

# Fast Exact String to D-Texts Alignments

Njagi Moses Mwaniki<sup>1</sup><sup>a</sup>, Erik Garrison<sup>2</sup><sup>b</sup> and Nadia Pisanti<sup>1</sup><sup>c</sup>

<sup>1</sup>University of Pisa, Italy

<sup>2</sup>University of Tennessee Health Science Center, U.S.A.

Keywords: D-string, Pangenome, Pattern Matching.

Abstract: A D-strings is a degenerate string representing similar and aligned strings by collapsing common fragments and highlighting variants. D-strings can represent a MSA or a pan-genome. In this paper we propose a new, fast and exact method to align a string to a D-string. In recent years, aligning a sequence to a pangenome has become a central problem in computational genomics and pangenomics. A fast and accurate solution to this problem can serve as a toolkit to many crucial tasks such as read-correction, Multiple Sequences Alignment (MSA), genome assemblies, and variant calling, just to name a few. An implementation of our tool is publicly available on github at <https://github.com/urbanslug/dsa>.

## 1 INTRODUCTION

In recent years, aligning a sequence to a pangenome has become a central problem in computational genomics and pangenomics (CPC, 2018). This problem has often been addressed with the purpose of aligning sequencing reads to a complex structure such as a degenerate string or a more general graph structure (E.Garrison et al., 2018; J.M.Eizenga et al., 2021; C.A.Darby et al., 2020; M.Rautiainen et al., 2019; M.Rautiainen and T.Marschall, 2020; H.Li et al., 2020; E.Birmelé et al., 2012), and possibly assuming the query string to be considerably shorter than the pangenome (E.Garrison et al., 2018; J.M.Eizenga et al., 2021; A.Cisłak et al., 2018; C.A.Darby et al., 2020). As sequencing data is now offering longer sequences with high accuracy, the attention has recently been moved towards the problem of aligning to a graph-like structure a string which is approximately as long. A fast and accurate solution to this problem would serve as a toolkit to many crucial tasks such as Multiple Sequences Alignment (MSA), genome assembly, and variant calling, just to name a few. The requirement of an alignment being at the same time exact and accurate raises computational challenges: current exact methods that perform alignments to long strings would have a time complexity quadratic in the

strings size.

Consider the following MSA of three closely-

related sequences

	GCAATCGGGTATT
	GCAATCGGGAATT
	GCACGCTGGATT

and its degenerate string (*D-string*) compact representation  $\hat{T} = \mathbf{GCA}_{[AT/CG]}\mathbf{C}_{[G/T]}\mathbf{GG}_{[TA/AA/AT]}\mathbf{TT}$

A D-string (or D-text)  $\hat{T}$  contains both deterministic (shown in bold) and non-deterministic (alternative variants) segments. Formally,  $\hat{T}$  is a sequence of  $n$  sets of strings where the  $i^{th}$  set contains strings of the same length  $\ell_i$  (possibly = 1 in the deterministic case) but this length can vary between different sets.

A great deal of research has been conducted in the bioinformatics literature on a special case of D-strings where all the alternative variants are single letters. These are equivalent to a sequence written in the IUPAC notation (IUPAC-IUB Commission on Biochemical Nomenclature, 1970) to represent a position in a DNA sequence that can possibly have multiple alternatives (H.Soldano et al., 1995; N.Pisanti et al., 2005; N.Pisanti et al., 2009; K.Abrahamson, 1987; M.Crochemore et al., 2017; C.S.Iliopoulos and J.Radoszewski, 2016). These are commonly used to encode the consensus of a population of sequences in a MSA (P.Peterlongo et al., 2008; CPC, 2018; M.Alzamel et al., 2018; M.Alzamel et al., 2020; P.Gawrychowski et al., 2020).

The more general notion of ED-strings (Elastic D-string, where variants can have different sizes within a degenerate position), and over them the short read

<sup>a</sup> <https://orcid.org/0000-0002-4858-2375>

<sup>b</sup> <https://orcid.org/0000-0003-3821-631X>

<sup>c</sup> <https://orcid.org/0000-0003-3915-7665>

matching problem *Elastic-Degenerate String Matching* (EDSM) problem has attracted some attention in the combinatorial pattern matching community. Since its introduction in 2017 (C.S.Iliopoulos et al., 2017), a series of results have been published both for the exact (R.Grossi et al., 2017; G.Bernardini et al., 2019; K.Aoyama et al., 2018; G.Bernardini et al., 2022) as well as for the approximate (G.Bernardini et al., 2020; G.Bernardini et al., 2017; N.M.Mwaniki and N.Pisanti, 2022) version of the problem. Even if they are designed and optimised for short read matching, some of the algorithms above would also work for long reads, but failing to be efficient. In particular, when rather than matching the task is finding an optimal alignment (that is, computing the edit distance) of a query sequence and the pangenome, then the heuristic-free method of (G.Bernardini et al., 2020; G.Bernardini et al., 2017) would require  $O(d^2WG + dN)$  time, where  $d$  is the computed edit distance,  $W$  is the length of the query string, and  $N$  and  $G$  are the total size and the total number of strings of the degenerate string.

In this paper we propose a new, fast and exact method to align a string to a D-string, the latter possibly representing an MSA or a pan-genome. Ours is a base-level heuristic-free alignment method that allows affine gap penalty score functions and the use of arbitrary scores as well as weights. D-strings are one out of many others possible pan-genome representation (V.Carletti et al., 2019). Our algorithm exploits the assumption that the similarity the alignment must detect is high enough to suitably adapt the seminal idea of (E.W.Myers, 1986) to D-strings, combining at the same time *partial order alignments* (C.Lee et al., 2002) and the *wavefront* paradigm (S.Marcosola et al., 2021) managing to work with affine gap penalty function to score gaps, which is a desirable property when the target of the alignment is to account for INDELs variant events.

This paper is organised as follows: Section 2 gives some preliminary definitions on D-strings and notions of partial order and wavefront alignment, Section 3 describes our algorithm, and Section 4 the experimental validation of the resulting tool we realized. The implementation of DSA is publicly available at <https://github.com/urbanslug/dsa>, and our D-strings random generator can be found at <https://github.com/urbanslug/simed/>.

## 2 PRELIMINARY NOTIONS

### 2.1 D-strings

A *string*  $X$  is a sequence of elements on an *alphabet*  $\Sigma$ , where  $\Sigma$  is a non-empty finite set (of size  $|\Sigma|$ ) of letters. The set of all finite strings over an alphabet  $\Sigma$ , including the *empty string*  $\varepsilon$  of length 0, is denoted by  $\Sigma^*$ , while  $\Sigma^+$  denotes the set  $\Sigma^* \setminus \{\varepsilon\}$ . The set of all strings of length  $k > 0$  over  $\Sigma$  is denoted by  $\Sigma^k$ . For any string  $X$ , we denote by  $X[i, j]$  the *substring* of  $X$  that *starts* at position  $i$  and *ends* at position  $j$ . In particular,  $X[0, j]$  is the *prefix*<sup>1</sup> of  $X$  that ends at position  $j$ , and  $X[i, |X|]$  is the *suffix* of  $X$  that starts at position  $i$ , where  $|X|$  denotes the *length* of  $X$ .

We define a *Degenerate String* (*D-string*)  $\hat{S} = \hat{S}\{1\}\hat{S}\{2\} \dots \hat{S}\{n\}$  of *length*  $n$  over an alphabet  $\Sigma$  as a finite sequence of  $n$  degenerate letters  $\hat{S}\{i\}$ . Each *degenerate letter*  $\hat{S}\{i\}$  has *width*  $\ell_i > 0$  and is a finite non-empty set of  $|\hat{S}\{i\}|$  strings of the same length  $\ell_i$  (i.e.  $\hat{S}\{i\}[1], \dots, \hat{S}\{i\}[|\hat{S}\{i\}|] \in \Sigma^{\ell_i}$ ).

For any  $1 \leq i \leq n$ , we remark that a degenerate letter  $\hat{S}\{i\}$  with  $|\hat{S}\{i\}| = \ell_i = 1$  is just a simple letter of  $\Sigma$ . Whenever this happens, we will say that  $i$  is a *solid position* of  $\hat{S}$ . We now define some parameters that measure the degeneracy of a D-string.

The *total size*  $N$  and *total width*  $w(\hat{T}) = W$  of a D-string  $\hat{S}$  are respectively defined as  $N = \sum_{i=1}^n |\hat{S}\{i\}| \cdot \ell_i$  and  $W = \sum_{i=1}^n \ell_i$ .

In our running example of D-string  $\hat{T} = \text{GCA[AT/CG]C[G/T]GG[TA/AA/AT]TT}$ ,  $\ell_4 = \ell_9 = 2$  while all other  $\ell_i$ 's are 1, and  $|\hat{S}\{4\}| = |\hat{S}\{6\}| = 2$ ,  $|\hat{S}\{9\}| = 3$  while all the other  $|\hat{S}\{i\}|$ 's are 1; finally, the solid positions of  $\hat{T}$  are 1, 2, 3, 5, 7, 8, 10, 11.

A D-string of width  $W$  actually represents a set of linear strings of length  $W$ , corresponding to all the strings that can be read by making any choice at a degenerate positions. Formally, given a D-string  $\hat{T}$  of width  $W$ , and a string  $T$  with  $|T| = W$  being any string in such set, we say that  $T$  belongs to  $\hat{T}$  ( $T \in \hat{T}$ ).

For example, the strings GCACGCTGGAATT and GCAATCTGGTATT are two of the twelve strings that belong to the D-string  $\hat{T} = \text{GCA[AT/CG]C[G/T]GG[TA/AA/AT]TT}$ .

The  $i^{\text{th}}$  position  $\hat{S}\{i\}$  ( $1 \leq i \leq n$ ) of  $\hat{S}$  should not be confused with its  $i^{\text{th}}$  width  $\hat{S}[i]$  ( $1 \leq i \leq W$ ):

For any D-string  $\hat{S}$ , its  $i^{\text{th}}$  width (for  $1 \leq i \leq W$ )  $\hat{S}[i]$  is the letter, or the set of alternative letters, that can appear in  $T[i]$  for string  $T \in \hat{T}$ . For any D-string  $\hat{S}$ , we denote by  $\hat{S}[i_1, i_2]$  the *D-substring* of  $\hat{S}$  that starts at width  $i_1$  and ends at width  $i_2$  with  $1 \leq i_1 \leq i_2 \leq W$ .

<sup>1</sup>We start from position 0 as we will need to consider the empty prefix  $X[0, 0]$  for the dynamic programming.

We will also use the notation  $s_i$  for both  $|\hat{S}\{i\}|$  and  $|\hat{S}[i]|$ .

For example, the D-string  $\hat{T} = \text{GCA[AT/CG]C[G/T]GG[TA/AA/AT]TT}$  has length  $n=11$ , size  $N=20$ , and width  $W=13$ . We have that at width, say, 3 there is a solid position  $A=\hat{S}[3]$ , and at width 4 there is  $[A/C]=\hat{S}[4]$ . The D-substring  $\hat{S}[3,7]$  is  $A[AT/CG]C[G/T]$ . We show below positions and widths for the D-string  $C[AT/CG]C[G/T]$  having length 4, width 5, and size 8.

D-string	C	[A	T	/	C	G]	C	[G	/	T]
width	1	2	3		2	3	4	5		5
position	1			2			3		4	

## 2.2 Partial Order Alignment

The use of Partial Order (graphs) for sequence Alignments (POA) was introduced in (C.Lee et al., 2002) with the purpose of improving the iterative step of progressive (D.-F.Feng and Doolittle, 1987) multiple sequence alignments (MSA), that is the step where a new sequence is added to an MSA (*starting MSA*). Traditionally, this was performed by first reducing the starting MSA to a linear profile, and then aligning it to the new sequence in order to obtain a new MSA (*resulting MSA*), with the downside that the reduction of the starting MSA to a linear profile might carry and propagate alignment errors. The idea behind POA is to perform a direct pairwise dynamic programming alignment between the new sequence and a suitable graph representation of the starting MSA (thus replacing the somewhat lossy linear profile representation), with the outcome of maintaining, in the resulting MSA, the optimal alignment of the new sequence with respect to each one of those of the starting MSA.

Such graph representation, the *partial order graph*, basically replaces the linear MSA profile with a DAG whose edges represent a partial order between the letters of the MSA to be used as an alternative to the traditional total order of the rows of an alignment.

## 2.3 WFA with Affine Gap Penalty

In (E.W.Myers, 1986), Gene Myers introduced the linear time and space dynamic programming (DP) alignment for similar strings: when aligning two strings of size  $N$  whose distance is known to be upper-bounded by  $D$ , rather than computing the whole table of size  $N^2$ , only a stripe of  $O(D)$  diagonals are computed, as their similarity guarantees their optimal alignment to lay therein. The result is a  $O(DN)$  algorithm replacing the quadratic one, and an algorithm that, rather than filling in a matrix row by row, proceeds with wave-fronts along diagonals (WFA: Wave

Front Alignment).

The *Affine Gap Penalty* function to score the cost of gaps in alignments evaluates a series of  $k$  consecutive gaps as  $w(k) = o + k \cdot e$  (where  $o$  is the cost of opening the gap, and  $e$  that of extending it), rather than the sum of  $k$  single gap's costs. Using such gap penalty score is almost mandatory in genomic sequences analysis as this forces the optimal alignment to detect and highlight INDELS variations that typically involve several consecutive nucleotides. In order to restore the optimal substructure of the alignment problem when using affine gap penalty score function, the DP algorithm requires three alignment tables instead of one: matrices  $I, D, M$  that store the score of the best alignments ending - respectively - with an Insertions, a Deletions, or a (Mis)Match.

In (S.Marco-Sola et al., 2021), the linear time and space optimization of Myers was extended to the exact computation of an optimal pairwise alignment of strings using the affine gap penalty score function. Roughly, for the alignment of two strings of size  $N$  and  $M$ , the three matrices  $I, D, M$  of size  $N \times M$  are replaced by *wave front* records  $I_{d,k}, D_{d,k}, M_{d,k}$  that store, for each score/distance  $d$  and diagonal  $k$ , the furthest-reaching offset in  $k$  that scores  $d$ . The dynamic programming recurrence is then operated on increasing values of  $d$  and adjacent diagonals  $k$ , up to the final distance: the one whose wave front reaches the end of both input strings.

## 3 THE DSA ALGORITHM

Let us first formally state the problem we solve:

[StoDS] Given a D-string  $\hat{T}$  of width  $W$  and size  $N$ , a pattern  $P$  of length  $m$ , and penalty scores  $a, x, o, e$ , find an optimal alignment between  $P$  and  $\hat{T}$  using scores  $a$  for match,  $x$  for mismatch,  $o$  for gap opening, and  $e$  for gap extension.

In the problem statement above, by optimal alignment we mean the alignment between  $P$  and a string  $T \in \hat{T}$  (see Section 2.1) that minimizes the distance computed by scoring  $a$  for a match,  $x$  for a mismatch, and gap affine penalty function with  $o$  for gap opening and  $e$  for gap extension.

In this section we describe our algorithm DSA that optimally solves STODS. We start with Section 3.1 where we show how we adapted the *Partial Order Alignment* (POA) framework to work with alignment to D-strings. Then, in Section 3.2 we describe our extension of *Wave Front Alignment* (WFA) to D-strings and show how we merge the two techniques into our algorithm DSA and analyse its complexity.

-	A	C	G	T	A	
-	0	1	2	3	4	5
A	1	0	1	2	3	4
C	2	1	0	1	2	3
G	(3,3)	(2,2)	(1,1)	(0,1)	(1,2)	(2,2)
T	(4,4)	(3,3)	(2,2)	(1,2)	(1,1)	(2,3)
A	5	4	3	2	2	1

Figure 1: Dynamic Programming table for the POA of the string ACGTA against the D-string AC[GC/AT]A partially ordered as shown by oriented edges, and using scores: 0 for match, 1 for mismatch, and 2 for gap.

### 3.1 POA with D-strings

In this section, we show how to use partial order alignment to be able to perform base level alignments with D-strings. Without loss of generality, for the sake of simplifying the notation, within this section we will not use the gap affine penalty score (whose implementation with D-strings will be described in the next section), but rather account a cost  $g$  to any gap.

Given a D-string  $\hat{T}$  of width  $W$  and pattern  $P$  of length  $m$ , we build a  $(W+1) \times (m+1)$  DP table  $M$  having a row  $i$  for each  $i^{\text{th}}$  width of  $\hat{T}$ , and a column for each letter of  $P$  (plus the usual first row and first column for their empty prefixes), and that will store in  $M[i, j]$  the best score of aligning  $P[1, j]$  to  $\hat{T}[1, i]$  (an example is shown in Figure 1). When a row falls within a non-solid position having  $s$  strings of length  $\ell$ , then that row is not associated with a simple letter, but rather with  $s$  variants. The entries of these rows will thus contain a tuple of size  $s$ : if row  $i$  has letters  $a_{i,1}, \dots, a_{i,h}, \dots, a_{i,s}$ , then  $M[i, j]$  is a tuple of  $s$  values  $\langle M[i, j]^1, \dots, M[i, j]^h, \dots, M[i, j]^s \rangle$  where, for each  $1 \leq h \leq s$ , a match is accounted if and only if  $a_{i,h} = P[j]$ .

On top of this tuple representation, a partial order is assumed for the  $N$  letters of the D-string: within solid positions, the usual order of sequences applies; in a degenerate position, instead, distinct letters in the same tuple are not comparable, and nor is letter  $a_{i,h}$  of tuple  $a_{i,1}, \dots, a_{i,h}, \dots, a_{i,s_i}$  at row  $i$  comparable with the other letters of all the  $\ell$  adjacent rows composing the same degenerate position, except for the  $\ell$  other letters  $a_{i',h}$  that are also the  $h^{\text{th}}$  letter in their tuples. Within all comparable letters, the order corresponds to that of the rows of  $M$ .

We use this partial order to drive the alignment

along the D-string: wherever the traditional dynamic programming alignment algorithm refers to the previous row and/or previous column, here we refer only to entries that - instead - are preceding according to the partial order defined above. For example, the partial order of AC[GC/AT]A is defined by the graph shown on the letters at the rows of the DP table of Figure 1.

We now formalize with dynamic programming recurrence relations the way we apply the ideas sketched above. For any two letters  $\sigma_1, \sigma_2 \in \Sigma$ , we define a function  $m(\sigma_1, \sigma_2)$  as equal to  $a$  if  $\sigma_1 = \sigma_2$ , and equal to  $x$  otherwise. Denoting with  $s_i$  the size of the tuple at row  $i$ , we compute  $M[i, j]$  for all  $1 \leq j \leq m$  distinguishing the following cases:

$s_{i-1} = s_i = 1$ . In this case no degeneracy is encountered and the traditional dynamic programming alignment framework applies.

$s_{i-1} = 1$  and  $s_i = s > 1$ . In this case, row  $i$  corresponds to the opening of a degenerate position of  $s_i$  strings of length  $\ell$ , this row (like the next  $\ell - 1$  rows) represents  $s_i$  alternative letters  $a_{i,1}, \dots, a_{i,s_i}$  and each entry  $M[i, j]$  will contain tuples of size  $s_i$  computed as follows for  $1 \leq h \leq s_i$ :

$$M[i, j]^h = \min \begin{cases} M[i-1, j-1] + m(P[j], a_{i,h}) \\ M[i-1, j] + g \\ M[i, j-1]^h + g \end{cases} \quad (1)$$

$s_{i-1} = s_i = s > 1$  and rows  $i-1, i$  fall within the same degenerate position. In this case, this row still represents  $s$  alternative letters  $a_{i,1}, \dots, a_{i,s_i}$  and each entry  $M[i, j]$  will contain tuples of size  $s_i$  computed as follows for  $1 \leq h \leq s_i$ :

$$M[i, j]^h = \min \begin{cases} M[i-1, j-1]^h + m(P[j], a_{i,h}) \\ M[i-1, j]^h + g \\ M[i, j-1]^h + g \end{cases} \quad (2)$$

$s_{i-1} = s > 1$  and  $s_i = 1$ . In this case, row  $i-1$  was the last letter of a degenerate position, and row  $i$  is a solid position representing a single letter  $\hat{T}[i]$ . The entry  $M[i, j]$  will contain a value computed as follows:

$$M[i, j] = \min \begin{cases} M[i-1, j-1]^1 + m(P[j], \hat{T}[i]) \\ \dots & \dots & \dots \\ M[i-1, j-1]^{s_{i-1}} + m(P[j], \hat{T}[i]) \\ M[i-1, j]^1 + g \\ \dots & \dots \\ M[i-1, j]^{s_{i-1}} + g \\ M[i, j-1] + g \end{cases} \quad (3)$$

$s_{i-1} > 1$  and  $s_i = s > 1$  and rows  $i-1, i$  fall within the same degenerate position. In this case, row

$i - 1$  was the last letter of a degenerate position and row  $i$  is the first of a new different degenerate position. This row represents  $s$  alternative letters  $a_{i,1}, \dots, a_{i,s_i}$  and each entry  $M[i, j]$  will contain tuples of size  $s_i$  computed as follows for  $1 \leq h \leq s_i$ :

$$M[i, j]^h = \min \begin{cases} M[i-1, j-1]^1 + m(P[j], a_{i,h}) \\ \dots \\ M[i-1, j-1]^{s_{i-1}} + m(P[j], a_{i,h}) \\ M[i-1, j]^1 + g \\ \dots \\ M[i-1, j]^{s_{i-1}} + g \\ M[i, j-1]^h + g \end{cases} \quad (4)$$

For example, Figure 1 shows the DP table for the alignment of the D-string AC[GC/AT]A against ACGTA.

### 3.2 WFA with D-strings

In this section we show how to perform Wave Front Alignment of Partially Ordered D-Strings with affine gap penalty score in almost linear time using a D-strings customization of Wave Front Alignment.

Similarly to (S.Marco-Sola et al., 2021), we replace the three affine gap penalty dynamic programming matrices  $I, D, M$  of size  $W \times m$  each ( $W$  being the width of  $\hat{T}$  and  $m$  the length of  $P$ ) with as many *wave front* records  $\tilde{I}_d, \tilde{D}_d, \tilde{M}_d$  starting with an initial distance  $d = 0$ . The value of  $d$  can thus only increase along the computation, and for each partial score  $d$ , the records  $\tilde{I}_d$  (resp.  $\tilde{D}_d, \tilde{M}_d$ ) store the following values:

- (i) the values  $lo_d$  and  $hi_d$  that define the range of diagonals that allow to reach alignment score  $d$ ;
- (ii) for each diagonal  $k$  in such range: tuple  $off_{d,k}$  for offsets, and tuple  $abd_{d,k}$  for abandoned.

Indeed, the values  $lo_d, hi_d$  define the span of  $WF_d$ . Initially,  $lo_d = hi_d = 0$  and the only single starting diagonal is the main diagonal  $k = W - m$ . For each partial distance  $d$  and diagonal  $k$ ,  $off_{d,k}$  stores the furthest-reaching offset in diagonal  $k$  that scores  $d$ , while the boolean tuple  $abd_{d,k}$  tells us whether, for having that distance  $d$  along that diagonal  $k$ , we should abandon a specific variant of the tuple. We will use the generic notation  $off_{d,k}$  as well as the more specific notation  $off_{d,k}^D$  (resp.  $off_{d,k}^I, off_{d,k}^M$ ) to specifically mean the offsets in  $\tilde{D}_d$  (resp.  $\tilde{I}_d, \tilde{M}_d$ ).

With the wave-front, there is no need to spend a quadratic  $W \times m$  space to store the matrices, nor to spend a quadratic time to fill them in. Observe, however, that  $d, k$ , and  $off_{d,k}$ , actually identify a specific diagonal and how far you can go down along it while keeping score  $d$ , and therefore  $k$  and  $off_{d,k}$  actually define a cell in what would have been the matrices

$D, I$  and  $M$ . Let us name  $u$  the row and  $v$  the column of the entry defined by  $off_{d,k}$ . Now, if  $\hat{T}[u]$  is a single/solid letter, then the offset  $off_{d,k}^D, off_{d,k}^I$ , and  $off_{d,k}^M$  are a single value, and else they are a tuple of size  $s_u$ : a wave front for each variant in  $\hat{T}[u]$ .

Starting with initial distance  $d = 0$  and the sole central diagonal, we compute the wavefront for growing values of  $d$  until the bottom right cell has been reached by the wavefront. The value of  $d$  is increased, and new offsets are computed, when the borders of the wave front have been reached: no further match is possible, and either a mismatch score  $x$  must be accounted (and new offset computed along the same diagonal  $k$  for distance  $d + x$ ), or a gap must be accounted and therefore new diagonals must be explored (and new offset computed for distance that has increased according to the gap penalty function). This is done according to the formulae (5), (6), and (7) below that formalize the recurrence relations that show how to update the offsets information; these are basically those of (S.Marco-Sola et al., 2021) with the tuple information added.

Formula (5) computes the new offsets for  $\tilde{I}_{d,k}$ , assuming the general case in which the entry on which we do the recurrence contains a tuple.

$$off_{d,k}^I = 1 + \max \begin{cases} 1 + \max \text{ in tuple } off_{d-o-e,k-1}^M \\ 1 + \max \text{ in tuple } off_{d-e,k-1}^I \end{cases} \quad (5)$$

When computing  $\tilde{I}_{d,k}$  we are assuming that the last event has been an insertion, in which case the previous event could either have been (i) a (mis)match, in which case a new gap is being opened, or (ii) another insertion, in which case an existing gap is being extended. In case (i) the new offset is that of  $\tilde{M}_{d-o-e}$  because the distance is increased by  $o + e$ , while in case (ii) the new offset is that of  $\tilde{I}_{d-e}$  because the distance is only increased by the gap extension  $e$ . In both cases, the new offset has to be picked from the preceding diagonal  $k - 1$ .

Dually, the following formula computes the new offsets for  $\tilde{D}_{d,k}$  assuming the general case in which the entry on which we do the recurrence contains a tuple:

$$off_{d,k}^D = \max \begin{cases} 1 + \max \text{ in tuple } off_{d-o-e,k+1}^M \\ 1 + \max \text{ in tuple } off_{d-e,k+1}^D \end{cases} \quad (6)$$

Since  $\tilde{D}_{d,k}$  is computed assuming that the last event has been a deletion, the previous event could either have been a (mis)match, in which case a new gap is being opened, or another deletion, in which case an existing gap is being extended. The new offset is then

either in  $\tilde{M}_{d-o-e}$  or in  $\tilde{D}_{d-e}$ , and in both cases the new offset has to be picked from the next diagonal  $k-1$ .

Finally, we show how to compute the new offsets for  $\tilde{M}_{d,k}$ :

$$off_{d,k}^M = \max \begin{cases} off_{d,k}^I & \text{(Insertion)} \\ off_{d,k}^D & \text{(Deletion)} \\ 1 + \max \text{ in tuple } off_{d-x,k}^M & \text{(Mismatch)} \end{cases} \quad (7)$$

The wavefront boundaries for  $\tilde{M}_{d,k}$  are computed assuming that the last event has been a (mis)match, in which case we stay in the same diagonal  $k$ , and the previous event could either have been another (mis)match, or an insertion or deletion. The new offset is then either in  $\tilde{M}_{d-x}$  or in  $\tilde{D}_d$  or  $\tilde{I}_d$ . In the latter formula, the case of a match is not taken into account because these offsets are recomputed only when a match is not available.

Notice that, among possible values, the maximum is always sought here because the distance is a fixed parameter  $d$  in  $\tilde{M}_{d,k}, \tilde{I}_{d,k}, \tilde{D}_{d,k}$ , and we want to go as far as possible along  $k$  keeping that distance (that can only increase otherwise), whatever alignment event must be assumed. In this way, we eventually minimize the final  $d$  whose  $WF_d$  will reach the bottom right final entry of the matrix. In all formulae above, whenever incurring in a gap and increasing or decreasing the diagonal number, then this has to be intended as updating  $lo_d$  or  $hi_d$ .

The pseudocode of our algorithm is shown in Appendix A. The main function, DWF\_ALIGN, calls DWF\_EXTEND to extend the wavefront for all diagonals  $k$  between  $lo_d$  and  $hi_d$  by means of the function DWF\_ROLL, which is the core of DSA as this is where the offsets of  $\tilde{M}_{d,k}$  are actually increased because at least one match in the tuple takes place, making  $WF_d$  roll along diagonal  $k$ . When this happens, then the function DWF\_ROLL:

(i) Possibly updates the width of the offset tuple with that of the current row and resets its content to account for the case in which we extend a wavefront from row  $i$  to row  $i+1$  such that  $|\hat{T}[i]| \neq |\hat{T}[i+1]|$  (the size of the tuple changes) or we are anyhow stepping into a different degenerate letter.

(ii) It checks, for each possible variant, whether there is a match and it has not been abandoned, in which case the offset is incremented. Else, it sets the variant as abandoned.

When DWF\_ROLL cannot extend further the  $WF_d$  by means of matches, then the lead gets back to DWF\_ALIGN which increases  $d$  and calls the function DWF\_NEXT that sets the offsets according to formulae (5), (6), and (7) to update the borders of the wavefront.

The final result we have obtained is an exact algorithm that computes the edit distance (or any distance accounting for insertions, deletions, and substitutions with any desired score, including affine gap penalty score), between a string and a D-string in time and space proportional to  $D \cdot N$ , where  $N$  is the size of the D-string, and  $D$  is the final edit distance. This is regardless of the alphabet size.

We remark that both the dynamic programming solution we suggested can naturally make use of weights associated to single letters or positions to be included in the optimization distance function.

We implemented our algorithm in a prototype tool DSA that computes the edit distance and an optimal alignment (thus keeping all the information needed for tracing back the optimal path).

## 4 PRELIMINARY EXPERIMENTAL VALIDATION

To the best of our knowledge, there is no published software tool implementing an algorithm specifically designed for a global alignment of a linear string and a D-text. Some algorithm are designed for *semiglobal* alignment, that is with the pattern being (substantially) shorter than the D-string (E.Garrison et al., 2018; J.M.Eizenga et al., 2021; A.Cisłak et al., 2018; A.Cisłak and S.Grabowski, 2020; M.Federico and N.Pisanti, 2009; C.A.Darby et al., 2020); some are designed for more general purposes, such as aligning a linear string to a graph structure that generalizes D-strings (E.Garrison et al., 2018; J.M.Eizenga et al., 2021; C.A.Darby et al., 2020; M.Rautiainen et al., 2019; M.Rautiainen and T.Marschall, 2020; H.Li et al., 2020). Some of these and other tools are not exact and hence do not guarantee to find the optimal alignment or solution (e.g (M.Rautiainen and T.Marschall, 2020; M.Rautiainen et al., 2019; H.Li et al., 2020; H.Li, 2016; H.Li, 2018; H.Li, 2021). Finally, some algorithms that basically solve a very similar problem, actually do not compute any distance or alignment, but rather output a consensus string that suitably merges the input strings (Y.Gao et al., 2021; P.Ivanov et al., 2020; P.Ivanov et al., 2022b; P.Ivanov et al., 2022a).

The Variation Graph Toolkit VG suite (E.Garrison et al., 2018; J.M.Eizenga et al., 2021) contains a tool that can solve STODS aligning a string to a more general data structure than D-strings (Variation Graphs indeed), but this is specifically designed to map reads that are shorter than the graph/text, rather than making a global alignment, and therefore a comparison would not be fair. The same holds for VARGAS (C.A.Darby

et al., 2020) as well as for SOPANG and SOPANG2 (A.Cisłak et al., 2018; A.Cisłak and S.Grabowski, 2020) that, moreover, only detect exact matches without allowing gaps nor mismatches. We thus considered the tool abPOA (Y.Gao et al., 2021), a C library tool to align a sequence to a directed acyclic graph that also uses partial order alignment and supports global alignment, and which is the state of the art as of base-level exact alignments, but it cannot handle sequences as long as 100,000b. The same holds for the tool Astarix (P.Ivanov et al., 2020; P.Ivanov et al., 2022b; P.Ivanov et al., 2022a).

We therefore compared our DSA with GRAPHALIGNER (M.Rautiainen and T.Marschall, 2020; M.Rautiainen et al., 2019) and MINIGRAPH (H.Li et al., 2020; H.Li, 2016; H.Li, 2018; H.Li, 2021) on solving STODS. Both of them are designed to align strings on a more general graph structure than D-strings and therefore the comparison we show below should be viewed as a validation of the performance of DSA in solving STODS, and not as claiming that DSA is in general a better tool than any of the other two. GRAPHALIGNER uses a seed and extend method and the bitvector alignment extension algorithm of (M.Rautiainen et al., 2019). MINIGRAPH is a well maintained and highly optimized software tool that uses minimizers to find strong colinear chains as starting point to build the alignment (H.Li et al., 2020; H.Li, 2016; H.Li, 2018; H.Li, 2021). Therefore, out of DSA, GRAPHALIGNER, and MINIGRAPH, our DSA is the only one which does base-level exact alignments.

**D-strings generation.** We randomly generated a D-string of width  $W = 100,000b$  by first generating a random string of length  $W$  on  $\{A, C, G, T\}$ , and then inserting<sup>2</sup> therein  $deg$  (input parameter given as a percentage of  $W$ ) degenerate non-solid positions as follows: for each such position we pick at random a value for its size between 1 and  $S$  (another input parameter), and a randomly chosen length between 1 and  $L$  (input parameter again). We generated D-strings using width  $W = 100,000b$  in all tests, degeneracy frequencies values  $deg = 1\%, 10\%$ , maximum variance values  $S = 2, 5$ , and maximum variant lengths  $L = 1, 4$ . As a consequence the width of tested D-strings will always be  $W = 100,000b$ , while its total size  $N$  will depend from input parameters  $deg, S, L$ .

**Pattern generation.** From the obtained synthetic D-string  $\hat{T}$ , we extracted a ground truth exact pattern  $P_0$  of size  $W$  (that is, a string  $P \in \hat{T}$  that thus matches  $\hat{T}$  with distance 0), and we (possibly) modified  $P_0$

<sup>2</sup>The insertion was done forcing the width to remain  $W$ .

into the actual input query  $P$  with different possible divergences using real .vcf files<sup>3</sup>. The divergences we tested were to insert (i) no divergence at all, (ii) 0, 1% SNPs, (iii) 1% SNPs, (iv) 0, 1% INDELS; (percentages are on  $W$ ). Hence, the size  $m = |P|$  of the query string will be in  $\Theta(W)$  and so will the distance  $d$  between  $P$  and  $\hat{T}$ .

We have run experiments for all values  $deg, S, L$  mentioned above in D-string generation paragraph, resulting in D-strings of size ranging from  $N = 101,000$  (for  $deg = 1, S = 2, L = 1$ ) to  $N = 160,327$  (for  $deg = 10, S = 5, L = 4$ ). All tests were ran on a laptop (single threaded) Intel® Core™ i7-11800H  $\times 16$  with 16.0 GiB RAM. Space and time was measured using `/usr/bin/time -f"%S\t%M"` to extract system time (seconds) and maximum resident set size (kbytes). Time was reported 0 when  $< 0.001s$ . In all tests we used alignment scores  $a = 0, x = 1, o = 2, e = 1$ . For space reasons, we only report results for two parameters' sets that sample the comparative results. Table 1 shows results with a D-string of size  $N = 106,147$  generated with  $deg = 1\%$  of degenerate positions with up to  $S = 5$  variants of length up to  $L = 4$  (little frequency of highly degenerated positions). Table 2 shows results with a D-string of size  $N = 110,000$  generated with  $deg = 10\%$  of degenerate positions with up to  $S = 2$  variants of length up to  $L = 1$  (high frequency of little degenerated positions).

We report time and memory peak, as well as the number of detected events on optimal alignment: number = for matches, X for mismatches, I for insertions, and D for deletions. With no pattern divergence, then for a correct alignment it must be  $I = D = X = 0$  and 100,000 matches. When the pattern divergence is only SNPs, then it must be  $I = D = 0$ , and  $X$  should be approximately equal to the number of SNPs (as by chance the divergence may not change the DNA base). For the experiments involving INDELS divergence in the pattern, we also report the number  $G$  of gaps that are opened: with  $W = 100,000b$  and 0.1% INDELS, in a correct alignment it must be  $G = 100$ . For accuracy evaluation, for each experiment the last line shows the ground truth.

In all experiments, and for all tools, time was below 0.2 seconds. For both data sets (Tables 1, 2), in the first experiment (no pattern divergence) for an exact match, DSA always finds (also in those not shown here) the exact solution with 100000 matches, 0 mismatches, and 0 gaps (like GRAPHALIGNER does) taking less memory than MINIGRAPH and much less than GRAPHALIGNER. In the second and third exper-

<sup>3</sup>The .vcf file format is the standard in bioinformatics to encode variants such as SNPs (letter substitutions) and INDELS.

Table 1: D-string of size  $N = 106,147$  with degeneracy  $deg = 1\%$ ,  $S = 5$ ,  $L = 4$ .

Tool	Pattern divergence	Peak memory (kbytes)	Time (seconds)	Alignment events
DSA	none	3188	0	100,000= 0X 0I 0D
GRAPHALIGNER	"	7418	0	100,000= 0X 0I 0D
MINIGRAPH	"	5585	0	99989= 0X 0I 0D
	Ground truth	—>	—>	100000= 0X 0I 0D
DSA	0.1 % SNPs	44096	0.005	99900= 100X 0I 0D
GRAPHALIGNER	"	26799	0.004	99901= 99X 0I 0D
MINIGRAPH	"	8038	0	99889= 100X 0I 0D
	Ground truth	—>	—>	99900= 100X 0I 0D
DSA	1 % SNPs	475668	0.120	99007= 993X 0I 0D
GRAPHALIGNER	"	26682	0.005	99005= 993X 2I 2D
MINIGRAPH	"	7910	0.001	98989= 1000X 0I 0D
	Ground truth	—>	—>	99007= 993X 0I 0D
DSA	0.1 % INDELS	131311	0.018	99813= 0X 135I 174D 100G
GRAPHALIGNER	"	26816	0.003	99717= 109X 135I 174D 145G
MINIGRAPH	"	8213	0.001	99813= 1X 136I 175D 100G
	Ground truth	—>	—>	99813= 0X 135I 174D 100G

Table 2: D-string of size  $N = 110,000$  with degeneracy  $deg = 10\%$ ,  $S = 2$ ,  $L = 1$ .

Tool	Pattern divergence	Peak memory (kbytes)	Time (seconds)	Alignment events
DSA	none	3174	0	100,000= 0X 0I 0D
GRAPHALIGNER	"	16405	0	100,000= 0X 0I 0D
MINIGRAPH	"	5609	0	99994= 0X 0I 0D
	Ground truth	—>	—>	100000= 0X 0I 0D
DSA	0.1 % SNPs	46942	0.005	99900= 100X 0I 0D
GRAPHALIGNER	"	50386	0.004	99903= 97X 0I 0D
MINIGRAPH	"	8078	0	99903= 97X 0I 0D
	Ground truth	—>	—>	99900= 100X 0I 0D
DSA	1 % SNPs	456316	0.120	99022= 978X 0I 0D
GRAPHALIGNER	"	50250	0.005	99020= 976X 4I 0D
MINIGRAPH	"	7934	0.001	98947= 997X 0I 0D
	Ground truth	—>	—>	99022= 978X 0I 0D
DSA	0.1 % INDELS	105904	0.018	99817= 0X 183I 171D 100G
GRAPHALIGNER	"	50223	0.003	99708= 109X 183I 171D 160G
MINIGRAPH	"	8145	0.001	99806= 0X 183I 171D 100G
	Ground truth	—>	—>	99817= 0X 183I 171D 100G

iments we introduced SNPs (in 0.1% and 1% of the positions, respectively): DSA is the only one which is exact at a cost of higher memory consumption. Finally, with INDELS, our prototype is always exact (and MINIGRAPH almost is). GRAPHALIGNER is not accurate as it is not designed for affine gap penalty function.

Summing up, DSA (I) outperforms both MINIGRAPH and GRAPHALIGNER on exact match, (II) outperforms accuracy of MINIGRAPH with SNPs, and (III) outperforms accuracy of GRAPHALIGNER with INDELS.

## 5 CONCLUSIONS AND FURTHER WORK

As mentioned above, at the moment our implementation of DSA is just a promising prototype. We plan to improve its memory consumption using the ideas of (J.M.Eizenga and B.Paten, 2022). Also, we are working on improve its speed (as well as consequent memory use) by removing hopeless diagonals suffixes that are currently being kept in the range.

Finally, we remark that with our dynamic programming method, it is very natural to add weights to let-

ters and to use a sum of weights modification of the actual score as objective function, in order to account for possible useful metadata such as confidence level in the query string or D-strings bases, or abundance in the MSA represented by the D-string.

## ACKNOWLEDGEMENTS

This paper is part of a project ALPACA (ALgorithms for PAngenome Computational Analysis) that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 956229. Co-financed by the Connecting Europe Facility of the European Union.

## REFERENCES

- A.Cisłak, S.Grabowski, and J.Holub (2018). SOPanG: online text searching over a pan-genome. *Bioinformatics*, 34(24):4290–4292.
- A.Cisłak and S.Grabowski, S. (2020). SOPanG 2: online searching over a pan-genome without false positives. *arXiv:2004.03033*.
- C.A.Darby, R.Gaddipati, M.C.Schatz, and B.Langmead (2020). Vargas: heuristic-free alignment for assessing linear and graph read aligners. *Bioinformatics*, 36(12):3712–3718.
- C.Lee, C.Grasso, and M.F.Sharlow (2002). Multiple Sequence Alignment Using Partial Order Graphs. *Bioinformatics*, 18(3):452–464.
- CPC, T. C. P.-G. C. (2018). Computational Pan-Genomics: Status, Promises and Challenges. *Briefings in Bioinformatics*, 19(1):118–135.
- C.S.Iliopoulos and J.Radoszewski (2016). Truly Subquadratic-Time Extension Queries and Periodicity Detection in Strings with Uncertainties. In *27th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 54 of *LIPICs*, pages 8:1–8:12.
- C.S.Iliopoulos, R.Kundu, and S.P.Pissis (2017). Efficient Pattern Matching in Elastic-Degenerate Texts. In *11th International Conference on Language and Automata Theory and Applications (LATA)*, volume 10168 of *Springer LNCS*, pages 131–142.
- D.-F.Feng and Doolittle, R. (1987). Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *Journal of Molecular Evolution*, 25(4):351–360.
- E.Birmelé, P.Crescenzi, R.A.Ferreira, R.Grossi, V.Lacroix, A.Marino, N.Pisanti, G.A.T.Sacamoto, and M.-F.Sagot (2012). Efficient bubble enumeration in directed graphs. In *19th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7608 of *Springer LNCS*, pages 118–129.
- E.Garrison, J.Sirén, A.M.Novak, G.Hickey, J.M.Eizenga, E.T.Dawson, W.Jones, S.Garg, C.Markello, M.F.Lin, B.Paten, and R.Durbin (2018). Variation Graph Toolkit Improves Read Mapping by Representing Genetic Variation in the Reference. *Nature Biotechnology*, 36(9):875–879.
- E.W.Myers (1986). An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266.
- G.Bernardini, N.Pisanti, S.P.Pissis, and G.Rosone (2017). Pattern matching on elastic-degenerate text with errors. In *24th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 10508 of *Springer LNCS*, pages 74–90.
- G.Bernardini, N.Pisanti, S.P.Pissis, and G.Rosone (2020). Approximate pattern matching on elastic-degenerate text. *Theoretical Computer Science*, 812:109–122.
- G.Bernardini, P.Gawrychowski, N.Pisanti, S.P.Pissis, and G.Rosone (2019). Even Faster Elastic-Degenerate String Matching via Fast Matrix Multiplication. In *46th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 132 of *LIPICs*, pages 21:1–21:15.
- G.Bernardini, P.Gawrychowski, N.Pisanti, S.P.Pissis, and G.Rosone (2022). Elastic-Degenerate String Matching via Fast Matrix Multiplication. *SIAM Journal of Computing*, 51(3):549–576.
- H.Li (2016). Minimap and Miniasm: Fast Mapping and de Novo Assembly for Noisy Long Sequences. *Bioinformatics*, 32(14):2103–2110.
- H.Li (2018). Minimap2: Pairwise Alignment for Nucleotide Sequences. *Bioinformatics*, 34(18):3094–3100.
- H.Li (2021). New strategies to improve minimap2 alignment accuracy. *Bioinformatics*, 37(23):4572–4574.
- H.Li, X.Feng, and C.Chu (2020). The Design and Construction of Reference Pangenome Graphs with Minigraph. *Genome Biology*, 21(265).
- H.Soldano, A.Viari, and M.Champesme (1995). Searching for flexible repeated patterns using a non-transitive similarity relation. *Pattern Recognition Letters*, 16(3):233–246.
- IUPAC-IUB Commission on Biochemical Nomenclature (1970). Abbreviations and symbols for nucleic acids, polynucleotides, and their constituents. *Biochemistry*, 9(20):4022–4027.
- J.M.Eizenga, A.M.Novak, E.Kobayashi, F.Villani, C.Cisar, S.Heumos, G.Hickey, V.Colonna, B.Paten, and E.Garrison (2021). Efficient dynamic variation graphs. *Bioinformatics*, 36(21):5139–5144.
- J.M.Eizenga and B.Paten (2022). Improving the time and space complexity of the wfa algorithm and generalizing its scoring. *bioRxiv*.
- K.Abrahamson (1987). Generalized string matching. *SIAM Journal of Computing*, 16(6):1039–1051.
- K.Aoyama, Y.Nakashima, T.I. S.Inenaga, H.Bannai, and M.Takeda (2018). Faster Online Elastic Degenerate String Matching. In *29th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 105 of *LIPICs*, pages 9:1–9:10.

- M.Alzamel, L.A.K.Ayad, G.Bernardini, R.Grossi, C.S.Iliopoulos, N.Pisanti, S.P.Pissis, and G.Rosone (2018). Degenerate String Comparison and Applications. In *18th International Workshop on Algorithms in Bioinformatics (WABI)*, volume 113 of *LIPICs*, pages 21:1–21:14.
- M.Alzamel, L.A.K.Ayad, G.Bernardini, R.Grossi, C.S.Iliopoulos, N.Pisanti, S.P.Pissis, and G.Rosone (2020). Comparing Degenerate Strings. *Fundamenta Informaticae*, 175(1-4):41–58.
- M.Crochemore, C.S.Iliopoulos, T.Kociumaka, J.Radoszewski, W.Rytter, and T.Walen (2017). Covering problems for partial words and for indeterminate strings. *Theoretical Computer Science*, 698:25–39.
- M.Federico and N.Pisanti (2009). Suffix tree characterization of maximal motifs in biological sequences. *Theoretical Computer Science*, 410(43):4391–4401.
- M.Rautiainen and T.Marschall (2020). Graphaligner: rapid and versatile sequence-to-graph alignment. *Genome Biology*, 21(253).
- M.Rautiainen, V.Mäkinen, and T.Marschall (2019). Bit-parallel sequence-to-graph alignment. *Bioinformatics*, 35(19):3599–3607.
- N.M.Mwaniki and N.Pisanti (2022). Optimal sequence alignment to ED-strings. In *18th International Symposium on Bioinformatics Research and Applications (ISBRA)*, volume 13760 of *Springer LNCS*.
- N.Pisanti, H.Soldano, and M.Carpentier (2005). Incremental inference of relational motifs with a degenerate alphabet. In *16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 3537 of *Springer LNCS*, pages 229–240.
- N.Pisanti, H.Soldano, M.Carpentier, and J.Pothier (2009). A relational extension of the notion of motifs: Application to the common 3d protein substructures searching problem. *Journal of Computational Biology*, 16(12):1635–1660.
- P.Gawrychowski, S.Ghazawi, and G.M.Landau (2020). On indeterminate strings matching. In *31st Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 161 of *LIPICs*, pages 14:1–14:14.
- P.Ivanov, B.Bichsel, H.Mustafa, A.Kahles, G.Rätsch, and M.T.Vechev (2020). Astarix: Fast and optimal sequence-to-graph alignment. In *24th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, volume 12074 of *Springer LNCS*, pages 104–119.
- P.Ivanov, B.Bichsel, and M.T.Vechev (2022a). Fast and optimal sequence-to-graph alignment guided by seeds. In *26th Annual International Conference on Research in Computational Molecular Biology (RECOMB)*, Springer LNCS In Press.
- P.Ivanov, B.Bichsel, and M.Vechev (2022b). Fast and optimal sequence-to-graph alignment guided by seeds. *bioRxiv*.
- P.Peterlongo, N.Pisanti, F.Boyer, do Lago, A., and M.-F.Sagot (2008). Lossless filter for multiple repetitions with hamming distance. *Journal of Discrete Algorithms*, 6(3):497–509.
- R.Grossi, C.S.Iliopoulos, C.Liu, N.Pisanti, S.P.Pissis, A.Retha, G.Rosone, F.Vayani, and L.Versari (2017). On-Line Pattern Matching on Similar Texts. In *28th Annual Symposium on Combinatorial Pattern Matching (CPM)*, volume 78 of *LIPICs*, pages 9:1–9:14.
- S.Marco-Sola, J.C.Moure, M.Moreto, and A.Espinosa (2021). Fast Gap-Affine Pairwise Alignment Using the Wavefront Algorithm. *Bioinformatics*, 37(4):456–463.
- V.Carletti, P.Foggia, E.Garrison, L.Greco, P.Ritrovato, and M.Vento (2019). Graph-based representations for supporting genome data analysis and visualization: Opportunities and challenges. In *12th International Workshop on Graph-Based Representations in Pattern Recognition (GbrPR)*, volume 11510 of *Springer LNCS*, pages 237–246.
- Y.Gao, Y.Liu, Y.Ma, B.Liu, Y.Wang, and Y.Xing (2021). abPOA: an SIMD-based C library for fast partial order alignment using adaptive band. *Bioinformatics*, 37(15):2209–2211.