination, to achieve quasi-linear time, we cannot afford to test all possible destinations in each round of the algorithm.

To address this problem, we maintain a catalogue of all possible destinations (i.e. regions between unique pairs), and we also maintain a status on each such destination. The status can achieve one of three values: discarded, waiting or valid. A possible destination is discarded, if it does not satisfy at least one condition out of D3–D6. The destination is valid, if it satisfies D2 (it has another occurrence in $X$). A valid destination may or may not satisfy conditions D3–D6, but we have never seen those conditions violated in any check done so far. Finally, waiting destinations do not satisfy condition D2, and we have never seen conditions D3–D6 violated.

Waiting destinations may become valid destinations by emergence of new sources after removal of some of the duplications, and valid destinations can also become waiting. However, once a destination is discarded (by checking conditions D3–D6 and finding that one of them is not satisfied), it can never become a valid destination again, as outlined in lemma 4.1.

Lemma 4.1 (Life cycle of possible duplications). If a possible duplication bordered by two unique pairs fails one of the conditions D3–D6, then it will never meet all conditions of definition 2.1 in the course of the history reconstruction algorithm.

Proof. Clearly, condition D3 can be changed only by a change in the unique pairs at the boundaries which would lead to the disappearance of this particular possible duplication. The remaining conditions D4–D6 concern the sets of predecessors and successors of the symbols at the proposed breakpoints. Let us assume, without loss of generality, that the condition which is not satisfied involves the list of successors of symbol $a$. The only time a set of successors of $a$ changes is when we create a breakpoint after $a$. However, that means that in the rest of the history $a$ cannot be used as a left side of the breakpoint, and therefore no duplication involving a breakpoint after $a$ will ever satisfy all conditions of definition 2.1.

To maintain the list and status of possible destinations, we use several simple data structures. First of all, we augment the table of predecessors and successors of each symbol with the list of occurrences for each pair of symbols $a$ and $b \in \pi_X(a)$. Coordinates of occurrences are kept with respect to the original input string before any deletions. The list of occurrences is kept in a balanced binary search tree, facilitating deletions of occurrences in $O(\log n)$ time.

The entire string is kept in an augmented binary search tree with the original position used as the key. This allows us to find predecessor and successor of each symbol of the current string and to find a character at a given position in either current or original coordinates. All of these operations can be performed in $O(\log n)$ time; deletion of any character is also $O(\log n)$ time.

Finally, we keep coordinates of all unique pairs in a binary search tree with the original coordinate of the first letter in the pair as the key. For each unique pair, we also maintain the status of the possible destination in the region between this pair and its successor. The set of valid destinations is kept in a separate binary search tree, indexed by the original coordinate.

Upon removal of destination $D$, all these data structures need to be updated. In particular

— all symbols of $D$ need to be removed from the binary tree storing $X$.
— predecessor and successor data structures need to be updated for every pair of adjacent symbols from $D$.
— let $a$ and $b$ be symbols adjacent to $D$ before its deletion. We need to add pair $ab$ to the successor and predecessor tables.
As a consequence of these steps, there can be new unique pairs created as well as some of the previous unique pairs deleted. Such pairs can be easily identified during the above-mentioned update steps. The binary tree storing unique pairs thus needs to be updated. Finally, we create a ‘due for revision’ list of potential destinations whose status needs to be recalculated after the removal of \( D \). This list includes newly created destinations, that is those that are adjacent to newly created unique pairs as well as those that were created by merging adjacent possible destinations after unique pairs were removed. Some existing destinations might also change status from waiting to valid, if a copy was created by removal of \( D \). All such destinations must contain pair \( ab \) as a substring. We can find all destinations containing \( ab \) using our data structures and add them to the ‘due for revision’ list. The only destination that may change status from valid to waiting is the source of the current destination \( D \), and we treat it separately in the algorithm.

(c) Algorithm and time complexity

Now, we are ready to state the final version of our algorithm. Its key part is the operation \( \text{CHECK}(k, \ell) \) that checks if a possible destination \( X[k \ldots \ell] \) in the current string \( X \) satisfies the conditions of definition 2.1, and returns the new state of the destination: discard, valid or waiting; if the new state is valid, then it also returns one possible source \( S \).

The running time of this procedure is \( O(\log^2 n \log \log n \log^* n) \) time. We need \( O(\log n) \) time to convert between different forms of coordinates and one call to operation FIND in the dynamic text data structure.

At the beginning we initialize the dynamic text data structure, create all necessary binary trees and test the status of every possible destination. Afterwards, we proceed as follows:

1. Let \( D \) be one of the valid destinations. Remove \( D \) from the list of valid destinations. If there are no more valid destinations, finish.
2. Use \( \text{CHECK} \) to classify \( D \). If the result is ‘discarded’, mark \( D \) as ‘discarded’ and repeat from step 1. Otherwise, let \( S \) be the returned source. (At this step, the result cannot be ‘waiting’.)
3. Remove \( D \) from the dynamic text data structure.
4. Remove \( D \) from the data structures used for the maintenance of possible destinations. Let \( R \) be the list of destinations ‘due for revision’.
5. For each \( D' \) from \( R \), run \( \text{CHECK}(D') \) and update its state.
6. If \( S \) is a possible destination, run \( \text{CHECK}(S) \) and update its state.
7. Repeat from step 1.

To analyse the running time, first note that throughout the whole algorithm each symbol of the alphabet may be the left part of at most two unique pairs, and therefore overall at most \( O(|\Sigma|) \) unique pairs are both created and deleted. The overall number of possible destinations considered in the algorithm is also \( O(|\Sigma|) = O(n) \).

Status of each new possible destination is checked once in step (5), when it is created. Each possible destination is handled at most once in step (2), and afterwards, it is either discarded or undone. Step (6) is run \( O(n) \) times in the whole algorithm, at most once for every duplication. Finally, step (5) is also run for all possible destinations containing currently created new pair \( ab \). For each \( a \in \Sigma \), this happens at most once owing to the no-breakpoint-reuse assumption. Therefore, for each occurrence of \( a \) in the original string \( X \), this rule triggers the check of the possible destination currently containing \( a \) at most once. The total number of calls to \( \text{CHECK} \) is thus \( O(n) \).

The remaining operations necessary to update and use our data structures take either \( O(\log n) \) per iteration or \( O(\log n) \) per deleted character, leading to an \( O(n \log n) \) contribution to the overall running time. The total running time of the algorithm is thus \( O(n \log^2 n \log \log n \log^* n) = \tilde{O}(n) \).
5. Proof of correctness

Here, we formalize the relationship of our algorithm to the approach by Zhang et al. [1] and prove that our algorithms indeed find a correct duplication history (theorem 2.2).

Our algorithms are based on efficient detection of candidate destinations, whereas Zhang et al. [1] use the notion of candidate duplications defined by a slightly different set of conditions. We start by paraphrasing their definition and proving that it is equivalent to ours. The following definition of a candidate duplication is rephrased from Zhang et al. [1], because we express it for an arbitrary string $X$, whereas Zhang et al. [1] assume that each pair of linked symbols is collapsed to a single symbol.

**Definition 5.1 (Candidate duplication).** A candidate duplication in a string $X$ is a pair of non-empty substrings $S = X[i \ldots j]$ (source) and $D = X[k \ldots ℓ]$ (destination) such that all of the following conditions hold.

- **Z1** Intervals $[i, j]$ and $[k, ℓ]$ are disjoint.
- **Z2** Substrings $X[i \ldots j]$ and $X[k \ldots ℓ]$ are identical.
- **Z3** Symbols $x_{i−1}$, $x_i$, and $x_{k−1}$ are pairwise distinct; similarly $x_i$, $x_{j+1}$ and $x_{ℓ+1}$ are also three distinct symbols.
- **Z4** Let $X'$ be the string obtained by deleting $D$ from $X$, i.e. $X' = X[1 \ldots k-1]X[ℓ+1 \ldots n]$. Then, symbols $x_{k−1}$ and $x_{ℓ+1}$ are linked in $X'$.
- **Z5** Similarly, symbols $x_{i−1}$ and $x_i$ as well as $x_j$ and $x_{j+1}$ are also linked in $X'$.

**Lemma 5.2 (Destination detection).** Substring $D = X[k \ldots ℓ]$ is a candidate destination in a string $X$ according to definition 2.1 if and only if there a substring $S = X[i \ldots j]$ such that the pair $(S, D)$ is a candidate duplication according to definition 5.1.

**Proof.** We first prove that conditions D1–D6 hold for the destination of each candidate duplication. Most conditions have two symmetrical parts, and we generally prove only the first of them; the other is proved analogously. Let $S = X[i \ldots j]$ and $D = X[k \ldots ℓ]$ be a candidate duplication in a string $X$, and let $X'$ be the string obtained from $X$ by deleting $D$. First, condition D2 follows directly from conditions Z1 and Z2 of definition 5.1. Condition D3 is also easy to see, because $x_i = x_j$ and $x_j \neq x_{k−1}$ by condition Z3.

The remaining conditions are concerned with pairs of characters in $X$. Let $P$ be the set of all words of length 2 that occur as substrings of $X$ and similarly let $P'$ be such a set of pairs in $X'$. Note that $P' \subseteq P \cup \{x_{k−1}x_{ℓ+1}\}$, because only one new pair can be created by deleting $D$. Conversely, $P \subseteq P' \cup \{x_{k−1}x_k, x_kx_{ℓ+1}\}$, because all pairs completely contained within $D$ occur also in $S$, and as such they do not disappear completely from the string after deleting $D$. The only potentially disappearing pairs are thus those spanning the boundaries of $D$.

To prove condition D4, first note that $s_X(x_{k−1}) = \{x_{ℓ+1}\}$ by condition Z4. Moreover, we have proved in D3 that $x_{k−1} \neq x_i$. Now using our observation about sets $P'$ and $P$, we obtain $s_X(x_{k−1}) \subseteq s_{X'}(x_{k−1}) \cup \{x_k\} = \{x_{k}, x_{ℓ+1}\}$.

By extending this idea, we can also prove D1. As we have noted, $s_X(x_{k−1}) = \{x_{ℓ+1}\}$. By definition Z3, we have $x_{ℓ+1} \neq x_i = x_k$, and therefore all occurrences of $x_{k−1}$ followed by $x_k$ must be destroyed by deleting $D$ from $X$. If any such occurrence was completely inside $D$, then its copy in $S$ would remain in $P'$. Therefore, such occurrences must span boundaries of $D$. The right boundary cannot be an occurrence of $x_{k−1}x_k$, because $x_{k−1} \neq x_{ℓ}$. Thus, the left boundary of $D$ is the only possible occurrence of this pair.

To prove condition D5, we choose $c = x_{i−1}$ (and by symmetry $d = x_{j+1}$). Condition D1 implies that $x_{k−1}x_k$ is a unique pair. However, $x_k$ has another copy at $x_i$, which implies that $c = x_{i−1} \neq x_{k−1}$. Thus, we see that $(c, x_{k−1}) \subseteq p_Xx_k$. Moreover, $p_X(x_k) = \{c\}$ by condition Z5. Any additional predecessors of $x_k$ thus must have been deleted from $S$ with $D$, but, because $x_{ℓ+1} \neq x_k$ by D3, the only such deleted predecessor is $x_{k−1}$, which implies $p_X(x_k) = \{c, x_{k−1}\}$. 

Finally, to prove D6, note that by condition Z5, \( s^X(x_{i-1}) = x_i = x_k \). Character \( x_k \) also clearly follows \( x_{i-1} \) in \( X \) at the left boundary of \( S \). We have already established that \( x_{i-1} \neq x_i \); in addition \( x_{i-1} \neq x_j \) by condition Z3. Therefore, by comparing sets \( P \) and \( P' \), there can be no additional successor of \( x_{i-1} \) in \( P \) compared with \( P' \).

This completes the proof that the destination of any candidate duplication satisfies conditions D1–D6. We now show that if a string \( D = X[k \ldots \ell] \) satisfies conditions D1–D6, it is the destination of some candidate duplication satisfying the conditions Z1–Z5 of definition 5.1. In particular, owing to condition D2, \( X[k \ldots \ell] \) has at least one other copy, say \( X[i \ldots j] \). We prove that \( X[i \ldots j] \) is a proper source for destination \( X[k \ldots \ell] \).

**Condition Z1:** if intervals \([i, j]\) and \([k, \ell]\) overlap, either \( k \) or \( \ell \) must be inside \([i, j]\). This is because \([i, j]\) and \([k, \ell]\) have the same length. Without loss of generality, assume that \( k \) is inside \([i, j]\) and because \( i \neq k \) and \( k - 1 \) is also inside \([i, j]\). String \( x_{k-1}x_k \) is thus completely inside \( X[i \ldots j] \), and therefore it has another copy inside \( X[k \ldots \ell] \), which is impossible owing to condition D1.

**Condition Z2:** this condition is clearly satisfied by our choice of \( X[i \ldots j] \).

**Condition Z3:** first, we assume that \( x_{i-1} = x_{i-1} \). Because \( x_i = x_j \), then \( x_{i-1}x_i = x_{k-1}x_k \) which contradicts condition D1. Second, if \( x_j = x_{k-1} \) then \( x_i \) is followed by \( x_k \) or \( x_{i+1} \) by condition D4. The former is impossible because, by condition D1, pair \( x_{k-1}x_k \) is unique. For the latter, because \( x_j = x_{i+1} \) this also contradicts D1.

Third, let \( x_{i-1} = x_j \). Because \( x_j = x_{i+1} \), we have \( s^X(x_{i-1}) = [x_{i+1}, d] \) for some \( d \neq x_{i+1} \) by condition D5, and thus either \( x_i = x_{i+1} \) or \( x_j = d \). The former leads to contradiction, because \( x_{i-1}x_i = x_{i+1}x_j \) and by condition D1, \( x_{i+1}x_j \) is a unique pair. This would imply that \( i = \ell + 1 \), but by condition D3, \( x_{i+1} \neq x_i \). On the other hand, if \( x_j = d \), by condition D6, then we have \( p_X(d) = p_X(x_j) = p_X(x_k) = \{x_i\} \). Thus, \( x_{k-1} = x_{i+1} \) which contradicts D3. Proof of the second part of Z3 is analogous.

**Condition Z4:** by condition D4, \( x_{k-1} \) is followed only by symbols \( x_k \) and \( x_{i+1} \). Symbols \( x_{k-1} \) and \( x_k \) form a unique pair in \( X \), and this pair is disrupted by the removal of \( D \) from \( X \). Thus, \( s^X(x_{i-1}) = \{x_{i+1}\} \). Similarly, we can argue that \( p_X(x_{i+1}) = \{x_{k-1}\} \). Therefore, symbols \( x_{k-1} \) and \( x_{i+1} \) are linked in \( X' \).

**Condition Z5:** by condition D5, we have \( p_X(x_i) = p_X(x_k) = \{x_{k-1}, c\} \) for some \( c \neq x_{k-1} \). Because \( x_{i-1} \in p_X(x_i) \) and \( x_{k-1} \neq x_{i-1} \) by condition Z3 proved above, we have \( c = x_{i-1} \). By condition D6, \( s^X(x_{i-1}) = \{x_{k-1}\} \). This means that symbols \( x_{i-1} \) and \( x_i \) are almost linked in \( X \), except for the unique pair \( x_{k-1}x_k \). When we delete \( X \), unique pair \( x_{k-1}x_k \) is removed and a new pair \( x_{k-1}x_{i+1} \) is created. As we have shown in Z3, \( x_{k-1} \neq x_{i-1} \) and \( x_{i+1} \neq x_i \); and thus symbols \( x_{i-1} \) and \( x_i \) are linked in \( X' \). Similarly, we can prove that symbols \( x_j \) and \( x_{i+1} \) are also linked in \( X' \).

Our proof of theorem 2.2 is based on the proof from Zhang et al. [1], but we simplify it for our model without inversions and tandem duplications, and we use both definitions 2.1 and 5.1. Before proceeding to the proof of the theorem, we introduce several useful lemmas. The first actually proves statement (1) of the theorem.

**Lemma 5.3.** Consider a history without breakpoint-reuse producing a string \( X \). Then, the destination of most recent duplication in this history is a candidate destination in \( X \).

**Proof.** Let the most recent duplication be a pair \((S, D)\). We prove that it satisfies conditions Z1–Z5 of definition 5.1. Owing to lemma 5.2, it also satisfies conditions of definition 2.1.

Clearly, conditions Z1 and Z2 are satisfied by every duplication. Condition Z3 is satisfied, because otherwise the duplication would use the same breakpoint twice. Finally, in the starting string of any history, every symbol occurs only once, and therefore every symbol is linked to both its neighbours. If \( a \) and \( b \) are linked before a duplication but not after the duplication, then the duplication needs to use a breakpoint after \( a \) or before \( b \). If one of the conditions Z4 and Z5 was not satisfied, then one of the characters \( x_{i-1}, x_i, x_j, x_{i+1}, x_{k-1}, x_{k+1} \) could never become linked to its neighbour without breakpoint reuse.

Before proceeding with additional lemmas, we introduce notation useful for referring to particular regions of a string and mapping these regions between different stages in a duplication history. Let \( A = X[i \ldots j] \) be a substring of string \( X \). We treat \( A \) both as a string and as a specific
region in $X$. Relation $A = B$ denotes string equality, and relation $A \equiv B$ denotes two identical regions. Therefore, $A \equiv B$ implies $A = B$, but the converse is not always true, because $A$ can have multiple occurrences in $X$. Let $X'$ be a string obtained from $X$ by deletion of some region $D = X[k \ldots \ell]$ and let $A = X[i \ldots j]$ be a region disjoint with $D$. Then, by $A(X')$ we denote the region in $X'$ corresponding to $A$, i.e. if $j < k$, then we have $A(X') = X'[i \ldots j]$ and if $i > \ell$, we have $A(X') = X'[i - (\ell - k + 1) \ldots j - (\ell - k + 1)]$. Mapping of regions from $X$ to $X'$ is not defined for regions overlapping $D$. We also use the inverse mapping from $X'$ to $X$ such that $B \equiv A(X')$ if and only if $B(X) \equiv A$. Finally, we can extend mapping to a pair of strings separated by several events in a duplication history by mapping regions through all intermediate strings.

As each candidate destination is a possible destination bordered by unique pairs, no two candidate destinations overlap. In lemma 5.4, we prove that they cannot be even adjacent, and moreover, one destination cannot start with the same character as the one following the other destination.

**Lemma 5.4.** Let $D = X[k \ldots \ell]$ and $D' = X'[k' \ldots \ell']$ be two candidate destinations in string $X$, where $D \neq D'$. Then $x_{\ell+1} \neq x_{k'}$, $x_{\ell+1} \neq x_{k'}$, $x_{k-1} \neq x_{\ell}$, and $x_{k-1} \neq x_{\ell}$.

*Proof.* Let us assume that $x_{\ell+1} = x_{k'}$, all other cases can be proved analogously. According to condition D2, $D$ must have a source $S = X[i \ldots j]$ and $D'$ must have a source $S' = X'[i' \ldots j']$, thus $x_j = x_\ell$ and $x_{k'} = x_{\ell+1}$. Predecessors of $x_{\ell+1}$ include $x_{k-1}, x_\ell$ and $x_{\ell-1}$ (not all of them are necessarily unique). We consider two cases: if $D'$ immediately follows $D$, and if $D'$ does not immediately follow $D$.

If $D'$ immediately follows $D$, according to D5 applied to $D$, $s_X(x_\ell) = \{x_{\ell+1}, x_{j+1}\}$. At the same time, if we apply D4 to $D'$, we obtain $s_X(x_{\ell}) = \{x_{\ell+1}, x_{j+1}\}$. Comparing the two sets, we obtain $x_{j+1} = x_{\ell+1}$. By D6 applied to $D$, $p_X(x_{j+1}) = \{x_j\}$, thus $x_{\ell} = x_j$. Because $x_\ell x_{\ell+1}$ is a unique pair, $S$ must be located at the right boundary of $D'$.

At the same time, by D5 applied to $D'$, we have $s_X(x_{\ell}) = \{x_{\ell+1}, x_{j'+1}\}$. By similar argument as before, $S'$ must be located at the right boundary of $D$. If one of $D$ and $D'$ is longer than the other, corresponding region $S'$ or $S$ would overlap a unique pair, which would lead to a contradiction. Therefore, $D$ and $D'$ must be of the same length and $D = D'$. However, such an arrangement would violate D3.

In the case that $D'$ does not immediately follow $D$, because both $x_\ell x_{\ell+1}$ and $x_{k-1} x_{\ell+1}$ are unique pairs (D1), and $p_X(x_{\ell+1})$ must have two distinct elements (D5 applied to $D'$), $x_\ell = x_{\ell-1}$. Moreover, $S'$ must immediately follow $D$. That would mean that according to D6 applied to $D'$, $x_\ell$ must be always followed by $x_{\ell+1}$, but this contradicts D5 applied to $D$ stating that there must be another symbol following $x_\ell$ at the right boundary of $S$.

**Lemma 5.5.** Let $D$ and $D'$ be two candidate destinations in a viable string $X$ and let $X'$ be obtained from $X$ by deleting $D'$. Then, region $D(X')$ is a candidate destination in $X'$, unless $D = D'$ and string $D$ occurs only once in $X'$.

*Proof.* Denote $D = X[k \ldots \ell]$ and $D' = X'[k' \ldots \ell']$. We prove that the region $D(X')$ satisfies conditions D1–D5 in $X'$, unless the condition stipulated above holds.

Removal of $D'$ creates only one new pair $x_{k-1} x_{\ell+1}$, which is linked in $X'$ according to D4 applied on $D'$. The set of successors will change only for $x_{k-1}$ and $x_{\ell}$, but, in both cases, the resulting set will be a singleton. Similarly, the set of predecessors changes only for $x_{k'}$ and $x_{\ell+1}$. This implies that condition D6 for $D$ stays satisfied in $X'$, because even if $c$ is one of the characters for which successors change, the resulting set is a singleton.

Condition D1 also stays satisfied, because disrupting it would require $x_k = x_{\ell+1}$ or $x_k = x_{k-1}$, both of which are not possible (lemma 5.4). The same lemma also implies that $D$ and $D'$ are non-adjacent in $X$, so deleting $D'$ does not affect D3. For condition D4 on $s_X(x_{k-1})$ to become violated in $X'$, it would have to hold $x_{k-1} = x_{\ell+1}$. Because $x_{k-1} x_{\ell+1}$ is linked in $X'$, it follows $x_k = x_{\ell+1}$ which again violates lemma 5.4; symmetric argument holds for $p_X(x_{\ell+1})$. 

Now, let us assume that condition D2 (existence of a copy) is not satisfied for $D^{(X)}$. This means that deletion of $D'$ disrupts source $S$ that existed in $X$ for $D$. Note that no boundary of $D'$ can be strictly inside $S$, because $S$ has another copy in $X$ and boundaries of $D'$ are flanked by unique pairs. Therefore, if $S$ is disrupted by deletion of $D'$ from $X$, then it must be deleted as a whole. In this case, $D$ is a substring of $S'$, which means that $D^{(X)}$ has another copy, unless $D$ itself is inside $S'$. Owing to unique pairs flanking $D$ this means that $D \equiv S'$, and thus $D = D'$. In such case, however, the conditions of the claim would be violated, because for $D = D'$ two copies are explicitly required.

Condition D2 implies that $D$ has a copy $S$ in $X'$ and owing to condition D1, $D$ and $S$ are preceded by different characters $x_{k-1}$ and $c$. To prove D5, it remains to show that no other character precedes $x_k$. This is evident from the fact that characters changing their set of predecessors after deletion of $D'$ end up with singleton sets. The proof for successors of $x_k$ is analogous.

**Lemma 5.6.** Let $D$ and $D'$ be two candidate destinations in a string $X$ such that string $X'$ obtained by deleting $D'$ from $X$ still has $D^{(X)}$ as a candidate destination. Then, $D'$ is a candidate destination in string $X''$ obtained by deleting $D$ from $X$.

**Proof.** If $D \neq D'$, we can apply lemma 5.5. If on the other hand $D = D'$, then $D^{(X)}$, as a possible source of $D'$, must be flanked by linked pairs in $X'$, and therefore cannot be a candidate destination.

**Lemma 5.7.** Consider a duplication history without breakpoint-reuse consisting of $k$ duplications $O_1, \ldots, O_k$, and let $X_i$ be the string after duplications $O_1, \ldots, O_i$. Let $D$ be the destination of some duplication $O_i$ such that $D^{(X)}$ remains a candidate destination in all subsequent strings $X_j$ for $j = i, \ldots, k$. Then, we can always find a history $O'_1, \ldots, O'_k$ without breakpoint-reuse, leading to the same $X_k$, in which $O'_k$ has $D$ as a destination.

**Proof.** We use induction on the difference $k - i$. If $k - i = 0$, then we simply set $O'_j = O_j$ for all $j$. We now prove the claim for $k - i = \ell \geq 1$, assuming it holds for differences up to $\ell - 1$. Using this assumption, we can find a history $O'_1, \ldots, O'_{k-1}$ in which the last duplication has $D$ as a destination and which produces the same string $X_{k-1}$ as $O_1, \ldots, O_{k-1}$. Let $D'$ be the destination of duplication $O_k$, and let $Y$ be the string obtained by deleting $D$ from $X_k$. Because $D$ is a candidate destination in both $X_{k-1}$ and $X_k$ and $D'$ is a candidate duplication in $X_k$ (owing to lemma 5.3), we can apply lemma 5.6, obtaining that $D'$ is a candidate duplication in $Y$. We set $O'_j = O'_j$ for $j < k - 2$; $O'_{k-1}$ is a duplication creating $Y$ by inserting $D'$, and $O'_k$ is the duplication creating $X_k$ by inserting $D$. Because destination of each duplication $O'_j$ is a candidate destination in the string resulting from $O'_1 \ldots O'_{k-1}$, no breakpoint-reuse occurs in this new history.

**Lemma 5.8.** Consider a duplication history of $X$ without breakpoint-reuse consisting of $k$ duplications $O_1, \ldots, O_k$, where the last duplication $O_k$ copies source $D$ to destination $S$. If $D$ is also a candidate destination in $X$, there exists a sequence of $k$ duplications $O'_1, \ldots, O'_k$ for string $X$ such that the last operation $O'_k$ copies source $S$ to destination $D$.

**Proof.** Let $S = X[i \ldots j]$ and $D = X[k \ldots \ell]$. If both $D$ and $S$ are candidate destinations in $X$, then by condition D1, $x_{j-1}x_i$, $x_{j+1}x_{j+1}$, $x_{k-1}x_k$, and $x\ell_{x\ell+1}$ must all be unique pairs in $X$. In addition, pair $x_{j-1}x_i$ is linked after removing $D$, and thus $x_{j-1}$ must occur only once in $D$. Similar arguments can show that $x_{j+1}x_{k-1}$ and $x_{\ell+1}$ are also unique in $X$. As a result, the two segments $S$ and $D$ are bounded within unique symbols and thus form ‘two islands’. Any previous duplication related to $S$ or $D$ segments must be completely inside of either $S$ or $D$, and they do not share boundaries with $S$ or $D$. Therefore, to change the latest duplication from $O_k = (D, S)$ to $O'_k = (S, D)$, we simply ‘redirect’ all the duplications that are inside of $D$ to be inside of $S$, and keep the rest the same. This creates a new sequence of duplications $O'_1, \ldots, O'_k$ that creates $X$.

Finally, we are ready to prove the main theorem.
Proof of theorem 2.2. Statement (1) of the theorem is proved in lemma 5.3. Let \(O_1, O_2, \ldots, O_k\) be a series of \(k\) duplications constituting a duplication history of a viable string \(X\). To prove statements (2) and (3), we show that for any candidate destination \(D\), string \(X\) can also be created by a history \(O'_1, O'_2, \ldots, O'_k\) of the same length, where the last duplication \(O'_k\) inserts \(D\). The claims of the theorem are a direct consequence of this claim, proved by induction on the number of duplication events.

Now consider a candidate destination \(D\) in the sequence \(X\). If we look at the duplication history in reverse, we can show that \(D\) is always a candidate destination until one of the following happens (see lemma 5.5): (A) either \(D\) is deleted as a destination of some duplication \(O_i\) or (B) all the sources matching \(D\) are deleted, and the role destination is in fact lost by a duplication \(O_i\) using \(D\) as a source.

In the case (A), we apply lemma 5.7 to find a history of length \(k\) that will create sequence \(X\), where \(D\) is the destination of the latest duplication.

In the case (B), the role of the destination has been gained by duplication \(O_i = (D, S)\). Immediately after this event, both \(S\) and \(D\) are candidate destinations. Therefore, we can replace \(O_1, \ldots, O_i\) with some sequence of duplications \(O'_1, \ldots, O'_i\) such that we obtain the same intermediate string at time \(i\), where \(O'_i\) inserts \(D\) (lemma 5.8). Using the sequence of duplications \(O'_1, \ldots, O'_i, O_{i+1}, \ldots, O_k\), we reduce case (B) to case (A), for which we have already proven the claim.

\[\square\]

6. Conclusion and open problems

In this paper, we have presented an efficient algorithm to compute the shortest duplication history of a string under no-breakpoint-reuse rule. The original algorithm is due to Zhang et al. [1], who did not study its complexity or an efficient implementation.

Many questions in this area still remain open. First of all, our algorithm computes one possible history, and it can be easily extended to enumerate all possible histories in \(O(n^2)\) time per history using recursive search through all candidate destinations produced by the algorithm shown in §3. However, a faster algorithm should be possible by appropriate changes in the data structures. Instead of enumerating all histories, one might be interested in efficiently counting or sampling possible histories, which remains an open problem. A natural extension asks for a history with a fixed starting string. If the nice structure of the problem carries into this scenario, the problem may become one of the few tractable block edit distance problems.

Time complexity \(O(n \log^2 n \log \log n \log^2 n)\) hardly seems natural for this problem. It would be desirable to speed up the algorithm to \(O(n \log n)\) or even to \(O(n)\) time (recall that it is possible to compute the history length in linear time). The main obstacle is the lack of faster data structures for dynamic text that would verify possible sources for destinations in our algorithm. Such a data structure would be certainly useful in many other areas of string processing.

Another class of open problems concerns models in which the no-breakpoint-reuse assumption is partially or completely relaxed or in which additional operations, such as deletions, are allowed. An interesting scenario, with strong biology motivation, allows an operation of speciation. In this scenario, we work with a collection of strings, each from a different biological species. The history starts with a single string, and at any point in the duplication history, we may create another copy of one of the strings in the collection. No further copying between the different strings in the collection is allowed; all duplications must act within the confines of one of these strings (i.e. the strings in the collection evolve independently). Towards solution of this problem, Zhang et al. [1] state without a proof that under the no-breakpoint-reuse scenario, this problem can be solved by a slight modification of their algorithm, but a more detailed investigation of this problem is warranted.

Acknowledgements. An early version of this paper appeared in SPIRE 2011.

Funding statement. This research was partially supported by the National Science Foundation Award 0904246, Israel Science Foundation grant 347/09, Yahoo, grant no. 2008217 from the United States–Israel Binational Science Foundation (BSF) and DFG, and VEGA grant nos. 1/1085/12 and 1/0719/14.
References


