A mismatch-aware Relative Lempel-Ziv parser to detect Genome Rearrangements

Rodrigo T. Rocha[1] and Georgios Pappas Jr[1]

Department of cell biology, University of Brasilia, Brazil gpappas@unb.br

1 Introduction

Lempel-Ziv algorithms have been used in genomics for a while and found their use in different branches of genomics, both in the analysis of the structures (elements) of a genomic sequence [12, 11] and as a metric for the construction of phylogenetic trees [28]. However, over the years and the advent of high-throughput sequencing, a flood of genome sequences have been generated by independent laboratories and large consortia. Perspectives indicate that the rate of production of these data will not cease, on contrarily, it is estimated that 600 million human genomes will be deposited in databases worldwide by 2025 [2]. Due to this reason, most of the Lempel-Ziv algorithms developed and improved in the last two decades were focused on sequence compression in order to optimize the use of disk spaces and the traditional use of these algorithms as a bioinformatics tool to gain biological insights have been left besides. Briefly, the Lempel-Ziv algorithm detects the substrings (i.e. phrases) previously encountered when viewing the sequence from beginning to end. The process to define the phrases is called Lempel-Ziv parsing or factorization (LZ parsing hereafter) and the final number of different substrings found can be viewed as a measure of the complexity (entropy) of the analysed sequence [21, 39, 40].

The genome-wide evolutionary patterns left by genomic rearrangements have radically shaped the genomes of a multitude of species [4, 24, 27, 29]. Genomic rearrangements such as insertions, deletions, duplications, inversions and translocations are defined as structural variations ranging from hundred bases pairs to megabases that join normally distant sequences [23]. The genome comparison of multiple related individuals have shed light to the impact of genomic rearrangements on different phenotypes, including many diseases [1, 35, 34, 17]. In order to detect the presence of rearrangements between two or more sequences, an algorithm must not solely rely on global alignment approach [3], given the intrinsic assumption that individual bases are in the same order in two or more sequences, precluding its use to detect genome rearrangements [37]. A popular alternative is the use of regional constraints, in the form of gene lists, following the expectation that gene clusters form conserved blocks [38, 14]. To operate on a sequence-level instead of an ordered gene lists, alignment-free methods based on probabilistic methods allied to k-mer spectrum of sequences [15, 31] or a mixture of global and local alignment approaches have been employed to find rearrangements [3].

Before the use of LZ algorithms as a compression tool in genomics [**?**], it has been used as a tool to discover sequence features inside a genome. A method has

been proposed to detect local regularities in a genome based on the extent of deviation in a complexity measure, defined as the number of phrases generated by LZ parsing a sliding windows of fixed size along a genome [12]. A local region was considered important if the complexity measure of this region deviates from the mean number of phrases calculated by parsing all sliding windows within a genome. The individual analysis of regions deviating from the mean, often exhibited conserved structural regularities found in DNA sequences such as stem-loop structures, tandem repeats, directed or inverted repeats. Beyond the analysis of local regions, statistics of the complexity measurement were used to estimate the complexity of genomes [11]. However, at that time, scant availability of genome sequences and high demand on computational resources restricted the analysis to small genomes (up to few megabases). In addition, the analysis was focused on self-genome comparison instead of a comparison between different genomes. Genome compression was another line of research tackled by LZ algorithm. In this approach, several related genomes are taken together and one genome sequence, the reference genome, is subjected to LZ parsing. The other genomes are encoded with the ensuing phrases from the query genome leading to a substantial compressed representation of a query genome in relation to a reference [19, **?**]. This LZ variant has been known as Relative Lempel-Ziv and encompasses a family of algorithms with different optimization and encoding schemes aiming to improve compression ratios of genomic collections [19, 20, 6, 7, 5]. However, as far as we know, none have been used as a bioinformatics tool to analyse genomic sequences, let alone in the detection of genome rearrangements.

In this paper, we leverage concepts deriving out of an efficient LZ algorithm [18, 25] and ideas from genomic read aligners based on Burrows-Wheeler transform (BWT) [22, 16] to devise the Relative Lempel-Ziv Inexact algorithm, an RLZ based algorithm that allows mismatches. The utility of this implementation is two fold. First, it could reduce the number of generated phrases leading to improvements on the compression ratio when dealing with collections of genomes belonging to the same species in which the main variations are point mutations such as single nucleotide polymorphism (SNP). To use it with this aim, a schema to compress the information concerning the mismatches implicitly parsed within the phrases must be think ahead - which is not the main aim of this paper. Second, the utility concerns the biological meaning of the phrases generated by RLZI algorithm. The phrases generated by the RLZ algorithm of a genome with respect to a reference genome must be permissive enough to allow the detection of related genomic regions that have meaningfully importance. For instance, we expect to detect genomic rearranged regions that have undergone mutations to yield a contiguous instead of being fragmented every time a SNP has been found.

Therefore, the purpose of this paper is to take advantage of the RLZ nature for comparative genomics, in the traditional implementation of RLZ the phrases obtained are not contiguous enough due to nucleotide variations among the genome used as a reference for parsing and the query genome. This implies that large syntenic regions between the genomes are not detected by traditional RLZ parsing algorithms. In contrast, the proposed RLZI algorithm allows

m-mismatches (m is a parameter) between phrases, allowing the detection of extensive synthetic regions. The paper begins with a formal definition of the concepts used by the algorithm, as well as a brief review of the LZ and RLZ traditional algorithms and the main data structures used. In section 3, the proposed algorithm, RLZI, is described and a concrete example is presented. After that, in section 4, the empirical correcteness of RLZI is evaluated through an example and the application of the algorithm aimed at detecting genomic rearrangements is demonstrated through simulated and real datasets.

2 Preliminaries

2.1 Strings

A *string* is defined as $T[1..|T|]$, where T is a finite sequence of $|T| = n$ symbols drawn from an alphabet Σ of size $|\Sigma|$. The i-th element of T is $T[i]$ and the substring from i-th to the j-th elements inclusively is $T[i:j]$. A prefix of T is a substring starting from first position of T, denoted $T[1:j]$, while a suffix of T is a substring $T[i:n]$ ending at the last position of T. The sequence T read backwards is denoted $T^{rev} = T[n \dots 1]$. The concatenation of two strings Q of length $|Q|$ and S of length $|S|$ is denoted QS. The length of $QS = |Q| + |S|$. A collection of l strings (i.e. sequences) is a set $S(\mathcal{T}) = \{T_1, T_2, \dots, T_l\}$. The concatenation of the l sequences within the set $S(\mathcal{T})$ yield the string $\mathcal{T} = T_1\$T_2\$ \dots T_l\$$ with each sequence T_i for $1 \leq i \leq l$ terminated with a special symbol \$ whose length is $|\mathcal{T}| = (|T_1| + 1) + (|T_2| + 1) + \cdots + (|T_l| + 1)$.

2.2 Lempel-Ziv parser

Let T be a given sequence over alphabet Σ, Lempel-Ziv algorithms considered that T can be generated through two operations: generation of a new symbol from Σ or copying of a substring already generated. Thus each operation generate a substring either of length one (new symbol operation) or more (copying operation). The act of partition T (i.e., generate T by a set of aforementioned operations) into substrings is known as parsing. Thus, after parsing, T can be represented by the concatenation of generated substrings (phrases):

$$T = T[1:i_1]T[i_1+1:i_2] \dots T[i_k+1:i_{k+1}] \dots [i_m+1:n]$$

Note that if a phrase is generated by a copying operation, it must be the longest already seen substring. Thus, if we have already parsed $T[1:i_1]$, the next phrase $T[i_1+1:i_2]$ is the longest prefix of $T[i_1+1 \dots n]$ that has already appeared in $T[1:i_1]$ appended with the unseen stretch of symbols. The Lempel-Ziv parsers were built as a method to compress strings/text and are specially useful for repetitive texts [26]. In fact, Lempel-Ziv (76) proposed the number of phrases z in which a string/text is parsed as a measurement of complexity based on their repetitiveness.

Many variants of the Lempel-Ziv compression algorithm were developed over the years with subtle differences among them [21, 39, 40]. The LZ76 is the "original" algorithm and is similar to the parsing aforementioned described with a difference in the copying operation [21]. This algorithm states that the copying substring is composed by the concatenation of the previous longest seen substring $T[i_k : i_{k+1}]$ plus the additional unseen character $T[i_{k+1}+1]$ located at the next position after the copied substring end. Therefore, every previous substring of T can be used to form a new phrase and the algorithm encodes the phrase as the triple $(i_k, i_{k+1}-i_k, T[i_{k+1}+1])$. Notice that the triple is formed, respectively, by the position, length of the phrase and unseen character. In order to compress a text T using less bits and more rapidly, the popular variant of LZ76, namely LZ77, was proposed [39]. It achieves this by restricting the locations in which a phrase can be found by a sliding window of length w, as a trade off it may not spot long enough repetitions. In the same way, in LZ78, a new phrase must be formed by extending any previous parsed phrase [40]. In exchange of increase in parsing time and bits allocated, the LZ76 is preferable to be used in repetitive text collections given that it can find long repetitions far away in the text [26].

In this sense, a set of genome sequences from a species or related species is a good representative of a collection of repetitive strings. To exploit the repetitiveness encountered in related genomic sequences, the Relative Lempel-Ziv (RLZ) algorithm has been proposed [19].

2.3 Relative Lempel-Ziv parser

Let R and S be a given sequences (i.e. text) over the same alphabet Σ, suppose that both sequences are almost identical except by N changes of characters (i.e. mismatches). The idea behind RLZ is to use a reference sequence as a dictionary to parse, in a LZ76 manner, the other sequence. In this case, suppose that R is the reference sequence, all phrases obtained by parsing the sequence S will be either a letter which does not occur in R or the longest prefix of $S[i : n]$ that matches a substring from R (similar to what was previous described for LZ76 parsing).

The generated phrases are encoded as (i_k, l_k), where i_k represent the position in R and l_k the length of the k-th phrase. For example (extracted from [5]), if

$$R = \texttt{ACATCATTCGAGGACAGGTATAGCTACAGTTAGAA}$$
$$S = \texttt{ACATGATTCGACGACAGGTACTAGCTACAGTAGAA}$$

The RLZ parsing of S relative to R generates the following phrases (1st line) and encoded phrases (2nd line):

$$\texttt{ACAT,GA,TTCGA,CGA,CAGGTA,CTA,GCTACAGT,AGAA}$$
$$(1,4),(10,2),(7,5),(9,3),(15,6),(24,3),(23,8),(32,4)$$

After the RLZ parsing, if we store the compressed index represented by the encoded phrases and the plain sequence R, we can delete S without loss of information. After the original implementation, the RLZ algorithm received many augmentations, including allowing a single mismatch character at the end of the copied phrase [20, 6] and adoption of relative pointers and run-length compression [6, 5]. All augmentations have the aim to offer higher compression ratios when compressing collections of genomes. The original RLZ algorithm uses the suffix array of the reference sequence for parsing the other sequences in $mathcalO(N\ log\ n)$ time where N is the length of all genome sequences in the collection and n the length of the reference sequence [19].

2.4 Suffix Arrays

A suffix array is a data structure storing the list of all suffixes indexes of a string R sorted lexicographically. The terminator character \$ is appended at the end of R to indicate the end of the string, therefore the suffix array SA_R of the text R has length $|R| + 1$ and contains a permutation of the integers $0 \dots |R| + 1$, such that $R[SA[0] : |R| + 1] < [SA[1] : |R| + 1] < \cdots < [SA[|R|] : |R| + 1]$. Notice that suffixes sharing the same prefix are clustered in the suffix array. For example, the prefix AT is clustered between positions 3-4 of SA in Fig. **??**. This arrangement allows efficiently search a pattern in R via binary search over SA_R given that we have access to the text R in $\mathcal{O}(|P|\ log|R|)$. The suffixes are stored by their positions requiring $\mathcal{O}(n\ log\ n)$ bits. In practise, the space used to store it is 4 times that of the text [32]. Therefore, we next describe how SA searching algorithm translates into data structures of smaller sizes.

2.5 BWT and FM-index

The Burrows-Wheeler Transform (BWT) is a paradigm widely used in compression algorithms. It rearranges the characters of a text into runs of similar characters [8, 9]. The usefulness of BWT lies in the fact that similar characters are clustered in blocks, favoring the compression. In addition, the Burrows-Wheeler transformation is reversible, meaning that we can reconstruct the original text without the aid of additional information other than the BWT [8, 9].

At the core of the reversibility and many other characteristics of BWT, is the property Last-To-First (LF) *mapping*. The LF *mapping* states that the i-th ocurrence of a character $c \in \Sigma$ on the last column of Burrows-Wheeler Matrix (BWM), build by sorting all the circular shifts of a text in lexicographic order, namely column L, corresponds to i-th occurence of c on the first column F. Basically, it means they represent the same position on text. This property allows a method to efficiently search for patterns on the BWT known as backward search algorithm. As well as LF mapping, and the suggestive name, the backward search algorithm works matching a pattern P from right-to-left (i.e. from the last character to the first). Two observations account for the effectiveness of the algorithm: all occurrences of P appear continuously in a range of rows in BWM and the characters corresponding to these range in BWT precede the occurrences

of P in the text. A method used to calculate the number of occurrences of a character $c \in \Sigma$ over an array L from the interval 1 to i (excluding i) is called the rank of c, denoted $rank_c(L, i)$ and the index identification of the i-th occurrence of c in L is calculated by $select_c(L, i)$. Generalizing, if P is a substring of a text R over an alphabet Σ and a character $a \in \Sigma$ is appended in front of P generating aP, the new range $[sp, ep]$ encompassing the pattern aP on BWT is

$$\begin{aligned} sp &= C[a] + rank_a(BWT(R), sp - 1) + 1 \\ ep &= C[a] + rank_a(BWT(R), ep) \end{aligned} \tag{1}$$

with $sp \leq ep$ if and only if aP is a substring (match) of R [8]. Notice that C is an ordered array of length $|\Sigma| + 1$ with the starting positions of a letter $\sigma \in \Sigma$ in column F. The BWT augmented with support to rank operations denotes the data structure FM-index [8]. Coupling the SA with FM-index it is possible to efficiently locate the positions in the text matching the pattern P in $\mathcal{O}(|P|)$ time using the backwards search algorithm.

Essentially, the task of building BWT is an active area of research [25, 32]. The context-wise bwt algorithm is used to build the BWT in the proposed algorithm because it constructs the BWT of a string in compressed space on nearly-uniform texts such as DNA sequences [30].

3 RLZ-Inexact Algorithm

We generalize the LZ76 parsing proposed by Navarro enabling it to work as a RLZ parsing. We also boost the algorithm with the capacity to allow mismatches when generating phrases applying concepts from short-read aligners based on BWT [22, 16]. The pseudo-code of our method is shown in Algorithm 1. The algorithm was implemented in C++ and the overall strategy begins concatenating, separated by the special character \$, the sequences of all chromosomes belonging to reference genome R generating the concatenated sequence R_c. Next, we build the suffix array A^r of the reverse concatenated sequences R_c^{rev}. The algorithm also builds the BWT of R_c reversed as the sequence of characters $L[1, n]$ in compressed succinct space (e.g. $|R_c|\mathcal{H}_k + o(n\mathcal{H}_k)$) and $\mathcal{O}(|R_c|\mathcal{H}_k)$ average time using the cw-bwt algorithm [30]. The algorithm also stores the array C informing the position of every character $c \in \Sigma$ in the first column of BWT. This concludes the definition of the FM-index data structure [9]. By applying successive backward search (BS) procedures over the pattern $P^{rev} = (S[i : i' - 1])^{rev}$ for increasing values of i' we're searching backwards for P^{rev} on the reverse reference sequence R_c^{rev}, therefore finding the ending positions of P (i.e. pattern not reversed) in R_c. Notice that the patterns are searched using backward searches over the BWT of reference genome, and thus as we increase i' in $S[i : i' - 1]$ we calculate the interval with respect to BWT of reference $[sp, ep]$ and, if the interval exists (i.e. $sp \leq ep$) we calculate i' as the finishing position of the pattern in R_c matching the left-most occurrences using a range maximum query data structure (rMq). If, at least *k-seed* successive rounds of the BS occur (*k-seed* is a parameter) we allow

a mismatch to appear in position i', otherwise we only allow exact matchings (Section 2.5). Making an analogy with short-read aligners, the *k-seed* parameter can be viewed as allowing only exact matching among the first k base pairs of the aligned reads. This heuristics leads to a decrease in the running time of the algorithm at the price of lowering the alignment accuracy. The rMq is build over the suffix array A^r informing the index of the maximum suffix array value between the interval $[sp, ep]$, namely the operation $A^r_c[rMq_{A^r}(sp, ep)]$ in Algorithm 1. We use the rMq implemented on Succinct Data Structure Library (sdsl) [10]. Similar to what has been done in the short-read aligners [22, 16], we use a minimum priority heap to store partial matching phrases based on alignment score. This allows to process the entries with the best score first. If we observe more than m mismatches (m is a parameter) within a k length bases, the phrase extension (i.e. BS steps) is interrupted. Note that if we set $m = 0$, RLZI works like a traditional RLZ algorithm [19]. The RLZI implementation is available on PUT GITHUB LINK.

Algorithm 1: RLZI($\mathcal{R}$,S,*kseed*,*mismatches*)

```
1  Concatenate the collection of sequences R (e.g. genome with multiple
     chromosomes) and reverse it to form R_c^rev[1, nR]
2  Build the suffix array A^r[1, nR] of R_c^rev
3  Build the FM-index of A^r, as the sequence L[1, nR] and array C[1, σ+1]
4  Build the structure for range maximum queries on A^r, rMq_A
5  i ← 0; p ← 1;
6  while i < |S| do
7      j ← i + 1; c ← L[p]; p ← LF(p);
       /* declare the parameters to control the number of
          mismatches, seed extension and ending position of
          phrases                                          */
8      m ← 0;d ← 0; i' ← 0;
9      [sp, ep] ← [C[c] + 1, C[c + 1]] /* start the BS with char c  */;
       /* initialize an empty priority-heap and start successive
          BS                                               */
10     heap = new priority-heap(); heap.push(i,sp,ep,i',m,p,j,d);
11     while !heap.empty() do
12         a = heap.top() /* get the alignment with highest score
            */;
13         [sp, ep] = [a.sp, a.ep];
14         c ← L[a.p]; p ← LF(a.p); i' ← nR − A^r[rMq_{A^R}(a.sp, a.ep)];
15         if m > threshold then
               /* do not continue if the number of mismatches
               found in a phrase is more than a given threshold */
16         end
17         if sp > ep then
               /* do not continue if the SA interval is not
               compatible                                   */
18         end
19         j ← a.j; d ← a.d;
20         if d ≥ kseed then
21             for each b ∈ {A,C,G,T} do
22                 sp ← C(b) + O(b, sp − 1) + 1; ep ← C(b) + O(b, ep);
23                 i' ← nR − A^r[rMq_{A^R}(sp, ep)];
24                 if sp ≤ ep then
25                     if b = R[i] then
26                         heap.push(a.i,sp,ep,i',m,p,j+1,d+1);
27                     else
28                         heap.push(a.i,sp,ep,i',m+1,p,j+1,0);
29                     end
30                 end
31             end
32         else
33             [sp, ep] ← ExactMatch(c, a.sp, a.ep);
34             if sp ≤ ep then
35                 i' ← nR − A^r[rMq_{A^R}(sp, ep)];
36                 heap.push(i,sp,ep,i',m,p,j+1,d+1);
37             end
38         end
39     end
40     return (a.i' − (a.j − a.i) − 1)
41 end
42 i ← a.j;
```

Therefore, given a genome sequence R as the "reference" and a single query sequence S, RLZI greedily parse S from left to right into phrases such that each phrase is a substring that inexactly matches some substring of R with at most k mismatches (k is a parameter). This algorithm outputs a set of phrases encoded as (i_k, l_k) where both terms refer, respectively, to the initial position in R and the length of the k-th phrase. We also store the i_k positions in the vector Q. For simplicity, we assume that the alphabets of S and R are the same. Furthermore, the algorithm outputs the compressed bitvectors $B[1..|R|]$, the array M and the compressed bitvector $I[1..|S|]$. The bitvector B have 1s in positions indicating where the reference differ from S. Given a position j between 1 and $|R|$, suppose that $B[j] = 1$, the rank operation $rank_1(B, j)$ computes in constant time the number of mismatches occurred before the position j. The vector M stores the mismatches, therefore the operation $M[rank_1(B, j)]$ outputs the mismatch character of position j if $B[j] = 1$. In addition, the compressed bitvector $I[1..|S|]$ marks with 1s the initial position of each phrase in relation to S.

The RLZI algorithm can handle the use of a reference genome with multiple chromosomes enabling the detection of genome rearrangements. This is achieved by keeping track of which chromosome a phrase belongs by storing a compressed bitvector $Ch[1..|R_c|]$ marking with 1s the initial positions of each chromosome relative to the string formed by the concatenation of each chromosome R_i within the collection $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$. The special character \$ separates each concatenated chromosome yielding the concatenated string R_c. This additional structures and constant access to R permits a modified version of the algorithm proposed to query for the j-th base of S [5]:

$$S[j] = \begin{cases} M[rank(B, j)] & \text{if } B[j] = 1 \\ R_c[Q[rank(I, j)] + j - select(I, rank(I, j)) & \\ \quad - select(Ch, rank(Ch, j))] & \text{otherwise} \end{cases} \tag{2}$$

For example, let the reference sequence be the collection of sequences $\mathcal{R} = \{R_1, R_1^{rc}, R_2, R_2^{rc}, R_3, R_3^{rc}\}$, composed by three sequences and their reverse complement denoted by R_i^{rc} for ($i \leq 3$).

$$\begin{aligned} R_1 &= \texttt{ACATCATTCGAGGACAGGTATAGCTACA} \\ R_1^{rc} &= \texttt{TGTAGCTATACCTGTCCTCGAATGATGT} \\ R_2 &= \texttt{GTCTACA} \\ R_2^{rc} &= \texttt{TGTAGAC} \\ R_3 &= \texttt{CCGA} \\ R_3^{rc} &= \texttt{TCGG} \end{aligned}$$

The concatenation of the sequences in $\mathcal{R}$ separated by the special character \$ gives the reference sequence R_c,

$$R_c = \texttt{ACATCATTCGAGGACAGGTATAGCTACA\$TGTAGCTATACC}$$
$$\texttt{TGTCCTCGAATGATGT\$GTCTACA\$TGTAGAC\$CCGA\$TCGG}$$

And let S be a sequence to be parsed by RLZI algorithm executed with parameters $k = 4$ and adjusted to forbid mismatches (i.e. $m = 0$). Furthermore, notice that sequence S is similar to reference sequence but for point mutations (red-colour characters), indels (blue-colour characters) and genome rearrangements (underline characters). The RLZI gives the following outputs (we omit the encoded phrases and focused on other outputs):

$$S = \texttt{ACATTGTAGCTATAGATTCGACGACAGGTACTAGCTACTGTAGAC}$$
$$RLZI(S|R_c) = \texttt{ACAT}, \texttt{TGTAGCTATA}, \texttt{GAT}, \texttt{TCGA}, \texttt{CGA}, \texttt{CAGGTA}, \texttt{CTA}, \texttt{GCTAC}, \texttt{TGTAGAC}$$
$$I = \texttt{100010000000001001000100100000100100001000000}$$
$$B = \texttt{000000000000000000000000000000000000000000000}$$
$$Q = \{\texttt{1, 30, 53, 8, 9, 15, 24, 23, 67}\}$$

In total, the RLZI parsing of S with respect to reference R_c, $RLZI(S|R_c)$, gives 9 factors as shown on the dotplot (Fig. 1-A) produced by flexidot tool [33]. Note the middle region of S between positions 15 to 38, the first mismatch up to the beginning of second rearrangement, is covered by 5 phrases due SNPs breaks. However, if we allow at least two mismatches per phrase (parameter $m = 2$), we parse this region in 2 phrases (Fig. 1-B, left-most frame). Regardless the number of allowed mismatches, RLZI algorithm will always break a phrase in indes (pos 31 of S). This fact is a limitation in RLZI algorithm.

4 Experimental Results

4.1 Correctness of the Algorithm

The correctness of the algorithm follows from empirical results in which we reconstruct sequence S using only RLZI outputs and access to R without requiring the explicit sequence S. The reconstruction is done based on the above equation.

We used simulated data in order to evaluate if the proposed algorithm could correctly perform the inexact-aware relative parsing of a genome. The lambda genome reference genome REFID was used as the reference genome. An additional genome, λ_{mut}, was generated by introducing 10 single nucleotide mutations in the reference genome using simuG tool [36]. Then, we parsed the mutated genome, λ_{mut} with respect to the reference genome λ using the proposed RLZI algorithm with a fixed seed length $k = 10$ and variying the numbers of mismatches allowed in a factor (parameter m), ranging from no mismatches ($m = 0$) up to 5 differences ($m = 5$). After the parsing and deletion of λ_{mut}, we

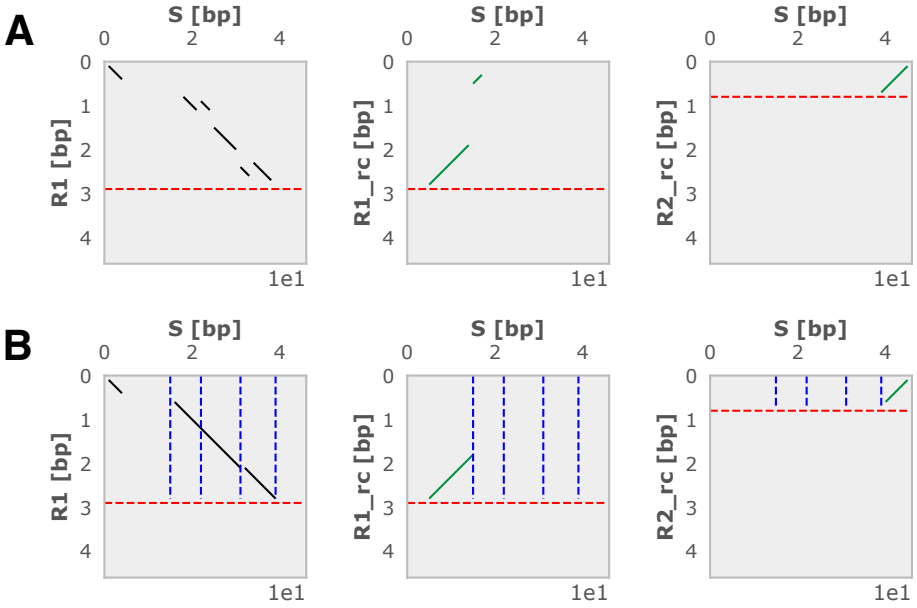

Fig. 1. A dotplot representation of the example given in Section 3. The x-axis denotes the S sequence and in y-axis, a chromosome from reference genome $\mathcal{R}$ or it's reverse complemented sequence flagged by rc. The horizontal red-line marks the length of this chromosome. Every diagonal line indicates a phrase generated through parsing S using $\mathcal{R}$ by RLZI algorithm. In (A), RLZI algorithm was executed with parameter $m = 0$ yielding 9 phrases. All phrases, have been broken in positions of S that differ from R. In (B), we allow up to 2 mismatches in every generated phrase (parameter $m = 2$), highlighting the mismatch positions with a vertical blue-line. This results in five phrases correctly identifying the proposed rearrangements.

could successfully reconstructed the mutated sequence using solely equation 2 in addition to pertinent data structures generated by RLZI concluding that RLZI algorithm correctly parses the query sequence. Furthermore, the RLZI algorithm generates more collinear, i.e. more contiguous, phrases than when the RLZ algorithm is simulated by setting the parameter $m = 0$. As we can see, the presence of a point mutation introduces at most two extra factors in traditional RLZ algorithms but RLZI parses it in a single phrase [20].

4.2 Application on Genome Rearrangement Detection

The proposed RLZI algorithm can be viewed as a glocal alingment tool, i.e. it identifies a substring of the reference that matches the entirety of a segment of the query (i.e. phrase) sequence. The segment is extended, allowing up to m-mismatches if the distance between consecutive mismatches are at least k (parameter seed-length). Therefore, while parsing the query sequence, if RLZI algorithm finds a DNA segment which has a hotspot of mismatches, it will break a phrase and RLZI will try to generate a new phrase more similar to this region. Therefore, the order of individual bases are broken and a new match can occur anywhere in the genome, enabling RLZI to find genome-wide rearrangements.

To test whether RLZI could correctly identify genome rearrangements, we ran RLZI on a simulated and a real dataset. The machine used for the tests had a 8-core 2.6 GHz Intel Core i7 CPU and 16 GB of RAM.

Simulated Dataset We used the simulated dataset generated to evaluate SMASH++ genome rearrangement detection tool [15]. It comprises 4 pairs of simulated sequences, four references and four query sequences, simulated with varying lengths, reverse complemented segments and mutation rates. For example, to build the synthetic reference A (Fig. 2-A), 3 random sequences of size 500 bp were generated and concatenated. Each random sequence is a random block from I to III. To build the corresponding synthetic query sequence, the reverse complement of the third block (III) with a 2% mutation rate was concatenated with second 500 bp block (II) and the first block (I) in reverse complement order. The operations applied to generate the other synthetic sequences are depicted on multi-square diagram and the the results of RLZI parsing of each query using the corresponding reference sequence is shown at the right as a ideogram draw (Fig. 2). The drawing has been done with RIdeogram package [13]. We used the generated phrases to draw the links between the sequences and filtered out phrases with length below 15 to avoid spurious links. Fig.2-A shows that RLZI algorithm applied with parameters $k = 5$ and allowing $m = 10$ mismatches could correctly identify both inversions, e.g. the reverse-complement of the first 500 bp block in reference has synteny with the last 500 bp of the query sequence, and a single phrase with 2% mismatches corresponds to a syntenic region at the middle of both sequences. In the case of Fig. 2-B, RLZI could detect the overall genome architecture differences among the sequences using parameters $k = 10$ and $m = 10$. Note that a segment corresponding to the 90% mutated block was wiped out because it is covered by many small phrases (less than 30 bp).

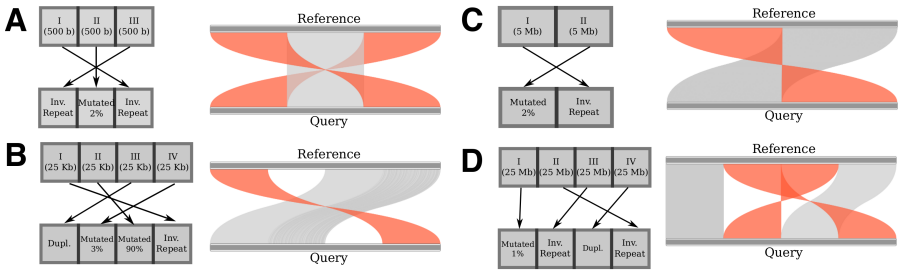

Fig. 2. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

In datasets Large and XLarge (Fig. 2-C and D), the sequences lengths are bigger, containing respectively, 10 Mb and 100 Mb. These lengths correspond to genome sizes found in prokaryotic and some eukaryotic organisms showing the methods robustness in detecting genome rearrangements even in larger sequences. The performance of RLZI algorithm was evaluated for every datasets by checking the RAM peak memory usage and elapsed time during algorithms execution (Table 1). The RAM memory requirements for the small, medium and larger synthetic datasets were under 200 MB while in XLarge dataset, the peak achieved almost 2 GB. An explanation for this behaviour might be associated with the use of a non-optimized priority minimum heap data structure in RLZI implementation. However, it is not a bottleneck given that, nowadays, 2 GB of memory is affordable in every personal computers. In comparison to SMASH++, the amount of RAM memory required to process the synthetic datasets was approximately steady in 1 GB irrespective to dataset size [15]. The small RAM memory footprint of RLZI algorithm for larger datasets ($<$ 30 Mb) makes it suitable to be run in parallel when comparing multiple prokaryotic or simpler eukaryotic genomes such as fungi.

On the other hand, the running time of RLZI increases substantially with sequence length (Tab 1). However, the most part of the time (X%) is spent on BWT construction of the reference sequence by cw-bwt algorithm and the remaining time on RLZI execution. Hence, our algorithm stores the reference BWT in a separate file detaching the requirement to build the BWT every time that the same reference sequence is used such as in comparative genomics. This speeds up the tool when finding genome rearrangements of multiple genomes using the same reference genome like the one performed in the following section.

Real Data We choose a set of *E. coli* genomes composed of 10 assemblies of different strains (NCBI ids XXX) as the real dataset to perform RLZI analysis.

Table 1. Table captions should be placed above the tables.

Dataset (Reference+Query)	Size (bases)	Memory (MB)	Elapsed time (s)
Synthetic			
Small	3,000	1.89	0.03
Medium	200,000	5.61	1.71
Large	10,000,000	116.27	132.27
XLarge	200,000,000	1,927.42	2994.79 (49m 54s)
XXLarge	2,000,000,000	9,939 GB	25h 48m 10s
Real			
Asp. terreus	2		

Among 10 genomes we pick genome NC_017634 as the reference genome for RLZI input and the other 9 genomes were query genomes.

References

1. Bennetzen, J.L.: Transposable elements, gene creation and genome rearrangement in flowering plants. Current Opinion in Genetics & Development **15**(6), 621–627 (Dec 2005). https://doi.org/10.1016/j.gde.2005.09.010
2. Birney, E., Vamathevan, J., Goodhand, P.: Genomics in healthcare: GA4GH looks to 2022. Preprint, Genomics (Oct 2017). https://doi.org/10.1101/203554
3. Brudno, M., Malde, S., Poliakov, A., Do, C.B., Couronne, O., Dubchak, I., Batzoglou, S.: Glocal alignment: Finding rearrangements during alignment. Bioinformatics **19**(Suppl 1), i54–i62 (Jul 2003). https://doi.org/10.1093/bioinformatics/btg1005
4. Covo, S.: Genomic Instability in Fungal Plant Pathogens. Genes **11**(4), 421 (Apr 2020). https://doi.org/10.3390/genes11040421
5. Cox, A.J., Farruggia, A., Gagie, T., Puglisi, S.J., Sirén, J.: RLZAP: Relative Lempel-Ziv with Adaptive Pointers. In: Inenaga, S., Sadakane, K., Sakai, T. (eds.) String Processing and Information Retrieval, vol. 9954, pp. 1–14. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-46049-9$_1$
6. Deorowicz, S., Grabowski, S.: Robust relative compression of genomes with random access. Bioinformatics **27**(21), 2979–2986 (Nov 2011). https://doi.org/10.1093/bioinformatics/btr505
7. Ferrada, H., Gagie, T., Gog, S., Puglisi, S.J.: Relative Lempel-Ziv with Constant-Time Random Access. In: Moura, E., Crochemore, M. (eds.) String Processing and Information Retrieval, vol. 8799, pp. 13–17. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-11918-2$_2$
8. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science. pp. 390–398. IEEE Comput. Soc, Redondo Beach, CA, USA (2000). https://doi.org/10.1109/SFCS.2000.892127
9. Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM **52**(4), 552–581 (Jul 2005). https://doi.org/10.1145/1082036.1082039

10. Gog, S., Beller, T., Moffat, A., Petri, M.: From Theory to Practice: Plug and Play with Succinct Data Structures. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Kobsa, A., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Terzopoulos, D., Tygar, D., Weikum, G., Gudmundsson, J., Katajainen, J. (eds.) Experimental Algorithms, vol. 8504, pp. 326–337. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-07959-$2_2$8
11. Gusev, V., Kulichkov, V., Chupakhina, O.: Global complexity analysis of genomes. Biosystems **30**(1-3), 201–214 (Jan 1993). https://doi.org/10.1016/0303-2647(93)90071-J
12. Gusev, V., Kulichkov, V., Chupakhina, O.: The Lempel-Ziv complexity and local structure analysis of genomes. Biosystems **30**(1-3), 183–200 (Jan 1993). https://doi.org/10.1016/0303-2647(93)90070-S
13. Hao, Z., Lv, D., Ge, Y., Shi, J., Weijers, D., Yu, G., Chen, J.: *RIdeogram* : Drawing SVG graphics to visualize and map genome-wide data on the idiograms. PeerJ Computer Science **6**, e251 (Jan 2020). https://doi.org/10.7717/peerj-cs.251
14. Hartmann, T., Middendorf, M., Bernt, M.: Genome Rearrangement Analysis: Cut and Join Genome Rearrangements and Gene Cluster Preserving Approaches. In: Setubal, J.C., Stoye, J., Stadler, P.F. (eds.) Comparative Genomics, vol. 1704, pp. 261–289. Springer New York, New York, NY (2018). https://doi.org/10.1007/978-1-4939-7463-4_9
15. Hosseini, M., Pratas, D., Morgenstern, B., Pinho, A.J.: Smash++: An alignment-free and memory-efficient tool to find genomic rearrangements. GigaScience **9**(5) (May 2020). https://doi.org/10.1093/gigascience/giaa048
16. Huang, L., Popic, V., Batzoglou, S.: Short read alignment with populations of genomes. Bioinformatics **29**(13), i361–i370 (Jul 2013). https://doi.org/10.1093/bioinformatics/btt215
17. Kjærbølling, I., Vesth, T., Frisvad, J.C., Nybo, J.L., Theobald, S., Kildgaard, S., Petersen, T.I., Kuo, A., Sato, A., Lyhne, E.K., Kogle, M.E., Wiebenga, A., Kun, R.S., Lubbers, R.J.M., Mäkelä, M.R., Barry, K., Chovatia, M., Clum, A., Daum, C., Haridas, S., He, G., LaButti, K., Lipzen, A., Mondo, S., Pangilinan, J., Riley, R., Salamov, A., Simmons, B.A., Magnuson, J.K., Henrissat, B., Mortensen, U.H., Larsen, T.O., de Vries, R.P., Grigoriev, I.V., Machida, M., Baker, S.E., Andersen, M.R.: A comparative genomics study of 23 Aspergillus species from section Flavi. Nature Communications **11**(1) (Dec 2020). https://doi.org/10.1038/s41467-019-14051-y
18. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theoretical Computer Science **483**, 115–133 (Apr 2013). https://doi.org/10.1016/j.tcs.2012.02.006
19. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval. In: Chavez, E., Lonardi, S. (eds.) String Processing and Information Retrieval, vol. 6393, pp. 201–206. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16321-$0_2$0
20. Kuruppu, S., Puglisi, S.J., Zobel, J.: Optimized relative Lempel-Ziv compression of genomes. In: Proceedings of the Thirty-Fourth Australasian Computer Science Conference - Volume 113. pp. 91–98. ACSC '11, Australian Computer Society, Inc., Perth, Australia (Jan 2011)
21. Lempel, A., Ziv, J.: On the Complexity of Finite Sequences. IEEE Transactions on Information Theory **22**(1), 75–81 (Jan 1976). https://doi.org/10.1109/TIT.1976.1055501
22. Li, H., Durbin, R.: Fast and accurate short read alignment with Burrows-Wheeler transform. Bioinformatics (Oxford, England) **25**(14), 1754–1760 (Jul 2009). https://doi.org/10.1093/bioinformatics/btp324

23. Mani, R.S., Chinnaiyan, A.M.: Triggers for genomic rearrangements: Insights into genomic, cellular and environmental influences. Nature Reviews Genetics **11**(12), 819–829 (Dec 2010). https://doi.org/10.1038/nrg2883
24. Maurer-Alcalá, X.X., Nowacki, M.: Evolutionary origins and impacts of genome architecture in ciliates. Annals of the New York Academy of Sciences **1447**(1), 110–118 (Jul 2019). https://doi.org/10.1111/nyas.14108
25. Navarro, G.: Compact Data Structures: A Practical Approach. Cambridge University Press, Cambridge (2016). https://doi.org/10.1017/CBO9781316588284
26. Navarro, G.: Indexing Highly Repetitive String Collections. arXiv:2004.02781 [cs] (Jun 2020)
27. Noureen, M., Tada, I., Kawashima, T., Arita, M.: Rearrangement analysis of multiple bacterial genomes. BMC Bioinformatics **20**(S23) (Dec 2019). https://doi.org/10.1186/s12859-019-3293-4
28. Otu, H.H., Sayood, K.: A new sequence distance measure for phylogenetic tree construction. Bioinformatics **19**(16), 2122–2130 (Nov 2003). https://doi.org/10.1093/bioinformatics/btg295
29. PCAWG Structural Variation Working Group, PCAWG Consortium, Rodriguez-Martin, B., Alvarez, E.G., Baez-Ortega, A., Zamora, J., Supek, F., Demeulemeester, J., Santamarina, M., Ju, Y.S., Temes, J., Garcia-Souto, D., Detering, H., Li, Y., Rodriguez-Castro, J., Dueso-Barroso, A., Bruzos, A.L., Dentro, S.C., Blanco, M.G., Contino, G., Ardeljan, D., Tojo, M., Roberts, N.D., Zumalave, S., Edwards, P.A.W., Weischenfeldt, J., Puiggròs, M., Chong, Z., Chen, K., Lee, E.A., Wala, J.A., Raine, K., Butler, A., Waszak, S.M., Navarro, F.C.P., Schumacher, S.E., Monlong, J., Maura, F., Bolli, N., Bourque, G., Gerstein, M., Park, P.J., Wedge, D.C., Beroukhim, R., Torrents, D., Korbel, J.O., Martincorena, I., Fitzgerald, R.C., Van Loo, P., Kazazian, H.H., Burns, K.H., Campbell, P.J., Tubio, J.M.C.: Pan-cancer analysis of whole genomes identifies driver rearrangements promoted by LINE-1 retrotransposition. Nature Genetics **52**(3), 306–319 (Mar 2020). https://doi.org/10.1038/s41588-019-0562-0
30. Policriti, A., Gigante, N., Prezza, N.: Average Linear Time and Compressed Space Construction of the Burrows-Wheeler Transform. In: Dediu, A.H., Formenti, E., Martín-Vide, C., Truthe, B. (eds.) Language and Automata Theory and Applications, vol. 8977, pp. 587–598. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-15579-1$_4$6
31. Pratas, D., Silva, R.M., Pinho, A.J., Ferreira, P.J.: An alignment-free method to find and visualise rearrangements between pairs of DNA sequences. Scientific Reports **5**(1) (Sep 2015). https://doi.org/10.1038/srep10203
32. Prezza, N.: Compressed Computation for Text Indexing. Ph.D. thesis (Jan 2017)
33. Seibt, K.M., Schmidt, T., Heitkam, T.: FlexiDot: Highly customizable, ambiguity-aware dotplots for visual sequence analyses. Bioinformatics **34**(20), 3575–3577 (Oct 2018). https://doi.org/10.1093/bioinformatics/bty395
34. Strope, P.K., Skelly, D.A., Kozmin, S.G., Mahadevan, G., Stone, E.A., Magwene, P.M., Dietrich, F.S., McCusker, J.H.: The 100-genomes strains, an *S. cerevisiae* resource that illuminates its natural phenotypic and genotypic variation and emergence as an opportunistic pathogen. Genome Research **25**(5), 762–774 (May 2015). https://doi.org/10.1101/gr.185538.114
35. Xia, L.C., Bell, J.M., Wood-Bouwens, C., Chen, J.J., Zhang, N.R., Ji, H.P.: Identification of large rearrangements in cancer genomes with barcode linked reads. Nucleic Acids Research **46**(4), e19–e19 (Feb 2018). https://doi.org/10.1093/nar/gkx1193
36. Yue, J.X., Liti, G.: simuG: A general-purpose genome simulator. Bioinformatics **35**(21), 4442–4444 (Nov 2019). https://doi.org/10.1093/bioinformatics/btz424

37. Zielezinski, A., Vinga, S., Almeida, J., Karlowski, W.M.: Alignment-free sequence comparison: Benefits, applications, and tools. Genome Biology **18**(1) (Dec 2017). https://doi.org/10.1186/s13059-017-1319-7
38. Zimao Li, Lusheng Wang, Kaizhong Zhang: Algorithmic approaches for genome rearrangement: A review. IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews) **36**(5), 636–648 (Sep 2006). https://doi.org/10.1109/TSMCC.2005.855522
39. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory **23**(3), 337–343 (May 1977). https://doi.org/10.1109/TIT.1977.1055714
40. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory **24**(5), 530–536 (Sep 1978). https://doi.org/10.1109/TIT.1978.1055934