

Permutation-constrained Common String Partitions with Applications

Manuel Lafond, Binhai Zhu

This version of the article has been accepted for publication, after peer review (when applicable) and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: <http://dx.doi.org/10.1007/s00453-024-01276-7>

Accessibility Disclaimer:

For a more accessible version of this document, please submit an accessibility request form through the Montana State University Library website.

Permutation-constrained Common String Partitions with Applications

Manuel Lafond* · Binhai Zhu

Received: date / Accepted: date

Abstract We study a new combinatorial problem based on the famous Minimum Common String Partition (MCSP) problem, which we call Permutation-constrained Common String Partition (PCSP for short). In PCSP, we are given two sequences/genomes s and t with the same length and a permutation π on $[\ell]$, the question is to decide whether it is possible to decompose s and t into ℓ blocks that can be matched according to some specified requirements, and that conform with the permutation π . Our main result is that PCSP is FPT in parameter $\ell + d$, where d is the maximum number of occurrences that any symbol may have in s or t . We also study a variant where the input specifies whether each matched pair of block needs to be preserved as is, or reversed. With this result on PCSP, we show that a series of genome rearrangement problems are FPT $k + d$, where k is the rearrangement distance between two genomes of interest.

Keywords Genome rearrangement · Permutation-constrained common string partition · Minimum common string partition · FPT

1 Introduction

Computing the distance between genomes is a fundamental problem in biology; in fact, the reversal and the corresponding breakpoint concept on sections of some chromosome (when a whole genome was not obtainable) were studied as early as in 1926 by Sturtevant and Dobzhansky (Sturtevant, 1926;

* Manuel Lafond (corresponding author)

Department of Computer Science, Université de Sherbrooke, Sherbrooke, Quebec J1K 2R1, Canada

E-mail: manuel.lafond@usherbrooke.ca ·

Binhai Zhu

Gianforte School of Computing, Montana State University, Bozeman, MT 59717, USA

E-mail: bhz@montana.edu

Sturtevant and Dobzhansky, 1936). The signed reversals were confirmed in the radish genome in 1988 (Makaroff and Palmer, 1988). Since then, a lot of rearrangement operations were studied, for instance, reversals, transpositions and block interchanges, etc. Here we focus more on operations on unichromosomal genomes — singleton genomes, each being a sequence; and we skip the details for genomic operations applied on multichromosomal genomes, like translocation and the more general double-cut-and-join operations. We refer the reader to (Fertin et al., 2009) for more information on string distances and their applications in understanding genome rearrangement.

Let us first describe some popular string operations. A *transposition* takes two adjacent substrings and swaps them. A *block-interchange* takes two non-overlapping substrings and swaps them. The recent *p-cut* partitions a string into p substrings and permutes them in any way (Bulteau et al., 2021b). On signed alphabets, which are alphabets with an involution acting on them, a *flip* switches the sign of each symbol of a chosen substring. A *reversal* first applies a flip to a substring, and then reverses the order of the characters. An *unsigned reversal* only reverses the order of the characters of a chosen substring. If $\mathcal{P}$ is a set of string operations, $d_{\mathcal{P}}(s, t)$ is the minimum number of operations from $\mathcal{P}$ needed to transform s into t . We also call $d_{\mathcal{P}}(s, t)$ the corresponding *distance* under operations in $\mathcal{P}$.

Computing most of these distances is NP-hard under different circumstances. The transposition distance problem (Bulteau et al., 2012) and the unsigned reversal distance problem (Caprara, 1999) are NP-hard even if s and t are permutations (i.e., 1-occurrence strings). The block-interchange distance problem is polynomially solvable for permutations (Christie, 1996); however, when s and t are sequences over a fixed-size alphabet the problem becomes NP-hard (Christie and Irving, 2001). Other types of distances that remain to be incorporated into our framework have recently been studied, including the NP-hard tandem duplication distance (Lafond et al., 2022b) and the copy-number distance (Qingge et al., 2018; Lafond et al., 2020).

In many practical applications, the value $k = d_{\mathcal{P}}(s, t)$ is small, and it makes sense to consider the FPT (fixed-parameter tractable) complexity of these problems — using parameter k . When s and t are permutations, both the transposition distance problem and the unsigned reversal distance problem are FPT since each such operation reduces at most three breakpoints (Mahajan et al., 2006). When s and t are d -occurrence sequences (i.e., each letter appears at most d times in s and t), whether the other distance problems are FPT in parameter $k + d$ is unknown, to the best of our knowledge. The d -occurrence variant is of practical interest in bioinformatics, since in genome comparison, repeated occurrences represent duplicated genes. Duplication is a rare event and the number of duplicates is usually less than a dozen. For instance, plants have undergone up to three rounds of whole genome duplications, resulting in a number of duplicates bounded by eight (see e.g. Zheng et al. (2009)).

Using a general framework first stated in Mahajan et al. (2006), if k operations can be applied on s to obtain t , then we can partition s and t into $\ell \in O(k)$ substrings, hereafter called *blocks*, such that the blocks of t are a per-

mutation of the blocks of s . It turns out that the problem of decomposing s and t (possibly with letter duplications) into a minimum set of common blocks was studied in another context called ortholog assignment (Chen et al., 2005). The problem was formally coined as Minimum Common String Partition (MCSP), which received quite some algorithmic attention due to its NP-hardness status (Goldstein et al., 2005; Chrobak et al., 2005). The best known approximation (to minimize the number of blocks) has a factor of $O(\log n)$ (Ganczorz et al., 2018). For these reasons, the fixed-parameter tractability of MCSP was considered in 2008 (Damaschke, 2008). For d -occurrence strings, the problem was first shown to be FPT in parameter $\ell + d$ in (Jiang et al., 2012b; Bulteau et al., 2013a), where ℓ is the solution size. In 2014, it was remarkably shown that MCSP is FPT when parameterized only by the solution size ℓ (Bulteau and Komusiewicz, 2014). Also note that common string partition problems with minimum length requirements on the blocks also received attention recently (Wang and Zhu (2010); Jiang et al. (2012a); Bulteau et al. (2013b); Lafond et al. (2022a)).

Although MCSP has been used to approximate various string distances, the applications of MCSP to compute *exact* distances is not always clear. When trying to solve, for instance, the transposition distance problem using MCSP, different string partitions with the same number of blocks lead to different transposition distances. It would seem necessary to enumerate all MCSP solutions to compute our string distances exactly. In fact, we show that restricting an MCSP instance to find blocks ordered according to a set of possible permutations Π is sufficient.

Our results. In this paper, we thus first establish a precise relationship between MCSP and string distance problems in terms of fixed-parameter tractability. We propose a string operation that generalizes all the ones mentioned above. We then define a variant of MCSP where the number ℓ of desired blocks is known, but the matched substrings in the partitions must satisfy a given block permutation π . We also allow the matched substrings to be affected by specified simple string functions, such as reversal or sign flips (allowed functions are defined precisely later on).

We call this variant of MCSP the Permutation-constrained Common String Partition problem (PCSP for short). We then show that deciding whether, under our general string operations, the distance between two strings s and t is at most k can be achieved by making a bounded number of calls to an algorithm that solves PCSP on the same input strings. The specified number of blocks ℓ in these calls is $O(k)$ for a fixed set of allowed operations, and the number of calls depends on the allowed operations and can be exponential in k , but not in the length of the strings. In particular, if PCSP is FPT in parameter $\ell + d$, then computing our distances is FPT in $k + d$.

We then show that PCSP is indeed FPT in $\ell + d$ if the allowed operations can reverse and/or flip the sign of permuted blocks, making the result applicable to several combinations of allowed operations. The recent results by Bulteau et al. imply that PCSP, parameterized by the number of blocks ℓ only, is W[1]-hard (Bulteau et al., 2021a), thereby motivating the need for

two parameters ℓ and d . Our algorithmic framework is strongly inspired by the work of Bulteau et al. (2013a), who showed that MCSP is FPT in $\ell + d$. Their algorithm uses a *match graph* that depicts which characters could be corresponding in a block partition. A branching strategy finds exactly one match per block in a solution. Our approach relies on a similar strategy, but requires non-trivial extensions to the framework. First, we want to find a block partition that uses a specific permutation π , and so our branching algorithm must maintain a set of matched substrings that always agrees with π . Moreover, we cannot assume that we want to find a partition of s and t that minimize the number of blocks as in MCSP, since π specifies the number of desired blocks (or, said more precisely, even though a minimum common string partition *could* be a solution to PCSP since we allow empty blocks, the set of minimum common string partitions may all fail to satisfy π). This prevents us from using structural results on optimal solutions (see e.g. Lemma 2 in Bulteau et al. (2013a)), which in turn requires a deeper case analysis. Finally, in our general framework, some matched substrings in a solution might be the reverse of one another. We develop novel machinery in the match graph to ensure that characters are matched consistently, i.e. that our branching matches all characters in a common substring in a forward or reverse manner, and not both.

As a consequence of our results, we show that several string distances are FPT in $k + d$ since they reduce to PCSP. The main results that can be derived from this PCSP reduction is that for strings s and t in which each symbol occurs at most d times, and with a pair of strings at distance at most k ,

- the transposition distance can be computed in time $k^{O(k)} d^{6k+2} n$.
- the block interchange distance can be computed in time $k^{O(k)} d^{8k+2} n$.
- the flip, reversal and unsigned reversal distances can be computed in time $k^{O(k)} d^{4k+2} n$.
- the p -cut distance can be computed in time $(pk)^{O(pk)} d^{2k(p-1)+2} n$.
- for any subset $\mathcal{P}$ of operations among transpositions, block interchanges, reversals, unsigned reversals or flips, computing $d_{\mathcal{P}}(s, t)$ can be done in time $k^{O(k)} d^{8k+2} n$,

where n is the length of the input strings.

This paper is organized as follows. In Section 2 we give necessary definitions. In Section 3 we present the definition and properties of the aforementioned simple string functions and the reduction to PCSP. In Section 4 we give the FPT algorithms for d -occurrence sequences (genomes). In Section 5 we conclude the paper. We comment that an extended abstract of this paper was presented as a conference paper (Lafond and Zhu, 2021), but due to space constraints, most of the proofs were omitted there.

2 Preliminary notions

For $\ell \in \mathbb{N}$, we denote $[\ell] = \{1, \dots, \ell\}$. If $X = (x_1, \dots, x_\ell)$ is an ordered sequence of arbitrary elements, we denote $\text{set}(X) = \{x_i : i \in [\ell]\}$. The empty

string is denoted λ . For a string s , we write $|s|$ for its length and $s[i]$ for its i -th character. For two integers a and b , we denote by $s[a, b]$ the substring of s containing positions a to b , inclusively. If $b < a$, then $s[a, b] = \lambda$. The concatenation of strings s_1 and s_2 is denoted s_1s_2 . For clarity, a string s may be denoted $s = \langle a_1, a_2, \dots, a_n \rangle$, where each a_i is an individual symbol. We denote a partition of a string s by $s_1|s_2|\dots|s_\ell$, where s is equal to the concatenation $s_1s_2\dots s_\ell$. We allow any of the s_i 's to be the empty string, which will have implications later on. Each s_i substring is called a *block* of the partition. Moreover, for each non-empty s_i , the position of the last character of s_i in s is called a *breakpoint*. Observe that position n is always a breakpoint.

A *marker* is a specific character of a string s . Each character $s[i]$ is a distinct marker, even if other positions contain the same symbol. For concreteness, one may think of a marker as a character $s[i]$ accompanied with a unique identifier. Two markers are *adjacent* (in s) if they are at consecutive positions in s , in either order. If a marker x of s is positioned before another marker y of s , we write $x \prec_s y$, or just $x \prec y$ if s is clear from the context. We also write $x \preceq_s y$ (or $x \preceq y$) if $x = y$ is possible. For a marker x and an integer i , we write $x +_s i$ to denote the marker located i positions to the right of x in s , and $x -_s i$ for the marker i positions to the left of x in s (if $x +_s i$ or $x -_s i$ do not exist because they go beyond the bounds of their sequence, then they are left undefined). In most cases, s will be clear from the context and we will write $x + i$ and $x - i$. For markers $x \preceq y$, we write $s[x, y]$ for the substring that starts at x and ends at y , inclusively (we will ensure that no confusion occurs with the notation $s[a, b]$ with a, b integers).

A *string function* is a function $h : \Sigma^* \rightarrow \Sigma^*$ that maps strings to strings. We require that $h(\lambda) = \lambda$. We say that h is *length-preserving* if $|h(s)| = |s|$ for any string s . Unless stated otherwise, we shall assume that $h(s)$ can be computed in time $O(|s|)$, in which case we say that h is *linear-time computable*. The identity function is denoted id and satisfies $id(s) = s$ for all s . Other functions of interest include those that apply on an alphabet in which each symbol x has a negation $-x$, and $-(-x) = x$ (which includes the possibility of fixed-points, i.e. $x = -x$):

- the *flip* function negates every symbol of s ;
 $flip(\langle a_1, \dots, a_n \rangle) = \langle -a_1, \dots, -a_n \rangle$;
- the *rev* function flips and reverses each symbol;
 $rev(\langle a_1, \dots, a_n \rangle) = \langle -a_n, \dots, -a_1 \rangle$;
- the *urev* function reverses the symbols;
 $urev(\langle a_1, \dots, a_n \rangle) = \langle a_n, \dots, a_1 \rangle$.

In the general Permutation-constrained Common String Partition (PCSP), we receive strings s and t , an integer ℓ , a permutation π of $[\ell]$ and string functions $f_1, \dots, f_\ell$. We must partition s and t in a way that the i -th block of t is equal to the $\pi(i)$ -th block of s after applying the f_i function to that block.

The Permutation-constrained Common String Partition (PCSP) problem

Input. Strings s and t of equal length, integer ℓ , permutation π of $[\ell]$, and a sequence of string functions $F = (f_1, f_2, \dots, f_\ell)$.

Question. Does there exist partitions $s_1 | \dots | s_\ell$ and $t_1 | \dots | t_\ell$ of s and t , respectively, such that $t_i = f_i(s_{\pi(i)})$ for each $i \in [\ell]$?

$$\ell = 5$$

$$f_1 = id, f_2 = rev, f_3 = rev, f_4 = rev, f_5 = id$$

$$\pi(1) = 1, \pi(2) = 4, \pi(3) = 3, \pi(4) = 5, \pi(5) = 2$$

$$s : a^+ b^+ c^+ d^+ a^- d^+ b^+ c^-$$

$$t : a^+ b^+ a^+ d^- c^+ b^- d^- c^+$$

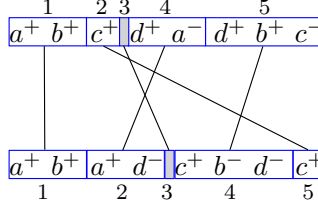


Fig. 1 An example of a PCSP instance, with a solution that satisfies $t_i = f_i(s_{\pi(i)})$ for each $i \in [5]$ on the right (the numbers correspond to the i subscripts). For example, $t_2 = rev(s_4)$ because $f_2 = rev$ and $\pi(2) = 4$. Note that s_3 and t_3 are empty strings.

Note that in terms of notation, we will usually reserve $f_1, \dots, f_\ell$ to denote string functions that are used to transform the blocks of s to those of t , and use the letter h for string functions with other purposes. An example of PCSP is shown in Figure 1. The ℓ parameter therefore refers to the desired number of substrings in our partitions. Recall that in the introduction, we used parameter k as the distance between two genomes. The exact relationship between ℓ and k depends on the specific distance of interest, as we shall detail later in the paper. Notice that in the classical MCSP problem, π and the f_i functions are not present (i.e. each f_i is the identity). The incorporation of string functions allows blocks to be affected by operations such as flips or reversals. As we mentioned, PCSP is W[1]-hard in parameter ℓ even if each f_i is the identity (Bulteau et al., 2021a).

Generic string operations

One of our goals is to establish a relationship between PCSP and string distances based on certain types of operations. We focus on operations that, given a string s , can (1) partition the string; (2) apply a string function on each block; and (3) apply a permutation on the blocks from a given list. Note that a similar string operation was proposed in Bulteau et al. (2021b), but without string functions and permutation list.

To define this more precisely, let p be an integer, let $H = (h_1, \dots, h_p)$ be a sequence of p string functions, and let Π be a set of permutations of $[p]$. We call the triple (p, H, Π) an *operation*. Given a string s , *applying* (p, H, Π) on s modifies the string by applying the following steps:

- partition s into a set of p substrings $s_1|s_2|\dots|s_p$ (some of which may be λ);
- for each $i \in [p]$, let $s'_i = h_i(s_i)$;
- choose $\pi \in \Pi$ and return the string $s' = s'_{\pi(1)}s'_{\pi(2)}\dots s'_{\pi(p)}$.

More succinctly, we can write $s' = h_{\pi(1)}(s'_{\pi(1)})\dots h_{\pi(p)}(s'_{\pi(p)})$. Intuitively, the $f_1, \dots, f_\ell$ functions from the PCSP problem definition will correspond to the composition of H functions that affected each relevant substring in a sequence of operations.

Let $\mathcal{P}$ be a set of operations, which can be thought of as a set of moves that are permitted. If, for each $(p', H, \Pi) \in \mathcal{P}$, we have $p' \leq p$, then we call $\mathcal{P}$ a set of p -operations. Let us write $s \xrightarrow{\mathcal{P}} t$ if there exists $(p, H, \Pi) \in \mathcal{P}$ that can be applied on s to obtain t . Also write $s \xrightarrow{\mathcal{P}, k} t$ if there exist strings $s_2, \dots, s_{k-1}$ such that $s \xrightarrow{\mathcal{P}} s_2 \xrightarrow{\mathcal{P}} \dots \xrightarrow{\mathcal{P}} s_{k-1} \xrightarrow{\mathcal{P}} t$. Given strings s and t , we denote by $d_{\mathcal{P}}(s, t)$ the minimum value of k that satisfies $s \xrightarrow{\mathcal{P}, k} t$.

Relationship with known string operations

The (p, H, Π) -operations generalize several well-known string operations. For instance, a *transposition* takes two consecutive substrings and exchanges them. This takes the form $s_1|s_2|s_3|s_4 \rightarrow s_1|s_3|s_2|s_4$, which can be modeled by putting $p = 4, H = \{id\}^4$, and Π only allowing to exchange blocks 2 and 3 (recall that s_1 and s_4 can be empty, allowing to move the ends of the strings). Similarly, a *block interchange* swaps two arbitrary substrings. This takes the form $s_1|s_2|s_3|s_4|s_5 \rightarrow s_1|s_4|s_3|s_2|s_5$, which takes $p = 5, H = \{id\}^5$ and Π allowing to swap blocks 2 and 4. The p -cut defined in Bulteau et al. (2021b), which allows cutting into p pieces and permuting them in any way, can also be modeled similarly.

For examples that do not use the identity function, the *reversal* operation, takes the form $s_1|s_2|s_3 \rightarrow s_1|rev(s_2)|s_3$, hence $p = 3, H = (id, rev, id)$ and Π only allowing the identity permutation. As another example, the *tandem duplication* replaces a substring $s[a, b]$ by $s[a, b]s[a, b]$. We can apply $s_1|s_2|s_3 \rightarrow s_1|s_2s_2|s_3$, with $p = 3$ and $h_2(s_2) = s_2s_2$ (and $h_1 = h_3 = id$). Note however that h_2 is not length-preserving and will not be supported in the results that follow.

Also note that our model allows several possible operations in a single move. It is possible that several mutations occur in parallel across independent locations on a genome. For instance, we may want to allow two (or more) reversals to occur at once on non-overlapping substrings, i.e. $s_1|s_2|s_3|s_4|s_5 \rightarrow s_1|rev(s_2)|s_3|rev(s_4)|s_5$.

Operations that cannot be modeled here include those that require some dependency between the substrings of the partition. For instance, in the general duplication operation, we take an arbitrary substring $s[a, b]$ and copy-paste it elsewhere. This could take the form $s_1|s_2|s_3|s_4|s_5 \rightarrow s_1|s_2|s_3|s_4s_2|s_5$, but our H functions do not allow h_2 and h_4 to “communicate” the contents of s_2 .

3 Reduction to PCSP for simple string functions

We will show that if the H functions allowed in the operations are “simple”, computing $d_{\mathcal{P}}(s, t)$ reduces to PCSP. We will need the ability to list all strings at distance at most k from s .

Lemma 1 *Let $\mathcal{P}$ be a set of p -operations that only use length-preserving string functions that are linear-time computable, let $k \in \mathbb{N}$, and let s be a string of length n . Define $N := (|\mathcal{P}|p)^k(n+p-1)^{k(p-1)}$. Then in time $O(nN)$, one can obtain a list R of size $O(N)$ such that t is in R if and only if $|t| = n$ and $d_{\mathcal{P}}(s, t) \leq k$.*

Proof We can simply brute-force recursively on every sequence of at most k operations on s that we can apply, and store each string encountered in a list (with possible duplicates). When listing the possible operations, there are $|\mathcal{P}|$ to choose from, then $\binom{n+p-1}{p-1} = \frac{(n+p-1)!}{n!(p-1)!}$ possible partitions to choose from, since empty substrings are allowed (note that the string changes after we apply some operations on it, but not its length — therefore we can use n here). Once the operation from $\mathcal{P}$ is chosen, there is only one way to apply the H functions. There are then up to $p!$ permutations to choose from. Thus each step has

$$|\mathcal{P}| \frac{(n+p-1)!}{n!(p-1)!} p! = |\mathcal{P}| \frac{(n+p-1)!}{n!} p = |\mathcal{P}| p(n+p-1)(n+p-2) \dots (n+1)$$

ways to modify a string, which we bound by $O(|\mathcal{P}|p(n+p-1)^{p-1})$. This yields a recursion tree of depth at most k and degree at most $O(|\mathcal{P}|p(n+p-1)^{p-1})$, and thus the recursion tree has at most $O((|\mathcal{P}|p)^k(n+p-1)^{k(p-1)}) = O(N)$ nodes, thereby bounding the size of the list. Moreover, it can be seen that the processing time in each node of the recursion tree is $O(n)$, to compute the result of the string function and permutation that yields its input. Note that a node can spend more than linear time to generate the inputs to all the recursive calls that it makes — however the (linear) processing time needed to generate each input passed to the recursive calls can be charged to the edges of the recursion tree. Hence the time to obtain the list is $O(nN)$. $\square$

Note that the list R above may contain duplicates. Later on, we will iterate over the entries of R , so in practice it may be more efficient to remove such duplicates, although this would add to the complexity without improving our theoretical guarantees. If needed, such duplicates could be removed in time $O(nN)$ with high probability by inserting each entry in a hash table, with each insertion costing $O(n)$ for hash computations. For our purposes, what matters is that each string at distance at most k is present in R , and that only such strings are present.

We can now define our notion of simplicity.

Definition 1 A string function h is *simple* if h is length-preserving and either:

- $h(s_1 s_2) = h(s_1) h(s_2)$ for any string s and any partition $s_1 | s_2$ of s ; or

- $h(s_1s_2) = h(s_2)h(s_1)$ for any string s and any partition $s_1|s_2$ of s .

Clearly, *id*, *flip*, *rev* and *urev* are simple. Note that if $s_1| \dots |s_\ell$ is a partition of s , then $h(s) = h(s_1) \dots h(s_\ell)$ or $h(s) = h(s_\ell) \dots h(s_1)$. In particular, if $s = \langle x_1, x_2, \dots, x_n \rangle$, then $h(s) \in \{\langle h(x_1), \dots, h(x_n) \rangle, \langle h(x_n), \dots, h(x_1) \rangle\}$, i.e. the function is applied to each character individually, and we decide to reverse or not. String functions with the $h(s_1s_2) = h(s_1)h(s_2)$ or $h(s_1s_2) = h(s_2)h(s_1)$ property are sometimes called *morphisms* and *anti-morphisms*, respectively, although length-preservation is not a requirement, see e.g. (Lothaire, 1997).

Since they can be applied on single symbols, simple functions can be represented as a lookup table that maps each symbol of Σ to another symbol, plus one bit that distinguishes whether to reverse or not. Moreover, assuming a total order on Σ , such a lookup table can be represented as a string of length $|\Sigma|$ (plus the reversal bit), where the i -th symbol of the string indicates where the function maps the i -th symbol of the alphabet. In what follows, we shall assume that each simple string function is given as a string of size $|\Sigma| + 1$, where the last symbol is reserved for whether the string is reversed or not.

Definition 2 Let $\mathbb{H}$ be a set of simple string functions. We say that an operation (p, H, Π) is $\mathbb{H}$ -restricted if $\text{set}(H) \subseteq \mathbb{H}$.

Let $\mathbb{H} = \{h_1, \dots, h_r\}$ be a set of simple string functions. By $\mathbb{H}\langle k \rangle$, we mean the set of functions obtained by composing any k functions or fewer from $\mathbb{H}$:

$$\mathbb{H}\langle k \rangle = \{id\} \cup \bigcup_{k'=1}^k \{h_{a_{k'}} \circ h_{a_{k'-1}} \circ \dots \circ h_{a_1} : a_1, \dots, a_{k'} \in [r]\}.$$

By convention, put $\mathbb{H}\langle 0 \rangle = \{id\}$. By $\mathbb{H}\langle k \rangle^\ell$, we mean the set of vectors of ℓ functions from $\mathbb{H}\langle k \rangle$. Before proceeding, we must show that $\mathbb{H}\langle k \rangle$ functions are simple, and that their corresponding strings can be computed easily.

Lemma 2 Let $\mathbb{H}$ be a set of linear-time computable and simple functions on alphabet Σ . Then $\mathbb{H}\langle k \rangle$ can be computed in time $O(|\Sigma| \log |\mathbb{H}\langle k \rangle| \cdot |\mathbb{H}| |\mathbb{H}\langle k-1 \rangle|)$. Moreover, any $h \in \mathbb{H}\langle k \rangle$ is simple and, for any string s , $h(s)$ can be computed in time $O(|s|)$.

Proof Let us first argue inductively that any function in $\mathbb{H}\langle i \rangle$ for $i = 0, \dots, k$ is simple. For $i = 0$, $\mathbb{H}\langle 0 \rangle = \{id\}$ and this is obvious. Assume $0 < i \leq k$ and that the statement holds for $\mathbb{H}\langle i-1 \rangle$. Then any function in $\mathbb{H}\langle i \rangle$ is obtained from $h \circ h'$, where $h \in \mathbb{H}$ and $h' \in \mathbb{H}\langle i-1 \rangle$. Note that h is simple by assumption and $h' \in \mathbb{H}\langle i-1 \rangle$ is simple by induction. It is not hard to see that composing two simple functions $h \circ h'$ yields a simple function. We argue this here for the completeness of the argument. Consider a string s and any partition $s_1|s_2$ of s into two parts. We get four cases, depending on which of h and/or h' reverses

its input string or not:

$$\begin{aligned}
\text{None reverses: } h \circ h'(s) &= h(h'(s_1 s_2)) = h(h'(s_1)h'(s_2)) = h(h'(s_1))h(h'(s_2)) \\
h' \text{ reverses: } h \circ h'(s) &= h(h'(s_1 s_2)) = h(h'(s_2)h'(s_1)) = h(h'(s_2))h(h'(s_1)) \\
h \text{ reverses: } h \circ h'(s) &= h(h'(s_1 s_2)) = h(h'(s_1)h'(s_2)) = h(h'(s_2))h(h'(s_1)) \\
\text{Both reverse: } h \circ h'(s) &= h(h'(s_1 s_2)) = h(h'(s_2)h'(s_1)) = h(h'(s_1))h(h'(s_2))
\end{aligned}$$

In each case, using the simplicity of h and h' , we get that $h \circ h'$ satisfies the desired properties of simple functions. Thus $h \circ h' \in \mathbb{H}\langle i \rangle$ is simple.

Now for the algorithm, recall that we represent each $h' \in \mathbb{H}\langle k \rangle$ as a string of length $O(|\Sigma|)$. Moreover for $h \in \mathbb{H}$, the string corresponding to $h \circ h'$ can then easily be computed in time $O(|\Sigma|)$. Consider the algorithm that starts with known $\mathbb{H}\langle 0 \rangle$ and $\mathbb{H}\langle 1 \rangle$, and then for each $i = 2, \dots, k$ in order, computes $\mathbb{H}\langle i \rangle$ from $\mathbb{H}\langle i-1 \rangle$. To do this, notice that it is sufficient to start from $\mathbb{H}\langle i-1 \rangle$, and add the new functions that are found in the set $\{h \circ h' : h \in \mathbb{H}, h' \in \mathbb{H}\langle i-1 \rangle \setminus \mathbb{H}\langle i-2 \rangle\}$. Therefore, for each $i = 2, \dots, k$, our algorithm will assume knowledge of both $\mathbb{H}\langle i-1 \rangle$ and $\mathbb{H}\langle i-1 \rangle \setminus \mathbb{H}\langle i-2 \rangle$, and compute both $\mathbb{H}\langle i \rangle$ and $\mathbb{H}\langle i \rangle \setminus \mathbb{H}\langle i-1 \rangle$. We simply store these sets of strings in balanced binary trees. Assuming a total order on Σ , inserting and searching strings of length $O(|\Sigma|)$ in a set with m elements can be done with $O(\log m)$ comparisons, each taking time $O(|\Sigma|)$. We thus assume complexity of $O(|\Sigma| \log m)$ for search and insertion operations. Note that we could use a hashmap instead to obtain complexities $O(|\Sigma|)$, but only with high probability.

Notice that initially for $i = 2$, the sets for $\mathbb{H}\langle 1 \rangle$ and $\mathbb{H}\langle 1 \rangle \setminus \mathbb{H}\langle 0 \rangle$ can be obtained in time $O(|\mathbb{H}| \cdot |\Sigma| \log |\mathbb{H}|)$.

For $i \geq 2$, we calculate all the $h \circ h'$ functions with $h \in \mathbb{H}, h' \in \mathbb{H}\langle i-1 \rangle \setminus \mathbb{H}\langle i-2 \rangle$. This requires computing $h(h'(\sigma))$ for each $\sigma \in \Sigma$ in order to yield the string corresponding to $h \circ h'$, which takes time $O(|\Sigma|)$. For each such function $h \circ h'$, we check whether it is already in the set $\mathbb{H}\langle i-1 \rangle$ in time $O(|\Sigma| \log |\mathbb{H}\langle i-1 \rangle|) = O(|\Sigma| \log |\mathbb{H}\langle k \rangle|)$. We insert each new $h \circ h'$ into $\mathbb{H}\langle i-1 \rangle$, and so it becomes $\mathbb{H}\langle i \rangle$ for the next iteration. Moreover, each new $h \circ h'$ is inserted into a new set, thus forming $\mathbb{H}\langle i \rangle \setminus \mathbb{H}\langle i-1 \rangle$. Overall, the i -th iteration requires performing $O(|\mathbb{H}|(|\mathbb{H}\langle i-1 \rangle \setminus \mathbb{H}\langle i-2 \rangle|))$ searches and insertions, each in time $O(|\Sigma| \log |\mathbb{H}\langle k \rangle|)$, for a total time of $O(|\Sigma| \log |\mathbb{H}\langle k \rangle| |\mathbb{H}|(|\mathbb{H}\langle i-1 \rangle \setminus \mathbb{H}\langle i-2 \rangle|))$.

The total time taken throughout the k iterations can be bounded, up to constant factors, by

$$\begin{aligned}
& |\mathbb{H}| \cdot |\Sigma| \log |\mathbb{H}| + \sum_{i=2}^k |\mathbb{H}| |\Sigma| \log |\mathbb{H}\langle k \rangle| (|\mathbb{H}\langle i-1 \rangle \setminus \mathbb{H}\langle i-2 \rangle|) \\
&= |\mathbb{H}| \cdot |\Sigma| \log |\mathbb{H}| + |\mathbb{H}| |\Sigma| \log |\mathbb{H}\langle k \rangle| \sum_{i=2}^k |\mathbb{H}\langle i-1 \rangle \setminus \mathbb{H}\langle i-2 \rangle| \\
&\leq |\mathbb{H}| \cdot |\Sigma| \log |\mathbb{H}| + |\mathbb{H}| |\Sigma| \log |\mathbb{H}\langle k \rangle| |\mathbb{H}\langle k-1 \rangle|
\end{aligned}$$

which justifies the claimed complexity. The final set storing $\mathbb{H}\langle k \rangle$ can be used to output all the string for the desired functions in time $O(|\Sigma||\mathbb{H}\langle k \rangle|) = O(|\Sigma||\mathbb{H}\langle k-1 \rangle|)$. Finally, since this computation gives access to the string of each $h \in \mathbb{H}\langle k \rangle$, it is clear that $h(s)$ can be computed in time $O(|s|)$. $\square$

We can now show that distances that use any combination of operations based on simple string functions reduce to PCSP. Beforehand, we need tools to represent permuted matches between a common partition in a succinct manner. We define a special alphabet for this purpose. Let $\mathcal{F}$ be the set of all simple string functions (which includes id). We will treat the set $\Sigma_{\mathcal{F}} = \mathbb{N} \times \mathcal{F}$ as a special alphabet, each symbol consisting of a pair with an integer and a function. For any $f \in \mathcal{F}$ and any $(i, h) \in \Sigma_{\mathcal{F}}$, we define $f((i, h)) = (i, f \circ h)$ (noting that $(i, f \circ h) \in \mathbb{N} \times \mathcal{F}$ since $f \circ h$ is simple by Lemma 2).

Computing distances based on simple string functions reduce to finding partitions on this alphabet. Although the next lemma requires some tedious detail handling, it has a clear intuition. If $d_{\mathcal{P}}(s, t) \leq k$, then each of the k operations creates at most $p-1$ breakpoints, and there are at most $1 + k(p-1)$ of them (counting the last position which is always a breakpoint). These breakpoints define a partition of s and t into $\ell \leq 1 + k(p-1)$ corresponding blocks, and if this partition into ℓ substrings is known, we could replace each i -th block by the symbol (i, id) in s , and by $(\pi(i), f_i)$ in t . Figure 2 illustrates this intuition.

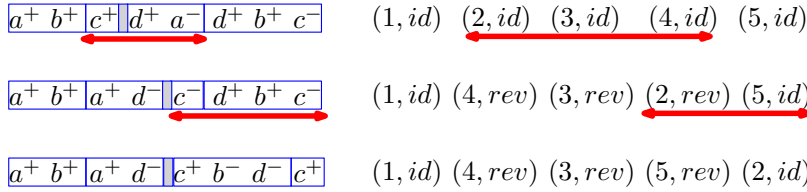


Fig. 2 An illustration of Lemma 3 in the case that $\mathbb{H} = \{rev\}$. Here s can be transformed into t using two reversals. Given that the solution and its blocks are known, block i can be replaced by (i, id) and, because the functions are simple, the same sequence of operations can be applied to the single symbols instead of the whole blocks.

Lemma 3 *Let $\mathbb{H}$ be a set of simple string functions and let $\mathcal{P}$ be a set of $\mathbb{H}$ -restricted p -operations.*

Then $d_{\mathcal{P}}(s, t) \leq k$ if and only if there exists a partition $s_1 | \dots | s_{\ell}$ of s , a partition $t_1 | \dots | t_{\ell}$ of t , a permutation π of $[\ell]$, and $f_1, \dots, f_{\ell} \in \mathbb{H}\langle k \rangle$ such that all the following conditions hold:

1. $\ell = 1 + k(p-1)$;
2. $t_i = f_i(s_{\pi(i)})$ for each $i \in [\ell]$;
3. $d_{\mathcal{P}}(\langle (1, id), (2, id), \dots, (\ell, id) \rangle, \langle (\pi(1), f_1), (\pi(2), f_2), \dots, (\pi(\ell), f_{\ell}) \rangle) \leq k$.

Proof ($\Rightarrow$) Assume that $d_{\mathcal{P}}(s, t) = k' \leq k$ for some k' (note that $k' = 0$ is possible). Let $D_1, \dots, D_{k'}$ be a sequence of operations transforming s into

t , and for $j \in [k']$, let s^j be the string obtained after applying the first j operations. Define $s^0 = s$ and note that $s^{k'} = t$.

We first argue that adjacent markers of s that are never separated by a breakpoint created by an operation remain adjacent in each s^j . Although this is usually obvious under string operations used in genomics such as transpositions or block interchanges, this requires care when dealing with string functions; in particular, markers change their symbols and are moved by the D_j 's, so one must keep track of the correspondence between markers. For each $j \in [k]$, we will use $s^j \langle i \rangle$ to refer to the marker at position i of s^j (to distinguish it from the symbol $s^j[i]$). We will think of each marker $s^j \langle i \rangle$ of s^j as changing symbol and moving from s^j to s^{j+1} , but remaining “the same” marker. One may think of assigning a unique id to each marker, and this id remains unchanged after each move.

To be a bit more specific, consider s^j for some $j \in \{0, 1, \dots, k' - 1\}$. The operation D_{j+1} first chooses a partition $s_1^j | \dots | s_{p'}^j$, then applies some function $h_i(s_i^j)$ to each substring s_i^j , and then applies a permutation π . This results in the string

$$s^{j+1} = h_{\pi(1)}(s_{\pi(1)}^j) h_{\pi(2)}(s_{\pi(2)}^j) \dots h_{\pi(p')}(s_{\pi(p')}^j)$$

Consider some substring s_i^j and its marker $s_i^j \langle q \rangle$ at position q . If h_i does not reverse its input, then the corresponding marker in s^{j+1} is the one at position q of the $h_i(s_i^j)$ substring in s^{j+1} . If h_i reverses its input, then the corresponding marker is the one at position $|s_i^j| - q + 1$ of the $h_i(s_i^j)$ substring in s^{j+1} . For a marker x of s^j , we write $x \rightsquigarrow y$ if y is the unique marker of s^{j+1} corresponding to x as described above.

For a marker x of s and a marker x' of s^j , $j \in \{1, \dots, k'\}$, we say that x' is a descendant of x if there are markers $x_1, x_2, \dots, x_{j-1}$ such that $x \rightsquigarrow x_1 \rightsquigarrow \dots \rightsquigarrow x_{j-1} \rightsquigarrow x'$. Note that each marker of s has a unique descendant in each s^j . Consider two adjacent markers x and y of s , with x occurring before y , and let $j \in [k']$. We say that x is *separated from y in s^j* if, denoting by x' and y' the respective descendants of x and y in s^j , either

- x' occurs before y' in s^j , and the partition used by operation D_j has a breakpoint at the position containing x' ; or
- y' occurs before x' in s^j , and the partition used by operation D_j has a breakpoint at the position containing y' .

Note that x and y can be separated in multiple s^j 's, but only the fact that at least one separation exists is relevant. Next, consider the partition $Q_0 = s_1 | s_2 | \dots | s_{\ell'}$ of s such that there is a breakpoint at position i if and only if there exists $j \in [k']$ such that the marker $s \langle i \rangle$ is separated from $s \langle i + 1 \rangle$ in s^j .

We claim that $\ell' \leq 1 + k(p - 1)$. To see this, observe that at most $p - 1$ pairs of markers are separated in s^j , for each $j \in [k']$, because the partition used by operation D_j has at most $p - 1$ breakpoints aside from the one at position n , which does not separate markers. Hence, at most $k(p - 1)$ pairs are separated in some s^j and Q_0 contains at most $1 + k(p - 1)$ breakpoints and no empty substrings, and therefore $\ell' \leq 1 + k(p - 1)$.

Then, consider the partition $Q := s_1|s_2|\dots|s_{\ell'}|s_{\ell'+1}|\dots|s_\ell$ obtained from Q_0 by appending $\ell - \ell'$ empty strings $s_{\ell'+1}, \dots, s_\ell = \lambda$. We use the partition Q for the rest of the proof. Note that Condition 1 of the lemma is satisfied by Q .

We argue Conditions 2 and 3 by induction on j . More precisely, we argue that for each $j \in \{0, 1, \dots, k'\}$, there is a partition $\hat{s}_1|\dots|\hat{s}_\ell$ of s^j , a permutation π of $[\ell]$, and $h_1, \dots, h_\ell \in \mathbb{H}\langle j \rangle$ such that $\hat{s}_i = h_i(s_{\pi(i)})$ for each $i \in [\ell]$, and such that the distance between $\langle (1, id), \dots, (\ell, id) \rangle$ to $\langle (\pi(1), h_1), \dots, (\pi(\ell), h_\ell) \rangle$ is at most j .

As a base case, when $j = 0$, we can choose π as the identity, as well as each $h_1 = \dots = h_\ell = id$. The conditions clearly hold in this case.

Assume that $j > 0$. By induction, there is a partition $\hat{Q} := \hat{s}_1|\dots|\hat{s}_\ell$ of s^{j-1} , a permutation π , and functions $h_1, \dots, h_\ell \in \mathbb{H}\langle j-1 \rangle$ such that $\hat{s}_i = h_i(s_{\pi(i)})$ for each $i \in [\ell]$. Consider operation $D_j := (p', H', \Pi')$ and the partition

$$s^{j-1} = s_1^{j-1}|\dots|s_{p'}^{j-1}$$

of s^{j-1} that it uses. Furthermore, let $\pi' \in \Pi'$ be the permutation applied by the operation. Note that for each $i \in [\ell]$, $\hat{s}_i$ is a substring of one of $s_1^{j-1}, s_2^{j-1}, \dots, s_{p'}^{j-1}$ (otherwise there would be a breakpoint at a marker x of $\hat{s}_i$ adjacent to $x+1$ also in $\hat{s}_i$, making x and $x+1$ separated in s^j and contradicting our initial partition of s). This means that each s_i^{j-1} is the concatenation $\hat{s}_{a_i}\hat{s}_{a_i+1}\dots\hat{s}_{b_i}$ of substrings of $\hat{Q}$, for some sequence of consecutive indices $a_i, a_i+1, \dots, b_i$. Note that some of the $\hat{s}_{a_i+q}$ substrings may be empty (corresponding to the $\ell - \ell'$ empty substrings we initially appended). Also note that some s_i^{j-1} may itself be empty, in which case we define $b_i = a_i - 1$, so that the concatenation yields the empty string.

Denote $H' = (h'_1, \dots, h'_{p'})$, which is the list of functions used by D_j . Then after applying D_j , we get

$$s^j = h'_{\pi'(1)}(s_{\pi'(1)}^{j-1})h'_{\pi'(2)}(s_{\pi'(2)}^{j-1})\dots h'_{\pi'(p')}(s_{\pi'(p')}^{j-1})$$

Now let us focus on some substring $h'_{\pi'(i)}(s_{\pi'(i)}^{j-1})$, where $i \in [p']$. To ease notation slightly, denote $i' = \pi'(i)$. This is equal to

$$h'_{i'}(s_{i'}^{j-1}) = h'_{i'}(\hat{s}_{a_{i'}}\hat{s}_{a_{i'}+1}\dots\hat{s}_{b_{i'}})$$

which, depending on whether $h'_{i'}$ reverses its input or not, is equal to one of

$$\begin{aligned} &h'_{i'}(\hat{s}_{a_{i'}})h'_{i'}(\hat{s}_{a_{i'}+1})\dots h'_{i'}(\hat{s}_{b_{i'}}) \quad \text{or} \\ &h'_{i'}(\hat{s}_{b_{i'}})\dots h'_{i'}(\hat{s}_{a_{i'}+1})h'_{i'}(\hat{s}_{a_{i'}}) \end{aligned}$$

by simplicity. Note that this holds even if there are empty substrings in the above concatenations, since they remain empty under string functions. Since this is true for every s_i^{j-1} , we see that s^j is a concatenation of substrings of the form $h'_{c_i}(\hat{s}_i)$ for some $h'_{c_i} \in H', i \in [\ell]$. Recall that $\hat{s}_i = h_i(s_{\pi(i)})$ for each $i \in [\ell]$, so that $h'_{c_i}(\hat{s}_i) = h'_{c_i}(h_i(s_{\pi(i)}))$. This means that there is some

permutation π'' of $[\ell]$ and $h_1^*, \dots, h_\ell^* \in \{h'_{c_1} \circ h_1, \dots, h'_{c_\ell} \circ h_\ell\}$ such that s^j can be written as

$$s^j = h_1^*(s_{\pi''(1)}) h_2^*(s_{\pi''(2)}) \dots h_\ell^*(s_{\pi''(\ell)}).$$

which includes the possible empty strings that were affected by some h_i^* and relocated. We note that for each $i \in [\ell]$, there is some q such that $h_i^* = h'_{c_q} \circ h_q \in \mathbb{H}\langle j \rangle$. Thus s^j has the desired form.

We also know by induction that the distance between $\langle (1, id), \dots, (\ell, id) \rangle$ and $u^{j-1} := \langle (\pi(1), h_1), \dots, (\pi(\ell), h_\ell) \rangle$ is at most $j-1$. Let us mimic operation D_j on u^{j-1} . That is, define $z_1, \dots, z_{p'}$ as follows for each $i \in [p']$:

- if $s_i^{j-1} = \lambda$, then $z_i = \lambda$;
- otherwise, $z_i = \langle (\pi(a_i), h_{a_i}), (\pi(a_i+1), h_{a_i+1}), \dots, (\pi(b_i), h_{b_i}) \rangle$.

where we recall that $s_i^{j-1} = \hat{s}_{a_i} \dots \hat{s}_{b_i} = h_{a_i}(s_{\pi(a_i)}) \dots h_{b_i}(s_{\pi(b_i)})$. We apply operation D_j to u^{j-1} using the partition

$$u^{j-1} = z_1 | z_2 | \dots | z_{p'}$$

which turns it into

$$u^j := h'_{\pi'(1)}(z_{\pi'(1)}) h'_{\pi'(2)}(z_{\pi'(2)}) \dots h'_{\pi'(p')}(z_{\pi'(p')})$$

Notice that this is identical to s^j , but with each $h'_{i'}(s_{i'}^{j-1})$ substring replaced by $h'_{i'}(z_{i'})$ (recall that $i' = \pi'(i)$).

Let us focus on one $h'_{i'}(z_{i'})$ with $i \in [p']$. Then

$$h'_{i'}(z_{i'}) = h'_{i'}(\langle (\pi(a_{i'}), h_{a_{i'}}), \dots, (\pi(b_{i'}), h_{b_{i'}}) \rangle)$$

which, depending on whether $h'_{i'}$ reverses its input or not, is equal to either

$$\begin{aligned} & \langle h'_{i'}((\pi(a_{i'}), h_{a_{i'}})), \dots, h'_{i'}((\pi(b_{i'}), h_{b_{i'}})) \rangle \\ &= \langle (\pi(a_{i'}), h'_{i'} \circ h_{a_{i'}}), \dots, (\pi(b_{i'}), h'_{i'} \circ h_{b_{i'}}) \rangle \end{aligned}$$

or to

$$\begin{aligned} & \langle h'_{i'}((\pi(b_i), h_{b_i})), \dots, h'_{i'}((\pi(a_i), h_{a_i})) \rangle \\ &= \langle (\pi(b_i), h'_{i'} \circ h_{b_i}), \dots, (\pi(a_i), h'_{i'} \circ h_{a_i}) \rangle \end{aligned}$$

In either case, this substring is the same as $h'_{i'}(s_{i'}^{j-1})$, but with each $h'_{i'}(\hat{s}_{a_{i'}+q})$ replaced by $(\pi(i), h'_{i'} \circ h_{a_{i'}+q})$. Because this is true for every $i \in [p']$, and because the $h'_{\pi'(i)}(z_{\pi'(i)})$ substrings of u^j appear in the same order as the $h'_{i'}(s_{i'}^{j-1})$ substrings of s^j , we get that u^j can be obtained from s^j by replacing each $h_i^*(s_{\pi''(i)})$ substring by $(\pi''(i), h_i^*)$. It follows that

$$u^j = \langle (\pi''(1), h_1^*), (\pi''(2), h_2^*), \dots, (\pi''(\ell), h_\ell^*) \rangle$$

and that the distance from $\langle (1, id), \dots, (\ell, id) \rangle$ to this string is at most j . This completes the induction step.

To conclude the proof, note that the induction works up to $j = k'$. Since $s^{k'} = t$, the forward direction of the proof is complete. That is, we have shown that there exists a partition $t_1 | \dots | t_\ell$ of t , a permutation π of ℓ , and $f_1, \dots, f_\ell \in \mathbb{H}\langle k' \rangle \subseteq \mathbb{H}\langle k \rangle$ such that $t_i = f_i(s_{\pi(i)})$ for each $i \in [\ell]$, and such that the distance between $\langle (1, id), \dots, (\ell, id) \rangle$ and $\langle (\pi(1), f_1), \dots, (\pi(\ell), f_\ell) \rangle$ is at most $k' \leq k$. This satisfies Conditions 2 and 3 of the lemma.

($\Leftarrow$) Let $s_1 | \dots | s_\ell$ and $t_1 | \dots | t_\ell$ be partitions of s and t , respectively, and let π be a permutation and $f_1, \dots, f_\ell \in \mathbb{H}\langle k \rangle$ satisfying the three conditions of the statement. The reader may bear in mind that some of the s_i 's and t_i 's can be empty. Let us denote $u = \langle (1, id), (2, id), \dots, (\ell, id) \rangle$ and $v = \langle (\pi(1), f_1), (\pi(2), f_2), \dots, (\pi(\ell), f_\ell) \rangle$. Thus we have

$$d_{\mathcal{P}}(u, v) = k' \leq k.$$

The idea is simply to replace (i, id) by s_i in the sequence of transformations from u to v . Let $D_1, \dots, D_{k'}$ be a sequence of operations that transform u to v , and for $j \in [k']$ let u^j be the string obtained after applying $D_1, \dots, D_j$ to u , with $u^0 = u$.

Note that for each $j \in \{0, 1, \dots, k'\}$, there is some permutation $\hat{\pi}$ and functions $h_1, \dots, h_\ell \in \mathbb{H}\langle j \rangle$ such that

$$u^j = \langle (\hat{\pi}(1), h_1), (\hat{\pi}(2), h_2), \dots, (\hat{\pi}(\ell), h_\ell) \rangle$$

because simple functions affect individual characters and our special alphabet stores the composition of the functions applied in its second coordinate. We claim that for any such u^j , there is a sequence of j operations that transform s into

$$s^j = h_1(s_{\hat{\pi}(1)})h_2(s_{\hat{\pi}(2)}) \dots h_\ell(s_{\hat{\pi}(\ell)})$$

The proof is by induction on j . For $j = 0$, this is trivial, since we can take $\hat{\pi}$ as the identity and each $h_i = id$. Assume $j > 0$. Let $\hat{\pi}$ and $h_1, \dots, h_\ell \in \mathbb{H}\langle j-1 \rangle$ be such that

$$u^{j-1} = \langle (\hat{\pi}(1), h_1), (\hat{\pi}(2), h_2), \dots, (\hat{\pi}(\ell), h_\ell) \rangle.$$

By induction, we can write

$$s^{j-1} = h_1(s_{\hat{\pi}(1)})h_2(s_{\hat{\pi}(2)}) \dots h_\ell(s_{\hat{\pi}(\ell)})$$

Assume that the D_j operation partitions u^{j-1} into

$$u_1^{j-1} | u_2^{j-1} | \dots | u_{p'}^{j-1}$$

where each non-empty u_i^{j-1} is a substring of the form $\langle (\hat{\pi}(a_i), h_{a_i}), (\hat{\pi}(a_i+1), h_{a_i+1}), \dots, (\hat{\pi}(b_i), h_{b_i}) \rangle$. Let $H = (h'_1, \dots, h'_{p'})$ and π' be the list of functions and the permutation used by D_j , respectively. After applying D_j , u^{j-1} becomes

$$u^j := h'_{\pi'(1)}(u_{\pi'(1)}^{j-1})h'_{\pi'(2)}(u_{\pi'(2)}^{j-1}) \dots h'_{\pi'(p')}(u_{\pi'(p')}^{j-1})$$

If we focus on one non-empty substring $h'_{\pi'(i)}(u_{\pi'(i)}^{j-1})$, and denote $i' = \pi'(i)$, it is equal to

$$h'_{i'}(u_{i'}^{j-1}) = h'_{i'}(\langle (\hat{\pi}(a_{i'}), h_{a_{i'}}), \dots, (\hat{\pi}(b_{i'}), h_{b_{i'}}) \rangle)$$

which, depending on whether $h'_{i'}$ reverses its input or not, is equal to either

$$\begin{aligned} & \langle h'_{i'}(\langle (\hat{\pi}(a_{i'}), h_{a_{i'}}), \dots, (\hat{\pi}(b_{i'}), h_{b_{i'}}) \rangle) \rangle \\ &= \langle (\hat{\pi}(a_{i'}), h'_{i'} \circ h_{a_{i'}}), \dots, (\hat{\pi}(b_{i'}), h'_{i'} \circ h_{b_{i'}}) \rangle \end{aligned}$$

or to

$$\begin{aligned} & \langle h'_{i'}(\langle (\hat{\pi}(b_{i'}), h_{b_{i'}}), \dots, (\hat{\pi}(a_{i'}), h_{a_{i'}}) \rangle) \rangle \\ &= \langle (\hat{\pi}(b_{i'}), h'_{i'} \circ h_{b_{i'}}), \dots, (\hat{\pi}(a_{i'}), h'_{i'} \circ h_{a_{i'}}) \rangle \end{aligned}$$

Now let us apply D_j on s^{j-1} on the partition

$$s_1^{j-1} | s_2^{j-1} | \dots | s_{p'}^{j-1}$$

where, for each $u_i^{j-1} = \lambda$, we have $s_i^{j-1} = \lambda$, and for each non-empty u_i^{j-1} , we have $s_i^{j-1} = h_{a_i}(s_{\hat{\pi}(a_i)}) \dots h_{b_i}(s_{\hat{\pi}(b_i)})$. This results in s^j , which becomes

$$s^j = h'_{\pi'(1)}(s_{\pi'(1)}^{j-1}) h'_{\pi'(2)}(s_{\pi'(2)}^{j-1}) \dots h'_{\pi'(p')}(s_{\pi'(p')}^{j-1})$$

If we focus on a non-empty $h'_{\pi'(i)}(s_{\pi'(i)}^{j-1})$ and denote $i' = \pi'(i)$, it is equal to

$$h'_{i'}(s_{i'}^{j-1}) = h'_{i'}(h_{a_{i'}}(s_{\hat{\pi}(a_{i'})}) \dots h_{b_{i'}}(s_{\hat{\pi}(b_{i'})}))$$

which, depending on whether $h'_{i'}$ reverses its input or not, is equal to either

$$\begin{aligned} & h'_{i'}(h_{a_{i'}}(s_{\hat{\pi}(a_{i'})})) \dots h'_{i'}(h_{b_{i'}}(s_{\hat{\pi}(b_{i'})})) \text{ or to} \\ & h'_{i'}(h_{b_{i'}}(s_{\hat{\pi}(b_{i'})})) \dots h'_{i'}(h_{a_{i'}}(s_{\hat{\pi}(a_{i'})})) \end{aligned}$$

In either case, the result is identical to $h'_{i'}(u_{i'}^{j-1})$, but with every $(\hat{\pi}(a_{i'} + q), h'_{i'} \circ h_{a_{i'}+q})$ replaced with $h'_{i'}(h_{a_{i'}+q}(s_{\hat{\pi}(a_{i'})}))$.

To summarize, as in the previous direction, we see that the $h'_{c_i}(h_i(s_{\hat{\pi}(i)}))$ substrings of s^j appear in the same order as the $(\hat{\pi}(i), h'_{c_i} \circ h_i)$ characters of u^j . Therefore, there is a permutation π'' and functions $h_1^*, \dots, h_\ell^*$ such that

$$\begin{aligned} u^j &= \langle (\pi''(1), h_1^*), (\pi''(2), h_2^*), \dots, (\pi''(\ell), h_\ell^*) \rangle \\ s^j &= h_1^*(s_{\pi''(1)}) h_2^*(s_{\pi''(2)}) \dots h_\ell^*(s_{\pi''(\ell)}) \end{aligned}$$

Since the induction argument holds up to k' , and because $u^{k'} = \langle (\pi(1), f_1), \dots, \pi(\ell), f_\ell) \rangle$, there is a sequence of k' operations that transform s into

$$f_1(s_{\pi(1)}) f_2(s_{\pi(2)}) \dots f_\ell(s_{\pi(\ell)}) = t_1 t_2 \dots t_\ell = t$$

which concludes the proof. $\square$

Lemma 3 shows that the exact content of the matched blocks does not truly matter, as we may reduce to computing a distance between strings of length ℓ . This allows us to relate distances and PCSP using Algorithm 1. The idea is simply to start from $u = \langle (1, id), \dots, (\ell, id) \rangle$, get every string at distance at most k from it, and check if they correspond to a string that satisfies Lemma 3. Note that in this algorithm, the brute-force procedure from Lemma 1 is used on u , which is on alphabet $[\ell] \times \mathbb{H}\langle k \rangle$. The lemma also assumes that string functions are linear-time computable, although it must work on strings with symbols of the form (i, h) and transform them into the form $(i, h' \circ h)$ in $O(1)$ time. To achieve this, we assign a distinct identifier $\alpha(h)$ to each function $h \in \mathbb{H}\langle k \rangle$ and we pre-compute a composition table μ that, for $h \in \mathbb{H}$ and $h' \in \mathbb{H}\langle k \rangle$, gives access to the identifier $\alpha(h \circ h')$ quickly (if $h \circ h' \notin \mathbb{H}\langle k \rangle$, we do not store it). This is detailed in the proof of complexity of Algorithm 1.

Let us also note that the proof would work if we required the blocks of s and t to be non-empty, except that we would need to consider blocks of length *at most* ℓ (and thus we would add a loop for lengths from 1 up to ℓ). However, we need to allow PCSP to output empty blocks, as it is unknown how to solve PCSP if blocks are required to be non-empty. This is explained in more detail in the next section.

```

function computeDistance( $s, t, \mathcal{P}, k$ )
  Compute  $\mathbb{H}\langle k \rangle$  and assign a distinct integer  $\alpha(h)$  to each  $h \in \mathbb{H}\langle k \rangle$ 
  Compute  $\mu(\alpha(h), \alpha(h')) := \alpha(h \circ h')$  for  $h \in \mathbb{H}, h' \in \mathbb{H}\langle k \rangle$ 
  Let  $\ell = 1 + k(p - 1)$ 
  Let  $u = \langle (1, id), (2, id), \dots, (\ell, id) \rangle$ 
  Compute the list  $R$  such that  $v$  is in  $R$  if and only if  $d_{\mathcal{P}}(u, v) \leq k$ ,
    using  $\mu$  and Lemma 1
  foreach element  $\langle (i_1, f_1), (i_2, f_2), \dots, (i_\ell, f_\ell) \rangle$  of  $R$  do
    Let  $\pi$  be the permutation such that  $\pi(1) = i_1, \dots, \pi(\ell) = i_\ell$ 
    Let  $F = (f_1, \dots, f_\ell)$ 
    Compute PCSP on input  $(s, t, \ell, \pi, F)$ 
    if the answer is yes then return yes
  end
  return no

```

Algorithm 1: Main $d_{\mathcal{P}}$ algorithm

Theorem 1 *Let $\mathbb{H}$ be a set of simple string functions, and let $\mathcal{P}$ be a set of $\mathbb{H}$ -restricted p -operations. Assume that any PCSP instance (s, t, ℓ, π, F) satisfying $F \in \mathbb{H}\langle k \rangle^\ell$ can be solved in time $g(\ell, n)$. Then Algorithm 1 decides whether $d_{\mathcal{P}}(s, t) \leq k$ in time*

$$O(n|\mathbb{H}||\mathbb{H}\langle k \rangle|^3 + (|\mathcal{P}|p)^k(k(p-1) + p)^{k(p-1)} \cdot g(1 + k(p-1), n)).$$

Proof Consider Algorithm 1 and let us argue that it correctly solves the problem. The algorithm checks whether the three conditions of Lemma 3 can be satisfied, which is equivalent to checking whether $d_{\mathcal{P}}(u, v) \leq k$. Assume that this is the case. Then with $\ell = 1 + k(p-1)$, by Condition 3 of the lemma, there

must be $v = \langle (\pi(1), f_1), \dots, (\pi(\ell), f_\ell) \rangle$ such that $d_{\mathcal{P}}(u, v) \leq k$, and such that s, t can be partitioned so as to satisfy Condition 2 of the lemma. By Lemma 1, v is in R , so the algorithm will eventually consider it. Then, since Condition 2 can be satisfied, PCSP must return *yes* on input s, t, ℓ, π, F (unless Algorithm 1 returns *yes* before even getting to v , which is correct). We thus also correctly return *yes*. If $d_{\mathcal{P}}(u, v) > k$, then for each v of R , Condition 2 cannot be satisfied, and so PCSP must return *no*, in which case we also return *no*.

As for the complexity, we know by Lemma 2 that $\mathbb{H}\langle k \rangle$ can be computed in time $O(|\Sigma| \log |\mathbb{H}\langle k \rangle| \cdot |\mathbb{H}| |\mathbb{H}\langle k-1 \rangle|)$. Here, we may assume that Σ is the set of symbols used by s and t , or that can be obtained from those by composing at most k functions from $\mathbb{H}$. We will therefore assume that $|\Sigma| \leq n |\mathbb{H}\langle k \rangle|$, so that Lemma 2 takes time $O(n |\mathbb{H}\langle k \rangle| \log |\mathbb{H}\langle k \rangle| \cdot |\mathbb{H}| |\mathbb{H}\langle k-1 \rangle|)$. We crudely upper bound this by $O(n |\mathbb{H}| |\mathbb{H}\langle k \rangle|^3)$.

Next, define the *extended alphabet* $\Sigma' = [\ell] \times \mathbb{H}\langle k \rangle$, with which the brute-force algorithm from Lemma 1 must work. We compute μ as defined in the algorithm to guarantee linear-time computability of string functions over this alphabet. For $h \in \mathbb{H}\langle k \rangle$, we may simply assign $\alpha(h)$ as the order of enumeration of h in $\mathbb{H}\langle k \rangle$ as returned by Lemma 2. Then for each $h \in \mathbb{H}$ and $h' \in \mathbb{H}\langle k \rangle$, we compute the function string corresponding to $h \circ h'$ and scan every element of $\mathbb{H}\langle k \rangle$ to search for its identifier, if it exists. This takes time $O(|\Sigma| |\mathbb{H}\langle k \rangle|)$. If $h \circ h'$ does exist, we associate the pair $\alpha(h), \alpha(h')$ with $\alpha(h \circ h')$. This results in the desired composition table μ , which can be represented as a 2D table of dimension $|\mathbb{H}| |\mathbb{H}\langle k \rangle|$. Filling this table takes total time $O(|\mathbb{H}| |\mathbb{H}\langle k \rangle| |\Sigma| |\mathbb{H}\langle k \rangle|)$, which we can bound by $O(n |\mathbb{H}| |\mathbb{H}\langle k \rangle|^3)$.

Then the brute-force algorithm applies $\mathcal{P}$ operations on strings over the extended alphabet Σ' . It needs to calculate $h(\sigma)$ in linear time, where $\sigma = (i, h') \in \Sigma'$ (note that only operations in $\mathbb{H}$ are used at each step of the brute-force algorithm, since $\mathcal{P}$ is $\mathbb{H}$ -restricted). This in turn requires computing $h((i, h')) = (i, h \circ h')$ in constant time, which can be done using μ . This allows achieving the bound of Lemma 1. Notice that since the brute-force algorithm has recursion depth of at most k , it composes at most k functions on each character in the explored strings, ensuring that all enumerated strings only use functions in $\mathbb{H}\langle k \rangle$.

We have argued that the steps prior to obtaining R can be done in time $O(n |\mathbb{H}| |\mathbb{H}\langle k \rangle|^3)$. Then by Lemma 1, the time required to compute R from a string of length ℓ is bounded by $O(\ell (|\mathcal{P}|p)^k (\ell + p - 1)^{k(p-1)})$, which is executed only once by the algorithm. Since $\ell = 1 + k(p-1)$, we may replace $\ell + p - 1$ with $k(p-1) + p$ in this expression. Defining $N := (|\mathcal{P}|p)^k (k(p-1) + p)^{k(p-1)}$, the brute-forcing time is $O(\ell N)$ and, as stated in Lemma 1, the number of elements in R we iterate on is $O(N)$. Then, inside the *while* loop of Algorithm 1, recovering the permutation π and constructing F take time $O(\ell)$, and then we take time $g(\ell, n)$ to call PCSP. We assume that PCSP takes time $\Omega(\ell)$ (one way to see this is that $\ell \leq n$ should hold, and PCSP takes time at least n to

read its input). It follows that the time of Algorithm 1 is

$$\begin{aligned}
& O(n|\mathbb{H}||\mathbb{H}\langle k \rangle|^3 + \ell N + N \cdot g(\ell, n)) \\
&= O(n|\mathbb{H}||\mathbb{H}\langle k \rangle|^3 + N(\ell + g(\ell, n))) \\
&= O(n|\mathbb{H}||\mathbb{H}\langle k \rangle|^3 + N \cdot g(\ell, n)) \\
&= O(n|\mathbb{H}||\mathbb{H}\langle k \rangle|^3 + (|\mathcal{P}|p)^k(k(p-1) + p)^{k(p-1)}g(1 + k(p-1), n))
\end{aligned}$$

as desired. $\square$

4 FPT algorithms for d -occurrence PCSP with reversals

In this section, we show that if F only contains $id, flip, rev$ and $urev$, and if each symbol occurs at most d times in the input strings, then PCSP is FPT in parameter $d + \ell$. Since empty blocks are allowed, we do not have the obligation of satisfying the given permutation exactly, provided that the order of the non-empty blocks agrees with π .

For the rest of this section, we assume that we are given strings s and t , an integer ℓ , a permutation π , and functions $F = (f_1, \dots, f_\ell)$. We will say that the partitions $s = s_1 | \dots | s_\ell$ and $t = t_1 | \dots | t_\ell$ form a *solution* if $t_i = f_i(s_{\pi(i)})$ for each $i \in [\ell]$. In the rest of the section, for a marker x of s or t , we reserve the notations $x + i$ and $x - i$ with respect to s or t , that is, if x is in s (resp. in t), then $x + i = x +_s i$ (resp. $x +_t i$).

For a string w , a *numbered partition* of w is a sequence of ordered pairs $(w_{a_1}, a_1), \dots, (w_{a_r}, a_r)$ where $w_{a_1} | \dots | w_{a_r}$ is a partition of w into *non-empty* substrings, all the a_i 's are distinct, and $a_1, \dots, a_r \in [\ell]$. Note that $r \leq \ell$ is implicit, but that $r < \ell$ is possible. We denote a numbered partition by $P_w = (w_{a_1}, a_1) | \dots | (w_{a_r}, a_r)$. If $a_1 < a_2 < \dots < a_r \leq \ell$, then P_w is called an *ordered partition*. Let $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ and $P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ be ordered partitions of s and t , respectively. We say that (P_s, P_t) *agrees* with π and F if, for each $j \in \{b_1, \dots, b_r\}$, we have $\pi(j) \in \{a_1, \dots, a_r\}$ and $t_j = f_j(s_{\pi(j)})$.

Roughly speaking, an ordered partition allows us to “tag” each block with its position in a solution to PCSP. There may be holes between some a_i and a_{i+1} , which correspond to empty blocks in a solution.

Lemma 4 *The instance (s, t, π, ℓ, F) is a Yes-instance of PCSP if and only if there exist ordered partitions $(s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ and $(t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ that agree with π and F .*

Proof ($\Rightarrow$): Assume that (s, t, π, ℓ, F) is a Yes-instance of PCSP, and consider a solution $s_1 | \dots | s_\ell$ and $t_1 | \dots | t_\ell$. These may contain empty strings. Note that $t_j = \lambda$ if and only if $s_{\pi(j)} = \lambda$, since $t_j = f_j(s_{\pi(j)})$ and string functions map λ to λ . In particular, the number of empty substrings in both partitions is the same. Let $a_1, \dots, a_r \in [\ell]$ be the indices for which s_{a_i} is non-empty, ordered so that $a_1 < \dots < a_r$. Likewise, let $b_1, \dots, b_r$ be the indices for which t_{b_j} is non-empty, with $b_1 < \dots < b_r$. Then $(s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ and $(t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$

are ordered partitions that agree with π and F , since $t_j = f_j(s_{\pi(j)})$ for each $j \in \{b_1, \dots, b_r\}$.

($\Leftarrow$): Suppose that there are ordered partitions $(s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ and $(t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ that agree with π and F . Consider the partition $s'_1 | s'_2 | \dots | s'_\ell$ of s where, for each $i \in [\ell]$,

$$s'_i = \begin{cases} s_i & \text{if } i \in \{a_1, \dots, a_r\} \\ \lambda & \text{otherwise} \end{cases}$$

Similarly, define the partition $t'_1 | t'_2 | \dots | t'_\ell$ of t where for each $j \in [\ell]$,

$$t'_j = \begin{cases} t_j & \text{if } j \in \{b_1, \dots, b_r\} \\ \lambda & \text{otherwise} \end{cases}$$

Consider t'_j , $j \in \{b_1, \dots, b_r\}$. Thus $t'_j = t_j$ and, since the ordered partition agrees with π , $t_j = f_j(s_{\pi(j)})$ with $\pi(j) \in \{a_1, \dots, a_r\}$. In this case, $s_{\pi(j)} = s'_{\pi(j)}$, and thus we have $t'_j = f_j(s'_{\pi(j)})$. Now consider t'_j with $j \notin \{b_1, \dots, b_r\}$. Then $t'_j = \lambda$. Moreover, π restricted to $\{b_1, \dots, b_r\}$ forms a bijection with $\{a_1, \dots, a_r\}$, and thus $\pi(j)$ cannot be in $\{a_1, \dots, a_r\}$. It follows that $s'_{\pi(j)} = \lambda$, and once again $t'_j = f_j(s'_{\pi(j)})$ (here we use the fact that $f_j(\lambda) = \lambda$). Since $t'_j = f_j(s'_{\pi(j)})$ for every $j \in [\ell]$, we have constructed a solution to the PCSP instance. $\square$

4.1 Bounded occurrences and reversal functions

Note that just like Theorem 1, Lemma 4 holds for any set of linear-time computable simple functions F and without restrictions on the number of occurrences. From now on, we assume that each symbol occurs at most d times in s , and at most d times in t . We also assume that s and t are on an alphabet Σ where each symbol x has a unique reverse $-x$, where $-(-x) = x$, and that each function of F is in $\{id, flip, rev, urev\}$. For a marker x of s or t , denote by $S(x) \in \{s, t\}$ the string that contains x . Our goal is to find an ordered partition that agrees with π and F .

Our algorithm is based on the approach of Bulteau et al. (2013a) for the classical common string partition problem in which there is no permutation or function constraint (although we note that the approach also allows deletions of “extra” symbols). This algorithm guesses a pair of matching markers in each block of a solution. We apply the same idea here, but develop new tools to account for the permutation π and the F functions.

A *fixed-match* is a tuple (x, y, i, j) where x is a marker of s , y is a marker of t , $i, j \in [\ell]$, $i = \pi(j)$, and $y = f_j(x)$ (this is slightly abusing notation, since x and y are markers, not strings — it is understood here that x and y are treated as their symbols in this equality). The interpretation of (x, y, i, j) is that in the desired ordered partition, x belongs to block (s_i, i) of s , y belongs to block (t_j, j) of t , and $t_j = f_j(s_{\pi(j)})$. Of course, the blocks are unknown, but

x and y serve as a witness of their existence. Also note that i is redundant in the definition of a fixed-match, since it is given by $\pi(j)$. However, it will help making some notions clearer later on.

A *fixed-match set* is a set M of fixed-matches such that, for any distinct $(x, y, i, j), (x', y', p, q) \in M$, we have $x \neq x', y \neq y'$, and $j \neq q$ (which implies $i \neq p$). The idea is that a fixed-match set provides at most one match per block, and our goal is to eventually find *exactly* one fixed-match per block of a solution, if such a solution exists. With the following, we ensure that fixed-match sets can satisfy the requirements of an ordered partition.

Definition 3 A fixed-match set M is *order-consistent* if, for any $(x, y, i, j), (x', y', p, q) \in M$, $x \prec x'$ implies $i < p$ and $y \prec y'$ implies $j < q$.

Let $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$, $P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ be ordered partitions that agree with π and F . We know which substring t_j corresponds to which $f_j(s_i)$ substring. However, we also need a way to determine which markers are in correspondence, which depends on whether f_j reverses the string or not.

Let $j \in \{b_1, \dots, b_r\}$, so that $t_j = f_j(s_{\pi(j)})$. For each $i \in [|s_{\pi(j)}|]$, if $f_j \in \{id, flip\}$, match the i -th marker of $s_{\pi(j)}$ with the i -th marker of t_j , and otherwise match the i -th marker of $s_{\pi(j)}$ with the $(|t_j| - i + 1)$ -th marker of t_j . Two markers x of $s_{\pi(j)}$ and y of t_j matched as described above are said to be *matched by* (P_s, P_t) . Note that each marker x of s is matched to a distinct marker y of t by (P_s, P_t) . Therefore, the graph in which the vertices are the markers of s and t , with an edge between markers matched by (P_s, P_t) , forms a perfect matching. In essence, this matching simply indicates the new location of the x marker after applying f_j . We next relate fixed-match sets with numbered partitions (which are not necessarily ordered).

Definition 4 Let M be a fixed-match set with r elements. We say that M is *complete for s and t* if there exist numbered partitions $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ of s and $P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ of t such that all the following conditions hold:

1. for each $j \in \{b_1, \dots, b_r\}$, $t_j = f_j(s_{\pi(j)})$;
2. for each $(x, y, i, j) \in M$, it holds that $i \in \{a_1, \dots, a_r\}$ and $j \in \{b_1, \dots, b_r\}$.
Moreover, x is in the s_i substring, y is in the t_j substring, and x, y are matched by (P_s, P_t) .

We call (P_s, P_t) a *completion* of M . Furthermore, we say that M is *completable* if there exists a fixed-match set $M' \supseteq M$ such that M' is complete.

For short, we say that M is complete to mean that it is complete for s and t . Note that the definition of complete depends not only on s and t , but also on π and F . We only mention s and t since, in what follows, the notion of complete always refers to π and F as fixed in the beginning of the section, whereas the nature of s and t may change in some of the proofs. Also note that the definition of complete does not require $a_1 < \dots < a_r$ nor $b_1 < \dots < b_r$. In fact, not all complete fixed-match sets correspond to a solution, as order-consistency is required.

Lemma 5 *There exist ordered partitions of s and t that agree with π and F if and only if there exists a complete and order-consistent fixed-match set M .*

Proof ($\Rightarrow$) Suppose there are ordered partitions $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ of s and $P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ of t that agree with π and F . We construct a complete and order-consistent fixed-match set M . Let $j \in \{b_1, \dots, b_r\}$, and let y be the first marker of t_j . If $f_j \in \{id, flip\}$, let x be the first marker of $s_{\pi(j)}$, and if $f_j \in \{rev, urev\}$, let x be the last marker of $s_{\pi(j)}$. Since $t_j = f_j(s_{\pi(j)})$, we observe that x, y are matched by (P_s, P_t) . Then add to M the fixed-match $(x, y, \pi(j), j)$ (note that $y = f_j(x)$, as required). We repeat this for every $j \in \{b_1, \dots, b_r\}$.

Note that no two fixed-matches of M share a marker x, y or block index i, j . Thus M satisfies the definition of a fixed-match set. To see that M is complete, note that $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ and $P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ is a numbered partition that satisfies every condition of completeness. That is, Condition 1 is satisfied since $t_j = f_j(s_{\pi(j)})$ for each $j \in \{b_1, \dots, b_r\}$, by agreement; Condition 2 is also satisfied by our construction of M since we chose exactly one match per numbered block, and for each s_i and t_j substring, we chose one of their markers that were matched by (P_s, P_t) .

It only remains to argue that M is order-consistent. Let $(x, y, i, j), (x', y', p, q) \in M$ be distinct. Suppose that $x \prec x'$ but $i \geq p$. Note that by construction, $i \neq p$, so we have $i > p$. Since P_s is ordered, P_s has the form $\dots | (s_p, p) | \dots | (s_i, i) | \dots$. This contradicts $x \prec x'$, since x is in s_i and x' is in s_p . Similarly, suppose that $y \prec y'$ but $j \geq q$. As we know, $j \neq q$ and thus $j > q$. Thus P_t has the form $\dots | (t_q, q) | \dots | (t_j, j) | \dots$, contradicting $y \prec y'$ since y is in t_j and y' is in t_q . Since both possibilities lead to a contradiction, we deduce that M must be order-consistent.

($\Leftarrow$) Let M be an order-consistent and complete fixed-match set. Let $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ and $P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ be the numbered partitions of s and t , respectively, that satisfy all conditions of completeness. By Condition 1 of complete fixed-match sets, $t_j = f_j(s_{\pi(j)})$ for each $j \in \{b_1, \dots, b_r\}$. Therefore, the numbered partitions agree with π and F . We need to argue that P_s and P_t are *ordered* partitions. For that, it remains to show that $a_1 < a_2 < \dots < a_r$ and $b_1 < b_2 < \dots < b_r$.

As we show, this follows from the fact that M is order-consistent. First consider blocks t_{b_j} and $t_{b_{j+1}}$ of P_t , where $j \in [r-1]$. By Condition 2 of completeness, M contains fixed-matches $(x, y, \pi(b_j), b_j)$ and $(x', y', \pi(b_{j+1}), b_{j+1})$ such that y is in t_{b_j} and y' is in $t_{b_{j+1}}$. By our partitioning of t , we know that $y \prec y'$. The fact that M is order-consistent implies that $b_j < b_{j+1}$. Since this is true for every j , $b_1 < \dots < b_r$ holds. Now consider blocks s_{a_i} and $s_{a_{i+1}}$ of P_s , where $i \in [r-1]$. As before, there are fixed-matches (x, y, a_i, b_j) and (x', y', a_{i+1}, b_h) in M . By our partitioning of s , we have $x \prec x'$ and, since M is order-consistent, $a_i < a_{i+1}$, which implies $a_1 < \dots < a_r$. $\square$

4.2 Constructing a complete and order-consistent fixed-match set

In the rest of the section, we aim to construct a complete and order-consistent fixed-match set. We adopt a branching strategy that adds one fixed-match at a time. If, at any point in the search tree, we reach a fixed-match set M that is not order-consistent, it cannot lead to the desired numbered partitions and we can stop trying to complete M (this is because we can only add elements to a fixed-match set, which cannot repair the lack of order-consistency).

Termination rule. If M is not order-consistent, then terminate, since there exists no complete and order-consistent fixed-match set M' such that $M' \supseteq M$.

This is where we extend the framework of Bulteau et al. (2013a). Let M be a fixed-match set, and let u be a marker of s or t that is not in any fixed-match of M . Let $l_{s,t,M}(u)$ (resp. $r_{s,t,M}(u)$) be the closest marker to the left (resp. right) of u that belongs to some fixed-match of M . We may write $l_M(u)$ and $r_M(u)$ if s and t are clear from the context. Note that $l_M(u)$ or $r_M(u)$ may not exist, in which case they are left undefined. If u is in a fixed-match, we also leave $l_M(u)$ and $r_M(u)$ undefined. Let $(x, y, i, j) \in M$. We are interested in pairs of markers that could belong to the same block as x and y . For this purpose, let $\{u, v\}$ be a pair of markers such that u is in s and v is in t . There are four types of matches that are represented in Figure 3.

- (*LL match*) $\{u, v\}$ is a *left-left match* of (x, y, i, j) if $f_j \in \{id, flip\}$, $r_M(u) = x, r_M(v) = y$ and $t[v, y] = f_j(s[u, x])$;
- (*RR match*) $\{u, v\}$ is a *right-right match* of (x, y, i, j) if $f_j \in \{id, flip\}$, $l_M(u) = x, l_M(v) = y$ and $t[y, v] = f_j(s[x, u])$;
- (*LR match*) $\{u, v\}$ is a *left-right match* of (x, y, i, j) if $f_j \in \{rev, urev\}$, $r_M(u) = x, l_M(v) = y$ and $t[y, v] = f_j(s[u, x])$;
- (*RL match*) $\{u, v\}$ is a *right-left match* of (x, y, i, j) if $f_j \in \{rev, urev\}$, $l_M(u) = x, r_M(v) = y$ and $t[v, y] = f_j(s[x, u])$.

Note that a pair $\{u, v\}$ can be multiple types of matches simultaneously. Also note that we defined $\{u, v\}$ as an *unordered* pair, and assumed that u was

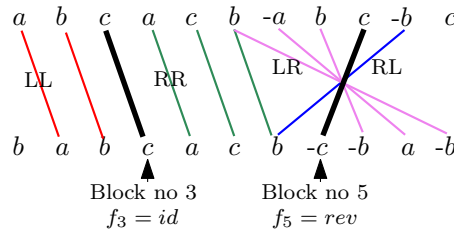


Fig. 3 A simple example depicting the four types of matches. The black edges represent fixed-matches, red edges are *LL* matches, green are *RR*, pink are *LR* and blue are *RL*. Hypothetically, the leftmost black edge belongs to some fixed-match (x, y, i, j) satisfying $j = 3$ (its block number), and $f_3 = id$. Similarly, the rightmost black edge belongs to a fixed-match (x', y', p, q) with $q = 5$ and $f_5 = rev$.

in s and v was in t . This means that if $\{u, v\}$ is a left-right match, for instance, then $\{v, u\}$ is also a left-right match. This is an important notational detail: when a pair $\{u, v\}$ of markers from s and t is given, u could be in s and v in t , or vice-versa. In most cases, this will not matter, but since this may impact the LR and RL types, we will need to specify the strings of origin of u and v when required.

We will view the markers and matches as a bipartite edge-colored multi-graph. Given a fixed-match set M , the graph $G_{s,t}(M)$ contains one vertex for each marker in s or t . We will not make the distinction between markers and vertices from now on, and will omit the s, t subscript when they are clear from the context. We add one edge for each match of any type, and label the edge by the type. That is, for each $(x, y, i, j) \in M$, add an edge $\{u, v\}$ labeled LL (respectively RR , LR , RL) if $\{u, v\}$ is a left-left (resp. right-right, left-right, right-left) match of (x, y, i, j) . We will call (x, y, i, j) the *fixed-match* of $\{u, v\}$. Furthermore, for each $(x, y, i, j) \in M$, add the edge $\{x, y\}$ and label it *fixed*.

Note that it is possible that an edge $\{u, v\}$ is present more than once, but each has a distinct label. Hence, given an edge $\{u, v\}$ and its label, the fixed-match of $\{u, v\}$ is uniquely defined. The basic properties of $G(M)$ are listed in the following.

Lemma 6 *Let u be a vertex of $G(M)$. Then u is incident to at most two edges and, in addition, the following holds:*

- if u is a marker of s , then u is not incident to two edges whose labels are both in $\{LL, LR\}$, or whose labels are both in $\{RR, RL\}$;
- if u is a marker of t , then u is not incident to two edges whose labels are both in $\{LL, RL\}$, or whose labels are both in $\{RR, LR\}$;
- if u is in a fixed-match in M , then u is incident to exactly one edge, which is labeled *fixed*.

Proof Let u be a marker of s that is not in a fixed-match of M . Thus u is not incident to a *fixed* edge. Notice that u cannot be incident to two edges with the same label, since only $l_M(u)$ and $r_M(u)$ “give” a single label to an edge with u , and $l_M(u)$ and $r_M(u)$ must give a different label. Also, u cannot be incident to both an LL edge and an LR edge. Indeed, being in an LL edge requires $r_M(u)$ to be in a fixed-match $(r_M(u), y, i, j) \in M$ with $f_j \in \{id, flip\}$, whereas an LR edge requires $f_j \in \{rev, urev\}$. By a symmetric argument, u cannot be incident to both an RR and an RL edge because of the requirements on $l_M(u)$. Thus u is incident to at most one edge with label in $\{LL, LR\}$, and at most one edge with label in $\{RL, RR\}$. Thus u is incident to at most 2 edges.

Assume that u is a marker of t instead. Using the same arguments, u is not incident to two edges with the same label, and cannot be incident to both an LL edge and an RL edge because of the function requirements on the fixed match with $r_M(u)$. By the same reasoning, it cannot be incident to both an RR edge and an LR edge because of $l_M(u)$.

Finally, for a marker u that is in a fixed-match of M , it is clear from the construction that u is incident to exactly one *fixed* edge (since by definition, u cannot be in an LL , RR , LR or RL match). $\square$

It follows from the above that $G(M)$ has maximum degree 2 and consists of a collection of disjoint cycles and paths. As in Bulteau et al. (2013a), our goal is to identify a subset of vertices that prevents us from extending M to a complete fixed-match set.

Definition 5 Let M be a fixed-match set, let $M' \supseteq M$ be a complete fixed-match set and let (P_s, P_t) be a completion of M' . Let u be a marker of s or t . Then u is *unseen* by M in (P_s, P_t) if, for any $(x, y, i, j) \in M$, u is not in the same substring as x in P_s and u is not in the same substring as y in P_t .

Lemma 7 Let M be a fixed-match set and suppose that, for any complete $M' \supseteq M$ and any completion (P_s, P_t) of M' , there exists a marker unseen by M in (P_s, P_t) . Then M is not complete.

Proof We prove the statement by contraposition. Assume that M is complete and let $r = |M|$. Then there are numbered partitions P_s and P_t , each with r substrings, that satisfy all the conditions of Definition 4, and such that (P_s, P_t) is a completion of M . By definition, for each distinct $(x, y, i, j), (x', y', p, q) \in M$, x and x' belong to distinct substrings in P_s (because $i \neq p$) and y, y' belong to different substrings in P_t (because $j \neq q$). Because M has r elements and P_s, P_t each have r substrings, it follows that the substring of P_s (resp. P_t) are in bijection with the fixed-matches of M . Consequently, for any marker u of s or t , u is in the same substring as some marker of M . Thus u is not unseen by M in (P_s, P_t) . As this holds for any u , there does not exist an unseen marker by M in (P_s, P_t) , which proves the lemma. $\square$

An *odd path* is a connected component of $G(M)$ with an odd number of vertices. Note that because $G(M)$ is bipartite and because each vertex has degree at most 2, such a connected component must be a path. Also note that a vertex of degree 0 is an odd path. By Lemma 6, such a path cannot contain *fixed* edges, and does not contain two consecutive edges with the same label. Moreover, we show that for any odd path $u_1, \dots, u_h$ and any completion (P_s, P_t) of a superset of M , there is at least one vertex u_c of the path unseen by M in (P_s, P_t) . The idea of the proof is that if we assume that each u_c is not unseen by some completion (P_s, P_t) , u_1 must be matched with u_2 , then u_3 with u_4 , and so on until u_h cannot be matched with any marker.

Lemma 8 Suppose that M is completable, and suppose that there is an odd path $(u_1, \dots, u_h)$ in $G(M)$. Then for any complete $M' \supseteq M$ and any completion (P_s, P_t) of M' , at least one of $u_1, \dots, u_h$ is unseen by M in (P_s, P_t) .

Proof Assume that $(u_1, \dots, u_h)$ is an odd path but that the statement is false. That is, there is $M' \supseteq M$ and a completion (P_s, P_t) of M' in which no u_c of the path is unseen by M in (P_s, P_t) . Denote $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$ and

$P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$. As we know, the graph G' in which each marker of s or t shares an edge with the marker that it is matched with by (P_s, P_t) , forms a perfect matching.

Consider a vertex u_c on the odd path. We claim that u_c is matched with a marker v by (P_s, P_t) , such that v is a neighbor of u_c in $G(M)$. Because u_c is not unseen by M , then u_c is in the same substring as some marker of a fixed-match of M . Moreover, u_c can only be in the same substring as either $l_M(u_c)$ or $r_M(u_c)$. Suppose first that u_c is in the same substring as $l_M(u_c)$. By the definition of a completion, we know that $l_M(u_c)$ is part of a fixed-match (x, y, i, j) such that x and y are matched by (P_s, P_t) (here, $l_M(u_c)$ is either x or y , depending on whether u_c is in s or t). If u_c is in the same substring as x , then v is in the same substring as y , and if u_c is with y , then v is with x . Either way, $\{u_c, v\}$ forms either an RR , RL , or LR match and forms an edge in $G(M)$. Likewise, suppose that u_c is in the same substring as $r_M(u_c)$. The latter is in a fixed-match (x, y, i, j) with x, y matched by (P_s, P_t) . Then u_c is in the same substring as x and v in the substring as y , or vice-versa, but either way, $\{u, v\}$ form either an LL , LR or RL match and forms an edge in $G(M)$.

It follows that in the perfect matching G' , each vertex u_c of the odd path is matched with one of its neighbors in $G(M)$. This is a contradiction, since no perfect matching exists in a graph that contains a connected component that is an odd path. $\square$

This leads to our main branching rule.

Branching rule. If there is an odd path $(u_1, \dots, u_h)$, branch into all ways of adding a fixed-match that contains one of $u_1, \dots, u_h$ to M .

Note that because our functions are *id*, *flip*, *rev*, and *urev*, if in an odd path $(u_1, \dots, u_h)$ the u_1 marker corresponds to symbol $w \in \Sigma$, then each of $u_1, \dots, u_h$ is a marker that corresponds to symbol w or $-w$. Since each has at most d occurrences in s and in t , the maximum number of vertices in an odd path is $4d$. Also, if only *id* and *urev* are allowed, negated symbols are not present and there are at most $2d$ vertices on such an odd path. Either way, the number of possible vertices on such a path to add to a fixed-match of M is bounded by a function of d , as well as the number of “partners” of these vertices in a fixed-match. In fact, this is the only reason why our branching algorithm does not work for any given sequence of simple string functions F , since if F has arbitrary simple functions, we cannot guarantee that the number of vertices on an odd path is a function of d only.

Our next step is to show that once all odd paths have been handled, we have reached a complete fixed-match set. At a high level, this can be shown by induction: if we find a *fixed* edge $\{x, y\}$ that has an LL , RR , LR or RL edge next to it, we can assume that this latter edge can belong to the same substring as x and y in a solution. We will need the following definition.

Definition 6 Let $\{u, v\}$ be an edge of $G(M)$ such that u is a marker of s and v is a marker of t . Let (x, y, i, j) be the fixed-match of $\{u, v\}$. We say that $\{u, v\}$ is *next to its fixed-match* if one of the following holds:

- $\{u, v\}$ is an LL edge and $u + 1 = x, v + 1 = y$;
- $\{u, v\}$ is an RR edge and $u - 1 = x, v - 1 = y$;
- $\{u, v\}$ is an LR edge and $u + 1 = x, v - 1 = y$;
- $\{u, v\}$ is an RL edge and $u - 1 = x, v + 1 = y$.

Our goal is to implement the following strategy. We would like to show that if $G(M)$ has no odd path, there is always an edge $\{u, v\}$ next to its fixed-match such that, after removing u and v , the resulting graph still has no odd path. As a first step, this requires showing that the edges of $G(M)$ remain unchanged after removing u and v (except $\{u, v\}$ itself, see Lemma 10). This edge removal process could be repeated until only fixed-matches remain, showing that M is complete (as can be deduced from Lemma 11). However, removing an arbitrary edge next to its fixed-match may create odd paths. For instance, if $G(M)$ has an even path, then an edge removal could split it into two odd paths. We will thus show that if $G(M)$ has an even path, then there is some degree one vertex whose edge is next to its fixed-match. Removing that edge is harmless. Let us proceed with the details.

We state a basic result first without proof, as it follows from the definition of the match types.

Lemma 9 *Let $\{u, v\}$ be an edge of $G(M)$, where $S(u) = s$ and $S(v) = t$. Then*

- *if $\{u, v\}$ is labeled LL , then $\{u + 1, v + 1\}$ is either labeled LL or fixed;*
- *if $\{u, v\}$ is labeled RR , then $\{u - 1, v - 1\}$ is either labeled RR or fixed;*
- *if $\{u, v\}$ is labeled LR , then $\{u + 1, v - 1\}$ is either labeled LR or fixed;*
- *if $\{u, v\}$ is labeled RL , then $\{u - 1, v + 1\}$ is either labeled RL or fixed.*

Lemma 10 *Let $\{u, v\}$ be an edge of $G(M)$ next to its fixed-match, where $S(u) = s$ and $S(v) = t$. Let s' and t' be the strings obtained by removing marker u from s and v from t . Then $G_{s',t'}(M)$ is the graph obtained from $G_{s,t}(M)$ by removing u, v and their incident edges.*

Proof Note that u and v are not in a fixed-match of M , and so the left and right fixed-match markers of any marker w of s' or t' cannot change by removing u and v . To be precise, this means that for any marker w of s' or t' , $l_{s',t',M}(w) = l_{s,t,M}(w)$ and $r_{s',t',M}(w) = r_{s,t,M}(w)$.

Let u' and v' be markers of s' and t' , respectively. Suppose that $\{u', v'\}$ is an edge of $G_{s',t'}(M)$, and let $(x, y, i, j) \in M$ be the fixed-match of $\{u', v'\}$. Assume that $\{u', v'\}$ is an LL edge of $G_{s',t'}(M)$ (hence, $f_j \in \{id, flip\}$), but not of $G_{s,t}(M)$. This is caused by the removal of u and v , so we must have $u' \prec u \prec x$ or $v' \prec v \prec y$. If the former holds, because $\{u, v\}$ is chosen to be next to a fixed-match, it must be that $u + 1 = x$, since $u - 1$ is not in a fixed-match. Thus $\{u, v\}$ is labeled LL or LR , but it cannot be labeled LR since $f_j \in \{id, flip\}$. This implies that $\{u, v\}$ is an LL edge in $G_{s,t}(M)$. But then, $f_j(s[u', x]) = t[v', y]$ holds because $f_j(s'[u', x]) = t'[v', y]$ and $f_j(s[u, x]) = t[v, y]$, implying that $\{u', v'\}$ should also be an LL edge in $G_{s,t}(M)$ as well. The reasoning is identical

if v is between v' and y , and the case in which $\{u', v'\}$ is an RR edge is symmetric.

Assume next that $\{u', v'\}$ is an LR edge of $G_{s', t'}(M)$, but not in $G_{s, t}(M)$. Again, u must be between u' and x or v between y and v' . If the former holds, we must have $u' \prec u \prec x$ and $x = u + 1$. Then $\{u, v\}$ must be next to x and y , and thus $\{u, v\}$ must form an LR edge. In this case, $\{u', v'\}$ must also be an LR edge of $G_{s, t}(M)$, since $f_j(s[u', x]) = t[y, v']$ because $f_j(s'[u', x]) = t'[y, v']$ and $f_j(s[u, x]) = t[y, v]$. The same holds if v is between y and v' , and the reasoning is symmetric if $\{u', v'\}$ is an RL edge.

Therefore, every edge of $G_{s', t'}(M)$ is in $G_{s, t}(M)$.

Now, consider an edge $\{u', v'\}$ of $G_{s, t}(M)$ where $S(u') = s$, $S(v') = t$, and u' and v' are distinct from u and v . Assume that $\{u', v'\}$ is not in $G_{s', t'}(M)$. Suppose that in $G_{s, t}(M)$, $\{u', v'\}$ is a match of $(x, y, i, j) \in M$. Let us first consider the case where $\{u', v'\}$ is an LL edge of $G_{s, t}(M)$. Then u must be between u' and x or v between v' and y . If $u' \prec u \prec x$, it must be that $\{u, v\}$ is next to x and y (since there is no other fixed-match that $\{u, v\}$ can be next to). This implies $x = u + 1$ and $y = v + 1$. By the simplicity of f_j , we have that

$$f_j(s[u', x]) = f_j(s[u', u - 1]) f_j(s[u, u]) f_j(s[x, x]), \text{ which equals} \\ t[v, y] = t[v', v - 1] t[v, v] t[y, y]$$

Since f_j is simple, it follows that

$$f_j(s'[u', x]) = f_j(s[u', u - 1]) f_j(s[x, x]) = t[v', v - 1] t[y, y] = t'[v', y].$$

Thus $\{u', v'\}$ is an LL match of (x, y, i, j) in $G_{s', t'}(M)$. The same holds if v is between v' and y , and the case in which $\{u', v'\}$ is an RR edge is symmetric. Suppose instead that $\{u', v'\}$ is an LR edge of $G_{s, t}(M)$. If u is between u' and x , then $u' \prec u \prec x$ and again, $\{u, v\}$ must be next to x and y . This implies $u + 1 = x$ and $v - 1 = y$. Using the same approach as in the previous case, by removing u and v , u' and v' will still form an LR edge. The same holds if v' is between y and v , and the case in which $\{u', v'\}$ is an RL edge is symmetric.

Therefore, every edge of $G_{s, t}(M)$ not incident to u or v is in $G_{s', t'}(M)$. $\square$

Lemma 11 *Let $\{u, v\}$ be an edge of $G(M)$ next to its fixed-match, where $S(u) = s$ and $S(v) = t$. Let s' and t' be the strings obtained by removing marker u from s and v from t . If M is complete for s' and t' , then M is also complete for s and t .*

Proof Let $P_{s'} = (s'_{a_1}, a_1) | \dots | (s'_{a_r}, a_r)$ and $P_{t'} = (t'_{b_1}, b_1) | \dots | (t'_{b_r}, b_r)$ be ordered partitions of s' and t' , respectively, such that $(P_{s'}, P_{t'})$ is a completion of M . We construct ordered partitions $P_s = (s_{a_1}, a_1) | \dots | (s_{a_r}, a_r)$, $P_t = (t_{b_1}, b_1) | \dots | (t_{b_r}, b_r)$ of s and t , respectively, such that (P_s, P_t) is a completion of M . Since the values $a_1, \dots, a_r, b_1, \dots, b_r$ are the same as in $P_{s'}, P_{t'}$, it only remains to define the s_i and t_j substrings, $i \in \{a_1, \dots, a_r\}$ and $j \in \{b_1, \dots, b_r\}$.

Let (x, y, i, j) be the fixed-match of $\{u, v\}$. Let $(x', y', p, q) \in M$ be a fixed-match other than (x, y, i, j) . Consider the substring $s'_p = s'[w_1, w_2]$ as defined

in $P_{s'}$. Note that s'_p contains x' and no other marker in a fixed-match. In s , $w_1 \preceq u \preceq w_2$ is not possible, for it would imply that u is next to x' instead of x . Likewise, for the substring $t'_q = t'[z_1, z_2]$, we cannot have $z_1 \preceq v \preceq z_2$ in t since v would be next to y' . Thus s'_p and t'_q are both substrings of s and t , respectively. In this case, we define $s_p = s'_p$ and $t_q = t'_q$, so that $t_q = f_q(s_p)$. It only remains to define s_i and t_j . Consider the substrings $s'_i = s'[w_1, w_2]$ of $P_{s'}$ and $t'_j = t'[z_1, z_2]$ of $P_{t'}$, which respectively contain x and y . We have two cases, depending on whether f_j reverses its input or not.

Case 1: suppose that $f_j \in \{id, flip\}$. Then $\{u, v\}$ is either labeled LL or RR . Since x and y are matched by $(P_{s'}, P_{t'})$, we have $t'[z_1, y] = f_j(s'[w_1, x])$ and $t'[y, z_2] = f_j(s'[x, w_2])$. Note that $\{w_1, z_1\}$ and $\{w_2, z_2\}$ are edges of $G_{s', t'}(M)$, and thus of $G_{s, t}(M)$ by Lemma 10 (note that these could be fixed edges, if $w_1 = x, z_1 = y$, or if $w_2 = x, z_2 = y$). It follows that $t[z_1, y] = f_j(s[w_1, x])$ and $t[y, z_2] = f_j(s[x, w_2])$. Hence, $t[z_1, z_2] = f_j(s[w_1, w_2])$.

Assume that $\{u, v\}$ is an LL edge, so that $u + 1 = x, v + 1 = y$. If $w_1 \prec x$, then $z_1 \prec y$ and $s[w_1, w_2]$ and $t[z_1, z_2]$ contain u and v , respectively. In this case, putting $s_i = s[w_1, w_2], t_j = t[z_1, z_2]$ shows that (P_s, P_t) is a completion of M . It is possible that $w_1 = x$, in which case we put $s_i = s[u, w_2]$ and $t_j = t[v, z_2]$. Since $v = f_j(u)$, this also results in a completion of M . Assume instead that $\{u, v\}$ is an RR edge, i.e. $x = u - 1, y = v - 1$. If $x \prec w_2$, then $y \prec z_2$ and $s_i = [w_1, w_2], t_j = [z_1, z_2]$ contain u and v and result in a completion of M . If $x = w_2$, then $s_i = [w_1, u], t_j = [z_1, v]$ results in a completion of M .

Case 2: suppose that $f_j \in \{rev, urev\}$. Then $\{u, v\}$ is labeled LR or RL . Since x and y are matched by $(P_{s'}, P_{t'})$, we have $t'[z_1, y] = f_j(s'[x, w_2])$ and $t'[y, z_2] = f_j(s'[w_1, x])$. Here, $\{z_1, w_2\}$ and $\{z_2, w_1\}$ are edges of $G_{s', t'}(M)$ and $G_{s, t}(M)$. It follows that $t[z_1, y] = f_j(s[x, w_2]), t[y, z_2] = f_j(s[w_1, x])$, and hence that $t[z_1, z_2] = f_j(s[w_1, w_2])$.

Suppose that $\{u, v\}$ is an LR edge, i.e. $u + 1 = x, v - 1 = y$. If $w_1 \prec x$, then $y \prec z_2$ and u, v are in $s[w_1, w_2], t[z_1, z_2]$, respectively. In this case, these two latter substrings form our completion. If $w_1 = x$, then $z_2 = y$ and putting $s_i = [u, w_2], t_j = [z_1, v]$ results in our completion. Finally, suppose that $\{u, v\}$ is an RL edge, i.e. $u - 1 = x, v + 1 = y$. If $x \prec w_2$, then $z_1 \prec y$, and $s[w_1, w_2], t[z_1, z_2]$ form our completion since they contain u and v . If $x = w_2$, then $y = z_1$, and here $s_i = s[w_1, u], t_j = [v, z_2]$ form our completion. $\square$

We can then handle degree 1 vertices of even paths.

Lemma 12 *Suppose that $G(M)$ has a degree 1 vertex that is not in an edge labeled fixed. Then $G(M)$ has a degree 1 vertex u with neighbor v , such that $\{u, v\}$ is an edge of G next to its fixed-match.*

Proof Let u be a degree 1 vertex of $G(M)$, and let v be its neighbor. Let (x, y, i, j) be the fixed-match of $\{u, v\}$. We handle four separate cases depending on whether $\{u, v\}$ is an LL, RR, LR , or RL edge. Note that u could be in s or t .

Case 1: $\{u, v\}$ is an LL edge. Suppose that u is a marker of s . If $u + 1 = x, v + 1 = y$, we are done, so suppose otherwise. Note that for each i such that $u \prec u + i \prec x$, $\{u + i, v + i\}$ is an LL edge by Lemma 9. We claim that for each such i , $u + i$ also has degree 1. Suppose this is false, and let i be the smallest value for which $u + i$ has degree 2.

Then $u + i$ has a neighbor w , with whom it forms an RR or RL edge (recall that $u + i$ cannot be adjacent to two edges with a label in $\{LL, LR\}$, by Lemma 6). If it is an RR edge, then $\{u + i - 1, w - 1\}$ is either fixed or RR , and if it is an RL edge, then $\{u + i - 1, w + 1\}$ is fixed or RL (both by Lemma 9). These edges cannot be fixed, so both contradict our choice of i , and hence this i does not exist.

Suppose next that u is a marker of t . Assume that i is the smallest value for which $u + i$ has degree 2, again assuming that such an i exists. Then $u + i$ forms an RR or LR match with some w by Lemma 6. In both cases, by Lemma 9 $u + i - 1$ is *fixed* or in an RR or LR edge, with the same contradiction.

We can thus choose i such that $\{u + i, v + i\} = \{x - 1, v - 1\}$ is an LL edge, which is next to its fixed-match with $u + i$ of degree 1, as desired.

Case 2: $\{u, v\}$ is an RR edge. Suppose that u is in s . If $u - 1 = x, v - 1 = y$, we are done, so suppose otherwise. Note that for each i such that $x \prec u - i \prec u$, $\{u - i, v - i\}$ is an RR edge. We claim that for each such i , $u - i$ also has degree 1. Suppose this is false, and let i be the smallest value for which $u - i$ has degree 2. Then $u - i$ has a neighbor w , with whom it forms an LL or LR edge. In both cases, $u - i + 1$ is labeled *fixed*, LL , or LR , contradicting our choice of i .

If $u - i$ is in t , then again assume there is some smallest i such that $u - i$ has degree 2. Then $u - i$ is in an LL or RL edge. In both cases, $u - i + 1$ is *fixed* or in an LL or RL edge, with the same contradiction. Thus we can choose $u - i$ of degree 1 in an edge next to its fixed-match.

Case 3: $\{u, v\}$ is an LR edge or an RL edge. Since the proof is analogous to the previous cases, we handle them both at once (note that all cases could be handled in one single generic case, but readability would greatly suffer). Define $\delta = 1$ if $\{u, v\}$ is an LR edge (respectively, $\delta = -1$ if $\{u, v\}$ is an RL edge). Suppose that u is in s . If $u + \delta = x, v - \delta = y$, we are done. If not, let i be the smallest value for which $u + \delta i$ has degree 2, with $u \prec u + \delta i \prec x$ (resp. $x \prec u + \delta i \prec u$). Then $u + \delta i$ has a neighbor w with whom it forms either an RR edge or RL edge (resp. LL or LR). In both cases, $u + \delta i - \delta$ is in an edge labeled *fixed*, RR , or RL (resp. LL , or LR). Both contradict the choice of i , so no such i exists. Thus we can choose $u + \delta i$ of degree 1 next to x . If u is in t , let i be the smallest value for which $u - \delta i$ has degree 2, with $y \prec u - \delta i \prec u$ (resp. $u \prec u - \delta i \prec y$). Then $u - \delta i$ has a neighbor w with whom it forms an LL or RL edge (resp. RR or LR). In both cases, $u - \delta i + \delta$ is labeled *fixed*, LL , or RL (resp. *fixed*, RR or LR), contradicting our choice of i . Thus we can choose $u - \delta i$ next to y . $\square$

Lemma 13 *A fixed-match set M is complete if and only if $G(M)$ has no odd path.*

Proof We already know, by combining Lemma 7 and Lemma 8, that if $G(M)$ has an odd path, then M is not complete. We focus on the converse direction, i.e. that if $G(M)$ has no odd path, then M is complete. We show that this is true for any pair of strings s and t that contain all the markers that are in a fixed-match of M . We proceed by induction on the number of markers of s and t that are not included in a fixed-match of M . As a base case, suppose that all the markers of s and t are in a fixed-match. Then M gives the partition directly. That is, for each fixed-match $(x, y, i, j) \in M$, we add (x, i) and (y, j) to the numbered partitions. By definition, the resulting partition will agree with π and F .

Now assume that the statement is true for any s' and t' with fewer markers not in a fixed-match of M than s and t , i.e. that if $G_{s',t'}(M)$ has no odd path, then M is complete for s' and t' .

Since $G(M)$ has maximum degree 2, we know that $G(M)$ has only even paths (i.e. with an even number of vertices) and cycles. First assume that $G(M)$ contains an even path that does not consist of a fixed-match edge. Then $G(M)$ has a degree 1 vertex in that path and, by Lemma 12, there is an edge $\{u, v\}$ next to its fixed-match such that u has degree 1. Consider the strings s' and t' obtained by removing u and v from their respective string. By Lemma 10, $G_{s',t'}(M)$ is the same as $G_{s,t}(M)$ but with the s and t vertices removed. Thus each connected component of $G_{s,t}(M)$ that does not contain u or v is unchanged in $G_{s',t'}(M)$. Furthermore, we have removed $\{u, v\}$, the start of an even path. Thus the connected component containing u and v in $G_{s,t}(M)$ is still an even path in $G_{s',t'}(M)$, but with two less vertices (possibly 0). It follows that $G_{s',t'}(M)$ has no odd path. We can thus apply induction and deduce that M is complete for s' and t' . Then by Lemma 11, we deduce that M is complete for s and t as well.

Thus, if $G(M)$ has an even path other than those formed by fixed-match edges, we can find a numbered partition of s and t . Assume that $G(M)$ has no such even path. Since the base case was covered, s and t have at least one marker not in a fixed-match of M . Then every such marker is in a cycle and thus has degree exactly 2. Let $\{u', v'\}$ be any non-fixed edge of $G(M)$. We can deduce from Lemma 9 that some edge $\{u, v\}$ is next to its fixed matched (if $\{u', v'\}$ is LL or RR , then this is true for some $u' + i, v' + i$ with i possibly negative, and if $\{u', v'\}$ is LR or RL , then this is true for some $u' + i, v' - i$). We know that $\{u, v\}$ is in a cycle of $G(M)$. Consider s' and t' obtained by removing u and v . By Lemma 10, the connected components of $G_{s',t'}(M)$ are the same as in $G_{s,t}(M)$, except the one that contains u and v . Since this connected component is a cycle in $G_{s,t}(M)$, it is now either an even path in $G_{s',t'}(M)$, or the connected component was removed altogether (in the case that u, v formed a 2-cycle). Thus $G_{s',t'}(M)$ has no odd path. As before, we can apply induction to deduce that M is complete for s' and t' , and thus also complete for s and t by Lemma 11. $\square$

The main consequence is that branching over odd paths is all that is required. We can now put everything together in Algorithm 2 (with $M = \emptyset$ initially).

```

function  $PCSP(s, t, \ell, \pi, F, M)$ 
  if  $M$  is not order-consistent then return “No solution”
  if  $G(M)$  has an odd path  $(u_1, \dots, u_h)$  then
    if  $|M| = \ell$  then return “No solution”
    foreach  $u$  on the path and each marker  $v$  with the same symbol
      or negated symbol, such that  $S(v) \neq S(u)$  and  $v$  is not in a
      fixed-match of  $M$  do
      foreach  $j \in [\ell]$  not already in a fixed-match of  $M$  do
         $\sigma_1 = u, \sigma_2 = v$ 
        if  $u$  is in  $t$  then  $\sigma_1 = v, \sigma_2 = u$ 
        if  $f_j(\sigma_1) = \sigma_2$  then
          Call  $PCSP(s, t, \ell, \pi, F, M \cup \{(\sigma_1, \sigma_2, \pi(j), j)\})$ 
          if a positive answer was returned then
            return “Yes”
        end
      end
    end
    return “No solution”
  else
    return “Yes”

```

Algorithm 2: Main $PCSP$ algorithm

Theorem 2 *If F is a subset of $\{id, urev\}$, then $PCSP$ can be solved in time $O(d^{2\ell}(2\ell)^\ell n)$.*

Furthermore, if F is a subset of $\{id, flip, rev, urev\}$, then $PCSP$ can be solved in time $O(d^{2\ell}(4\ell)^\ell n)$.

Proof Algorithm 2 implements all the ideas discussed above. We argue its correctness inductively on the height of the search tree. At the terminal cases, if M is not order-consistent, by Lemma 5 we know that M cannot lead to a solution so we can return. If $G(M)$ has no odd path, we know that M is complete (and order-consistent) and so we may return that a solution exists, by Lemma 13. If G has an odd path, then by Lemma 7, M is not complete. If $|M| = \ell$, then we cannot add any more block, so M cannot lead to a solution. This takes care of the terminal cases.

If Algorithm 2 makes a recursive call, it is because there is an odd path and we branch over every possible unseen vertex on it. If M is not completable, then by induction, all the recursive calls return a negative answer, and so we also correctly return that there is no solution. Suppose instead that M is completable. That is, there exists a complete and order-consistent fixed-match set $M' \supseteq M$ with completion (P_s, P_t) . By Lemma 8, some vertex u on the odd path is unseen by M in (P_s, P_t) . Since M' is complete, P_s and P_t have substrings s_i and t_j , respectively, with $i = \pi(j)$ and $t_j = f_j(s_i)$, such that one of s_i or t_j contains u . Again because M' is complete, there is $(x, y, i, j) \in M'$

such that x is in s_i , y is in t_j , and x, y are matched in (P_s, P_t) . Let v be the unique marker of s or t such that u and v are matched in (P_s, P_t) . Let $M_u = (u, v, i, j)$ if u is in s , and $M_u = (v, u, i, j)$ otherwise. It is not hard to see that $M'' := (M' \setminus \{(x, y, i, j)\}) \cup \{M_u\}$ is also a fixed-match set satisfying $M'' \supseteq M$, and that (P_s, P_t) is a completion of M'' (we have merely changed the witness of s_i, t_j). Now, while iterating over recursive PCSP calls, $M \cup \{M_u\}$ is among the possible fixed-match sets that it can branch into (in particular, if u is in s , then $v = f_j(u)$ must hold, and if u is in t , then $u = f_j(v)$ must hold, as verified by the algorithm). Thus, the algorithm will either return “Yes” correctly before branching into $M \cup \{M_u\}$, or eventually branch into it. We know that $M \cup \{M_u\}$ leads to solution M'' and, by induction, we may assume that this recursive call will return a positive answer. Thus the algorithm will also return the correct positive answer in this case.

Consider the complexity of the algorithm. We can argue that each recursive call takes time $O(n)$. First, we must check that M is order-consistent. Note that M has $O(n)$ fixed-matches, and that we can sort M with respect to the order of appearance of their markers of s , from left to right, in time $O(n)$ (using e.g. bucket sort on the position of the markers). In this sorted order, we can then check that the i integers appear in strictly increasing order, as required by Definition 3. We can do the same sorting procedure for the markers of t , thereby checking order-consistency in time $O(n)$. The construction of $G(M)$ can also be done in time $O(n)$. Indeed, the values of each $l_M(v), r_M(v)$ can be computed in linear time for each marker v in one pass through s and t . We can then find the LL edges as follows. For each fixed-match of (x, y, i, j) of M that uses *id* or *flip*, we check whether $\{x - h, y - h\}$ should be an LL edge, starting with $h = 1$ and increasing it iteratively by 1. As long as $f_j(x - h) = y - h$, we mark $\{x - h, y - h\}$ as an LL edge, and we stop iterating when either: $f_j(x - h) \neq y - h$; one of $x - h$ or $y - h$ does not exist because it is out of the bounds of s or t ; or one of them is in another fixed-match of M . Since f_j is simple, this process of extending to the left as far as possible guarantees that $\{x - h, y - h\}$ is labeled LL if and only if $f_j(s[x - h, x]) = t[y - h, x]$ for the valid values of $x - h$ and $y - h$. This finds LL edges in time $O(n)$, since every marker is considered by at most one fixed-match edge. A similar process can be used to find LR, RR , and RL edges. Checking for odd paths in $G(M)$ can be done in linear time since it has $O(n)$ edges. Then, the linear time spent to generate each input sent to the recursive calls can be charged to the edges of the recursion tree. Finally, note that there is no additional complexity hidden in the PCSP algorithm.

Let us now bound the number of recursive calls made. When branching on an odd path, there are at most $2d$ markers on the path if $F \subseteq \{id, urev\}$, i.e. at most d in s and at most d in t that are all markers with the same symbol. If $F \subseteq \{id, flip, rev, urev\}$, there are at most $4d$ markers on the path, since we must count the symbol and its negation. Moreover, we assign j as one of the ℓ possible block numbers for the next fixed-match to be formed. Once this block number is chosen, we match u with at most d possible markers from the other string (this is true whether $f_j \in \{id, urev\}$ or whether $f_j \in$

$\{\text{flip}, \text{urev}\}$). Therefore, we always branch into at most $2d \cdot \ell \cdot d$ possibilities when $F \subseteq \{\text{id}, \text{urev}\}$, and at most $4d \cdot \ell \cdot d$ otherwise. We never make more than ℓ matches, so we have a search tree of degree at most $2d^2\ell$ and depth at most ℓ in the first case, and degree at most $4d^2\ell$ and depth at most ℓ in the second case. Since each call takes time $O(n)$, we get the claimed complexity. $\square$

4.3 Consequences.

Theorem 1 and Theorem 2 can be combined to deduce that several distances can be computed in FPT time with respect to parameter $k + d$. It suffices to express these distances as operations and plug in the complexity values.

Theorem 3 *Assume that s and t have at most d occurrences of the same character and let k be the distance between s and t , for a given distance. Then the following results hold:*

- the transposition distance can be computed in time $k^{O(k)}d^{6k+2}n$.
- the block interchange distance can be computed in time $k^{O(k)}d^{8k+2}n$.
- the flip, reversal and unsigned reversal distances can be computed in time $k^{O(k)}d^{4k+2}n$.
- the p -cut distance can be computed in time $(pk)^{O(pk)}d^{2k(p-1)+2}n$.
- for any subset P of operations among transpositions, block interchanges, reversals, unsigned reversals or flips, $d_P(s, t)$ can be computed in time $k^{O(k)}d^{8k+2}n$,

where the constants in the $O(k)$ exponent depend on the specific type of distance.

Proof We use Algorithm 1 to compute the distances, which in turn uses Algorithm 2 as a PCSP solver. Notice that Algorithm 1 only returns *yes* or *no* answers. To find the actual distance between s and t , we iterate from $i = 1$ to k and ask whether $d_P(s, t) \leq i$ for each such i . This requires k calls to Algorithm 1, which multiplies the complexity by k . We ignore this factor in our analysis and hide it in the $k^{O(k)}$ factor that is present in the complexity statements.

As for the claimed time bounds, first take notice that in every case, we will argue that $|\mathbb{H}\langle k \rangle|$ has constant size. We will therefore ignore the $n|\mathbb{H}||\mathbb{H}\langle k \rangle|^3$ term in the complexity stated in Theorem 1, as the $(|\mathcal{P}|p)^k(k(p-1)+p)^{k(p-1)}g(\ell, n)$ term will always dominate by assuming that $g(\ell, n) \in \Omega(n)$.

For the transposition distance, as we mentioned it can be modeled as an operation of the type $s_1|s_2|s_3|s_4 \rightarrow s_1|s_3|s_2|s_4$. This is an operation that uses $p = 4$ and $\mathbb{H} = \{\text{id}\}$ (which implies $|\mathbb{H}\langle k \rangle| = 1$). Given that PCSP can be solved in time $O(d^{2\ell}(2\ell)^\ell n)$, we can plug in the values in the statement of Theorem 1, using $\ell = 1 + k(p-1) = 3k+1$, and decide whether $d_P(s, t) \leq k$ in time $O(4^k(3k+4)^{3k}d^{2(3k+1)}(2 \cdot (3k+1))^{3k+1}n)$. This complexity is $k^{O(k)}d^{6k+2}n$.

For block interchange distances, we model them with operation $s_1|s_2|s_3|s_4|s_5 \rightarrow s_1|s_4|s_3|s_2|s_5$. This is the same as above, but with $p = 5$, which changes the complexity to $k^{O(k)}d^{2(4k+1)}n$.

For flips and reversals, we use the operation $s_1|s_2|s_3 \rightarrow s_1|f(s_2)|s_3$, where $f \in \{\text{flip}, \text{rev}, \text{urev}\}$. Here, $p = 3$ and $|\mathbb{H}| = 2$ since each block is either affected by id or f . We thus use $\ell = 2k + 1$ and the $O(d^{2\ell}(4\ell)^\ell n)$ time algorithm for PCSP. Moreover, $|\mathbb{H}(k)| = 2$ since composing id and f functions can only yield the identity or the reverse, for each of $f \in \{\text{flip}, \text{rev}, \text{urev}\}$. Theorem 1 gives a complexity of $O(3^k(2k+3)^{2k}d^{4k+2}(4 \cdot (2k+1))^{2k+1}n)$, which is $k^{O(k)}d^{4k+2}n$.

For the more general p -cut distance, we have $|\mathbb{H}| = |\mathbb{H}(k)| = 1$ and $\ell = 1 + k(p-1)$. We can plug in the values in Theorem 1 to get $O(p^k(k(p-1) + p)^{k(p-1)}d^{2(1+k(p-1))}(2 \cdot (1+k(p-1)))^{1+k(p-1)}n)$. Assuming $k > 0$, aside from the exponential factor with base d , the other factors can be upper-bounded by $(2pk)^{pk}$, which makes the contribution of these factors $(pk)^{O(pk)}$.

Finally, allowing $\mathcal{P}$ to contain any subset of transpositions, block interchanges, reversals, unsigned reversals and flips, we get that each operation $\mathcal{P}$ is a 5-operation. We thus use $p = 5$ and $\ell = 4k + 1$. Moreover, $|\mathcal{P}| \leq 5$, and $|\mathbb{H}(k)|$ is bounded by a constant $b \leq 4$ since composing id , flip , rev and urev can only transform a string in at most four possible ways (it can be seen that the four functions listed form a subgroup, so that from s only $\text{id}(s)$, $\text{flip}(s)$, $\text{rev}(s)$, or $\text{urev}(s)$ can be attained by composition). Plugging into Theorem 1, the complexity is bounded by $O((25)^k(4k+5)^{4k}d^{8k+2}(4 \cdot (4k+1))^{4k+1}n)$. This is $k^{O(k)}d^{8k+2}n$. $\square$

5 Conclusion

We studied a new version of the famous Minimum Common String Partition problem, called the Permutation-constrained Common String Partition (PCSP), where we are given two sequences s and t with the same content and a permutation π on $[\ell]$, and the problem is to decide whether the two sequence can be decomposed into ℓ blocks while agreeing with π . Although PCSP is W[1]-hard when parameterized by ℓ , we show that it is FPT for parameters ℓ and d , i.e. if s and t are both d -occurrence sequences. Moreover, we have shown that computing the distance between genomes exactly reduces to PCSP under several well-known string operations. This leads to FPT algorithms in parameters k and d , where k is the distance between the genomes. Our framework is general and support any finite combination of rearrangements that can be represent as permutations of blocks affected by simple string functions.

Let us mention that our PCSP algorithm made implicit use of the fact that some substrings of the partition could be empty. Indeed, this allowed us to stop branching immediately after finding a complete set of fixed-matches, even if the size of the common partition was strictly below ℓ . Although this is inconsequential for the rearrangement distances of interest in this work, requiring a partition into exactly ℓ non-empty substrings is an interesting question. It is not obvious that our algorithm could be extended to this case, as one would need a post-processing step to disentangle the missing blocks of a complete fixed-match set. We leave this question open, as well as the question of extending our framework to string functions that are not simple.

Acknowledgments. We sincerely thank the anonymous reviewers of this paper, who made several important suggestions on improving the complexity of our algorithms and on presenting the proofs in a clearer and more rigorous manner. They were a tremendous help in polishing the whole manuscript.

Conflict of interest. The authors declare no conflict of interest.

References

- Bulteau, L., Fellows, M., Komusiewicz, C., and Rosamond, F. (2021a). Parameterized string equations. In *arXiv*, number 2104.14171.
- Bulteau, L., Fertin, G., Jean, G., and Komusiewicz, C. (2021b). Sorting by multi-cut rearrangements. In *Proceedings of 47th International Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM'21)*, volume 12607 of *LNCS*, pages 593–607. Springer.
- Bulteau, L., Fertin, G., Komusiewicz, C., and Rusu, I. (2013a). A fixed-parameter algorithm for minimum common string partition with few duplications. In *International Workshop on Algorithms in Bioinformatics (WABI'13)*, pages 244–258. Springer.
- Bulteau, L., Fertin, G., and Rusu, I. (2012). Sorting by transpositions is difficult. *SIAM Journal on Discrete Mathematics*, 26(3):1148–1180.
- Bulteau, L., Fertin, G., and Rusu, I. (2013b). Maximal strip recovery problem with gaps: Hardness and approximation algorithms. *J. Discrete Algorithms*, 19:1–22.
- Bulteau, L. and Komusiewicz, C. (2014). Minimum common string partition parameterized by partition size is fixed-parameter tractable. In *Proceedings of 25th ACM-SIAM Symposium on Discrete Algorithms, SODA'14*, pages 102–121. SIAM.
- Caprara, A. (1999). Sorting permutations by reversals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12(1):91–110.
- Chen, X., Zheng, J., Fu, Z., Nan, P., Zhong, Y., Lonardi, S., and Jiang, T. (2005). Assignment of orthologous genes via genome rearrangement. *IEEE/ACM Trans. Computational Biology and Bioinformatics*, 2(4):302–315.
- Christie, D. A. (1996). Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165–169.
- Christie, D. A. and Irving, R. W. (2001). Sorting strings by reversals and by transpositions. *SIAM Journal on Discrete Mathematics*, 14(2):193–206.
- Chrobak, M., Kolman, P., and Sgall, J. (2005). The greedy algorithm for the minimum common string partition problem. *ACM Trans. Algorithms*, 1(2):350–366.
- Damaschke, P. (2008). Minimum common string partition parameterized. In *Proceedings of the 8th International Workshop on Algorithms in Bioinformatics (WABI'08)*, volume 5251 of *LNCS*, pages 87–98. Springer.

- Fertin, G., Labarre, A., Rusu, I., Tannier, E., and Vialette, S. (2009). *Combinatorics of Genome Rearrangements*. Computational molecular biology. MIT Press.
- Ganczorz, M., Gawrychowski, P., Jez, A., and Kociumaka, T. (2018). Edit distance with block operations. In *Proceedings of ESA'2018, LIPIcs*, volume 112, pages 33:1–33:14. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Goldstein, A., Kolman, P., and Zheng, J. (2005). Minimum common string partition problem: Hardness and approximations. *European J. Combinatorics*, 12.
- Jiang, H., Li, Z., Lin, G., Wang, L., and Zhu, B. (2012a). Exact and approximation algorithms for the complementary maximal strip recovery problem. *J. Comb. Optim.*, 23(4):493–506.
- Jiang, H., Zhu, B., Zhu, D., and Zhu, H. (2012b). Minimum common string partition revisited. *Journal of Combinatorial Optimization*, 23(4):519–527.
- Lafond, M., Liyanage, A., Zhu, B., and Zou, P. (2022a). The complexity of finding common partitions of genomes with predefined block sizes. In *RECOMB International Workshop on Comparative Genomics*, pages 105–123. Springer.
- Lafond, M. and Zhu, B. (2021). Permutation-constrained common string partitions with applications. In *Proceedings of the 28th International Symposium on String Processing and Information Retrieval (SPIRE'21)*, volume 12944 of *LNCS*, pages 47–60. Springer.
- Lafond, M., Zhu, B., and Zou, P. (2020). Genomic problems involving copy number profiles: Complexity and algorithms. In *Proceedings of CPM'2020, LIPIcs*, volume 161, pages 22:1–22:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
- Lafond, M., Zhu, B., and Zou, P. (2022b). Computing the tandem duplication distance is NP-hard. *SIAM Journal on Discrete Mathematics*, 36(1):64–91.
- Lothaire, M. (1997). *Combinatorics on words*, volume 17. Cambridge university press.
- Mahajan, M., Rama, R., Raman, V., and Vijaykumar, S. (2006). Approximate block sorting. *International J. of Foundations of Computer Science*, 17(2):337–356.
- Makaroff, C. and Palmer, J. (1988). Mitochondrial dna rearrangements and transcriptional alternatives in the male sterile cytoplasm of ogura radish. *Mol. Cell. Biol.*, 8:1474–1480.
- Qingge, L., He, X., Liu, Z., and Zhu, B. (2018). On the minimum copy number generation problem in cancer genomics. In *Proceedings of the 10th ACM Conference on Bioinformatics, Computational Biology, and Health Informatics, BCB'18*, pages 260–269, New York, NY. ACM Press.
- Sturtevant, A. (1926). A crossover reducer in *drosophila melanogaster* due to inversion of a section of the third chromosome. *Biol. Zent. Bl.*, 46:697–702.
- Sturtevant, A. and Dobzhansky, T. (1936). Inversions in the third chromosome of wild races of *drosophila pseudoobscura*, and their use in the study of the history of the species. *Proc. Nat. Acad. Sci. USA*, 22:448–450.

- Wang, L. and Zhu, B. (2010). On the tractability of maximal strip recovery. *J. Comput. Biol.*, 17(7):907–914.
- Zheng, C., Kerr Wall, P., Leebens-Mack, J., de Pamphilis, C., Albert, V. A., and Sankoff, D. (2009). Gene loss under neighborhood selection following whole genome duplication and the reconstruction of the ancestral populus genome. *Journal of Bioinformatics and Computational Biology*, 7(03):499–520.