Università degli Studi di Milano - Bicocca

Dipartimento di Informatica, Sistemistica e Comunicazione

Corso di Laurea in Informatica

Detection of Genomic Inversions Using Sample-Specific Strings

Relatore: Prof.ssa Paola Bonizzoni

Correlatore: Dott. Luca Denti

Tesi di Laurea 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 an efficient algorithm to detect genomic inversions from sequencing data. More precisely, it focuses on leveraging long-read sequencing technology and utilizes sample-specific strings to accurately identify these inversions within genomic sequences.

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 core of the thesis is a complete description of the algorithm, with a complete theoretical analysis of its design, its computational complexity and the logic that lies behind it.

Practical implementation aspects are then explored, highlighting experiments conducted with the SVDSS tool to validate the algorithm's performance. The results are finally presented and discussed, demonstrating the algorithm's effectiveness.

Contents

1. Introduction

1.1 Context

As defined by the Human Genome Variation Society[1] (HGVS), inversions 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 base pairs (bp). They may delete, insert, duplicate, invert, or move genomic sequences[1] (see Figure 1.1).

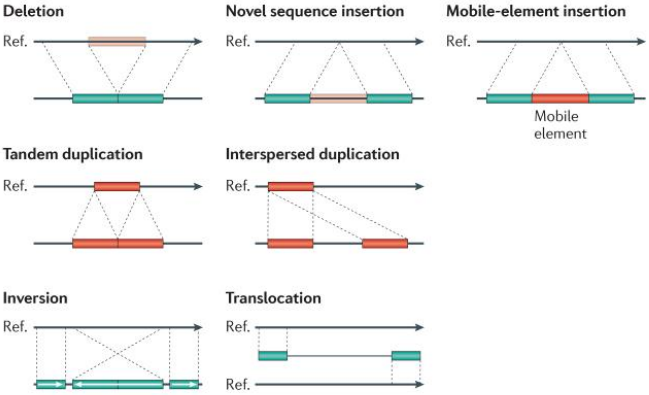


Figure 1.1: Outline of different SV-archetypes.

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[2]. Although inversions constitute only a small fraction of SVs across various organisms, ranging from 0.5% to 7%[3], they can span over 100 Mb and collectively account for up to 10% of the genome, holding considerable functional and evolu-

[1] https://hgvs-nomenclature.org/stable/recommendations/DNA/inversion/

tionary significance.

Inversions were first identified in 1917 by Alfred Sturtevant[4], who found out that they act as suppressors of recombination between chromosomes. Since then, inversions have been increasingly drawing attention, due to their crucial role in driving genome evolution, as well as to their link to the evolutionary processes of many species. In fact, it was eventually discovered 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[5]. Inversions thus facilitate genomic isolation, which can lead to the formation of novel genotypes and phenotypes, contributing to the genetic diversity we can observe today.

However, inversions are not only related to evolution: they are involved in the development of several diseases, as will be shown in Chapter 2. In fact, a recent study shows that the contribution of chromosomal inversions to phenotypic diversity can even affect brain development and lead to neurological disorders[6].

1.2 Inversions detection

Due to their significance, the challenge of detecting inversions has been widely addressed. The first methods developed to this scope, such as cytogenetics or approaches based on polymerase chain reaction (PCR), were labor-intensive and lacked resolution, which limited their effectiveness[3]. With the advent of high-throughput next-generation sequencing (NGS), large-scale population studies of inversions finally became possible, enabling the discovery of polymorphic inversions and their association with phenotypic traits. Additionally, highly effective, long-read DNA sequencing technologies such as PacBio HiFi sequencing or Oxford Nanopore duplex sequencing now allow to generate sequencing reads between 10 kb and 2 Mb. These long reads can span repetitive and complex genomic regions, thus making identifying inversions a more straightforward task. Despite all this technological progress, inversions still remain one of the most underascertained forms of structural variation in human and non-human primate genomes, limiting our understanding of their significance in evolution[7]. The main challenge here continues to be the precise determination of inversion breakpoints.

1.3 Project

The scope of this thesis is to present 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. The practical experimentation will use a `Python` implementation of the algorithm, in combination with the SVDSS[8] tool, that will also be implemented in the code and will help detecting the number, position, and length of the sample-specific strings.

Chapter 2 will present the necessary preliminaries to understand how the algorithm works, including the definitions used throughout the thesis and the biological background that underlies inversions, other than their practical biological consequences.
Chapter 3 will explain the algorithm in detail, including pseudocode and a complete theoretical analysis. This chapter will address, among other things, time and space complexity and proof of correctness.
Chapter 4 will focus on the practical experimentation, 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

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.

2.1 Definitions

In this paragraph will be introduced the definitions that will be used throughout the thesis.
The definition of sample-specific string[9] is as follows:

Definition 1 (Sample-specific string (SFS))**.** *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.*

Subsequently, it will be necessary to formally define what constitutes an inverted segment of the target T in relation to the reference R. The following definitions establish these key concepts:

Definition 2 (Inversion)**.** *Let T be a target string and R a reference string. A substring $s = T[i, j]$ for $1 \leq i < j \leq |T|$ is a single inversion in the target T with respect to R if s^{rc} occurs as a substring of R. If the inversion in T occurs from i to j, it is denoted as a triple (T, i, j).*

Next, the concept of substring from i to j is defined, along with the notion of a single inversion:

Definition 3 (Substring)**.** *Given a string S, the substring $S[i : j]$ is defined as the sequence of characters in S that starts at position i (inclusive) and ends at position j (exclusive). Formally, $S[i : j] = s_i s_{i+1} \dots s_{j-1}$, where s_k represents the character at position k in the string S, and $0 \leq i < j \leq |S|$.*

Definition 4 (Single Inversion). *Given a string S, another string S' is obtained by a single inversion on S from i to j if S' is derived by reversing and complementing $S[i,j]$, denoted as $S[i,j]^{rc}$.*

Finally, the concept of inversion breakpoint is introduced:

Definition 5 (Inversion Breakpoint). *Given an inversion (S', i, j) in string S' with respect to string S, the indexes i and j are referred to as breakpoints of the inversion if $S[i,j] = S'[i,j]^{rc}$, and no index $i' < i$ or $j' > j$ exists such that $S[i',j'] = S'[i',j']^{rc}$.*

These definitions will be used in the description of the algorithm for solving the Detecting-Inversions problem, described in Chapter 3. In cases involving multiple inversions, the target may result from several disjoint inversions.

Since the Knuth-Morris-Pratt algorithm will be introduced in Section 2.4 below as the method used to verify the presence of the middle segment between two sample-specific strings (extracted from the target) within the reference, a few clarifying definitions[1] are presented here:

Definition 6 (Proper 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'. If $S = TT'$, T is a proper prefix of S if $T' \neq \varepsilon$, meaning T' is not the empty string.*

Definition 7 (Proper 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. If $S = TT'$, T' is a proper suffix of S if $T \neq \varepsilon$, meaning T is not the empty string.*

2.2 Biological Background

2.2.1 DNA

Before addressing the computational part of the project, it is useful to present an overview of its biological background.

Deoxyribonucleic acid (DNA) is the molecule that carries genetic information for the development and functioning of an organism[2]. 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, specifically 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

[1] https://proofwiki.org/wiki/Definition:Prefix

[2] https://www.genome.gov/genetics-glossary/Deoxyribonucleic-Acid

(C), as shown in Figure 2.1.

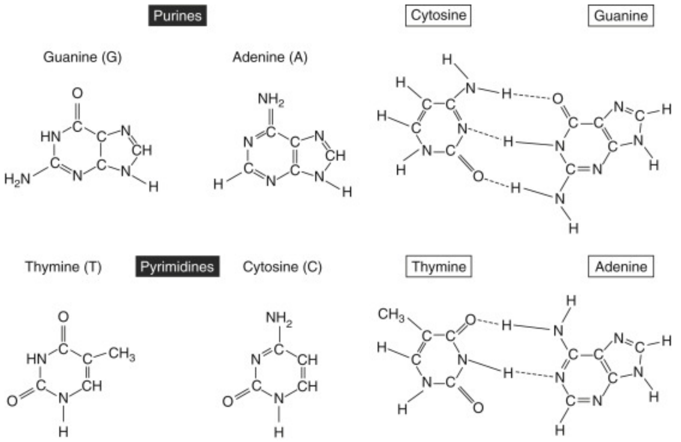


Figure 2.1: DNA nucleotide bases and base pairing.

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[10]. 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 by 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 types of DNA lesions. In addition to this, DSBs can also be produced during physiological processes, such as recombination or meiosis[11]. 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 a role in the generation of genetic diversity, in the repair of damaged DNA, in the homologous alignment of chromosomes required for successful completion of meiotic cell division, and in the generation of genomic alterations that lead to changes in gene expression. Classical homologous recombination is a type of genetic recombination in which nucleotide sequences are exchanged between two similar or identical molecules of DNA, as shown 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 exchange of DNA is an important source of the genomic variation seen among offspring[3]. 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[12].

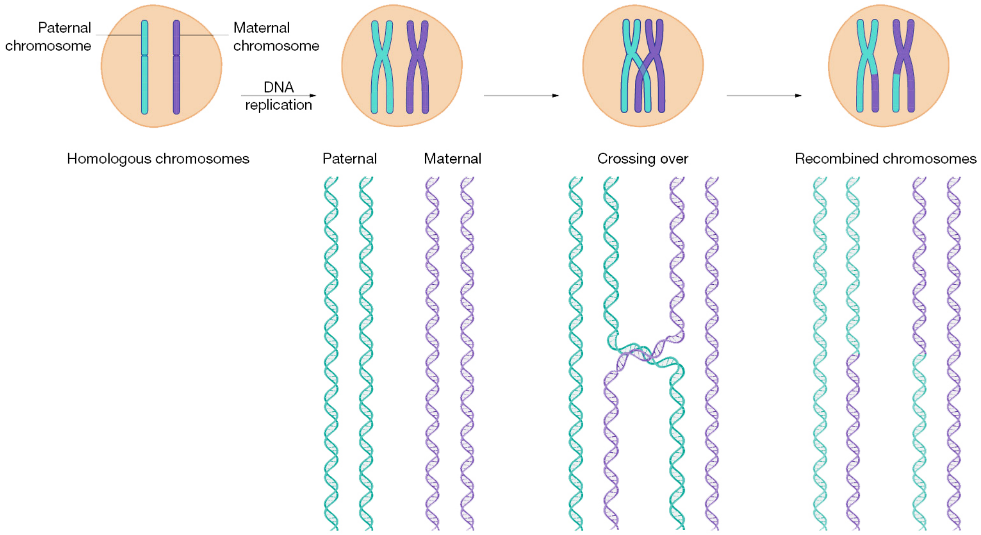


Figure 2.2: Homologous recombination between maternal and paternal chromosomes.

However, a similar recombination process can also occur 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).
It 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[13]. 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.

[3] https://www.genome.gov/genetics-glossary/homologous-recombination

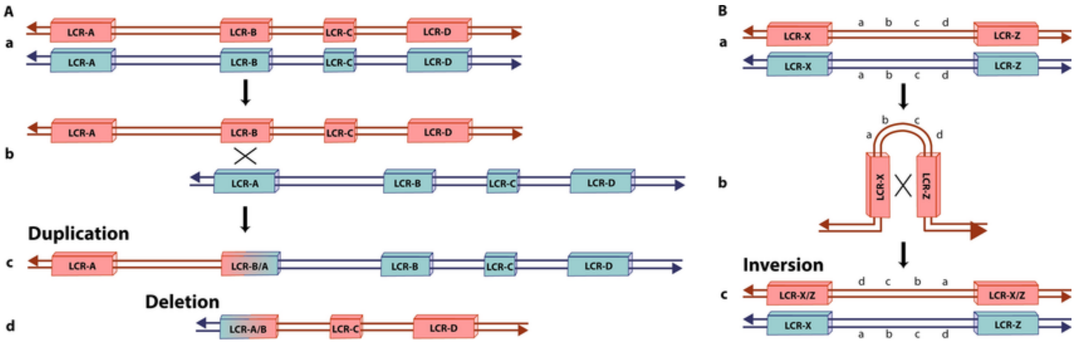


Figure 2.3: Non-Allelic Homologous Recombination (NAHR) mechanism. A. NAHR leading to the formation of duplication and deletion: (a) Normal chromosome pairing and alignment of Low Copy Repeats (LCRs) in the same orientation. (b) A misalignment between LCRs due to their high level of sequence identity 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 NAHR, 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[14].

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. These mainly invert heterochromatic sequences and are frequently observed in cytogenetic analysis[2]. However, not all inversions are harmless, and several diseases have been found to be occasionally caused by inversions, mostly due to direct disruption of one gene or alteration of 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 to have multiple patients with the same inversion, it is often problematic to assess whether the inversion present in the patient is actually associated with the phenotype. The exception is when the inversion breakpoint falls within or near a gene that has previously been associated with a 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 disease-triggering recurrent inversions is linked to 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[14].

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 shown by a study conducted in 2018[15]. This study analyzed the distribution of micro-inversions among 24 chromosomes in four types of cancer (hepatocellular, lung, pancreatic and bladder), and found that the average count of micro-inversions per individual in the normal samples was 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).

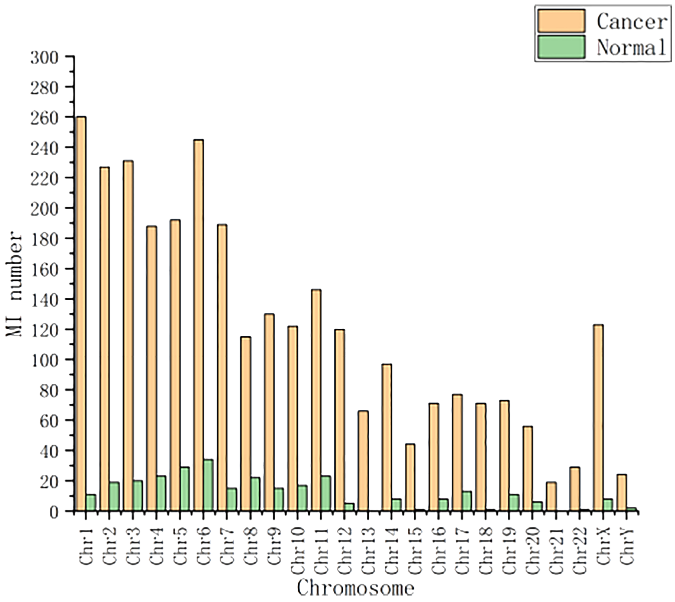


Figure 2.4: Micro-inversion (MI) distribution among 24 chromosomes. Average count of MIs per individual among the four types of cancer.

2.4 Knuth-Morris-Pratt Algorithm

Knuth-Morris-Pratt[16] (KMP) is the algorithm that will be used in this project to detect the position of 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 naïve 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. 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

KMP 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 certainly match, thus reducing the number of comparisons needed and ultimately speeding up the search process.

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. $LPS[0]$ is set to 0 because there is no proper prefix for a single character. Two pointers are also initialized: i, which iterates through the pattern, is set to 1, and j, which will track the length of the previous longest proper prefix suffix, 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 determine the next smaller prefix that could still match. If $j == 0$, there is no valid prefix to fall back to, so $LPS[i]$ is set to 0 and i is incremented in order to move to the next character.

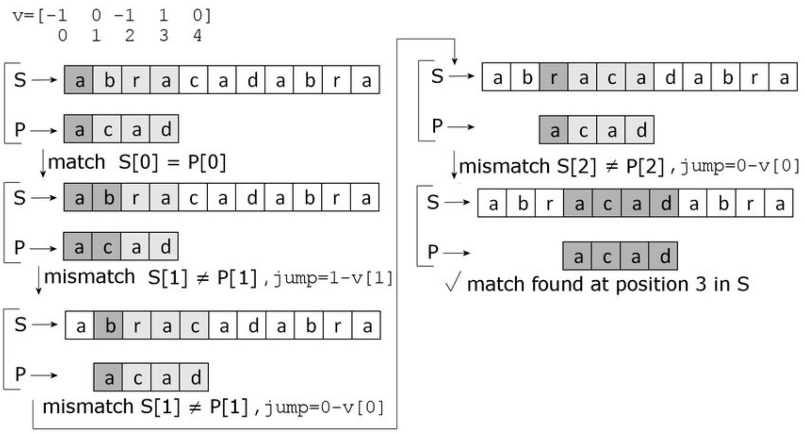
The process continues until i reaches the length of the pattern.

P :	A	B	C	A	B	D
LPS :	0	0	0	1	2	0

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 KMP 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 it happens in the naïve algorithm, KMP consults $LPS[i-1]$ in order to determine how many characters of the pattern can be skipped. Thus, 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.


v=[-1 0 -1 1 0]
0 1 2 3 4
S
P
a b r a c a d a b r a
a c a d
match S[0] = P[0]
mismatch S[1] ≠ P[1], jump=1-v[1]
mismatch S[1] ≠ P[1], jump=0-v[0]
mismatch S[2] ≠ P[2], jump=0-v[0]
✓ match found at position 3 in S

Among other solutions, like the suffix array or the Boyer-Moore algorithm, the KMP algorithm was chosen for this project because it is more suitable in contexts with a very limited alphabet, such as DNA (A, C, T, G). Additionally, KMP performs well even when the length of the reference is large, unlike, for example, the suffix array method, which which can become less efficient in such cases due to its preprocessing requirements. This makes KMP the optimal choice for detecting 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 strand of DNA. A read is defined as a raw sequence generated by a sequencing machine[4]. A read may consist of multiple segments. For sequencing data, reads are indexed in 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 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:** have lengths usually ranging 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, high-quality data at the lowest cost per base. On the other hand, short reads fail to generate a sufficient overlap between the DNA fragments. Overall, this means that it can be challenging to use them for sequencing highly complex, repetitive libraries, like the human genome. The dominant technologies in this field are Illumina's platform and Thermo Fisher Scientific's Ion Proton. However, multiple new sequencing technologies have surfacedin 2022, and have now 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[5].

- **Long reads:** have lengths usually ranging from 5000 to 50000 base pairs [17]. 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

[4] https://samtools.github.io/hts-specs/

[5] https://frontlinegenomics.com/long-read-sequencing-vs-short-read-sequencing/

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.

Sequencing data are the main input of several bioinformatics algorithms. The algorithm described in this work uses long read sequencing data. Indeed, due to their length, they are suited to detect enough long inversions.

2.6 FASTA format

FASTA is a text-based format used in bioinformatics to represent 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. As detailed in Chapter 4 below, FASTA will be used in this project to represent both the reference sequence and the target, which is a long read of the reference. A FASTA sequence 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 an optional description separated from the identifier by a white space or tab. The sequence data starts on the next line following the text line and ends at the appearance of another line starting with a ">", which signals 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 Problem definition

First, the problem will be formalized and will be referred to as Detecting-Inversions.

Problem 1 (Detecting-Inversions).
INPUT: *a collection of sample-specific strings for a target sequence T and a reference R.*
OUTPUT: *a list of disjoint segments $T[i, j]$ that are inverted in relation to the reference sequence R.*

The main idea of the algorithm is that the target is a long read coming from a genome G that differs from the reference R by possible operations of inversions. Such operations will be non-overlapping, meaning repeated inversions in overlapping positions will not be considered.

3.2 Pseudocode

Algorithm 1 Check For Inversions Using Sample-Specific Strings

```
Input: target: string, reference: string, indexes: list of integers, lengths: list of integers, sample_specific_strings: list of strings
Output: reverted: list of reverted sequences found
reverted ← [ ] {Initialize list to store inversions}
for $i$ ← 1 to sample_specific_strings.length − 1 do
    $j$ ← indexes[$i$] {Start index of current sample-specific string (SFS)}
    $k$ ← indexes[$i+1$] {Start index of next SFS}
    middle ← target[$j$ + lengths[$i$] : $k$] {Extract potential inversion}
    if middle == $\emptyset$ then
        CONTINUE
    end if
    middle ← REVERT_AND_COMPLEMENT(middle)
    (found, start) ← CHECK_SUBSTRING(reference, middle) {Search in reference}
    if found then
        left_increment ← 1
        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
            left_increment ← left_increment +1 {Extend left breakpoint}
        end while
        left_breakpoint ← target[$j$ + lengths[$i$] − left_increment : $j$ + lengths[$i$] − left_increment + 2]
        right_increment ← 0
        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
            right_increment ← right_increment +1 {Extend right breakpoint}
        end while
        right_breakpoint ← target[$k$ + right_increment − 1 : $k$ + right_increment + 1]
        inversion ← target[$j$ + lengths[$i$] − left_increment + 1 : $k$ + right_increment]
        APPEND reverted ← inversion {Store the inversion}
    end if
end for
return reverted
```

Algorithm 2 Revert and Complement DNA Sequence

```
Input: DNA sequence: string
Output: reverse and complement sequence: string
complements ← $\{'A' :' T',' C' :' G',' G' :' C',' T' :' A'\}$
reverted_complement ← $\varepsilon$
for $i$ ← sequence.length − 1 to 0 do
    $base$ ← sequence[$i$]
    $complement$ ← complements[$base$]
    APPEND reverted_complement ← $complement$
end for
return reverted_complement
```

Algorithm 3 Knuth-Morris-Pratt Prefix Function

```
Input: pattern: string
Output: LPS array: list of integers
$m$ ← length of pattern
$lps$ ← $[0] * m$
$j \leftarrow 0$
$i \leftarrow 1$
while $i < m$ do
    if pattern[$i$] == pattern[$j$] then
        $j \leftarrow j + 1$
        $lps[i] \leftarrow j$
        $i \leftarrow i + 1$
    else
        if $j \neq 0$ then
            $j \leftarrow lps[j - 1]$
        else
            $lps[i] \leftarrow 0$
            $i \leftarrow i + 1$
        end if
    end if
end while
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)

```
$n \leftarrow$ length of reference
$m \leftarrow$ length of string
$lps \leftarrow$ KMP Prefix Function($string$)
$i \leftarrow 0$
$j \leftarrow 0$
while $i < n$ do
    if string[$j$] == reference[$i$] then
        $i \leftarrow i + 1$
        $j \leftarrow j + 1$
    end if
    if $j == m$ then
        return (True, $i - j$)
    else if $i < n$ and string[$j$] $\neq$ reference[$i$] then
        if $j \neq 0$ then
            $j \leftarrow lps[j - 1]$
        else
            $i \leftarrow i + 1$
        end if
    end if
end while
return (False, $-1$)
```

3.3 High-Level Explanation

The algorithm for solving the Detecting-Inversions problem is structured in a way that involves different phases, as d etailed below. The approach assumes solving the Detecting-Inversions problem starting from SFSs.
The algorithm addresses an auxiliary problem where the input data has the additional information given by a list of SFSs of the target with respect to the reference R.
It is important to note that each SFS represents a substring that does not occur in R, and potentially each SFS is used to capture the breakpoints of an inversion. The algorithm proceeds as follows:

- **Setup (line 1):** an empty list, *reverted*, is initialized to store any detected inverted sequences. This list ensures that only valid inverted segments are stored throughout the process.

- **Iterate through SFS pairs and extraction of the segment (lines 2-8):** the algorithm focuses on the prime candidate for detecting inversions, which is the segment 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.

- **Reverse Complement Generation (line 9):** 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 middle segment 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 (see Algorithm 3). 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 (line 10):** 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 KMP algorithm, detailed in Chapter 2. If m_r is found in R, the starting index s of the occurrence of m_r in R is stored for further analysis.

- **Identify breakpoints (lines 12-23):** 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 (lines 12-16):** 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 (lines 17-21):** 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.

3.4 Preconditions for Algorithm Correctness

For this algorithm to function correctly, the middle segment extracted from the target must be sufficiently long to ensure its uniqueness within the reference sequence. This is crucial, because a middle segment that is too short might match multiple locations within the reference sequence, leading to a potential misalignment. This means that the algorithm might identify incorrect breakpoints or even wrongly identify an inversion that does not actually exist, resulting in a false positive.

A second prerequisite for algorithm accuracy is that 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 might not match the expected part of the reference sequence, leading to false negatives where actual inversions are missed.

3.5 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. An example is given below:

Example 2. *This is an example of the edge case where the two sample-specific strings are consecutive:*

- ***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.

3.6 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.

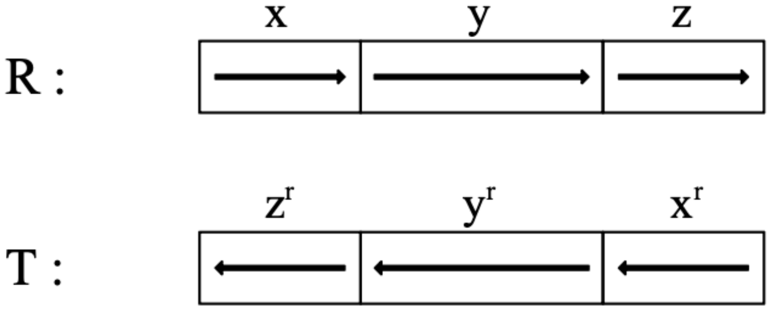


Figure 3.1: R represents the reference string, that in the following demonstration will be referred as S, and T is the target where the inversion is found, that will be later referred as S'

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

1. Start with $z^r (a_k \dots a_2 a_1) x^r$;
2. Insert a_{k+1} before $a_k \dots a_2 a_1$;

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

3.7 Time and Space Complexity

This analysis will start with time complexity.
During the initialization phase, the necessary parameters are set up, and the list for storing 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 a time that is 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 middle segment 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 KMP 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 may vary when multiple inversions having different lengths are found during the whole computation, the actual value analyzed here 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 the LPS array has been built, 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 of the average length of the sample specific strings $\bar{s}$. Since 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 KMP 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[8] 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 that the algorithm was functioning properly.

The tool is available at a public Git repository[1] on GitHub. Once the SVDSS binary file `SVDSS_linux_x86-64` is downloaded, 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
```

The final step is to 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:

[1] https://github.com/Parsoa/SVDSS

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

The fields in the file are as follows:

- The target sequence where 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.

In this case the tool 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. This function implements the main algorithm for detecting inversion breakpoints by analyzing the sample-specific strings.
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 section to analyze the results.
All the described `Python` files are available at a public Git repository[2] on GitHub.

[2] https://github.com/silviacambiago/bachelor-thesis

4.2 Results

The experimentation was conducted using various reference lengths and a target of fixed length, 50000 bp, making sure to simulate an actual long read. The reference lengths considered were 50000 bp, 100000 bp, 1000000 bp, and 10000000 bp. Since time complexity also depends on the number of inversions and their lengths, the generated samples contained 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 lengths - number of inversions - length of 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. 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.

These data demonstrates that the execution time increases dramatically with larger reference lengths, making it practically the dominant factor when it comes to complexity, as shown by the graph in Figure 4.1. For example, for 2 short inversions, the time rose from 0.0188 s at $r = 50000$ to 2.9765 s at $r = 10000000$, highlighting the substantial contribution of r to the overall complexity.

Additionally, the results show that execution times increase almost linearly with the number of inversions n. For instance, at $r = 100000$, the time for 2 short inversions was 0.0353 s, while for 6 short inversions, it was 0.1336 s: roughly a 3.7x increase for 3 times the number of inversions.
Also evident is a slight but consistent increase in execution time when dealing with

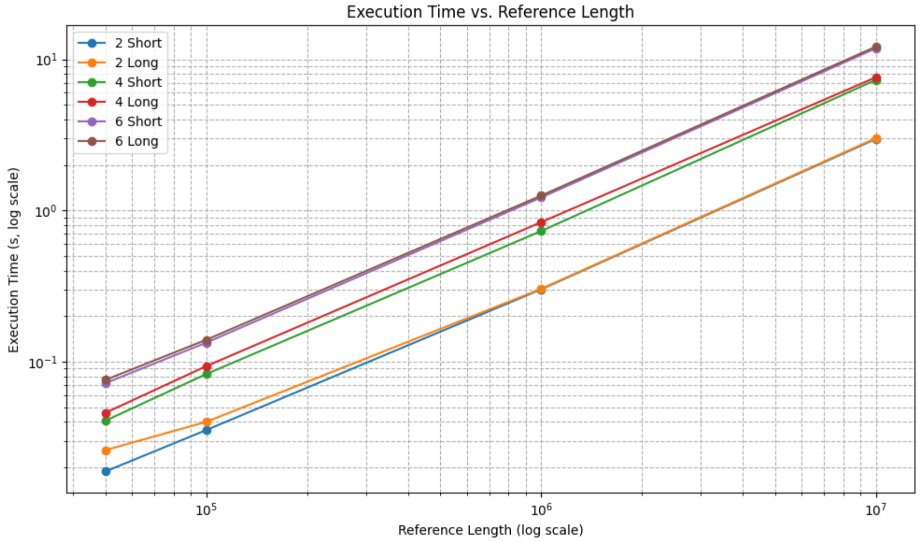


Figure 4.1: Execution time as a function of reference length for 2, 4, and 6 inversions, comparing short and long inversions. The plot uses a logarithmic scale on both axes, highlighting the linear relationship between the reference length and execution time, that is consistent with the algorithm's time complexity.

longer inversions. For example, at $r = 10000000$, the time for 2 short inversions was 0.3009 s, while for 2 long inversions and the same reference length, it was 0.3029 s. The impact of inversion length m is less pronounced than the impact of r or n, but still measurable.

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[18]. 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 the start and end indexes of the segment that will be inverted and inserted directly afterward.

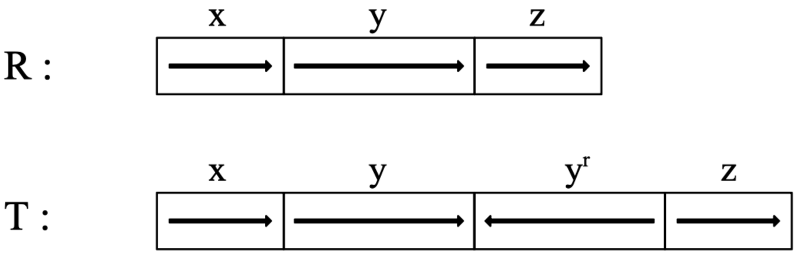


Figure 4.2: Reference sequence R and target sequence T that has undergone an inverted duplication event. In the reference, the sequence consists of three segments, x, y and z, all oriented in the same direction. However, in the target, the segment y is duplicated and inserted in the reverse orientation, indicated as y^r.

The experimentation shows 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 in the reference (see Figure 4.2 where, if y^r is reverted, y is obtained, which is present in the reference). The next step was to determine the breakpoints, just like in the case without inverted duplications.

5. Conclusions

5.1 Final Considerations

This study focuses on the development and evaluation of an algorithm specifically designed to detect inversions in genomic sequences obtained from long reads. The experimental results show that the algorithm performs efficiently for moderate reference lengths and inversion counts. However, its execution time increases significantly as the reference sequence length grows. This behavior is consistent with the anticipated time complexity of $\theta(r + n \cdot m)$, where the reference length r is the dominant factor. A significant advantage of the algorithm is its ability to effectively handle inverted duplications, expanding the method's applicability to real-world genomic data where such variations are common.

Overall, the algorithm provides a reliable approach for inversion detection, although further optimizations may be necessary to manage even larger datasets or more complex genomic structures effectively. It is also essential to note that the algorithm operates under the assumption that the input samples are error-free. Any presence of sequencing errors or mutations leads to incorrect identification of inversions, ultimately affecting the overall performance and accuracy of the algorithm.

5.2 Future improvements

Future improvements to the inversion detection algorithm will focus on two key areas: enhancing its capability to handle sequencing errors and mutations, and increasing its efficiency for processing long reference sequences that may span millions of base pairs. Given that it is almost impossible to obtain perfectly clean reads in real-life scenarios, the current implementation of the algorithm may not be advisable for practical use. Additionally, experimental results demonstrate that as the reference length increases, the execution time of the algorithm rises significantly, highlighting the need for optimization in this area.

Addressing these areas will significantly enhance the algorithm's reliability and applicability in genomic research, particularly when dealing with extensive datasets typical of modern genomic studies.

References

1. 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.

2. 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.

3. 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.

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

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

6. 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.

7. 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.

8. 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.

9. Parsoa Khorsand, Luca Denti, Human Genome Structural Variant Consortium, Paola Bonizzoni, Rayan Chikhi, and Fereydoun Hormozdiari. Comparative genome analysis using sample-specific string detection in accurate long reads. *Bioinformatics Advances*, 1(1):vbab005, June 2021.

10. 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.

11. 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.

12. 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.

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

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

15. 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.

16. 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.

17. 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.

18. 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.