On Deriving Synteny Blocks by Compacting Elements

Leonard Bohnenkämper*[1][0000−0003−4508−0078], Luca Parmigiani*[1][0000−0002−2139−3259], Cedric Chauve[2][0000−0001−9837−1878], and Jens Stoye[1][0000−0002−4656−7155]
* co-first authors

[1] Faculty of Technology and Center for Biotechnology (CeBiTec), Bielefeld University, Bielefeld, Germany
{leonard.bohnenkaemper,luca.parmigiani,jens.stoye}@uni-bielefeld.de
[2] Department of Mathematics, Simon Fraser University, Burnaby (BC), Canada
cedric.chauve@sfu.ca

Abstract. Genomic rearrangements are major drivers of evolution and genetic disease. However, studying rearrangements requires segmenting the genomes of interest into conserved regions, called synteny blocks, that highlight structural differences between genomes. Synteny blocks are typically defined from annotated genes or derived as a by-product of whole-genome alignments. As these procedures are heuristic and do not explicitly model rearrangements, they can obscure real variation, create false similarities, and affect phylogenetic inference.
The importance of synteny block definition has long been recognized, as shown for example by discussions on breakpoint reuse, where different definitions of synteny blocks led to different estimates of rearrangement complexity in mammalian genomes.
We present a formal framework for deriving synteny blocks directly from sequence data by partitioning genomic elements into blocks that do not contain breakpoints. A breakpoint is defined between a pair of genomes as an adjacency of shared elements that occurs in one genome but not in the other. Synteny blocks are therefore not allowed to span such boundaries, ensuring that rearrangements are not obscured. The framework is fully agnostic to the type of genomic element and applies to any genome representation expressed as sequences of elements, such as non-overlapping alignments, exact matches (MUMs/MEMs), k-mers, unitigs or minimizers.
We formalize two optimization problems: minimizing the total genome length after replacement by synteny blocks (the Minimum-Length Synteny Block Problem) and minimizing the number of distinct blocks (the Minimum-Size Synteny Block Problem). We show that both problems are NP-hard in general. However, when blocks are required to be collinear and to contain a conserved element, we provide a linear-time algorithm with respect to the number of input elements that simultaneously minimizes both objectives. The resulting method is simple, efficient, and produces large synteny blocks without obscuring rearrangements.

Keywords: Comparative genomics · Rearrangements · Synteny blocks.

1 Introduction

Comparative genomics aims to detect and explain similarities and differences between groups of evolutionarily related genomes; examples of such analyses include genome rearrangement studies or phylogeny reconstruction, to cite only two examples. With the availability of thousands of fully assembled genomes, from small viral genomes to large and complex genomes of eukaryotic species, it is now common for comparative studies to consider tens to hundreds of genomes at once. The size and complexity of most genomes prevents that whole-genome comparative studies can be done at the nucleotide level, and a preliminary step is to abstract the considered genomes into more compact representations over an "alphabet" of conserved genome segments called *synteny blocks*, a term coined by Pevzner and Tesler in [1].

Intuitively, synteny blocks partition genomes into non-overlapping segments, each segment belonging to a unique synteny block, such that both differences between segments from the same synteny block and similarities between segments from different synteny blocks can be ignored for the comparative analysis that is done. For example, when comparing genomes to understand their evolution in terms of large-scale genomic rearrangements, synteny blocks aim to be large homologous genome segments that are likely to have evolved from a single ancestral genome segment through nucleotide-level mutations, small-scale duplications and losses, and micro-rearrangements [1, 2, 3, 4]. In practice, synteny blocks are determined from small,

highly-conserved genome segments called *elements* (e.g., orthologous genes, unitigs, local pairwise or multiple sequence alignments), that are clustered into synteny blocks according to criteria that are typically parameterized by conservation of content and order; for example, locally collinear synteny blocks would require that all genome segments in a synteny block contain the set of elements in the same order and orientation.

Since the initial comparison between the human and mouse genomes, there has been a large corpus of methods for the computation of synteny blocks, using a wide variety of approaches [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]. Most methods are inherently heuristic, in that they do not rely on a formal definition of what a synteny block is; we refer the reader to the recent papers [20, 21, 23] for an overview of used approaches.

As a consequence of the many ways synteny blocks can be defined, the design of algorithms to identify synteny blocks is both challenging and prone to controversy. This is well illustrated by the debate on the rate of breakpoint reuse by genomic rearrangements which stemmed from initial comparisons of the human and mouse genomes [24, 25, 26, 27], and by two comparative studies of synteny block methods [28, 29]. In [29], which focused on rearrangement-based phylogenetic reconstruction, it was shown that "the tree inference method had less impact than the block determination method"; in [28], Ghiurcuta and Moret argued for "the need for a well founded, systematic approach to the decomposition of genomes into syntenic blocks."

In this work, we introduce a formal definition of synteny blocks as groups of genomic elements that do not contain breakpoints, where a breakpoint is defined between a pair of genomes as an adjacency of shared elements that occurs in one genome but not in the other. This definition ensures that genome rearrangements are not obscured. Our framework is fully agnostic to the type of elements and applies to any genome representation expressed as sequences of elements, such as genes, exact matches (MUMs/MEMs), k-mers, or unitigs. We then formalize the computation of synteny blocks as two optimization problems. The first one aims to find a set of synteny blocks minimizing the total length of the genomes after they are abstracted as (signed) sequences of synteny blocks (the Minimum-Length Synteny Block Problem), while the second one aims to find a set of synteny blocks minimizing the number of blocks (the Minimum-Size Synteny Block Problem). We show that both problems are NP-hard in general, but that when blocks are required to be collinear and to contain a conserved element, i.e., an anchor present in all genome segments forming a synteny block, both problems can be solved simultaneously in linear time with respect to the input size, measured as the sum of the elements spelling all genomes. The resulting approach is conceptually simple, runs efficiently, and preserves rearrangements while producing large synteny blocks.

2 Problem Definition

We formally define *synteny blocks* and the related algorithmic problems. For the sake of exposition, we consider unichromosomal linear genomes, although our definitions naturally extend to general genomic architectures.

Genomes and strings. Let E be a set of *elements*, where each element typically represents a family of genomic regions (e.g., k-mers, unitigs, orthologous genes).

A *string* $s = s[1] \dots s[n]$ of length n over E is a sequence of *signed* elements, each of the form $s[i] = oe$, where $e \in E$ and $o \in \{+, -\}$; we write $\mathrm{sgn}(s[i]) = o$ and $\mathrm{elm}(s[i]) = e$. We use $-x$ to indicate a reversal of a signed element x, i.e., $-+e = -e$ and $--e = +e$. The set of unsigned elements in a string s is denoted by $\mathrm{elm}(s) = \{\mathrm{elm}(s[i]) \mid i = 1, \dots, n\}$. We denote the *substring* of string s between positions i and j by $s[i : j] = s[i] \dots s[j]$.

For the moment we consider only strings in which each element occurs once, i.e., we assume that repeats have been masked. Such strings are known as *singular strings*. Our results can be generalized for arbitrary strings, with repeated elements, as outlined in Appendix A.

Adjacencies and Breakpoints. Rearrangements are usually characterized by their *breakpoints*. Because we aim to define blocks that reflect rearrangements and therefore breakpoints accurately, the definition of breakpoints is central to our notion of synteny blocks.

Each signed element $x = oe$ is associated with two *extremities*: a *tail* e^t and a *head* e^h. The order of these extremities depends on the sign of x: if $o = +$, the extremities are ordered (e^t, e^h); if instead $o = -$, the order of the extremities is reversed, (e^h, e^t).

The *adjacency set* of s, $\mathrm{Adj}(s)$, is the set of unordered pairs of extremities connecting consecutive elements:

$$\mathrm{Adj}(s) = \{\, \{\mathrm{elm}(s[i])^x, \mathrm{elm}(s[i+1])^y\} \mid 1 \leq i < n \,\} \cup \{\{\$, \mathrm{elm}(s[0])^x\}, \{\mathrm{elm}(s[n])^y, \$\}\},$$

where, $x = h$ when $\mathrm{sgn}(s[i]) = +$ and $x = t$ otherwise, and $y = t$ when $\mathrm{sgn}(s[i+1]) = +$ and $y = h$ otherwise. We call \$ the *telomere*.

To compare two strings s and t with unequal content, we project them onto their common elements as follows: the *projection* of s onto t, denoted $s|_t$, is the string obtained from s by removing every $s[i]$ for which $\mathrm{elm}(s[i])$ is not contained in $\mathrm{elm}(t)$. Rearrangements can then be characterized by their *breakpoints*. The *breakpoint set* between two strings s and t is the set of adjacencies present in only one of the two strings:

$$\mathrm{Bp}(s,t) = \mathrm{Adj}(s|_t) \cup \mathrm{Adj}(t|_s) \setminus (\mathrm{Adj}(s|_t) \cap \mathrm{Adj}(t|_s)).$$

Parsings and Partitions. We aim to define synteny blocks as a partition of the set of elements E. Recall that a partition $P = \{P_1, \ldots, P_k\}$ is a set of non-empty sets P_i, called *parts*, such that each element is contained in exactly one part. We call the partition $P = \{\{e\} \mid e \in E\}$, the *finest partition*.

Given a partition P, the maximal substrings in s consisting of elements from the same part P_i form a parsing of s. More formally, given a string s, a *parsing* of s is a decomposition into substrings $s[1 : r_1], s[r_1+1 : r_2], \ldots, s[r_{l-1}+1 : r_l]$. Each $s[r_{i-1}+1 : r_i]$ of a parsing is called a *phrase*. The parsing *induced* by partition P is the parsing, such that $s[i], s[i+1]$ are in the same phrase if and only if $\mathrm{elm}(s[i])$ and $\mathrm{elm}(s[i+1])$ are in the same part of P. An example is given in Fig. 1.

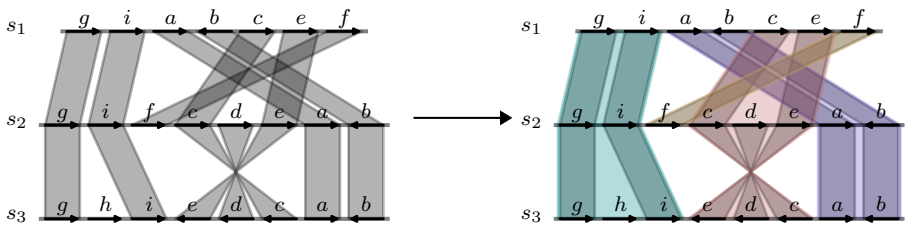

Fig. 1. Illustration of the parsing induced by partition $P = \{\{a,b\},\{c,d,e\},\{f\},\{g,h,i\}\}$ for three strings; the corresponding phrases are indicated by four different colors. The parsed strings are $+g+i| + a-b| + c+e| + f$, $+g+i| + f| + c+d+e| + a-b$, and $+g+h+i| - e-d-c| + a-b$. One possible way of orienting the phrases is to view each as forward except $-e-d-c$ in s_3 which one considers backward.

In the following we define synteny blocks as partitions of E such that (i) phrases of distinct parts do not interleave, (ii) synteny blocks do not contain internal rearrangements (are free of breakpoints), and (iii) phrases can be assigned an orientation. The partition in Fig. 1 fulfills these properties.

Synteny blocks. First, since our input strings are presumed to be singular, we expect synteny blocks to occur only once per string as well, uninterrupted by other blocks. An example where this is not the case is given in Fig. 2 (i). More formally: Given a part P_i and a signed string s, the *index set* of P_i in s is $\{j \mid \mathrm{elm}(s[j]) \in P_i\}$; P is *absent* from s if its index set is empty, *present* otherwise. We call P_i *contiguous* in s if it is absent from s or its index set defines a single substring $s[i : j]$ of s. We then write $s^{P_i} = s[i : j]$ and call s^{P_i} the phrase *induced* by P_i. For a set of strings S, we write $S^{P_i} = \{s^{P_i} \mid s \in S, \text{ if } P_i \text{ is present in } s\}$.

Second, since we want synteny blocks to reflect the rearrangements in the original strings, we want blocks not to contain breakpoints like shown in Fig. 2 (ii). A contiguous part P_i is called *breakpoint free* on string set S if for any two strings $s, t \in S$, the induced phrases s^{P_i} and t^{P_i} have no breakpoints with respect to each other.

Third, to reflect rearrangements in the original strings, the phrases induced by synteny blocks need to be assigned an orientation. However, this is not always possible, even when the corresponding part is breakpoint free, as shown in Fig. 2 (iii). A part P_i is called *orientable* on string set S if for each of its induced phrases in S^{P_i}, we can assign an orientation $f : S^{P_i} \to \{+, -\}$, such that applying this orientation to $s \in S^{P_i}$, i.e., inverting it if $f(s) = -$, each element occurs only with one orientation. More formally, if $+e \in \{f(s)s[i] \mid s \in S^{P_i}\}$, then $-e \notin \{f(s)s[i] \mid s \in S^{P_i}\}$ and conversely. We call f an *orienting function* of P_i. Note that this is only required for the same element, i.e. different elements may have different signs according to f; for example $\{a, b\}$ in Figure 1 is orientable. Also note that negating f, one can always obtain another orienting function g of P_i, i.e., $g(s) = -f(s)$.

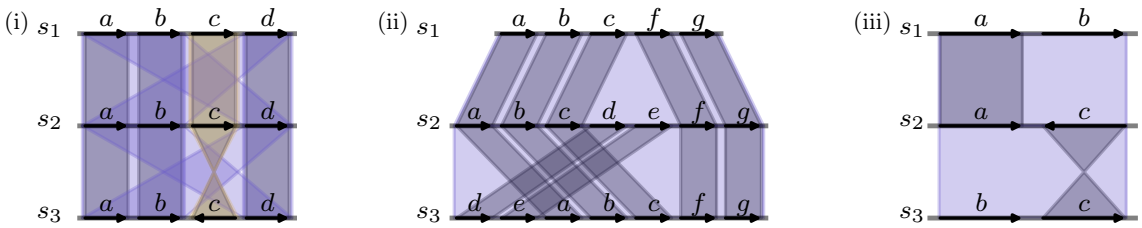

Fig. 2. Critically undesirable properties in synteny blocks. (i) The partition $\{\{a, b, d\}, \{c\}\}$ contains the part $\{a, b, d\}$ which is not contiguous. (ii) This block contains a transposition of elements d and e between s_2 and s_3. (iii) The block $\{a, b, c\}$ is not orientable. Trying to give this block an orientation leads to a contradiction: If it is forward in s_1, then it is forward in s_2 because of a, but also forward in s_3 because of b. However, since it is forward in s_2, it must also be backward in s_3 because of c, a contradiction.

We now formally define synteny blocks and the problem(s) of finding optimal sets of synteny blocks.

Definition 1 (Synteny block). *Let S be a set of strings over the set of elements E and $P = \{P_1, \ldots, P_k\}$ a partition of E. Part P_i is a* synteny block *if all of the following hold: (i) for all $s \in S$, P_i is contiguous in s, (ii) P_i is breakpoint free on S, and (iii) P_i is orientable on S. If each $P_i \in P$ is a synteny block, we call P a* synteny block partition *for S.*

For example, the partition illustrated on the right side of Fig. 1 is a synteny block partition.

As noted above, the phrases induced by blocks of P in each $s \in S$ define a unique parsing which we denote by $\rho_P(s)$. We denote by $\rho_P(S)$ the parsings of all strings in S induced by P. The parsing of S induced by P naturally defines an encoding of the strings in S as strings over the alphabet $\{1, \ldots, k\}$ where, for every string $s \in S$, its encoding $\mathrm{enc}_P(s)$ is obtained from s by replacing, for every part P_i, the phrase induced by P_i (if P_i is present in s) with $+i$ (resp. $-i$) if P_i is forward (resp. reversed) in s according to an orienting function f_{P_i}.

Note that the finest partition, i.e., the one where each part contains exactly only one element, is trivially a synteny block partition. However, this is also the largest partition, both in number of parts as well as regarding the resulting encoding. Since downstream rearrangement analyses may be computationally demanding, it makes sense to search for the smallest synteny block partition instead. We consider two different criteria for what constitutes a "smallest" partition.

Problem 1. Given a collection of strings $S = \{s_1, \ldots, s_m\}$ over an element set E, find a synteny block partition P for S such that one of the following criteria is optimized:

- *Minimum-Length Synteny Blocks Problem (MLSBP)*: minimize the total encoded length $\sum_{i=1}^{m} |\mathrm{enc}_P(s_i)|$;
- *Minimum-Size Synteny Blocks Problem (MSSBP)*: minimize the number of parts $|P|$.

Problem 1 is already meaningful in its original form, but in practice it is often desirable for the resulting synteny block partition to satisfy additional properties. We discuss these next.

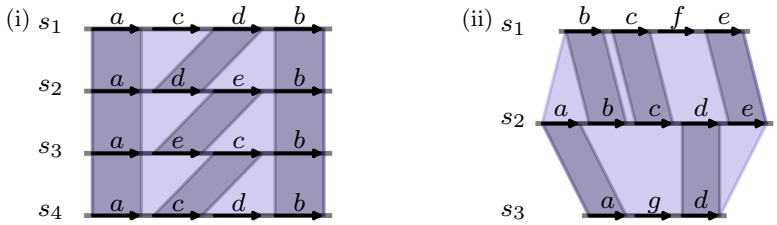

Fig. 3. Potentially undesirable properties of synteny blocks. (i) Despite being breakpoint free, this block does not have a (partial) ordering between elements c, d and e; it is thus not collinear. (ii) The phrases induced by this block in s_1 and s_3 do not share any elements.

Desirable Properties of Synteny Blocks. While enforcing breakpoint-freeness and orientability is sufficient to ensure that pairwise rearrangements are not hidden within synteny blocks, it is not sufficient to do so in a global comparison on strings with unequal content. Consider Fig. 3 (i): while none of the three strings s_1, s_2, s_3 has a breakpoint with respect to any other one, there is clearly disagreement about what the order of $+c$, $+d$ and $+e$ should be, indicating that there must have been a rearrangement at some point during the emergence of these "chromosomes". Additionally, there is no multiple alignment of the phrases induced by this synteny block that respects all element matches. Below, we introduce the concept of *collinearity*, which ensures that such a situation does not occur.

Definition 2 (Collinear). *Let S be a set of strings. A synteny block P_i with orienting function f is* collinear *in S if there exists a partial order $\prec$ on the signed elements in P_i such that, for every $s \in S$, $s^{P_i} = s^{P_i}[1] \dots s^{P_i}[k]$, for all $j, 1 \leq j < k$: $s^{P_i}[j] \prec s^{P_i}[j+1]$, if $f(s^{P_i}) = +$, or $s^{P_i}[j+1] \prec s^{P_i}[j]$, if $f(s^{P_i}) = -$.*

Collinearity implies breakpoint-freeness, as stated in the following lemma.

Lemma 1. *Let S be a set of strings over element set E, and P_i a contiguous, orientable part of a partition on E. If P_i is collinear in S, then P_i is breakpoint-free on S.*

Proof. (By contradiction) Assume for two strings s, t, the induced phrases s^{P_i}, t^{P_i} have a breakpoint with respect to each other. That is, for some elements $a, b \in p$, $s^{P_i}|_{t^{P_i}}$ contains $a^x b^y$ for some $x, y \in \{h, t\}$ and $t^{P_i}|_{s^{P_i}}$ does not. Without loss of generality, let $x = h$ and $y = t$.

We distinguish two cases: (I) $t^{P_i}|_{s^{P_i}}$ contains $a^h b^t$. (II) $t^{P_i}|_{s^{P_i}}$ contains $a^h c^z$ for some other shared element c with $z \in \{h, t\}$.

In case (I) we have $s^{P_i} = \dots + aX + b \dots$ and $t^{P_i} = \dots + aY - b \dots$ for substrings of non-shared elements X, Y. Then P_i is not orientable.

In case (II) we have $s^{P_i} = A + aX + bB$ and $t^{P_i} = W + aY \pm cC$ for some substrings A, X, B, W, Y, C where elements in X, Y are not shared between s^{P_i}, t^{P_i}. Therefore c does not occur in X and b does not occur in Y. Since both strings agree on the sign of a, we know the partial order should either rise or fall from left to right in both strings (see Def. 2). W.l.o.g. let the partial order rise. We further distinguish three cases.

(i) Let b occur in W. Then t^{P_i} implies the order: $b \prec a \prec c$. However, s^{P_i} implies $a \prec b$, a contradiction.

(ii) Let b occur in C and c occur in A. Then follows from the order in s^{P_i}: $c \prec a \prec b$, which contradicts with $a \prec c$ in t^{P_i}.

(iii) Let b occur in C and c occur in B. Then in s^{P_i} we have $a \prec b \prec c$ while in t^{P_i} we have $a \prec c \prec b$, a contradiction. □

Additionally, one may wish to avoid cases where completely unrelated phrases are induced by the same block, such as in Fig. 3 (ii). This can create false signals of homology or rearrangement. Other problems, such as inconsistent orientation (Fig. 2 (iii)) can also arise when phrases induced by a block share no common elements. To prevent this, we require an *anchor* element.

Definition 3 (Anchored). *Let S be a set of strings and $P = \{P_1, \dots, P_k\}$ a set of synteny blocks for S. We say that P_i is* anchored *if $\bigcap_{s \in S} \mathrm{elm}(s^{P_i})$ is not empty. We call any $e \in \bigcap_{s \in S} \mathrm{elm}(s^{P_i})$ an* anchor *of P_i. We say that P is anchored if all synteny blocks P_i are anchored.*

Requiring anchored synteny blocks has additional benefits, some of which will be discussed later. Importantly for subsequent analyses, it enables a bijection between breakpoints in the input strings and in the encoded strings which implies preservation of genome rearrangement distances.

Theorem 1. *Let P be an anchored synteny block partition for a set S of singular strings. For each pair of strings $s, t \in S$ there is a bijection $\beta_{s,t}$ between extremities in $\mathrm{Bp}(s,t)$ and in $\mathrm{Bp}(\mathrm{enc}_P(s), \mathrm{enc}_P(t))$, such that*

- *for each breakpoint $\{a, b\} \in \mathrm{Bp}(\mathrm{enc}_P(s), \mathrm{enc}_P(t))$, $\{\beta_{s,t}(a), \beta_{s,t}(b)\}$ is a breakpoint of s, t,*
- *for each breakpoint $\{c, d\} \in \mathrm{Bp}(s,t)$, $\{\beta^{-1}_{s,t}(c), \beta^{-1}_{s,t}(d)\}$ is a breakpoint of $\mathrm{enc}_P(s), \mathrm{enc}_P(t)$.*

Proof. We first define the mapping $\beta_{s,t}$ for each extremity of a block P_i. Let s^{P_i}, t^{P_i} be the phrases induced by p in s and t respectively. Both s^{P_i} and t^{P_i} contain the anchor, therefore they share at least one element. Let a be the first element in s^{P_i} that also occurs in t^{P_i}. Because s^{P_i} does not have any breakpoints with t^{P_i}, a must either be the first element in t^{P_i} and occur with the same orientation or occur in reverse orientation and as the last shared element of t^{P_i}. For the same reason, the last shared element b in s^{P_i} is the last shared element in t^{P_i} or occurs in reverse orientation as the first shared element in t^{P_i}. Without loss of generality, s^{P_i} be a forward occurrence of P_i. In $\beta_{s,t}$, we then map i^t to the first extremity of a and i^h to the last extremity of b.

Consider an adjacency $\{i^x, j^y\}$ in $\mathrm{enc}_P(s)|_{\mathrm{enc}_P(t)}$. Then $\{\beta_{s,t}(i^x), \beta_{s,t}(j^y)\}$ is an adjacency in $s|_t$. To see why, regard $\mathrm{enc}_P(s)$ and s. Let $s', s'_{\mathrm{enc}}, t', t'_{\mathrm{enc}}$ be the string obtained by replacing each signed element with its extremities in $s, \mathrm{enc}_P(s), t$ and $\mathrm{enc}_P(t)$ respectively.

We have $s'_{\mathrm{enc}} = \ldots i^x U j^y \ldots$ and accordingly, $s' = \ldots \beta_{s,t}(i^x) V \beta_{s,t}(j^x) \ldots$ for some substrings U, V. For the sake of contradiction, assume V contains extremities that occur in t'. Then U contains extremities that occur in t'_{enc}. However, then $\{i^x, j^y\}$ is not an adjacency in $\mathrm{enc}_P(s)|_{\mathrm{enc}_P(t)}$. Thus, $\{\beta_{s,t}(i^x), \beta_{s,t}(j^y)\}$ is an adjacency in $s|_t$. An analogous argument holds for adjacencies of $\mathrm{enc}_P(t)|_{\mathrm{enc}_P(s)}$. Since adjacencies are a superset of breakpoints, each breakpoint $\{i^x, j^y\}$ between $\mathrm{enc}_P(s)$ and $\mathrm{enc}_P(t)$ has a corresponding a breakpoint $\{\beta_{s,t}(i^x), \beta_{s,t}(j^y)\}$ between s, t.

Now regard a breakpoint $\{a^x, b^y\}$ between s, t. W.l.o.g. let $\{a^x, b^y\}$ be an adjacency in $s|_t$, but not in $t|_s$. Then a, b are in different parts of P. Otherwise the part would not be breakpoint free or not be contiguous.

Let thus a, b be in different parts $P_i, P_j \in P$. Observe the phrases of P_i and P_j in s. Let again $s', s'_{\mathrm{enc}}, t', t'_{\mathrm{enc}}$ be the string obtained by replacing each signed element with its extremities in $s, \mathrm{enc}_P(s), t$ and $\mathrm{enc}_P(t)$ respectively.

Then, $s' = \ldots a^x U_1 U_2 U_3 b^y \ldots$ for some substrings U_1, U_2, U_3 of U, where the phrase induced by P_i includes $a^x U_1$ and the phrase induced by P_j includes $U_3 b^y$. Since $U = U_1 U_2 U_3$ does not have any extremities that occur in t', $\beta_{s,t}(i^{x'}) = a^x$ and $\beta_{s,t}(j^{y'}) = b^y$ for some $x', y' \in \{h, t\}$. Note that since U_2 does not contain any elements in t, there are no synteny blocks between P_i and P_j that are also present in t. Thus $\{i^{x'}, j^{y'}\}$ is an adjacency in $\mathrm{enc}_P(s)|_{\mathrm{enc}_P(t)}$.

To see why it is a breakpoint, consider that in t, $\{a^x, b^y\}$ is not an adjacency in $t|_s$. We distinguish two cases: (I) $\{a^x, b^w\}$ is an adjacency in t for $w \neq y$ and (II) $\{a^x, c^z\}$ is an adjacency in $t|_s$ for some other element c with $z \in \{h, t\}$.

In Case (I), $\mathrm{enc}_P(t)|_{\mathrm{enc}_P(s)}$ contains the adjacency $\{i^{x'}, j^{w'}\}$ with $w' \neq y'$. Therefore, $\{i^{x'}, j^{y'}\}$ is a breakpoint. In Case (II), c must be in a third part P_l as otherwise P_i or P_j would not be breakpoint free. Then $\mathrm{enc}_P(t)|_{\mathrm{enc}_P(s)}$ contains some adjacency $\{i^{x'}, l^{z'}\}$. Therefore, $\{i^{x'}, j^{y'}\}$ is a breakpoint. □

The bijection between breakpoints is a powerful property, as many rearrangement models – such as the Reversal, Hannenhalli-Pevzner, Single-Cut-or-Join and Double-Cut-and-Join models – consider only breakpoints, not conserved adjacencies. This means that distances under such models are preserved for an anchored synteny block partition.

Finally, we propose the following variant of Problem 1, restricted to synteny blocks with these desirable properties, which plays a central role in our results.

Problem 2. Given a collection of strings $S = \{s_1, \ldots, s_m\}$ over an element set E, find a synteny block partition P for S in which all blocks are collinear and anchored, and which is optimal with respect to either the partition size or the encoded length, as defined in Problem 1.

3 Methods

With these definitions in place, we now consider the computational complexity of the problem and propose an optimal algorithm for the restricted version. First, we prove that both objectives in Problem 1 (MLSBP and MSSBP) are NP-hard in general. Then we consider the restricted case of Problem 2 where blocks must be collinear and anchored. Under these constraints, both problems have the same optimal solution, and this solution can be computed in linear time by a simple greedy merging procedure.

3.1 Finding a Smallest Synteny Block Partition is NP-hard

The optimization problems in Problem 1 admit no efficient solution unless P = NP. We establish this via two polynomial-time reductions: MLSBP is shown NP-hard by a reduction from Vertex Cover, and MSSBP is shown NP-hard by a reduction from SAT.

MLSBP is NP-hard. Here, we show that MLSBP is NP-hard via a reduction from vertex-cover. Let $G = (V, E)$ be a graph and let r be a non-negative integer. The Vertex Cover decision problem asks whether there exists a subset $C \subseteq V$ with $|C| \leq r$ such that every edge in E is incident to at least one vertex in C. We describe a polynomial-time reduction from this problem to an instance S_G of our problem MLSBP. For each vertex $v_i \in V$, we construct a gadget represented by a two-character string

$$X_i = xv_i,$$

where x is a common character shared among all vertex gadgets. The signs are always positive, therefore we omit them. For each edge $e_j = (v_p, v_q) \in E$, we construct two gadgets

$$Y_j = \{v_p v_q, v_q v_p\},$$

which introduce a breakpoint between the two corresponding vertex strings. The complete instance is then defined as the union of those gadgets $S_G = X_1 \cup X_2 \cup \ldots \cup X_{|V|} \cup Y_1 \cup Y_2 \cup \ldots \cup Y_{|E|}$. Since each edge gadget has a conflict they need to be separated into two different partitions, therefore into two different phrases each, i.e., $v_p|v_q, v_q|v_p$. Each vertex gadget can be either split into two phrases or kept together. We refer to a vertex where its gadget is split into two phrases as *selected*. With the following theorem we prove that MLSBP is NP-hard.

Theorem 2. *Let $G = (V, E)$ be a graph and r a non-negative integer, and let S_G be the set of strings constructed as above. Then G has a vertex cover of size at most r if and only if there exists an MLSBP of S_G such that the total number of phrases is $|V| + 4|E| + r$.*

Proof. If two nodes v_p and v_q are connected by an edge, they cannot be placed in the same partition, since they form a breakpoint. Therefore, at least one of the two vertices must be selected. If both are selected, they can be assigned to different partitions, as we aim to minimize the number of phrases and can use as many partitions as needed.

Let $\tau(G)$ denote the size of a minimum vertex cover of G. If $\tau(G) \leq r$, we can select $\tau(G)$ vertices in S_G, and the total number of phrases is $r' = |V| + 4|E| + \tau(G) \leq |V| + 4|E| + r$. This is because each edge contributes four phrases, and each selected vertex adds one additional phrase. This constitutes a valid solution for S_G, since the complement of a vertex cover is an independent set, so no two vertices in it share an edge. Therefore, the character x is in a partition only with characters that do not share an edge.

If $\tau(G) > r$, there are two possible cases: (1) if we select more than r vertices, it is clear that $r' > |V| + 4|E| + r$; (2) if we select at most r vertices, there remain $\tau(G) - r$ vertices that still share an edge, and therefore cannot be placed in the same partition as the character x, leading to a contradiction. □

MSSBP is NP-hard. In addition to MLSBP, MSSBP is also NP-hard. We show the reduction from the Boolean satisfiability problem (SAT).

In short, SAT asks whether a formula F in propositional logic can be satisfied. This formula is built over an alphabet of variables V, utilizing each $x \in V$ as *literals*, i.e., as "positive" x or "negative" $\neg x$. These literals

are connected with logical operators $\vee$ ("or") and $\wedge$ ("and"). Each variable can be assigned a Boolean value, i.e., "true" or "false".

The formula F is given in *conjunctive normal form*, that is $F = c_1 \wedge c_2 \ldots \wedge c_m$ where each *disjunctive clause* c_i consists of disjunctions of variable literals $c_i = l_i^1 \vee l_i^2 \ldots \vee l_i^{m_i}$, where each $l_i^j = x$ or $l_i^j = \neg x$ for some variable $x \in V$. The question asked in SAT is whether there is an assignment of Boolean values ("true", "false") to each variable x, such that F is true. SAT is NP-complete.

Theorem 3. *For any given SAT formula F in CNF with n variables and m clauses, there is an instance of MSSBP with size $\mathcal{O}(nm + n^2)$, such that the smallest synteny block partition has size $2n$ if and only if F is satisfiable. All synteny blocks in this partition are then also collinear.*

Proof. In this proof, all elements have positive orientation. For the sake of readability, we thus omit their (positive) signs.

Note that we can ensure that two elements e, d are in different partitions by adding the chromosomes ed and de. We write $\texttt{DIFF}(d, e) = \{de, ed\}$.

For each variable x_i, we add three elements, namely $\texttt{t}_i, \texttt{f}_i, \texttt{v}_i$. We model the boolean value of each variable x_i by grouping $\texttt{v}_i$ either with $\texttt{t}_i$ ("true") or $\texttt{f}_i$ ("false"). To ensure a variable is either true or false, we add the chromosomes $\texttt{DIFF}(\texttt{t}_i, \texttt{f}_i)$. For each pair of different variables $x_i \neq x_j$, we add the chromosomes $\texttt{DIFF}(a, b)$ for all $a \in \{\texttt{t}_i, \texttt{f}_i, \texttt{v}_i\}$ and all $b \in \{\texttt{t}_j, \texttt{f}_j, \texttt{v}_j\}$ to ensure distinct parts for each variable. Note that there are now at least $2n$ parts in any synteny block partition.

For each disjunctive clause c_i, we add an element $\texttt{c}_i$. For each clause c_i and variable x_j, we add $\texttt{t}_j\texttt{v}_j\texttt{c}_i$ if variable x_j occurs as a positive literal x_j in c_i and $\texttt{DIFF}(\texttt{t}_j, \texttt{c}_i)$ otherwise. We add $\texttt{c}_i\texttt{v}_j\texttt{f}_j$ if variable x_j occurs as a negative literal $\neg x_j$ in c_i and $\texttt{DIFF}(\texttt{f}_j, \texttt{c}_i)$ otherwise. In cases where a disjunctive clause contains both x_j and $\neg x_j$, we omit the clause as it is automatically satisfied by any Boolean assignment.

Note that if a clause element $\texttt{c}_i$ is grouped with $\texttt{t}_j$ (or $\texttt{f}_j$), due to contiguousness $\texttt{v}_j$ is also in $\texttt{t}_j$ (or $\texttt{f}_j$), i.e., the clause is satisfied by the corresponding boolean assignment to x_j. If there is a partition with $2n$ parts, we can thus derive a satisfying assignment of boolean values to all x_j.

Conversely, given a satisfying assignment to F, we can obtain a partition by grouping all $\texttt{v}_j$ with their corresponding boolean value and each $\texttt{c}_i$ with any variable that satisfies it, obtaining a partition of the minimum size $2n$.

Note that in each case, we have the partial order $\texttt{t}_i \prec \texttt{v}_i \prec \texttt{c}_j$ for $\texttt{c}_j$ being grouped with $\texttt{t}_i$ or $\texttt{c}_j \prec \texttt{v}_i \prec \texttt{f}_i$ for $\texttt{c}_j$ being grouped with $\texttt{f}_i$ that satisfies Definition 2. The blocks are thus collinear.

Moreover, the MSSBP instance is comprised of $\mathcal{O}(nm + n^2)$ strings, each with at most 3 elements. The size of the MSSBP instance is thus $\mathcal{O}(nm + n^2)$. □

3.2 Finding a Smallest Collinear, Anchored Synteny Block Partition is Polynomial

Although the unrestricted problems are hard, when adding the constraints that blocks should be collinear and have (at least) one anchor, the problem is much easier to solve. In fact, as we show in this section, the solution for both problems is the same.

We first formalize the notion that when every occurrence of an element forms an adjacency with the same element, they should belong to the same synteny block.

Definition 4 (Unique neighbor). *We say that extremity a^x has a* unique neighbor *extremity b^y on string set S, for some elements a, b and some $x, y \in \{h, t\}$, if $\{a^x, b^y\}$ is the only adjacency involving a^x on S. We write $a^x \to b^y$. When the concrete extremities x, y are irrelevant, we simply write $a \to b$.*

For two parts P_i, P_j of partition P, we write $P_i \to P_j$ if in any encoding $\mathrm{enc}_P(S)$, $i \to j$.

Note that the unique neighbor relationship is directed, i.e., for $a \to b$, whenever a occurs, b occurs adjacent to it, but the converse is not necessarily true. This is in contrast to the concept of compaction employed in compacted de Bruijn graphs, where two k-mers a, b are placed in the same unitig if a always occurs adjacent to b, but also b always occurs adjacent to a. In our notation, this would be $a \to b$ *and* $b \to a$. We also note that if $i^x \to j^y$ for some $x, y \in \{h, t\}$ in an encoding of a partition P with $P_i, P_j \in P$, we can always choose the orienting functions of P_i, P_j, such that $i^h \to j^t$.

We now show that one can create a set of collinear anchored synteny blocks by iteratively merging smaller (also collinear anchored) synteny blocks, where one of them is a unique neighbor of the other.

Lemma 2. *Let P_i, P_j be collinear synteny blocks of partition P with anchors c_i, c_j, such that $P_i \to P_j$. Then $P_i \cup P_j$ is a collinear synteny block with anchor c_j.*

Proof. In the encoding $\mathrm{enc}_P(s)$, P_i, P_j are represented by i, j. W.l.o.g. let $i^h \to j^t$. Given that P_i, P_j are synteny blocks, the only strings relevant to whether $P_i \cup P_j$ is a synteny block are those where both P_i and P_j are present. Since $i^h \to j^t$, in all strings where P_i and P_j are present, they occur in the same orientation and their induced phrases are adjacent. Thus $P_i \cup P_j$ is a synteny block and concatenating the partial orders of P_i and P_j, such that $e \prec d$ for all $e \in P_i, d \in P_j$ is a partial order that satisfies Definition 2 for each phrase. Moreover, in each string where P_i is present, P_j is also present, therefore anchor c_j is also present. Thus, $P_i \cup P_j$ has anchor c_j. □

This strategy can then be used to construct any collinear anchored synteny block partition.

Theorem 4. *Any collinear anchored synteny block partition $P = \{P_1, \dots, P_k\}$ on element set $E = \{e_1, \dots e_l\}$ can be constructed by iteratively merging pairs of parts $Q \cup R$, where $Q \to R$, starting from the finest partition $\{\{e_1\}, \dots, \{e_l\}\}$.*

Intuitively, starting from an anchor c, the closest elements to c in the partial order must have c as their unique neighbor and can thus be merged with it for a larger synteny block. The next closest elements then have the previously created partial block as their unique neighbor. Iterating this, one can construct any collinear anchored synteny block. Using this strategy, we now prove Theorem 4. Central to this is the following lemma, which allows the extension in one direction of the partial order.

Lemma 3. *Given an anchored, collinear synteny block $P_j = \{e_1, \dots e_m\}$ with anchor c and its partial order $\prec$. Let $P_\prec = \{e \in P_j \mid e \prec c\}$ and $P_\succ = \{e \in P_j \mid e \succ c\}$. Let $Q_0 = P_\prec \cup \{c\}$ and $Q_{i+1} = Q_i \cup \{q_{i+1}\}$ for a topological sorting $q_1 \dots q_l$ of $P_\succ$. Then each $(Q_i)_{i=0}^l$ is an anchored, collinear synteny block and $\{q_{i+1}\} \to Q_i$ for all $0 \le i < l$.*

Proof. Q_i is collinear, orientable and has anchor c because P_j is collinear, orientable and has anchor c.

Regard q_{i+1}. Let s be a string in which q_{i+1} occurs. Since c is an anchor, s also contains c. W.l.o.g. let c occur before q_{i+1} (otherwise simply invert the string). Then $s = \dots cT \pm q_{i+1} \dots$ for some substring T. Then all elements in T must be in P, otherwise P would not be contiguous in s. Additionally, for each element t in T must hold $t \prec q_{i+1}$. Thus for each element t in T, $t \in Q_i$. The same argument holds for all strings where q_{i+1} is present. Additionally, since P_j is orientable, q_{i+1} always occurs in the same orientation relative to c. Thus, if Q_i is contiguous, it is a synteny block and $\{q_{i+1}\} \to Q_i$.

We now show contiguousness by induction. Q_0 is contiguous, otherwise P_j would not be contiguous or not collinear. Assume Q_i is contiguous. Then $\{q_{i+1}\} \to Q_i$. Thus, using Lemma 2, $Q_{i+1} = Q_i \cup \{q_{i+1}\}$ is contiguous. □

By inverting the partial order one can then construct a block also in the other direction. Combining these two extensions yields the following proposition.

Proposition 1. *Any collinear, anchored synteny block $P_j = \{e_1, \dots e_m\}$ on string set S and element set E can be constructed by iteratively unifying pairs of parts R_i, R_l where $R_i \to R_l$ from some partition $\{\{e_1\}, \dots, \{e_m\}\} \cup R$ of E.*

Proof. Let $\prec$ be the partial order on P_j. Consider the two sets $P_\prec = \{x \in P \mid x \prec c\}$ and $P_\succ = \{x \in P \mid x \succ c\}$. Note that since c is an anchor, $P_\prec, P_\succ$ are disjoint.

Regard the set $Q = P_\prec \cup \{c\}$. It is contiguous and collinear because P is.

Using an inverted partial order $\prec'$, i.e., $x \prec' y \iff y \prec x$, we can then construct Q with unions as described in Lemma 3 starting with $Q_0 \coloneqq \{c\}$.

We can then construct P with unions as described in Lemma 3 starting with $Q_0 \coloneqq Q$. □

Since we can construct each part in this manner, this proves Theorem 4.

To characterize optimal sets of synteny blocks, it is helpful to keep track of only one anchor in case there are multiple anchors per block.

Definition 5 (Canonical anchor). *For a collinear anchored partition $P = \{P_1, \ldots, P_k\}$ on strings S on element set E, we call the smallest anchor $c_i \in P_i$ according to an (arbitrary) total order on E the* canonical anchor *of P_i.*

The canonical anchors identify the parts that they belong to, since each phrase must contain them. It is therefore possible to find an encoding of the input string set by simply removing all elements that are not canonical anchors.

Observation 1 *Let $P = \{P_1, \ldots, P_k\}$ be a collinear anchored synteny block partition of element set E with canonical anchors $c_1, \ldots c_k$ on string set S. Then the string set S' where each element but $c_1, \ldots, c_k$ is removed and then each $+c_i$ $(-c_i)$ is replaced by $+i$ $(-i)$ is an encoding for P.*

We can then construct a partition P where no further parts can be merged by iteratively unifying synteny blocks. The corresponding encoding S^* can then be created by removing elements that are not canonical anchors. To that end, we write $S \setminus e$ for a string set S where each occurrence of element e has been removed. Pseudocode is given in Algorithm 1 and a visual representation in Appendix A, Fig. 8.

Algorithm 1 Generate a partition P of collinear, anchored synteny blocks and its encoding S^*.

Require: A set of singular signed strings S on element set $E = \{e_1, \ldots, e_l\}$
  $P \leftarrow \{\{e_1\}, \ldots, \{e_l\}\}$
  while $P_i \to P_j$ for some $P_i, P_j \in P$ with canonical anchors c_i, c_j :
    $P \leftarrow P \setminus \{P_i, P_j\} \cup \{P_i \cup P_j\}$
    $r \leftarrow c_i$
    if $P_j \to P_i$: ▷ If they are each other's unique neighbor, remove the larger anchor
      $r \leftarrow \max(c_i, c_j)$
    $S \leftarrow S \setminus r$
  Obtain S^* by replacing each canonical anchor c_i with i, for each remaining part P_i
  return P, S^*

The way elements are merged here is similar to other compaction algorithms, such as the compaction step in maf2synteny [17]. However, while these algorithms are heuristics, Algorithm 1 is optimal for collinear, anchored MLSBP and MSSBP, as we show in the following. We begin by showing that, in the first iteration of the while loop, the removed element r from S cannot be a canonical anchor.

Lemma 4. *For two elements e, d with $e \to d$. If $d \not\to e$ or $d < e$ and e is a canonical anchor in an anchored, collinear partition $P = \{P_1, \ldots, P_k\}$, then there is a shorter partition P' with a shorter encoding than P, such that P' is also anchored and collinear.*

Proof. (By contradiction) Assume that e is a canonical anchor of part P_i. Then $d \notin P_i$. Let $d \in P_j$ with canonical anchor c. Then $P_i \to P_j$ (because $e \to d$), that is $P_i \cup P_j$ is a collinear synteny block and has anchor c according to Lemma 2. Observe that $P' = P \setminus \{P_i, P_j\} \cup \{P_i \cup P_j\}$ has both fewer parts and a shorter encoding than P. □

Note that removing an element also does not change whether the phrases induced by a block are contiguous or collinear, and as long as one anchor remains, the block also remains anchored. Thus, any optimal solution P on S still has an equivalent solution on $S \setminus r$.

Observation 2 *Given a set of strings $S = \{s_1, \ldots, s_m\}$ and a collinear anchored partition $P = \{P_1, \ldots, P_k\}$, let $P \setminus e$ be the set of parts and $S \setminus e$ be the set of strings obtained by removing an element e that is not a canonical anchor. Then $P \setminus e$ is a collinear anchored synteny block partition on $S \setminus e$.*

Let S_q be S in the q-th iteration of Algorithm 1. It follows from Observation 2 and Lemma 4 that any optimal solution on S_q still has a (possibly suboptimal) equivalent solution on S_{q+1} and the element r removed in S_{q+1} never is a canonical anchor of an optimal solution on S_q. Then any optimal solution on S^* is at most as large as any optimal solution on S.

Corollary 1. *Let S^* be the string set produced by Algorithm 1.*

Let P^*_{Size}, P_{Size} be the collinear, anchored partitions on S^* and on S respectively, such that $|P^*_{Size}|$ and $|P_{Size}|$ are minimized. Then $|P^*_{Size}| \leq |P_{Size}|$.

Let P^*_{Length}, P_{Length} be collinear, anchored partitions on S^* and on S respectively, such that the respective encoding lengths $p^* := \sum_{s^* \in S^*} |\operatorname{enc}_{P^*_{Length}}(s^*)|$ and $p := \sum_{s \in S} |\operatorname{enc}_{P_{Length}}(s)|$ are minimized. Then $p^* \leq p$.

Note that on S^* there are no further elements with unique neighbors. Theorem 4 implies that the only collinear anchored synteny block partition on S^* is the finest partition. Since $|P|$ is the number of elements in S^* and $\sum_{s_i \in S} |\operatorname{enc}_P(s_i)| = \sum_{s^*_i \in S^*} |s^*_i|$, the partition constructed by the algorithm is optimal for both MLSBP and MSSBP.

Theorem 5. *Algorithm 1 constructs a signed partition P that is a solution to collinear, anchored MLSBP and collinear anchored MSSBP in Problem 2.*

3.3 Finding a Smallest Collinear, Anchored Synteny Block Partition is Linear

With the correctness of the greedy algorithm established, we analyze its computational complexity. Let L denote the total number of extremity occurrences across all genomes, counted with multiplicity.

Lemma 5. *Algorithm 1 admits an implementation which is linear in time and space with respect to the input genome size L.*

A detailed proof is given below; we give a brief summary first. Algorithm 1 repeatedly merges elements that have a unique neighbor until no further merges are possible. Initially, all extremities with a unique neighbor are added to a queue. While the queue is not empty, the algorithm removes one extremity a^x, determines its unique neighbor b^y (with $a \to b$), and merges the corresponding element a into b. All extremities c^z that were adjacent to the opposite extremity of a^x are reconnected to b^y. If this update causes any of these neighboring extremities to have a unique neighbor, they are added to the queue. The algorithm terminates when the queue is empty. The complexity analysis has three main components: the number of iterations of the main while loop, the cost of updating adjacencies following a merge, and the cost of detecting new unique neighbors. We show that the while loop considers each adjacency at most twice, each corresponding to a possible merge between two elements. Merging an element a into b affects only the occurrences of a, and since merged elements are never reinserted, the total update cost remains linear in L. Finally, checking for new unique neighbors can be done in amortized linear time with respect to L, as each adjacency list is scanned only until one or two non-merged elements are found.

To keep track of the partitions formed, we maintain an array MERGEDTO of size equal to the number of distinct elements, where MERGEDTO$[e]$ stores the element into which element e was merged, initialized as MERGEDTO$[e] = e$ for all $e \in E$.

We store three main data structures. Each input string is represented as a linked list of occurrences of extremities. Let e^x_i denote the i-th occurrence of an extremity e^x in any of the input strings. For every occurrence e^x_i, we store the extremity adjacent to it as

$$\text{NEXT}(e^x_i) = f^y_j.$$

If e^x_i occurs at the end of a string, we set NEXT$(e^x_i) = \$$, where \$ denotes the telomere.

In addition, for each extremity e^x we maintain the list of its occurrences,

$$\text{OCC}(e^x) = [e^x_1, e^x_2, \dots],$$

which allows us to access all instances of that extremity across the input. The two extremities of the same element always have occurrence lists of the same length, and the i-th entry of OCC(e^t) corresponds to the same copy of the element as the i-th entry of OCC(e^h).

In addition, we also store for each extremity e^x the collection of extremities that appear adjacent to it in any of the input strings. We denote it by ADJ(e^x), and store it as a vector. Initially, it contains all extremities adjacent to e^x in any of the strings.

$$\text{ADJ}(e^x) = [f^y, g^z, \dots,].$$

An extremity e^x has a unique neighbor if all of the extremities in $\textsc{adj}(e^x)$ are the same.

We now prove that Algorithm 1 can be implemented in linear time and space. The complexity analysis consists of three parts: how many times the main while loop is executed, the cost of reconnecting adjacent values after merging one element with another, and the cost of checking whether any extremity after the update has a unique neighbor.

The while loop of Algorithm 1 is implemented as a queue that is processed until it becomes empty. Initially, the queue contains all extremities that have a unique neighbor. During each iteration, one extremity is removed from the queue, check if it still has a unique neighbor, and if the corresponding merge creates a new unique neighbor, it is added to the queue. Each adjacency between two extremities is considered for merging only once, and at most two extremities can be inserted into the queue for the same adjacency, in the case where each is the unique neighbor of the other. Therefore, the total number of queue insertions, and consequently the number of iterations of the while loop, is bounded by twice the number of adjacencies, which is linear in the total number of extremity occurrences L.

When the queue pops an extremity a^x with its unique adjacent extremity b^y, the element a is merged into b. Let $a^{\bar{x}}$ denote the opposite extremity of a^x,i.e., $\bar{x} = t$ if $x = h$, and $\bar{x} = h$ if $x = t$. All extremities that were adjacent to $a^{\bar{x}}$ must now be connected to b^y. To perform this update, we inspect every occurrence $a_i^{\bar{x}} \in \textsc{occ}(a^{\bar{x}})$, and for each $\textsc{next}(a_i^{\bar{x}}) = c_l^z$, we append c^z to $\textsc{adj}(b^y)$ and b^y to $\textsc{adj}(c^z)$. This is done for every single occurrence of $a_i^{\bar{x}}$, even if it implies adding multiple times c^z to $\textsc{adj}(b^y)$. At the same time, we update the linked list to preserve the structure of the strings: for each occurrence $a_i^x \in \textsc{occ}(a^x)$ we update the corresponding b_j^y adjacent to a_i^x, and we set $\textsc{next}(b_j^y) = c_l^z$. The same we do for each occurrence of c_l^z, by setting $\textsc{next}(c_l^z) = b_j^y$. This operation is performed exactly once for each occurrence of element a. Since removed elements are never reinserted, the total amortized cost of these updates is $\mathcal{O}(L)$. Moreover, the adjacency lists $\textsc{adj}(c^z)$ and of $\textsc{adj}(b^y)$ are updated simply by appending b^y and c^z, respectively, to the end of the lists.

Lastly, we need to check whether any extremities has now a unique neighbor. This must be done for each c^z and for b^y. To perform this efficiently, we use a function IsUniqueNeighbor defined in Algorithm 2, which takes in input a list of adjacencies $\textsc{adj}(e^x)$ and returns true if the there is only one adjacent extremity or false otherwise. This function scans $\textsc{adj}(e^x)$ until it finds an extremity e^x which was not merged yet. This can be checked in constant time by verifying that $\textsc{mergedto}(e) = e$. If such an extremity f^y is found, the function continues scanning for another non-merged extremity different from f^y. If a second one is found, or if all elements have already been merged, the function returns false; otherwise, it returns true. We also maintain an auxiliary data structure that stores the index of the first non-merged extremity, which we refer to as $\textsc{adjptr}(a^x)$. When two distinct non-merged neighbors are discovered, they are moved next to each other in the adjacency list so that subsequent checks can be performed in constant time.

Algorithm 2 Check if an extremity e^x has a unique neighbor. All arrays are 1-based index.

```
procedure IsUniqueNeighbor(e^x, adj, adjptr, mergedto)
    i ← adjptr(e^x)
    f^y = adj(e^x)[i]
    while i ≤ |adj(e^x)| ∧ mergedto[b] ≠ b :
        i ← i + 1
        f^y = adj(e^x)[i]
    if i ≤ |adj(e^x)| :
        g^z = adj(e^x)[i]
        while i ≤ |adj(e^x)| ∧ ( mergedto[c] ≠ c) ∨ g^z = f^y ) :
            i ← i + 1
            g^z = adj(e^x)[i]
        adjptr(e^x) ← i
        adj(e^x)[i] ← f^y
        return adjptr(e^x) = |adj(e^x)|
    return false
```

This function is called once for each occurrence of an element a which is merged, plus once for the element b into which a is merged. Each call takes either constant time or increases the scanning index monotonically until one or two non-merged elements are found. Since the total size of all adjacency lists ADJ is initially L, and each merge operation appends to these lists once per occurrence of an element, the total size remains $\mathcal{O}(L)$. Therefore, IsUniqueNeighbor runs in amortized $\mathcal{O}(L)$ time.

For similar reasons, the space used is linear in L.

4 Results

We implemented the linear version of Algorithm 1 and made it available at `github.com/gi-bielefeld/mice`. We refer to this implementation as *Markers Inferred by Compacting Elements (MICE)*. We tested three distinct modes of the algorithm. Since the theory presented here is based on singular elements, the default mode ("MICE") ignores duplicates as elements, but may still create blocks that span positions with duplicate elements. The mode "MICE - duplicates" uses the duplicate theory in Appendix A and includes duplicate elements that occur up to 10 times per genome. Lastly, we include a mode ("MICE - BP bijection") inspired by Theorem 1, in which duplicates are not merged and remain as singletons in the partition. While for singular breakpoints, a bijection of breakpoints can be guaranteed, this property does not hold for breakpoints of duplicates if they are merged or ignored. Of course, this mode produces generally smaller blocks than the first two.

In all experiments we used unitigs generated by ggcat [30] as elements, with parameter $k = 31$. The reason for using this slightly higher value for k than other synteny block tools, like SibeliaZ, is that for MICE the value of k primarily defines the threshold above which rearrangements are considered, not sensitivity for similarity. We proceeded to benchmark MICE against state-of-the-art synteny block detection methods.

SibeliaZ [20] is a heuristic algorithm that also works on elements (usually k-mers), selecting a block if there is a strong enough "backbone" of k-mers. It is currently one of the fastest tools to generate synteny blocks. In each case, we used the recommended k-value ($k = 15$ for bacteria, $k = 25$ otherwise) and two different k-mer frequency thresholds a, namely $a = 2 \times m$ and $a = 2 \times 10 \times m$, simulating the behavior of our default and duplicate modes.

Minigraph cactus [31] is a workflow that builds on minigraph [32] using the cactus framework [33, 34], which is an alignment-based, optimization-free way to construct pangenome graphs and synteny blocks.

Datasets. We used five pangenomic datasets to assess the performance of our method, namely a selection of 56 *Y. pestis* assemblies, 48 *E. coli* assemblies, all 29 currently available *S. cerevisiae* assemblies with NCBI assembly level "complete", all 5 currently available *A. thaliana* assemblies with NCBI assembly level "complete" as well as the 16 *M. musculus* genomes studied in the initial publication of SibeliaZ [20]. We give details to these datasets in Appendix B Table 3.

4.1 Benchmark

Runtime. We benchmarked MICE against SibeliaZ. For both methods we only evaluated the synteny block generation step, that is, we measured the time it takes to convert elements as input to blocks as output. To that end, we provided MICE with a GFF-format file representing unitigs in the ggcat graph and we provided SibeliaZ with the binary TwoPaCo graph and skipped SibeliaZ's alignment step. We did not include minigraph-cactus in this evaluation as the costly alignment step is central to the minigraph-cactus workflow. The runtimes for minigraph-cactus in each case were slower (see also Fig. 9 in Appendix B), but are not directly comparable to the measurements reported here.

We ran all algorithms on a single thread on a cluster machine. The results are visualized in Fig. 4. We see that despite being an exact algorithm with strict theoretical guarantees, MICE achieves similar performance to SibeliaZ, even outperforming the version with a high abundance filter. However, for both tools we observed that a significant portion of the runtime was spent on loading the input data from disk. We therefore see major potential for performance improvement by integrating ggcat directly with MICE. Because not all SibeliaZ runs finished within 24 hours on the *M. musculus* dataset, in the following experiments we only used the first 4 of the 16 assemblies of this dataset.

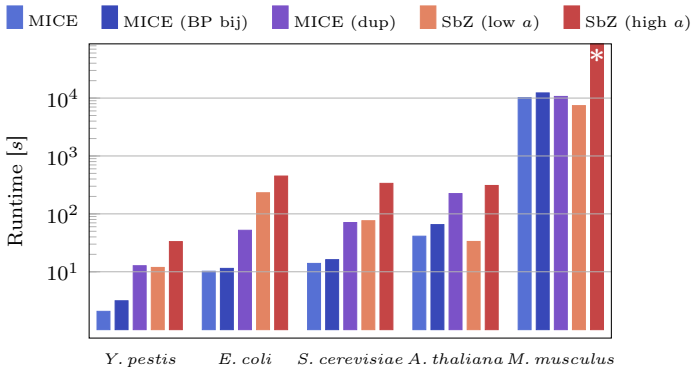

Fig. 4. Runtimes of three MICE variants (MICE, breakpoint-bijection mode "MICE (BP bij)", and duplicate mode "MICE (dup)") and two SibeliaZ variants (SbZ with low a and high a) on five benchmark datasets, measured with `/usr/bin/time` wall-clock time. The asterisk in SbZ (high a) for *M. musculus* indicates that it exceeded the 24-hour limit.

Genome coverage. We proceeded to evaluate the coverage of SibeliaZ, minigraph-cactus and MICE on the benchmark datasets. To that end, we examined how many blocks of 50 base pairs or longer it takes to cover a certain fraction of the input genomes. As a summary, we give the coverage and encoding sizes in Table 1 and the respective N50, N75 and N90 values in Fig. 5. Here Nx is defined as the length threshold with which one can cover x% of the total genome lengths using only block phrases of length Nx or higher. We show details in Fig. 10 of Appendix B. We see that for all datasets, the default version of MICE covers the same or more positions with fewer and larger blocks than SibeliaZ and minigraph-cactus. Running in duplicate and BP bijection mode produces slightly shorter blocks on low duplicate datasets, but has problems covering genomes with large numbers of duplicates.

Nonetheless, the coverage improvement of the default mode in these genomes is even more noticeable. Additionally, examining the proportion of shared blocks between pairs of genomes (Appendix B, Fig. 11) shows that the high coverage of MICE does not come at the expense of significantly reducing sensitivity in finding shared regions.

Table 1. Encoding sizes as well as coverage across all genomes using block occurrences ≥ 50 BP in the five benchmark datasets.

	Y. pestis		*E. coli*		*S. cerevisiae*		*A. thaliana*		*M. musculus*	
	enc. size	coverage	enc. size	coverage	enc. size	coverage	enc. size	coverage	enc. size	coverage
MICE	27535	93.68%	100182	94.73%	85613	95.82%	258296	97.02%	2043614	91.67%
MICE - BP bijection	102488	94.46%	156282	88.97%	151332	88.93%	694185	77.02%	14694541	74.58%
MICE - duplicates	58312	96.68%	151457	96.47%	106418	96.36%	731918	86.81%	15932964	80.40%
MG cactus	49723	62.44%	102950	81.73%	239559	95.51%	341340	87.38%	5298901	87.16%
SibeliaZ - low a	106243	97.96%	166098	96.65%	116994	97.51%	232359	90.78%	4603390	85.27%
SibeliaZ - high a	102369	98.94%	199547	96.90%	145731	98.56%	488621	93.46%	4574842	88.58%

4.2 Rearrangement Detection

While it is desirable for synteny blocks to be large and have high coverage, they must also avoid obscuring true rearrangements. To assess this, we measured, for each method and for four of the five datasets, how many breakpoints were obscured by the synteny-block construction. We excluded the *M. musculus* dataset because determining breakpoints on the original elements in this dataset was too computationally expensive. Breakpoints for duplicates are not well-defined outside of a concrete rearrangement and matching model. Therefore, we discarded any element appearing as a duplicate in any genome of the dataset for this analysis.

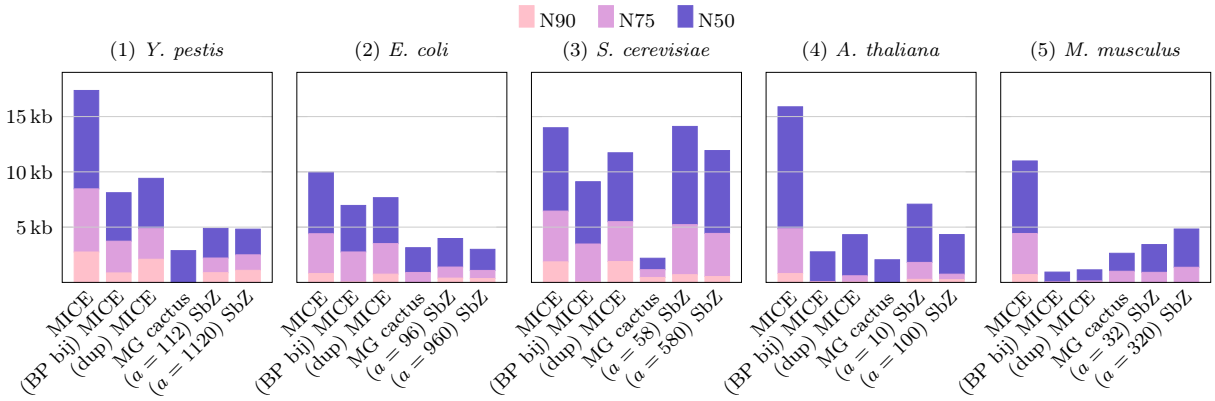

Fig. 5. N50, N75 and N90 values for blocks generated by MICE, MICE in breakpoint bijection mode ("(BP bij) MICE"), MICE with duplicates ("(dup) MICE"), minigraph-cactus (MG cactus) and SibeliaZ (SbZ) on five benchmark datasets.

For each dataset, we compared all genome pairs, projected their sequences to their shared 31-mers, and computed their breakpoints. We searched for two types of obscured breakpoints: (i) pairs of elements forming a breakpoint in some genome but are found at least once in phrases with the same block identifier, and (ii) breakpoints occurring entirely within a single block phrase; the second type being a subset of the first. We give an example of these types of breakpoints in Appendix A, Fig. 7.

In both cases, it is not possible for subsequent analyses to recognize the breakpoint using the synteny blocks alone. Fig. 6 shows the percentage of breakpoints that were not obscured (blue), obscured within the same block identifier (orange), and obscured within the same synteny block phrase (red) for all methods.

For our methods MICE and MICE BP–bijection, no breakpoints are obscured by construction (Theorem 1). In MICE duplicates, where duplicates are considered for merging, breakpoints are obscured when both elements forming a breakpoint are unified with the same duplicate, which occurs in about 10-13% of cases. Even though SibeliaZ uses a smaller k-mer size, both of its versions obscure 20-30% of breakpoints, with about 5-10% falling within the same block occurrences. Minigraph-cactus obscures 20-40% of breakpoints, which can be attributed to the fact that our measurement is k-mer based, while minigraph-cactus' construction is alignment-based.

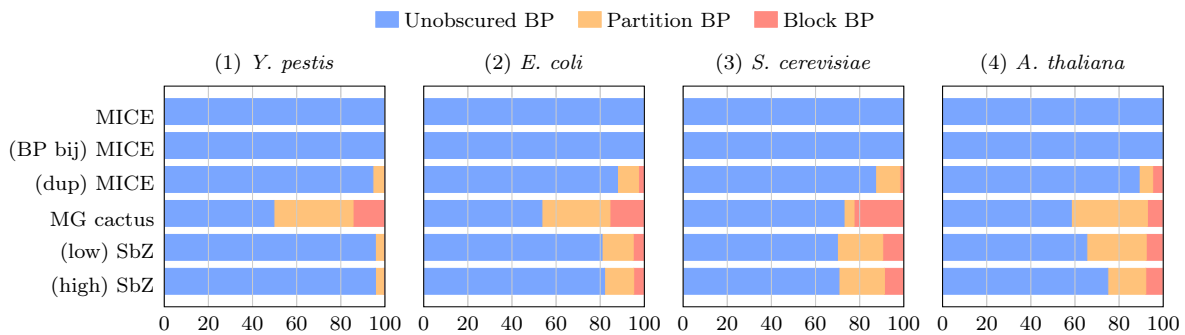

Fig. 6. Percentage of breakpoints that are not obscured (blue), obscured by being placed in the same type synteny block (orange), or obscured within the same synteny block (red), shown for all methods and for four of the datasets. Table 2 lists the corresponding values.

Table 2. Comparison of breakpoints in a partition (Part) and Breakpoints in a block (Block) across species for each method. The row "*Breakpoints*" represents the total number of breakpoints found from pairwise genome comparisons using 31-mers.

	Y. pestis		*E. coli*		*S. cerevisiae*		*A. thaliana*	
Breakpoints	3945		55867		20957		129338	
	Part	Block	Part	Block	Part	Block	Part	Block
MICE	0	0	0	0	0	0	0	0
MICE - BP bijection	0	0	0	0	0	0	0	0
MICE - duplicates heuristic	216	25	6815	1467	2679	373	14129	6196
MG cactus	1985	570	25946	8719	5676	4727	53892	9290
SibeliaZ - low a	172	22	10714	2833	6309	2003	44798	9993
SibeliaZ - high a	170	28	10073	2705	6158	1841	32493	10247

5 Discussion

We introduced a formal framework for deriving synteny blocks by compacting genomic elements in a way that preserves rearrangements. To our knowledge, this is the first such framework. We defined the MSSBP and MLSBP problems and showed that both are NP-hard in general. Yet, when blocks are required to be collinear and contain an anchor element, both objectives can be optimized simultaneously by a linear-time algorithm. Our experiments indicate that this algorithm is competitive with state-of-the-art synteny block construction methods while preserving all rearrangements of unique elements present in the input data.

While these results focus on representing rearrangements, we did not asses how well the sequences within the generated blocks align. Methods that explicitly model alignment, such as minigraph-cactus, likely produce blocks that are better suited for whole-genome alignment. Nonetheless, future work may combine rearrangement-based and alignment-based synteny block theory. In practice, the larger blocks generated by MICE could be used as a pre-segmentation step before applying slower alignment-based methods.

While our method supports duplicated elements, blocks in the singular setting come with more theoretical guarantees and show better coverage in our evaluation. This suggests that a different generalization may yield a more effective handling of duplicates while ensuring stricter guarantees regarding breakpoints. For example, the generalization so far guarantees only what is known in the literature as local collinearity (collinearity per block phrase), not global collinearity (collinearity w.r.t. the entire genome).

Another research direction is to explore relaxed variants of the theoretical properties presented here. For example, one could require that each pair of genomes shares at least one element in a block, without requiring it to be the same element across all pairs. However, this may come at the risk of a computationally harder problem or too dissimilar blocks. Finally, one could consider the frequency of breakpoints and allow merges across breakpoints that occur only rarely, which may enable stronger compression in closely related genomes, such as pangenome datasets.

Acknowledgments

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement number 872539 (PANGAIA). Most of the work was conceived and developed during a research stay of LP and LB at Simon Fraser University. We thank Kamil Hepak for helpful comments on readability, visualization and language.

References

[1] P. A. Pevzner and G. Tesler. "Genome rearrangements in mammalian evolution: lessons from human and mouse genomes". In: *Genome Research* 13.1 (2002), pp. 37–45. DOI: 10.1101/gr.757503.

[2] G. Bourque, P. A. Pevzner, and G. Tesler. "Reconstructing the genomic architecture of ancestral mammals: lessons from human, mouse, and rat genomes". In: *Genome Research* 14.4 (2004), pp. 507–516. DOI: 10.1101/gr.1975204.

[3] J. Ma et al. "Reconstructing contiguous regions of an ancestral genome". In: *Genome Research* 16.12 (2006), pp. 1557–1565. DOI: 10.1101/gr.5383506.

[4] C. Chauve and É. Tannier. "A methodological framework for the reconstruction of contiguous regions of ancestral genomes and its application to mammalian genomes". In: *PLoS Computational Biology* 4.11 (2008), e1000234. DOI: 10.1371/journal.pcbi.1000234.

[5] Y. Wang et al. "Mcscanx: a toolkit for detection and evolutionary analysis of gene synteny and collinearity". In: *Nucleic Acids Research* 40 (7 2012), e49–e49. DOI: 10.1093/nar/gkr1293.

[6] A. Sinha and J. Meller. "Cinteny: flexible analysis and visualization of synteny and genome rearrangements in multiple organisms". In: *BMC Bioinformatics* 8.1 (2007). DOI: 10.1186/1471-2105-8-82.

[7] B. Paten et al. "Enredo and pecan: genome-wide mammalian consistency-based multiple alignment with paralogs". In: *Genome Research* 18.11 (2008), pp. 1814–1828. DOI: 10.1101/gr.076554.108.

[8] S. Pham and P. A. Pevzner. "Drimm-synteny: decomposing genomes into evolutionary conserved segments". In: *Bioinformatics* 26.20 (2010), pp. 2509–2516. DOI: 10.1093/bioinformatics/btq465.

[9] M. Grabherr et al. "Genome-wide synteny through highly sensitive sequence alignment: satsuma". In: *Bioinformatics* 26 (9 2010), pp. 1145–1151. DOI: 10.1093/bioinformatics/btq102.

[10] S. Proost et al. "I-adhore 3.0—fast and sensitive detection of genomic homology in extremely large data sets". In: *Nucleic Acids Research* 40.2 (2011), e11–e11. DOI: 10.1093/nar/gkr955.

[11] B. Paten et al. "Cactus: algorithms for genome multiple sequence alignment". In: *Genome Research* 21 (9 2011), pp. 1512–1528. DOI: 10.1101/gr.123356.111.

[12] B. Brejová, M. Burger, and T. Vinař. "Automated Segmentation of DNA Sequences with Complex Evolutionary Histories". In: *Algorithms in Bioinformatics - 11th International Workshop, WABI 2011, Saarbrücken, Germany, September 5-7, 2011. Proceedings.* Vol. 6833. Lecture Notes in Computer Science. 2011, pp. 1–13. DOI: 10.1007/978-3-642-23038-7_1.

[13] I. Minkin et al. "Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes". In: *Algorithms in Bioinformatics - 13th International Workshop, WABI 2013, Sophia Antipolis, France, September 2-4, 2013. Proceedings.* Vol. 8126. Lecture Notes in Computer Science. 2013, pp. 215–229. DOI: 10.1007/978-3-642-40453-5_17.

[14] G. Drillon, A. Carbone, and G. Fischer. "Synchro: a fast and easy tool to reconstruct and visualize synteny blocks along eukaryotic chromosomes". In: *PLoS ONE* 9.3 (2014), e92621. DOI: 10.1371/journal.pone.0092621.

[15] J. Lee et al. "Synteny portal: a web-based application portal for synteny block analysis". In: *Nucleic Acids Research* 44 (W1 2016), W35–W40. DOI: 10.1093/nar/gkw310.

[16] J. M. Lucas and H. Roest Crollius. "High precision detection of conserved segments from synteny blocks". In: *Plos One* 12.7 (2017), e0180198. DOI: 10.1371/journal.pone.0180198.

[17] M. Kolmogorov et al. "Chromosome assembly of large and complex genomes using multiple references". In: *Genome Research* 28.11 (2018), pp. 1720–1732. DOI: 10.1101/gr.236273.118.

[18] E. Polevikov and M. Kolmogorov. "Synteny paths for assembly graphs comparison". In: *19th International Workshop on Algorithms in Bioinformatics, WABI 2019, September 8-10, 2019, Niagara Falls, NY, USA.* Vol. 143. LIPIcs. 2019, 24:1–24:14. DOI: 10.4230/lipics.wabi.2019.24.

[19] M. Goel et al. "Syri: finding genomic rearrangements and local sequence differences from whole-genome assemblies". In: *Genome Biology* 20.1 (2019). DOI: 10.1186/s13059-019-1911-0.

[20] I. Minkin and P. Medvedev. "Scalable multiple whole-genome alignment and locally collinear block construction with sibeliaz". In: *Nature Communications* 11.1 (2020). DOI: 10.1038/s41467-020-19777-8.

[21] K. Krasheninnikova et al. "Halsynteny: a fast, easy-to-use conserved synteny block construction method for multiple whole-genome alignments". In: *GigaScience* 9.6 (2020). DOI: 10.1093/gigascience/giaa047.

[22] J. Armstrong et al. "Progressive cactus is a multiple-genome aligner for the thousand-genome era". In: *Nature* 587 (7833 2020), pp. 246–251. DOI: 10.1038/s41586-020-2871-y.

[23] L. Coombe et al. "Multi-genome synteny detection using minimizer graph mappings". In: (2024). DOI: 10.1101/2024.02.07.579356.

[24] P. A. Pevzner and G. Tesler. "Human and mouse genomic sequences reveal extensive breakpoint reuse in mammalian evolution". In: *Proceedings of the National Academy of Sciences* 100.13 (2003), pp. 7672–7677. DOI: 10.1073/pnas.1330369100.

[25] D. Sankoff and P. Trinh. "Chromosomal breakpoint reuse in genome sequence rearrangement". In: *Journal of Computational Biology* 12.6 (2005), pp. 812–821. DOI: 10.1089/cmb.2005.12.812.

[26] Q. Peng, P. A. Pevzner, and G. Tesler. "The fragile breakage versus random breakage models of chromosome evolution". In: *PLoS Computational Biology* 2.2 (2006), e14. DOI: 10.1371/journal.pcbi.0020014.

[27] D. Sankoff. "The Signal in the Genomes". In: *PLOS Computational Biology* 2.4 (2006), e35. DOI: 10.1371/journal.pcbi.0020035.

[28] C. G. Ghiurcuta and B. M. E. Moret. "Evaluating synteny for improved comparative studies". In: *Bioinformatics* 30.12 (2014), pp. i9–i18. DOI: 10.1093/bioinformatics/btu259.

[29] A. Elghraoui et al. "Evaluating impacts of syntenic block detection strategies on rearrangement phylogeny using *mycobacterium tuberculosis* isolates". In: *Bioinformatics* 39.1 (2023). DOI: 10.1093/bioinformatics/btad024.

[30] Andrea Cracco and Alexandru I Tomescu. "Extremely fast construction and querying of compacted and colored de Bruijn graphs with GGCAT". In: *Genome Research* 33.7 (2023), pp. 1198–1207.

[31] Glenn Hickey et al. "Pangenome graph construction from genome alignments with Minigraph-Cactus". In: *Nature Biotechnology* 42.4 (2024), pp. 663–673. ISSN: 1546-1696. DOI: 10.1038/s41587-023-01793-w.

[32] Heng Li, Xiaowen Feng, and Chong Chu. "The design and construction of reference pangenome graphs with minigraph". In: *Genome Biology* 21.1 (2020), p. 265. ISSN: 1474-760X. DOI: 10.1186/s13059-020-02168-z.

[33] Benedict Paten et al. "Cactus Graphs for Genome Comparisons". In: *Journal of Computational Biology* 18.3 (2011), p. 469. DOI: 10.1089/cmb.2010.0252.

[34] Benedict Paten et al. "Superbubbles, Ultrabubbles, and Cacti". In: *Journal of Computational Biology* 25.7 (2018), pp. 649–663. DOI: 10.1089/cmb.2017.0251.