Detection of Genomic Inversions Using Sample-Specific Strings

Relatore: *Prof.ssa Paola Bonizzoni*

Correlatore: *Dott. Luca Denti*

Relazione della prova finale di:
Silvia Cambiago
Matricola 879382

Anno Accademico 2023-2024

You gotta dig deep, and bury all the thoughts
The thoughts that tell you, you're not good enough
The critics, the cynics, say you'll never make it
Prove 'em all wrong, they are mistaken

Hold on, 'cause you don't know what is going to happen
Stay strong, your life's worth more than you know

I Prevail - Crossroads

Abstract

This thesis presents the development of an efficient algorithm that detecting genomic inversions using sequencing data, specifically focusing on long reads and using sample-specific strings to identify the inversions.

It begins by introducing the concept of inversions, explaining their biological meaning, and providing a detailed overview of the data structures and definitions that form the basis of the algorithm.

The algorithm itself is then thoroughly described, with a complete theoretical analysis of its design, its computational complexity and the logic that lies behind it.

Later on, the thesis covers the practical implementation and experimentation, that was performed using the SVDSS tool to validate the algorithm. The results are finally summarized and discussed, demonstrating the algorithm's effectiveness.

Contents

1. Introduction

1.1 Context

Inversions, as defined by HGVS[1] (Human Genome Variation Society), are sequence changes where, compared to a reference sequence, more than one nucleotide replacing the original sequence is the reverse complement of the original sequence. They are a category of genomic structural variations (SVs), defined as alterations in the DNA that affect more than 50 bp that may delete, insert, duplicate, invert, or move genomic sequences[2] (see Figure 1.1).
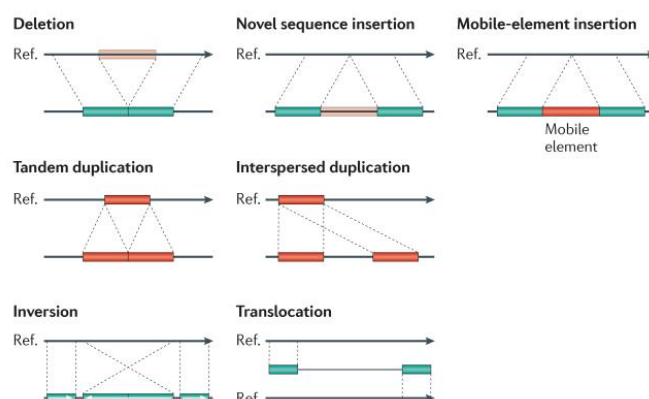
Figure 1.1: Outline of different SV-archetypes.

They are often generated by non-allelic homologous recombination (NAHR) between inverted repeats, but they can also be originated by double-strand break repair mechanisms, like non-homologous end joining (NHEJ), or replication-based mechanisms mediated by microhomology, like fork stalling and template switching[3]. Although inversions constitute only a small fraction of SVs across various organisms, ranging from 0.5% to 7%[4], they can span over 100 Mb and collectively account for up to 10% of the genome, holding considerable functional and evolutionary significance.

Inversions were first identified in 1917 by Alfred Sturtevant[5], and he found out they act as suppressors of recombination between chromosomes. Since then, inversions have been drawing increasing attention, due to their crucial role in driving genome evolution and their link to the evolutionary processes of many species. In fact, it was later found that suppressed recombination among genes within an inversion can lead to largely independent genome evolution between derived and ancestral arrangements, providing opportunities for divergence and speciation[6]. This genomic isolation, facilitated by inversions, can lead to the formation of novel genotypes and phenotypes, contributing to the genetic diversity we can observe today.

But inversions are not only related to evolution. In fact, they are involved in the formation of several diseases, as will be shown in Chapter 2. A recent study even shows that the contribution of chromosomal inversions to phenotypic diversity can even affect brain development and lead to disorders[7].

1.2 Inversions detection

Due to their significance, the challenge of detecting inversions has been widely addressed. The first methods developed for this scope, such as cytogenetics or PCR-based approaches, were labor-intensive and lacked resolution, limiting their effectiveness[4]. With the advent of high-throughput next-generation sequencing (NGS), large-scale population studies of inversions became finally possible, enabling the discovery of polymorphic inversions and their association with phenotypic traits. In addition to this, long-read DNA sequencing technologies such as PacBio HiFi sequencing or Oxford Nanopore duplex sequencing have enabled the generation of sequencing reads between 10 kb and 2 Mb, making them highly useful for detecting inversions. Therefore, identifying inversions has become more straightforward with the use of long reads that can span repetitive and complex genomic regions. However, despite all this technological progress, inversions remain one of the most underascertained forms of structural variation in human and non-human primate genomes, limiting our understanding of their evolution significance[8]. The main challenge continues to be the precise determination of inversion breakpoints.

1.3 Project

In this thesis will be presented an algorithm designed to detect the presence of genomic inversions from long-read sequencing data using sample-specific strings (which will be defined in detail in Chapter 2) to determine the breakpoints. A

`Python` implementation of the algorithm will be used for the practical experimentation, in combination with the SVDSS[9] tool, that will also be implemented in the code and will help in detecting the number, position, and length of the sample-specific strings.

In Chapter 2, the necessary preliminaries will be presented to understand how the algorithm works, including the definitions used throughout the thesis and the biological background that lies behind inversions, other than their pratical biological consequences.

In Chapter 3, the algorithm will be presented in detail, including the pseudocode and a complete theoretical analysis. This chapter will treat, among other things, time and space complexity and proof of correctness.

In Chapter 4, the practical experimentation will be discussed, explaining the `Python` implementation of the code and the use of the SVDSS tool. The results will be presented, summarized, and commented upon.

2. Preliminaries

2.1 Definitions

The algorithm's primary objective is to detect reverted sequences within a target DNA sequence, which are characterized by segments that have been inverted and complemented relative to a reference DNA sequence.

It uses sample-specific strings as possible anchors of an inverted portion occurring in the target. The definition of sample-specific string (cite khorsand) is as follows:

Definition 1 (Sample-specific string). *A sample-specific string S is a string that occurs in the target DNA string T, does not occur in the reference DNA string R, and for every string S' which is a substring of S, S' occurs in the reference string R.*

Since later in this chapter the Knuth-Morris-Pratt algorithm will be introduced as the method used to check the presence of the reverted string within the reference, a few definitions will be given[10] in order to better understand it:

Definition 2 (Prefix). *Given a string S, a string T is a prefix of S if and only if S can be formed by concatenating T with another string T'.*

Definition 3 (Proper prefix). *Given a string S, a proper prefix of S is a prefix of S which does not equal the whole of S.*

Definition 4 (Suffix). *Given a string S, a string T is a suffix of S if and only if S can be formed by concatenating another string T' with T.*

Definition 5 (Proper suffix). *Given a string S, a proper suffix of S is a suffix of S which does not equal the whole of S.*

2.2 Biological Background

2.2.1 DNA

Before treating the computational part of the project, an overview of the biological background will be provided.

The DNA, abbreviation for deoxyribonucleic acid, is the molecule that carries genetic information for the development and functioning of an organism[11]. Each DNA molecule consists of a long polymer made up of repeating units, the nucleotides, which are composed of a phosphate group, a sugar molecule (2-deoxyribose), and a nitrogenous base. There are four types of nitrogenous bases found in DNA that define the properties of the nucleotide: adenine (A), thymine (T), guanine (G), and cytosine (C), as can be seen in Figure 2.1.
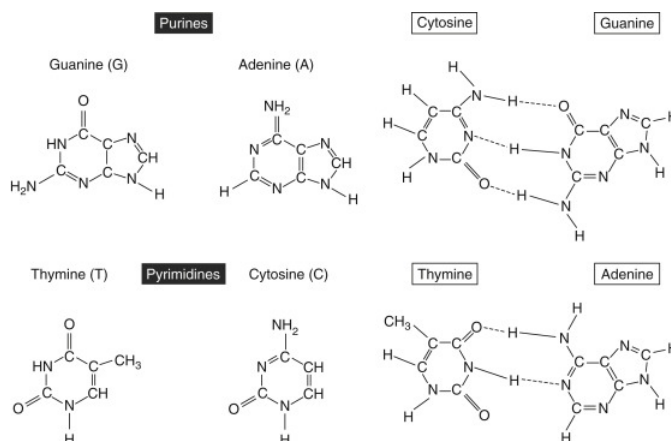

Figure 2.1: DNA nucleotide bases and base pairing.

In all eukaryotic organisms, DNA exists as a tightly associated pair of two long strands that intertwine, forming the shape of a double helix. The two strands of DNA are stabilized by hydrogen bonds between the nitrogenous bases attached to the two strands[12]. Watson-Crick base pairing involves adenines pairing with thymines and guanines pairing with cytosines. Strands of DNA that form matches among base pairs are called complementary strands.

2.2.2   Double strand breaks

Double-strand break (DSB) is the primary cytotoxic lesion generated from ionizing radiation, radio-mimetic chemicals such as camptothecin (CPT), mechanical stress on chromosomes or when the replication machinery encounters a single-strand DNA break or other type of DNA lesions. In addition to this, DSBs can also be produced during physiological processes, such as recombination or meiosis[13]. Double-stranded DNA breaks (DSBs) are a particularly dangerous form of DNA damage because they can lead to chromosome loss, translocations or truncations. Repair occurs via one of two pathways: non-homologous end-joining (NHEJ), in

which broken DNA ends are directly ligated; or homologous recombination (HR), in which a homologous chromosome is used as a template in a replicative repair process.

2.2.3 DNA recombination and formation of inversions

Genetic recombination is a fundamental cellular process that has been implicated in the generation of genetic diversity, the repair of damaged DNA, the homologous alignment of chromosomes required for successful completion of meiotic cell division, and the generation of genomic alterations that lead to changes in gene expression. Classical homologous recombination (HR) is a type of genetic recombination in which nucleotide sequences are exchanged between two similar or identical molecules of DNA, as can be seen in Figure 2.2. During the formation of egg and sperm cells (meiosis), paired chromosomes from the male and female parents align so that similar DNA sequences can cross over, or be exchanged, from one chromosome to the other. This exchanging of DNA is an important source of the genomic variation seen among offspring[14]. This allelic homologous recombination repairs double-stranded breaks (DSBs) in chromosomes by using the allele on the sister chromatid as a template. This mechanism is highly faithful because the allelic region of the sister chromatid is a nearly exact copy of the DNA lost in the DSB[15].
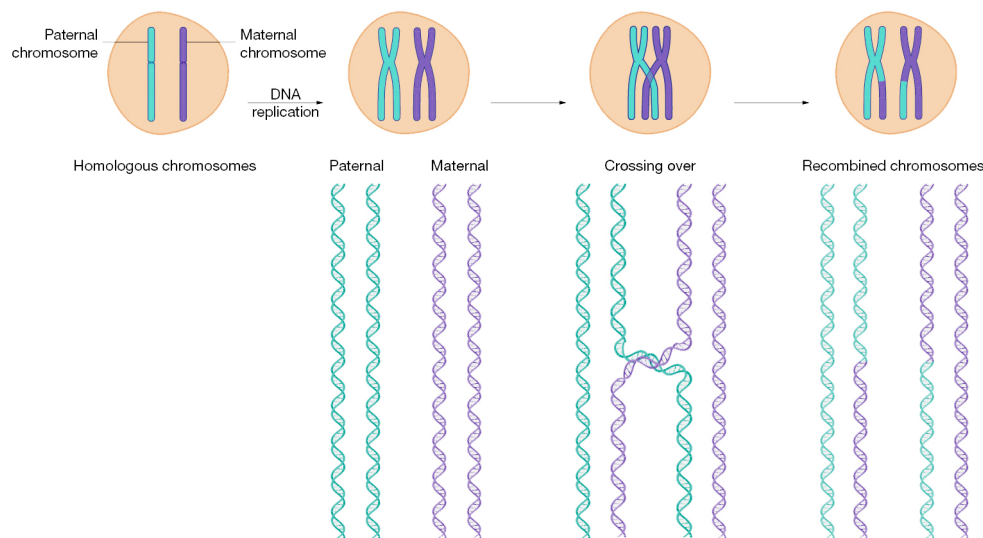

Figure 2.2: Homologous recombination between maternal and paternal chromosomes.

However, a similar mechanism can also act between repetitive DNA sequences that are similar but located in different (non-allelic) regions of the genome. This

process is called non-allelic homologous recombination (NAHR).

This kind of recombination occurs between Low Copy Repeats (LCRs), also known as Segmental Duplications, that are DNA blocks of 10 to 400 kb in size with over 97% identity between sequences[16]. NAHR can occur after a DSB during meiosis or mitosis when non-allelic copies of LCRs erroneously align due to their high level of sequence identity. This misalignment causes an unequal crossing over event generating genomic rearrangement in the daughter cells.
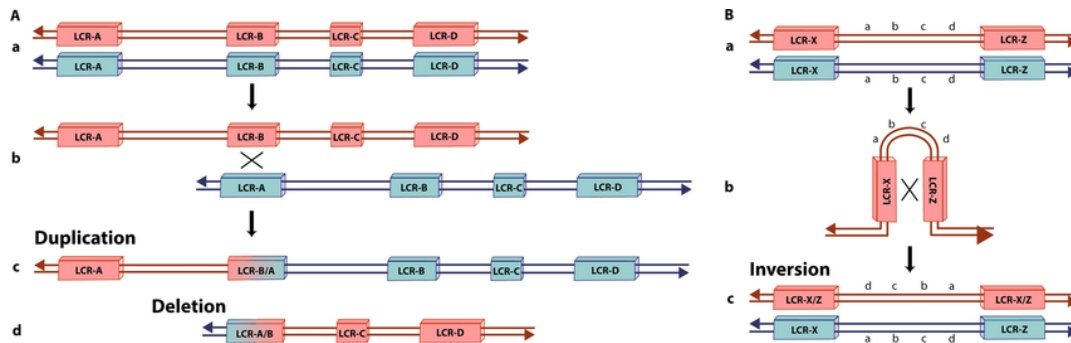

Figure 2.3: Non-Allelic Homologous Recombination (NAHR) mechanism. A. NAHR leading to the formation of duplication and deletion: (a) Normal chromosome pairing and alignment of Low Copy Repeats (LCRs) in the same orientation. (b) A misalignment between LCRs due to their high level of sequence identity leads to an unequal crossing over event that can generate (c) a duplication and (d) a deletion. B NAHR leading to the formation of inversions: (a) Normal chromosome pairing and alignment of Low Copy Repeats (LCRs). LCR-X and LCR-Z present similar DNA sequences but in opposite orientations. (b) A misalignment between LCRs due to their high level of sequence identity leads to an unequal crossing over event that can generate (c) an inversion

Through non-allelic homologous recombination (NAHR) - recombination events taking place between highly similar sequences - regions located between segmental duplications or highly identical repeat sequences may be deleted, duplicated or inverted. Inversions can be formed by this process if the duplicated sequences are in inverted orientation with respect to each other, as Figure 2.3B shows. Therefore, NAHR is considered the primary mechanism by which large (tens of kilobases) inversions are formed[17].

2.3   Inversion in human disorders

Many inversions traditionally detected in human karyotypes do not appear to have any phenotypic effects of clinical significance. This is the case of pericentric inversions (the inverted sequence includes the centromere) in chromosomes 1, 2, 3, 5, 9, 10 and 16, which mainly invert heterochromatic sequences and are frequently

observed in cytogenetic analysis[3]. However, not all inversions are harmless, and several diseases have been found to be occasionally caused by inversions, mostly by direct disruption of one gene or by altering its gene expression. These inversions appear de novo in patients or are inherited mutations restricted to a given family. Since inversions are relatively rare events, and it is unlikely that multiple patients with the same inversion are found, it is often problematic to assess whether the inversion present in the patient is actually associated with the phenotype. The exception is if the inversion breakpoint falls within or near a gene that has previously been associated with the disorder through other types of mutations. For recurrent inversions, the association between phenotype and genotype is more obvious, and a number of such loci have been described. One of the best-characterized recurrent inversions giving rise to disease causes hemophilia A, an X-linked disorder caused by mutations in the factor VIII gene. A recurrent inversion has been found in approximately 43% of patients. Molecular characterization of the breakpoints indicates that the inversion is a result of intra-chromosomal homologous recombination, originating almost exclusively in male germ cells. This recurrent inversion spans approximately 400 kb and is mediated by two inverted segmental duplications, one of which is located in intron 22 of the factor VIII gene, with two other copies being located approximately 400 kb telomeric to the gene. Other examples where recurrent inversions have been shown to lead to a disease phenotype are the disruption of the idunorate 2-sulphatase gene in mucopolysaccharidosis type II (Hunter syndrome), and disruption of the emerin gene in Emery-Dreifuss muscular dystrophy[17].

In addition to this, it was found that the presence of micro-inversions, defined as inversion in DNA shorter than 100 bp, can be linked to cancer, as a study conducted in 2018[18] demonstrates. In said study, the distribution of microinversions was analyzed among 24 chromosomes in four types of cancer (hepatocellular, lung, pancreatic and bladder), and it was found that the average count of micro inversions per individual in the normal samples is lower than that of any type of cancer, showing that there is a high chance that micro inversions may be associated with cancer development (see Figure 2.4).
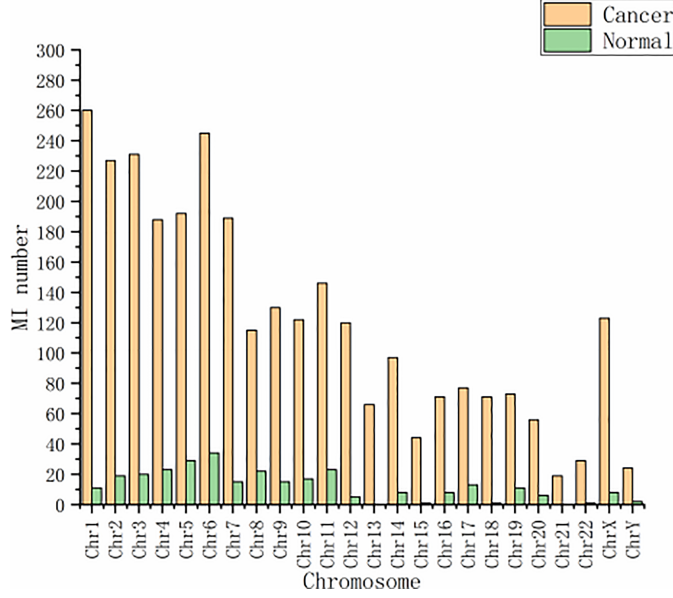
Figure 2.4: MI (micro-inversion) distribution among 24 chromosomes. Average count of MIs per individual among the four types of cancer.

2.4 Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt[19] algorithm is the one that will be used in this project to detect the position of the inverted segments within the reference. It can be seen as an evolution of the naïve string matching algorithm. In fact, given a text T having length n and a pattern P, the naive algorithm, for each possible starting position i, where $0 \leq i \leq n - m$, compares the substring starting at position i in the text with the pattern and, if a mismatch is found, it moves to the next position. If all the characters match, it returns the occurrence of the pattern at index i.

The idea behind the Knuth-Morris-Pratt algorithm is that, in case of mismatch, it is possible to perform a shift greater than 1, in order to avoid unnecessary comparisons.

2.4.1 LPS array

The Knuth-Morris-Pratt algorithm preprocesses the pattern by building an auxiliary array, the Longest Prefix Suffix (LPS) array. For each index i of the pattern P, $LPS[i]$ contains the length of the longest proper prefix of the substring $P[:i]$ which is also a suffix of said substring. This array is the key to the algorithm's efficiency. It helps in skipping characters that will match for sure, thus reducing the number of comparisons needed and ultimately speeding up the search process.

11

The process of building the LPS array can be described as follows:

- Initialization: an array $LPS[]$, of size equal to the length of the pattern is initialized with zeros, and $LPS[0]$ is set to 0 because there is no proper prefix for a single character. Two pointers are also initialized: i, that iterates through the pattern and is set to 1, and j, that will track the length of the previous longest proper prefix suffix and is set to 0.

- Iteration over the pattern: each character at index i in the pattern is compared with the character at index j in the text. If $P[i] == P[j]$, the current character in the pattern extends the longest proper prefix that is also a suffix, so both indexes i and j are incremented and $LPS[i] = j$. Otherwise, if $P[i]! = P[j]$, the previously computed $LPS[j-1]$ is checked to see the next smaller prefix that could still match. If $j == 0$, there is no valid prefix to fallback to, so $LPS[i]$ is set to 0 and i is incremented in order to move to the next character.

The process continues until i reaches the length of the pattern.

Figure 2.5: For the pattern P displayed in the figure, for the first three positions, $LPS[i]$ equals to 0 because there is no proper prefix for "A", "AB" or "ABC" which is also a suffix. Moving on, $LPS[3]$ is 1 because the proper prefix "A" of "ABCA" is also a suffix, $LPS[4]$ is 2 because the proper prefix "AB" of "ABCAB" is also a suffix. For the last position, $LPS[5]$ is, again, 0, since no proper prefix of "ABCABD" is also a suffix.

2.4.2 Pattern matching

The Knuth-Morris-Pratt algorithm iterates through the text T and through the pattern P, using two separate pointers. When a character in the reference matches a character in the pattern, both pointers are incremented to continue the comparison. When a mismatch is found at position j in the reference and i characters of the pattern have already been matched, instead of restarting from the beginning of the pattern as happens in the naïve algorithm, the KMP consults $LPS[i-1]$ in order to determine how many characters of the pattern can be skipped. Thus,

```
v=[-1  0 -1  1  0]
    0  1  2  3  4
```

S → | a | b | r | a | c | a | d | a | b | r | a |

P → | a | c | a | d |

↓ match S[0] = P[0]

S → | a | b | r | a | c | a | d | a | b | r | a |

P → | a | c | a | d |

↓ mismatch S[1] ≠ P[1], jump=1-v[1]

S → | a | b | r | a | c | a | d | a | b | r | a |

P → | a | c | a | d |

↓ mismatch S[1] ≠ P[1], jump=0-v[0]

S → | a | b | r | a | c | a | d | a | b | r | a |

P → | a | c | a | d |

↓ mismatch S[2] ≠ P[2], jump=0-v[0]

S → | a | b | r | a | c | a | d | a | b | r | a |

P → | a | c | a | d |

✓ match found at position 3 in S
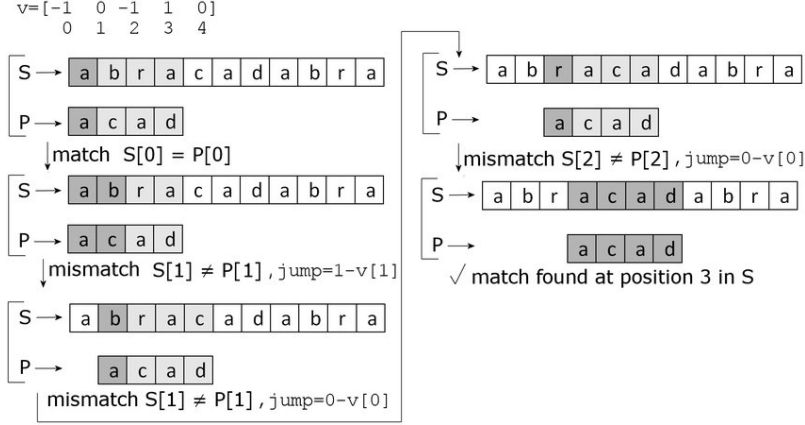
the KMP algorithm performs pattern matching in $\mathcal{O}(n+m)$ time, where n is the length of the reference and m is the length of the pattern.

Among other solutions, like the suffix array or the Boyer-Moore algorithm, the KMP algorithm was chosen in this project because it is more suitable for contexts where the alphabet is small, such as DNA (A, C, T, G). Additionally, the KMP algorithm performs well even when the length of the reference is large, unlike, for example, the suffix array, which which can become less efficient in such cases due to its preprocessing requirements. This makes KMP the optimal choice for detecting of inversions from long reads in the reference.

2.5   DNA sequencing

In bioinformatics, DNA sequencing is the process of determining the sequence of nucleotides (As, Ts, Cs, and Gs) in a piece of DNA. A read is defined as a raw sequence that comes off a sequencing machine[20]. A read may consist of multiple segments. For sequencing data, reads are indexed by the order in which they are sequenced. Most next-generation sequencing technologies fragment the genome prior to sequencing, and each sequenced fragment produces a read. The length of the read and how many are produced will depend on fragment size and the type of technology being used. As the fragments of DNA usually overlap, the reads can be pieced back together to reconstruct the genome. The length of the read is the number of bases that are read at one time, hence the number of letters that will appear in each read. In general, they can be divided into:

- **Short reads:**   read lengths usually range from 50 to 300 base pairs. They are effective for applications aimed at counting the abundance of specific sequences, identifying variants within otherwise well-conserved sequences, or for profiling the expression of particular transcripts. Short-read sequencing

is the best way to obtain high depth and high-quality data for the lowest cost per base. On the other hand, they fail to generate a sufficient overlap between the DNA fragments. Overall, this means that sequencing a highly complex and repetitive genome, like that of a human, can be challenging using these technologies. Illumina's platform dominates this field along with Thermo Fisher Scientific's Ion Proton technology. However, in 2022, multiple new sequencing technologies have surfaced and been introduced to the market, helping to drive short-read sequencing costs even lower. These include instruments from Element Biosciences, Ultima Genomics, MGI, Singular Genomics and the PacBio acquired technology, Omniome[21].

- **Long reads:** read lengths usually range from 5000 to 50000 base pairs[22]. They allow to identify complex SVs such as large insertions/deletions, inversions, repeats, duplications, and translocations. Long read NGS instruments have been on the market for the past decade but the lower yield, higher error rate, and higher costs of the instruments, have kept them from being more widely adopted. An additional downside is that the accuracy per read can be much lower than that of short-read sequencing. In the last decade, PacBio and Oxford Nanopore Technologies (ONT) have both been working to make long-read sequencing more accessible. Specifically, PacBio has improved the chemistry on their Sequel II instruments, enabling "HiFi sequencing" via circular consensus, which allows for sequencing of up to 15-20 kb pieces of DNA with error rate that are closer to short read sequencing.

In this project, long reads will be used, since the target is a long read extracted from the reference and the aim is to detect inversions.

2.6   FASTA format

In bioinformatics, FASTA format is a text-based format for representing DNA sequences, in which base pairs are represented using a single-letter code from the alphabet $\Sigma = \{A, C, G, T\}$, where each letter corresponds to the initial of one of the four nitrous bases that make up the DNA. The format also allows for sequence names and comments to precede the sequences. In this project, as will be shown in Chapter 4, this file format will be used to represent the reference sequence and the target, which is a long read of the reference. A sequence in FASTA format begins with a single-line identifier description, followed by lines of DNA sequence data. The identifier description line is distinguished from the sequence data by a greater-than ('>') symbol in the first column. The word following the ">" symbol is the identifier of the sequence, and the rest of the line is a description (optional) separated form the identifier by a white space or tab. The sequence data starts on

the next line following the text line and ends if another line starting with a ">"
appears, indicating the start of another sequence. The extension of this format is
usually `.fa`.

Example 1. *This is an example of a portion of a FASTA file that contains two
nucleotide sequences:*

```
>NC_045512.2
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCT
GTTCTCTAAACGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACT
>OL700521.1
GTTCTCTAAACGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACT
CACGCAGTATAATTAATAACTAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATC
```

3. Algorithm

3.1 Algorithm Overview

The algorithm's primary objective is to detect reverted sequences within a target DNA sequence, which are characterized by segments that have been inverted and complemented relative to a reference DNA sequence.
It uses sample-specific strings as possible anchors of an inverted portion occurring in the target. The definition of sample-specific string is as follows:

Definition 6 (Sample-specific string). *A sample-specific string S is a string that occurs in the target DNA string T, does not occur in the reference DNA string R, and for every string S' which is a substring of S, S' occurs in the reference string R.*

The key components of the algorithm are:

- Target Sequence: the DNA sequence in which the algorithm looks for reverted sequences. It is a long read extracted from the reference sequence;

- Reference Sequence: the DNA sequence used as a baseline to identify reverted sequences in the target;

- Sample-specific strings: see Definition 6. They expected to appear in order within the target sequence;

3.2 High-Level Explanation

This algorithm is structured in a way that involves different phases, detailed as follows:

- **Setup:** in this phase the working environment is set up, and relevant data structures are prepared in order to store relevant results. Given a

Algorithm 1 Check For Inversions Using Sample-Specific Strings

    Input: indexes: list of integers, lengths: list of integers, sample_specific_strings: list of strings, target: string, reference: string

  Output: list of reverted sequences found in target

1: reverted ← [ ]
2: **for** $i \leftarrow 1$ **to** sample_specific_strings.length $- 1$ **do**
3:    $j \leftarrow$ indexes$[i]$
4:    $k \leftarrow$ indexes$[i + 1]$
5:    **if** $j == -1$ **or** $k == -1$ **then**
6:      **CONTINUE**
7:    **end if**
8:    middle ← target$[j + $lengths$[i] : k]$
9:    **if** middle $== \emptyset$ **then**
10:     **CONTINUE**
11:    **end if**
12:    middle ← **REVERT_AND_COMPLEMENT**(middle)
13:    (found, start) ← **CHECK_SUBSTRING**(reference, middle)
14:    **if** found **then**
15:     left_increment ← 1
16:     **while** left_increment $\leq$ lengths$[i]$ **and** $(j + $lengths$[i] - $left_increment$) \geq 0$ **and** target$[j + $lengths$[i] - $left_increment$] == $**REVERT_AND_COMPLEMENT**(reference$[$start $+$ middle.length $+$ left_increment $- 1]$) **do**
17:       left_increment ← left_increment $+ 1$
18:     **end while**
19:     left_breakpoint ← target$[j + $lengths$[i] - $left_increment $: j + $lengths$[i] - $left_increment $+ 2]$
20:     right_increment ← 0
21:     **while** right_increment $<$ (reference.length $- $start $- $middle.length **and** $(k + $right_increment$) < $target.length **and** target$[k + $right_increment$] == $**REVERT_AND_COMPLEMENT**(reference$[$start $- 1 - $right_increment$]$) **do**
22:       right_increment ← right_increment $+ 1$
23:     **end while**
24:     right_breakpoint ← target$[k + $right_increment $- 1 : k + $right_increment $+ 1]$
25:     **PRINT** "*left_breakpoint* and *right_breakpoint* are breakpoints of an inversion"
26:     inversion ← target$[j + $lengths$[i] - $left_increment $+ 1 : k + $right_increment$]$

27:     **APPEND** reverted ← inversion
28:    **end if**
29: **end for**
30: **return** reverted

Algorithm 2 Revert and Complement DNA Sequence
___
   Input: DNA sequence: string

       Output: reverted   and   complemented   sequence: string

1: complements $\leftarrow \{'A' :' T','C' :' G','G' :' C','T' :' A'\}$
2: reverted_complement $\leftarrow \varepsilon$
3: **for** $i \leftarrow$ sequence.length $- 1$ **to** 0 **do**
4:    $base \leftarrow$ sequence$[i]$
5:    $complement \leftarrow$ complements$[base]$
6:    **APPEND** reverted_complement $\leftarrow complement$
7: **end for**
8: **return** reverted_complement
___

Algorithm 3 Knuth-Morris-Pratt Prefix Function
___
   Input: pattern: string

   Output: LPS array: list of integers

1: $m \leftarrow$ length of pattern
2: lps $\leftarrow [0] * m$
3: $j \leftarrow 0$
4: $i \leftarrow 1$
5: **while** $i < m$ **do**
6:    **if** pattern$[i] ==$ pattern$[j]$ **then**
7:      $j \leftarrow j + 1$
8:      lps$[i] \leftarrow j$
9:      $i \leftarrow i + 1$
10:   **else**
11:     **if** $j \neq 0$ **then**
12:       $j \leftarrow lps[j - 1]$
13:     **else**
14:       $lps[i] \leftarrow 0$
15:       $i \leftarrow i + 1$
16:     **end if**
17:   **end if**
18: **end while**
19: **return** lps
___

Algorithm 4 Check Substring Using Knuth-Morris-Pratt

    Input: reference: string, string: string

    Output: Tuple of boolean (True if string is a substring of reference) and integer (starting index if found, -1 otherwise)

1: $n \leftarrow$ length of reference
2: $m \leftarrow$ length of string
3: lps $\leftarrow$ KMP-Prefix/Function(string)
4: $i \leftarrow 0$
5: $j \leftarrow 0$
6: **while** $i < n$ **do**
7:    **if** string$[j]$ == reference$[i]$ **then**
8:      $i \leftarrow i + 1$
9:      $j \leftarrow j + 1$
10:   **end if**
11:   **if** $j == m$ **then**
12:     **return** $(\text{True}, i - j)$
13:   **else if** $i < n$ **and** string$[j] \neq$ reference$[i]$ **then**
14:     **if** $j \neq 0$ **then**
15:       $j \leftarrow \text{lps}[j - 1]$
16:     **else**
17:       $i \leftarrow i + 1$
18:     **end if**
19:   **end if**
20: **end while**
21: **return** $(\text{False}, -1)$

target sequence S and a reference sequence R, let $I = \{i_1, i_2, \ldots, i_n\}$ denote the list of indexes where sample-specific strings occur in S, and let $L = \{l_1, l_2, \ldots, l_n\}$ be the list of their corresponding lengths. Additionally, an empty list, *reverted*, is initialized to store any detected inverted sequences.

- **Iteration through sample-specific strings:** the algorithm focuses on the prime candidate for detecting inversions, which is the segment that lies between a couple of sample-specific strings. Therefore, the core of the algorithm involves iterating through each pair of adjacent sample-specific strings. It confronts every pair (x, y) of sample-specific strings within the range, where $x = S[i_j : i_j + l_j]$ and $y = S[i_{j+1} : i_{j+1} + l_{j+1}]$, and extracts the segment m that lies exactly in the middle of the two sample-specific strings x and y. More precisely, $m = S[i_j + l_j : i_{j+1}]$. The extracted segment m represents the region that could potentially be the inversion.

- **Reversing and Complementing:** when handling genomic data, an inversion is not just a reversal of the nucleotide sequence, but it also involves the complementing of bases, as seen on Chapter 2. After extracting the segment in the middle m, its reverse complement m_r can be easily obtained: first, the bases in m are reversed, then each base is replaced with its complement. The segment m_r is the transformed version of m that allows the algorithm to search for inverted regions in R.

- **Search for the Reverse Complement in the Reference String:** the logic behind this step is that if m is actually an inverted segment, this means that its reverse and complement m_r can be found within the reference R, since R represents the original, non-inverted version of the sequence. To efficiently locate m_r in R, the algorithm uses the Knuth-Morris-Pratt (KMP) algorithm, detailed in Chapter 2. This method is used in combination with a prefix function, the longest prefix suffix array (LPS) that avoids unnecessary character comparisons. It preprocesses the pattern m_r, creating an array that stores the lengths of the longest prefixes of the pattern that are also suffixes. When a mismatch occurs, the algorithm uses this information to skip ahead in the pattern, making the whole process more efficient. If m_r is found in R, the starting index s of the occurrence of m_r in R is stored for further analysis.

- **Identification of Breakpoints:** the identifications of breakpoints is essential to precisely locate the boundaries of the inversion within the target sequence S. So, if the reverse complement m_r is found in R, the algorithm proceeds to identify the inversion breakpoints, that define the boundaries of

the inverted segment in S. The breakpoints are determined by comparing characters from S and R in two stages:

- **Left Breakpoint Identification:** starting from the last character of the first sample-specific string x in S, that corresponds to $S[i_j + l_j]$ and the first character immediately following m_r in R, which is $R[s + m_r.length]$, the algorithm compares corresponding characters from S and the reverse complement of R. This comparison continues backward in S and forward in R until a mismatch occurs. At this point, the final position in S is marked as left breakpoint.

- **Right Breakpoint Identification:** the right breakpoint is determined symmetrically. The comparison starts from the first character of the second sample-specific string y in S, $S[i_{j+1}]$, and the character immediately preceding m_r in R, $R[s-1]$, and the search proceeds forward in S and backward in R until a mismatch is found, marking the right breakpoint in S.

3.3 Conditions for Proper Functioning

The correctness of this algorithm is ensured if the middle segment extracted from the target is sufficiently long to ensure its uniqueness within the reference sequence. This is crucial because if the middle segment is too short, it might match multiple locations within the reference sequence, leading to potential misalignment. This can result in the algorithm identifying incorrect breakpoints or even falsely identifying an inversion that does not actually exist, hence a false positive.

In addition to this, for the algorithm to function properly, the target sequence, and particularly the middle segment, must be free of errors or mutations. In fact, even a single error within the middle segment can disrupt the transformation process, where the segment is reversed and complemented. As a result, the transformed segment may not match the expected part of the reference sequence, leading to false negatives where actual inversions are missed.

3.4 Edge Cases

An important edge case to consider occurs when two sample-specific strings appear consecutively without any intervening characters in the target string. In this scenario, the middle segment will be an empty string. This happens because there are no characters between the end of the first sample-specific string and the beginning of the second one. For example:

- **Sample-specific strings:** 'S1', 'S2'

- **Target string:** '...S1S2...'

In this case the starting index of 'S1' is j and the starting index of 'S2' is k, making the middle segment target[j + S1.length : k], which evaluates to an empty string.

3.5 Proof of Correctness

Now that the algorithm has been explained in detail, the proof of its correctness will be provided.
Let $S = xyz$ be a string, where x, y, and z are substrings of S: Let also y^r be the reversal of y, the aim is to demonstrate that it is possible to reconstruct the sequence $S' = z^r y^r x^r$, where the superscript 'r' indicates the reversal of a string. For the sake of simplicity, the complement will be ignored. The proof will be made using mathematical induction.
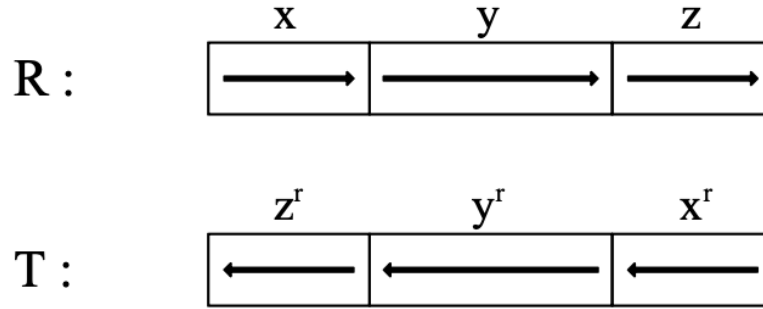

Figure 3.1: R represents the reference string, that in the following demonstration will be referred as S, and T is the target where the inversion is found, that will be later referred as S'

Theorem 1. *For any string $S = xyz$, where x, y, z are substrings of S, given y^r, reversal of y, it is possible to reconstruct the sequence $S' = z^r y^r x^r$.*

Proof. **Base case:** $\|y\| = 1$

$Let\|y\| = 1$, meaning y only consists of a single character 'a'. In this trivial case, $S = xaz$, and $y^r = a$. Hence, the string $S' = z^r y^r x^r$ can be reconstructed as follows:

1. Reverse z to obtain z^r;

2. Append y^r, which is simply 'a';

3. Append the reversal of x, which is x^r.

The resulting sequence S' is $z^r a x^r$, equivalent to $S' = z^r y^r x^r$ when $\|y\| = 1$.

Inductive step: $\|y\| > 1$

Assume that for some $k > 1$, it is possible to reconstruct $S' = z^r y^r x^r$ when $\|y\| = k$ (inductive hypothesis). It will be proven that this reconstruction is also possible when $\|y\| = k + 1$.

Let $y = a_1 a_2 \ldots a_k a_{k+1}$, where each a_i is a single character. Thus, $y^r = a_{k+1} a_k \ldots a_2 a_1$. According to the inductive hypothesis, it is possible to reconstruct $S' = z^r (a_1 a_2 \ldots a_k)^r x^r$. In order to obtain $S' = z^r y^r x^r$ for $\|y\| = k + 1$, the following steps are necessary:

1. Start with $z^r (a_k \ldots a_2 a_1) x^r$;

2. Insert a_{k+1} before $a_k \ldots a_2 a_1$;

The result is $z^r (a_1 a_2 \ldots a_k a_{k+1})^r x^r$, which corresponds to $S' = z^r y^r x^r$ for $\|y\| = k + 1$. $\qquad\square$

3.6   Time and Space Complexity

Now, the time and space complexity of the algorithm will be analyzed.
Starting with the first one, during the initialization phase, the necessary parameters are set up, and the list to store the reverted sequences is initialized. These steps are performed in constant time, hence $\mathcal{O}(1)$.

In the iteration phase, each pair of sample-specific strings is processed. Let n denote the total number of sample-specific strings, each interaction involves the extraction of the segment m between two adjacent sample-specifics strings. This step is repeated for each pair, and extracting the segment takes time proportional to its average length $\bar{m}$, so the overall time complexity of this phase is $\mathcal{O}(n \cdot \bar{m})$.

In the following phase, the segment in the middle is transformed by reverting its characters and complementing each base. This operation takes $\mathcal{O}(\bar{m})$ time per iteration, so the overall time complexity of this phase is, just like the previous one, $\mathcal{O}(n \cdot \bar{m})$.

For the search in the reference string, the Knuth-Morris-Pratt algorithm is used to locate the transformed segment m_r within the reference R. First, the LPS

array is constructed. The array is initialized of size m, where m is the length of the pattern. Keeping in mind that this data can vary when multiple inversions having different lengths are found during the whole computation, the actual value to analyze is the average length $\bar{m}$. Then, the algorithm iterates through the pattern and the LPS array is filled based on the comparison of the characters in the pattern. Each character is processed at most twice: once when a match is found, and again if a mismatch occurs (when the pointer j is reset based on the LPS array). Therefore, the time complexity required for the LPS computation is $\mathcal{O}(\bar{m})$, hence linear.

Once built the LPS array, the actual search of the substring begins, iterating over the characters in the reference string while using the information in the LPS array in order to skip unnecessary comparisons. Each character in the reference string is processed at most once. Additionally, the character comparisons are skipped according to the values in the LPS array. Therefore, said r the length of the reference, the time complexity of the search is $r + \mathcal{O}(\bar{m})$, which also makes the total time complexity of the whole phase.

In the last phase, the algorithm identifies the left and right breakpoints, which requires comparing characters in both target and reference sequences. Considering the search of the left breakpoint, where the algorithm starts at the last character of the first sample-specific string in the target and the character immediately following the segment in the middle segment extracted in the reference, the number of comparisons performed depends on the length of the first sample-specific string. Said s such length, in the worst case scenario, the algorithm compares s characters. The process is the same for the right breakpoint, but the length of the second sample-specific string may differ. Therefore, the time complexity must be proportional to the average length of the sample-specific strings, that will be referred as $\bar{s}$. Since the algorithm iterates over n sample-specific strings, the total time complexity of this phase is $\mathcal{O}(n \cdot \bar{s})$.

The overall time complexity is determined combining all the phases. Therefore, it can be expressed as $\mathcal{O}(n \cdot \bar{m} + n \cdot \bar{m} + r + \bar{m} + n \cdot \bar{s})$ and summarized as $\mathcal{O}(n \cdot \bar{m} + r + \bar{m} + n \cdot \bar{s})$. In this scenario, $\bar{m}$ is considerably smaller than the reference length r, and the same can be said for the average length of the sample specific strings $\bar{s}$. So, the dominant terms that actually determine the total time complexity of the algorithm are $n \cdot \bar{m}$ and s. Consequently, the result is $\mathcal{O}(r + n \cdot \bar{m})$.

Regarding space complexity, the algorithm requires $\mathcal{O}(n \cdot \bar{m})$ space to store the list of reverted sequences. Additionally, the LPS array used in the Knuth-Morris-Pratt algorithm demands $\mathcal{O}(\bar{m})$ space for each segment processed and temporarily

stored during the search phase. Temporary variables for indexing and counters occupy $\mathcal{O}(1)$ space. Although the middle segments are not stored long-term, the memory requirements are still significant due to the storage of reverted sequences. Therefore, the overall space complexity of the algorithm is $\mathcal{O}(n \cdot \bar{m})$.

4. Experimentation

4.1 Implementation

The practical experimentation was carried out using the SVDSS[23] tool on a Linux Ubuntu virtual machine installed on macOS, in order to detect the sample-specific strings. It requires two input files: the reference and the target, both in `FASTA` format. A `Python` script, `input_generator.py`, was used to generate mock examples. It creates a random DNA sequence and a modified copy with one or more inversions, producing two output files: `reference.fa` and `target.fa`. Numerous tests were conducted with strings of varying lengths in order to verify the algorithm's proper functionality.

Once the SVDSS binary file `SVDSS_linux_x86-64` is downloaded from (add link repo), the file permissions must be modified using the shell to make it executable, with the following command:

```
chmod +x SVDSS_linux_x86-64
```

Next, the reference file must be indexed, generating the output file `index.fmd` using the following command:

```
./SVDSS_linux_x86-64 index --reference reference.fa --index index.fmd
```

Finally, run the command to generate the `specifics.txt` file containing all the sample-specific strings detected and related data:

```
./SVDSS_linux_x86-64 search --index index.fmd --fastx target.fa

--bsize 10 > specifics.txt
```

The file `specifics.txt`, which is needed by the code implementing the algorithm described in Chapter 3, has the following structure, as generated by a random test:

```
T        90      19      0
*        240     19      0
*        492     16      0
*        625     15      0
```

The fields in the file are as follows:

- The target in which the sample-specific strings are detected, in comparison to the reference. If the sample-specific strings are detected in the same target as the previous row, the symbol "∗" is displayed;

- The index where the sample-specific string starts;

- The length of the sample-specific string;

- An additional field that will not be analyzed.

So, in this case the tool has found four sample-specific strings within a single target, at indexes 90, 240, 492 and 625 and having length 19, 19, 16 and 15. This file is given in input to the actual `Python` implementation of the algorithm, together with the `reference.fa` and `target.fa` FASTA files generated by `input_generator.py`. A function `read_fasta` reads `reference.fa` and `target.fa` and converts them into two strings, in order to be better analyzed. Another function `build_sample_specific_data` takes `target.fa` and the generated file `specifics.txt` and builds three lists: one of indexes, one of lengths, and one of sample-specific strings found in the target. This information will be needed by the function `check_reverted_sample_specific` that is the actual implementation of the algorithm.

The user will choose the boundaries, which are the indexes of the first and the last sample-specific strings that will be taken into consideration. This will help narrow down the analysis to a smaller and more manageable subset of sample-specific strings, and will be extremely helpful in the cases where regions with a high chance of having inversions are known.

In output are given the inverted sequences, the breakpoints of the inversions and the time taken to execute the program, that will be needed in the following paragraph to analyze the results.

All the described `Python` files are available at a public Git repository[24] on GitHub.

4.2   Results

The experimentation was conducted using various reference length and a target length fixed, of 50000 bp, making sure to simulate an actual long read. The reference lengths considered are 50000 bp, 100000 bp, 1000000 bp, and 10000000 bp.

Since the time complexity also depends from the number of inversions and their lengths, there have been generated samples containing 2, 4 and 6 inversions, both short or long. The first ones were, on average, 250 bp, while the latter ones 6500 bp on average.

4.2.1   No segmental duplications

This data was collected making 10 experimentations for each combination of reference length - number of inversions - length of the inversions, and calculating the average time measured in seconds, using the `time` library provided by `Python`. For this first set of experimentations, the samples have no segmental duplications. The Table 4.1 sums up the results:

Reference	Target	2 Inversions		4 Inversions		6 Inversions	
		Short	Long	Short	Long	Short	Long
50000	50000	0.0188	0.0260	0.0407	0.0459	0.0721	0.0758
100000	50000	0.0353	0.0400	0.0828	0.0934	0.1336	0.1396
1000000	50000	0.3009	0.3029	0.7305	0.8364	1.2190	1.2535
10000000	50000	2.9765	3.0082	7.3140	7.6067	11.8296	12.1238

Table 4.1: Execution times for various reference lengths and target lengths with different numbers of short and long inversions.

This data demonstrates that the execution time increases dramatically with larger reference lengths, making it, in practice, the dominant factor when it comes to complexity, as the graph in Figure 4.1 shows. For example, for 2 short inversions, the time rises from 0.0188 s at $r = 50000$ to 2.9765 s at $r = 10000000$, underlining how the contribution of r to the overall complexity is substantial.

In addition to this, the results show that the execution times grow almost linearly with the number of inversion n. For instance, at $r = 100000$, the time for 2 short inversions is 0.0353 s, while for 6 short inversions, it is 0.1336 s, roughly a 3.7x increase for 3 times the number of inversions.
It can be also seen that there is a slight but consistent increase in execution time when dealing with longer inversions. For example, at $r = 10000000$, the time for 2 short inversions is 0.3009 s, while for 2 long inversions and the same reference length, it is 0.3029 s. So, the impact of the inversion length m is less pronounced than r or n, though still measurable.
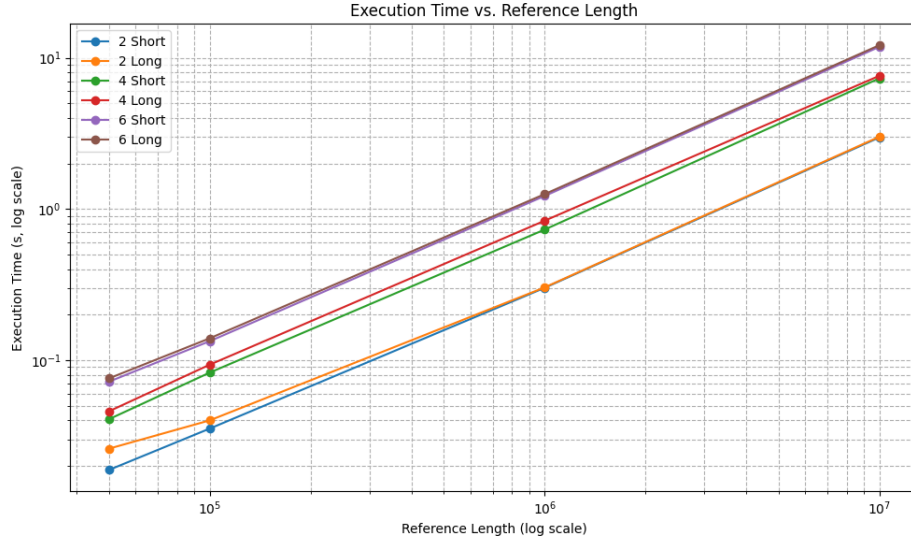
Figure 4.1: Execution time as a function of reference length for 2, 4, and 6 inversions, comparing short and long inversions. The plot uses a logarithmic scale on both axes, highlighting the linear relationship between the reference length and execution time, consistent with the algorithm's time complexity.

As r increases, the difference in execution times between short and long inversions diminishes in relative terms. In fact, at $r = 10000000$, the execution time for 2 short inversions is 2.9765 s, while for 2 long inversions is 3.0082 s. The difference is minimal compared to the absolute increase caused by the reference length.

4.2.2 With inverted duplications

Inverted duplications are a type of SV in which a segment of DNA is duplicated and the copy is inserted in the reverse orientation relative to the original. These duplications are often found in genomes and can be associated with various genetic disorders and evolutionary processes[25]. In an inverted duplication, the duplicated sequence is typically contiguous with the original, but flipped in orientation, leading to an inverted repeat structure, as summarized in Figure 4.2.

In order to generate samples that contain inverted duplications, another `Python` script was built, `input_duplications.py`. It allows setting the reference length, target length, and indexes that define the segment that will be inverted and inserted right after.

It was seen that the algorithm can also detect inversions in these situations, since sample-specific strings are detected at the beginning and end of the inverted segment in the target. This segment, when reversed, matches a segment the reference (referring to Figure 4.2, if y^r is reverted, y is obtained, which is present in the
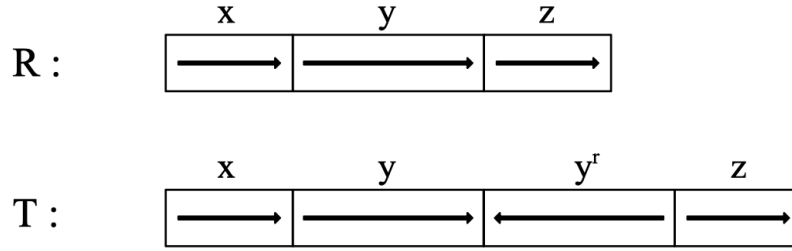
Figure 4.2: Reference sequence R and target sequence T that has undergone an inverted duplication event. In the reference, the sequence consists of three segments, x, y and z, all oriented in the same direction. However, in the target, the segment y is duplicated and inserted in the reverse orientation, indicated as y^r.

reference) and then the breakpoints are determined, just like in the case without inverted duplications.

5. Conclusions

For large reference genomes (e.g., in genomic studies where r can be several million base pairs), the algorithm's performance could become a bottleneck unless optimizations specific to handling larger sequences are applied. If many inversions are to be detected (e.g., during large-scale genomic rearrangement studies), the algorithm may need further optimizations in the loop or process that handles inversion detection.

References

1. Inversion - HGVS Nomenclature.

2. Marzieh Eslami Rasekh, Giorgia Chiatante, Mattia Miroballo, Joyce Tang, Mario Ventura, Chris T. Amemiya, Evan E. Eichler, Francesca Antonacci, and Can Alkan. Discovery of large genomic inversions using long range information. *BMC Genomics*, 18(1):65, December 2017.

3. Marta Puig, Sònia Casillas, Sergi Villatoro, and Mario Cáceres. Human inversions and their functional consequences. *Briefings in Functional Genomics*, 14(5):369–379, September 2015.

4. Haifei Hu, Armin Scheben, Jian Wang, Fangping Li, Chengdao Li, David Edwards, and Junliang Zhao. Unravelling inversions: Technological advances, challenges, and potential impact on crop breeding. *Plant Biotechnology Journal*, 22(3):544–554, March 2024.

5. A. H. Sturtevant. A Case of Rearrangement of Genes in Drosophila. *Proceedings of the National Academy of Sciences*, 7(8):235–237, August 1921.

6. Rui Faria, Kerstin Johannesson, Roger K. Butlin, and Anja M. Westram. Evolving Inversions. *Trends in Ecology & Evolution*, 34(3):239–248, March 2019.

7. Hao Wang, Carolina Makowski, Yanxiao Zhang, Anna Qi, Tobias Kaufmann, Olav B. Smeland, Mark Fiecas, Jian Yang, Peter M. Visscher, and Chi-Hua Chen. Chromosomal inversion polymorphisms shape human brain morphology. *Cell Reports*, 42(8):112896, August 2023.

8. David Porubsky, Ashley D. Sanders, Wolfram Höps, PingHsun Hsieh, Arvis Sulovari, Ruiyang Li, Ludovica Mercuri, Melanie Sorensen, Shwetha C. Murali, David Gordon, Stuart Cantsilieris, Alex A. Pollen, Mario Ventura, Francesca Antonacci, Tobias Marschall, Jan O. Korbel, and Evan E. Eichler. Recurrent inversion toggling and great ape genome evolution. *Nature Genetics*, 52(8):849–858, August 2020.

9. Luca Denti, Parsoa Khorsand, Paola Bonizzoni, Fereydoun Hormozdiari, and Rayan Chikhi. SVDSS: structural variation discovery in hard-to-call genomic regions using sample-specific strings from accurate long reads. *Nature Methods*, 20(4):550–558, April 2023.

10. Definition:Prefix - ProofWiki.

11. Deoxyribonucleic Acid (DNA).

12. Matt Carter, Rachel Essner, Nitsan Goldstein, and Manasi Iyer. Chapter 9 - Identifying Genes and Proteins of Interest. In *Guide to research techniques in neuroscience*. Academic Press, an imprint of Elsevier, London San Diego, CA Cambridge, MA Oxford, third edition edition, 2022.

13. Liu Ting, Huang Jun, and Chen Junjie. RAD18 lives a double life: Its implication in DNA double-strand break repair. *DNA Repair*, 9(12):1241–1248, December 2010.

14. Homologous Recombination.

15. Matthew M Parks, Charles E Lawrence, and Benjamin J Raphael. Detecting non-allelic homologous recombination from high-throughput sequencing data. *Genome Biology*, 16(1):72, December 2015.

16. Bruna Burssed, Malú Zamariolli, Fernanda Teixeira Bellucco, and Maria Isabel Melaragno. Mechanisms of structural chromosomal rearrangement formation. *Molecular Cytogenetics*, 15(1):23, June 2022.

17. Lars Feuk. Inversion variants in the human genome: role in disease and genome architecture. *Genome Medicine*, 2(2):11, 2010.

18. Li Qu, Huaiqiu Zhu, and May Wang. Micro-Inversions In Human Cancer Genomes. In *2018 40th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, pages 1323–1326, Honolulu, HI, July 2018. IEEE.

19. Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

20. HTS format specifications.

21. Long-read sequencing vs short-read sequencing.

22. Hayan Lee, James Gurtowski, Shinjae Yoo, Shoshana Marcus, W. Richard McCombie, and Michael Schatz. Error correction and assembly complexity of single molecule sequencing reads, June 2014.

23. Parsoa Khorsand. Parsoa/SVDSS, July 2024. original-date: 2020-03-10T17:55:53Z.

24. Silvia Cambiago. silviacambiago/bachelor-thesis, October 2024. original-date: 2024-03-02T13:51:12Z.

25. Karen E. Hermetz, Scott Newman, Karen N. Conneely, Christa L. Martin, Blake C. Ballif, Lisa G. Shaffer, Jannine D. Cody, and M. Katharine Rudd. Large Inverted Duplications in the Human Genome Form via a Fold-Back Mechanism. *PLoS Genetics*, 10(1):e1004139, January 2014.

È la terza volta che riscrivo questa pagina da capo. Le prime due, infatti, rileggendo ciò che avevo abbozzato, sentivo che quelle frasi non mi appartenevano più, quasi come fossero scritte da una persona totalmente diversa, con una storia da raccontare che con la me di oggi non c'entra davvero più nulla. Se non altro, tutto questo è testimonianza del fatto che negli ultimi tre anni ho davvero compiuto un percorso, ed ogni cambiamento è stato forse per la prima volta genuino al 100% e non frutto di un personaggio pilotato a tavolino.

Grazie a mio papà, nulla di tutto questo sarebbe stato possibile senza il tuo supporto economico, sempre costante e puntuale. Sappi che in questi tre anni ho cercato tutti i giorni di far valere al massimo queste spese che con tanta fiducia mi hai concesso, impegnandomi al massimo negli studi e cercando di amministrare il resto nella maniera più saggia possibile, come tu mi hai insegnato da sempre. Prometto che un giorno ti farò vedere davvero i frutti. Siamo due anime con obiettivi molto in contrasto e ciò che condividiamo sono più i difetti che i pregi, ma quando davvero conta so che posso sempre fidarmi di te e che sei sempre dalla mia parte. Non è scontato un supporto del genere, quindi sentiti davvero parte di questo traguardo. Rimani il compagno di viaggio migliore di sempre.

Grazie a mia zia Laura e a Philip, con cui ho vissuto alcuni tra i momenti più felici della mia vita. Siete la mia seconda famiglia, a tutto quello a cui non possono arrivare i miei genitori ci avete pensato voi.

Grazie ad Alessandra

Grazie a Parham, ci hai creduto prima ancora di me.

Grazie a Giorgio

E per ultima, ma non di certo per importanza, grazie a mia mamma, sei la persona più forte che conosca, mi hai sempre sostenuta insegnandomi a trovare la luce anche nei momenti più bui, perché c'è sempre modo di ricominciare. Voglio che tu sappia che nutro una stima infinita nei tuoi confronti, che tante volte ti ho sottovalutata, ma mai quanto ti sei sottovalutata tu. Sei infatti una persona brillante e sempre in evoluzione, sappi che sono tremendamente fiera di te. Grazie per l'estrema fiducia che mi concedi da sempre, anche quando al resto del mondo non sembrava opportuno. La tua completa sincerità //finire. Grazie per non avermi mai detto ciò che volevo sentirmi dire, ma ciò che avevo bisogno di sentire, in qualsiasi situazione, anche se in un primo momento potevo rimanere ferita. Ed è proprio una delle tue frasi quella che penso essere ciò che meglio rappresenta il

35

mio percorso, ma anche me stessa come persona. Anzi, rappresenta proprio noi due. Una frase semplice, ma proprio per questo estremamente impattante.

Perché non importa come si parte, ma come si arriva.