Universiteit Leiden
The Netherlands

Opleiding Informatica

Methods to Describe Tandem Repeats in DNA Sequences

Eline Rodoe

Supervisors:
Jonathan K. Vis & Mark A. Santcroos

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

15/01/2026

Abstract

Repeats in DNA sequences play an important role in both forensic and biomedical science, yet existing nomenclature systems such as HGVS lack sufficient rules to consistently describe repeat structures. This thesis addresses the problem of how repeats in a DNA sequence can be described.

We first study the problem of identifying all repeats in a string. A naive algorithm is introduced, followed by a detailed description of an existing linear time algorithm based on s-factorization and suffix trees. Building on this, we introduce the maximal cover method, which aims to describe a DNA sequence by selecting a set of non-overlapping repeats that together cover as many symbols as possible.

Several algorithms are developed to compute such descriptions, including a brute-force approach, two greedy algorithms, a dynamic programming algorithm, and a linear time algorithm.

Finally, experiments on synthetic sequences and real DNA data from dbSNP are conducted to evaluate both runtime performance and the quality of the resulting descriptions. The results show that, while the greedy algorithms are fast, they often fail to achieve the maximum number of covered symbols. In contrast, the dynamic programming and linear time algorithms do achieve this with efficient runtimes. Moreover, the experiments demonstrate that in some cases our method produces descriptions identical to those in dbSNP, while in other cases it describes more repeats. However, this sometimes results in longer descriptions.

Contents

1 Introduction

A tandem repeat is a pattern that occurs at least twice consecutively. Repeats within DNA sequences play an important role in several scientific fields, including forensic science and biomedical science. In forensic science, repeats are widely used for human identification, as both the length and the sequence composition of repeats vary significantly between individuals [But07]. In the biomedical domain, repeats are also of major importance. While many repeats normally occur in small copy numbers, abnormal expansions of repeat regions can occur in exceptional cases. Such expansions are associated with severe genetic disorders, including Huntington's disease [Xu24].
However, the study by Santcroos et al. [SKL+25] shows that the HGVS nomenclature lacks sufficient rules to consistently describe certain complex forms of repeats, known as mixed repeats. The HGVS nomenclature is an internationally recognized standard for the description of DNA sequence variants. Unlike simple repeats, mixed repeats consist of multiple, consecutive repeat motifs that differ in sequence.
Due to the absence of well-defined rules for mixed repeats, no standardized representation currently exists for describing such structures. As a consequence, mixed repeat descriptions are frequently rejected by official validation tools, such as the NCBI Variation Service. Consider, for example, the sequence `CACACACACTCTCTC`. This sequence can be described either as `CA[4]TC[3]C[1]` or as `C[1]AC[4]TC[3]`, where the number within square brackets indicates the number of repetitions of the preceding pattern.

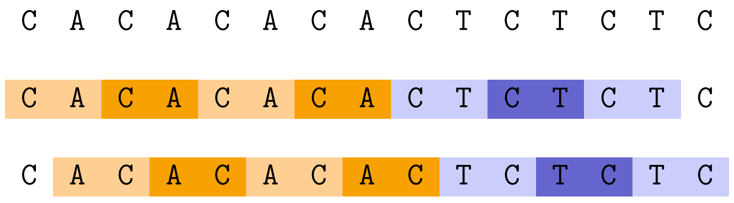

C A C A C A C A C T C T C T C
C A C A C A C A C T C T C T C
C A C A C A C A C T C T C T C

Figure 1: Consider the string S = `CACACACACTCTCTC`. This string can be described either as `CA[4]TC[3]C[1]` or as `C[1]AC[4]TC[3]`.

These multiple interpretations lead to challenges in the standardization, validation, and storage of such variants in existing databases.
Furthermore, the study by Van der Gaag and De Knijff [vd15] also highlights the need for a new way to describe repeats within DNA sequences. This need arises from developments in Massively Parallel Sequencing (MPS). This technology allows repeated DNA to be studied in greater detail. Unlike the current approach, capillary electrophoresis, which only determines how long a DNA fragment is, the new method reveals the exact order of the DNA building blocks. This makes it possible to detect differences within repeated regions that were previously invisible. However, this increased level of detail also necessitates a new nomenclature to accurately describe the observed variation.
To address these challenges, a method for describing repeats in DNA sequences is required. This leads to the following main research question:

RQ: How can tandem repeats in a four-character alphabet be described?

To answer this research question, multiple algorithms are developed. The first two algorithms extract all repeats from a given DNA sequence, whereas the remaining five algorithms describe the sequence based on the previously identified repeats.

2 Background

All living organisms are composed of cells, which are complex and dynamic structures containing a range of molecular components. These include the plasma membrane, cytoplasm, organelles, and deoxyribonucleic acid (DNA). [SuRM+24] This research focuses specifically on DNA. DNA contains the complete set of hereditary information required for an organism to function, grow, and reproduce.

The DNA molecule is composed of units known as nucleotide triphosphates. Each nucleotide consists of a triphosphate group, a deoxyribose sugar, and one of four nitrogenous bases: adenine (A), guanine (G), thymine (T), or cytosine (C). DNA exists as a double-stranded molecule arranged in a helical structure, where each base on one strand forms hydrogen bonds with its complementary base on the opposite strand. Adenine pairs exclusively with thymine, and cytosine pairs with guanine.

DNA can be extracted from cells and sequenced to determine the order of its bases. A DNA sequence is typically represented as a string over a four-letter alphabet, where each letter corresponds to one of the four nitrogenous bases.

Human body cells are diploid, meaning that they contain two complete genomes, two copies of all the genetic information. A single haploid genome consists of 23 chromosomes and contains approximately 3.2 billion base pairs (bp). The lengths of individual chromosomes vary substantially. For example, chromosome 1 is approximately 250 million bp long, whereas chromosome 22 contains roughly 50 million bp. Because diploid cells contain two copies of the genome, the total length of DNA in a human body cell is approximately 6.4 billion bp. [GLH11]

In essence, a DNA sequence can be regarded as a very long string over a four-letter alphabet. Within such sequences, various structural patterns can be observed, among which repeats are particularly common. A repeat is defined as a pattern that occurs at least twice consecutively within the DNA sequence. As discussed earlier in Section 1, repeats play an important role in multiple fields, including forensic science [But07, STR24] and biomedical research [Pau18, OZ07].

2.1 Repeats in Forensic Science

The aim of using genetic analysis for forensic casework is to produce a DNA profile that is highly discriminating. This allows biological evidence from the scene of a crime to be matched to an individual with a high level of confidence and can be very powerful forensic evidence. The ability to produce highly discriminating profiles depends on genetic variation between individuals, as no two individuals have identical DNA, not even monozygotic twins. [vvv+25] However, individuals are actually very similar at the genetic level. [GLH11] Humans share around 99.9% of our genetic code with each other. From a forensic perspective, there is little value in analyzing regions of human DNA that are shared among individuals. Fortunately, there are well characterized regions within the genome that are variable between individuals and these have become the focus of forensic genetics. One important category of tandem repeats that is widely used in forensic genetics is short tandem repeats (STRs).

Short Tandem Repeats (STRs) Short tandem repeats (STRs) are currently the most commonly used genetic markers in forensic genetics, as they satisfy the key requirements for forensic identification. STRs consist of short DNA patterns of 1 to 6 bp that are repeated consecutively. Variation between individuals arises from differences in the number of repeat units, typically resulting in total sequence lengths ranging from tens to several hundreds of base pairs. Although a large number of STRs have been identified, only approximately 20 are commonly used in forensic genetics. These commonly used STRs have repeat patterns of 4 or 5 bp and can be classified into four structural categories: simple repeats, simple repeats with non-consensus repeats, compound repeats, and complex repeats. [GLH11]. Before discussing these categories in more detail, two key terms are introduced: *locus* and *allele*. A locus refers to a specific position in the DNA where all individuals share the same type of genetic marker, whereas an allele represents the variant form of that marker present at a given locus in an individual.

- **Simple repeat:** A simple repeat consists of a single repeat pattern that is repeated consecutively without interruption or mixing with other repeat patterns.
- **Simple repeat with non-consensus repeats:** The consensus repeat is the standard and most frequently occurring repeat sequence used as the reference for a given locus. A simple repeat with a non-consensus structure refers to a locus with one type of repeat pattern in which most repeats conform to the consensus sequence, but one or a small number of repeats deviate slightly from it. Figure 2 illustrates an example of such a structure at the TH01 locus. At this locus, `AATG` is the consensus repeat. Allele 7 consists of seven complete `AATG` repeats, whereas allele 9.3 contains nine full `AATG` repeats and one truncated repeat (`ATG`), resulting in a slight deviation from the consensus sequence.
- **Compound repeat:** A compound repeat is composed of multiple different repeat patterns that occur in a fixed order. While the repeat order remains constant, the number of repeats for each pattern may vary between individuals. An example of a compound repeat is shown in Figure 2 for the FGA locus. The alleles at this locus consist of six distinct repeat pattern arranged in a consistent order. For instance, the repeat pattern `CTTT` occurs nine times in allele 17, whereas it occurs twelve times in allele 20.
- **Complex repeat:** A complex repeat consists of a combination of multiple repeat pattern with DNA sequences that are not considered part of the repeat structure. Figure 2 shows an example of a complex repeat at the D21S11 locus. This locus contains a mixture of repeat pattern such as `TCTA`, `TCTG`, and `TA`, which differ in length. In addition, several DNA regions are present that are not included in the repeats and therefore do not contribute to the allele designation.

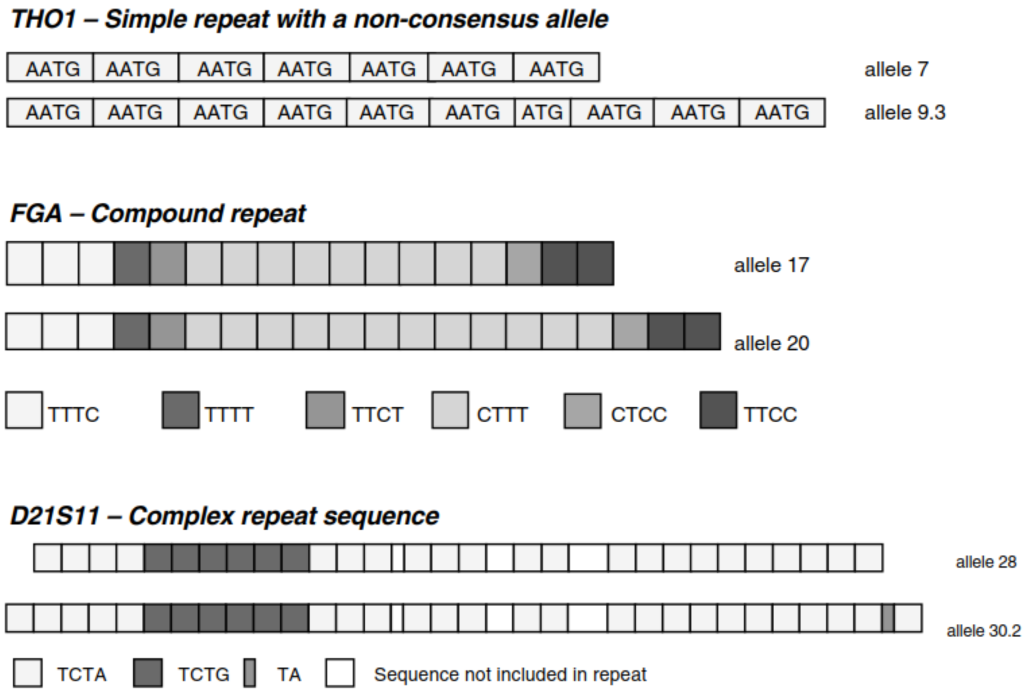


Figure 2: Example of a simple repeat with non-consensus allele, compound repeat, and complex repeat sequence. [GLH11]

2.2 Repeats in Biomedical Science

Repeats are common in the human genome and occur in many different genomic regions. Every individual contains numerous repetitive DNA sequences. In most cases, these repeats have a stable length and do not cause any problems. However, in some regions of the genome, repeats can become unstable and increase in length. When repeats expand far beyond their normal size, they can lead to genetic disorders known as *repeat expansion diseases*. Currently, more than forty hereditary diseases are associated with repeat expansions [Pau18].
Repeat expansion diseases differ in the length of the expansion, the repeated motif, and the genomic location of the repeat. One well-known example is Huntington's disease, which is caused by an expansion of the trinucleotide repeat `CAG`. In healthy individuals, this repeat usually occurs between 6 and 34 times. In contrast, individuals affected by Huntington's disease carry between 36 and 121 consecutive `CAG` repeats [OZ07].
Another example is myotonic dystrophy type 2 (DM2), which is caused by the expansion of a `CCTG` repeat. In unaffected individuals, this repeat typically occurs between 10 and 26 times. In patients with DM2, however, the `CCTG` pattern can be repeated consecutively from about 75 up to more than 10,000 times. [OZ07]

3 Primitive Maximal Repeats

Repeats are important structural features of DNA sequences and can occur in a wide variety of forms and genomic locations. Therefore, it is first necessary to identify all repeats in the sequence before a method for describing these repeats can be defined. In Section 2.1, four classes of repeats were described, which are specific to short tandem repeats (STRs) in forensic genetics. In this thesis, however, we do not distinguish between different types of repeats.

We consider a string S over a finite alphabet Σ. In this research, $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$, as the focus is on DNA sequences. Let $S_{i,j}$ denote the substring of S consisting of the consecutive symbols $S_i S_{i+1} \ldots S_j$, where $0 \leq i \leq j < n$ and $n = |S|$.
A *repeat* is a substring r of S that can be written as $r = u^c$, where u is a non-empty string and $c \geq 2$. The substring u is called the *repeat unit*, and its length $p = |u|$ is referred to as the *period* of the repeat. The integer value c is called the *count* of the repeat and indicates how many times the repeat unit occurs consecutively.
A repeat is *maximal* if it cannot be extended to the left or to the right by a single symbol without increasing its period. [KK99] In addition, a maximal repeat can have a *shift*. The shift is defined as the number of symbols by which the run could be shifted to the right. Equivalently, it corresponds to the length of the longest prefix of the maximal repeat that immediately follows the run. The shift value is always smaller than the period.
Finally, a maximal repeat is called *primitive* if the repeat unit u is primitive, meaning that u itself cannot be written as an integer power of a shorter string. [KK99] Formally, u is primitive if there exists no substring t and integer $c \geq 2$ such that $u = t^c$.
Each Primitive Maximal Repeat (PMR) is described by the annotation:

$$(start, period, count, shift).$$

The value *start* indicates the starting position of the PMR in the string. As an example, consider the string $S = \mathtt{CATCATACATACTACTAAAAA}$. This string contains four PMRs, as listed in Table 1. Consider the PMR r_2 with annotation $(9, 3, 2, 2)$. The repeat unit of this PMR is TAC, giving a period of 3. The PMR starts at position 9 and the repeat unit occurs twice consecutively, yielding a count of 2. The shift value equals 2, meaning that the PMR can be shifted two positions to the right. Consequently, the rotations ACT and CTA are also maximal repeats, starting at positions 10 and 11, respectively.

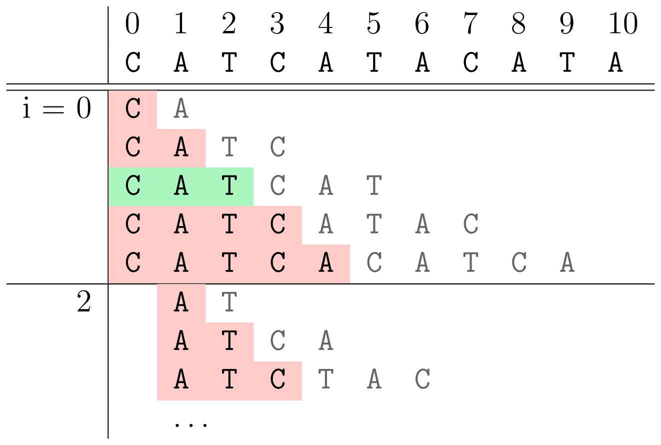
	Repeat unit	Annotation
r_0	CAT	(0,3,2,0)
r_1	CATA	(3,4,2,1)
r_2	TAC	(9,3,2,2)
r_3	A	(16,1,5,0)

Table 1: All primitive maximal repeats (PMRs) of the string $S = \mathtt{CATCATACATACTACTAAAAA}$. The last column provides the annotation of each PMR, consisting of four values: the starting index, the period, the count, and the shift.

To identify the PMRs in a string, one algorithm is implemented. The input to the algorithm is string S, and the output consists of all PMRs present in string S. Each PMR is represented by a data structure containing four fields: index, period, count, and shift.

3.1 Naive Algorithm

To identify all PMRs, we first developed a naive algorithm. This algorithm iterates over every symbol in the string S. For each position, it examines all possible period lengths and verifies how many times the corresponding repeat unit occurs consecutively. Additionally, it checks whether each detected repeat satisfies the requirements of a PMR. An illustrative example is provided in Figure 3. Consider the example $S =$ `CATCATACATA`. For the first symbol, there are five possible period lengths, namely $\{1, 2, 3, 4, 5\}$, since the maximum possible period length at position i is given by $\frac{|S|-i}{2}$. For each possible period, the algorithm verifies whether the associated repeat unit occurs at least twice consecutively. As shown in Figure 3, the first symbol has a period of length 3 whose associated pattern is repeated twice.



	0	1	2	3	4	5	6	7	8	9	10
	C	A	T	C	A	T	A	C	A	T	A
i = 0	🟥 C	A									
	🟥 C	🟥 A	T	C							
	🟩 C	🟩 A	🟩 T	C	A	T					
	🟥 C	🟥 A	🟥 T	🟥 C	A	T	A	C			
	🟥 C	🟥 A	🟥 T	🟥 C	🟥 A	C	A	T	C	A	
2		🟥 A	T								
		🟥 A	🟥 T	C	A						
		🟥 A	🟥 T	🟥 C	T	A	C				
		...									

Figure 3: Example of the string $S =$ `CATCATACATA` illustrating all possible period lengths at position 0 and the first possible period lengths at position 1. Repeat units highlighted in red do not form repeats, whereas the repeat unit highlighted in green forms a repeat. The symbols shown in light gray represent the symbols of the string immediately following each repeat unit.

3.1.1 Computational Complexity

Let $n = |S|$. To determine the computational complexity of the algorithm, we derive a coarse upper bound. Iterating over all symbols in the string S takes $O(n)$ time. For each position, the algorithm iterates over all possible periods, which are bounded by $n/2$ and therefore take $O(n)$ time. For each possible period, the algorithm checks how many times the repeat unit occurs, which is also bounded by $n/2$ and therefore takes $O(n)$ time.
Consequently, the overall computational complexity of this naive algorithm is:

$$O(n \cdot n \cdot n) = O(n^3),$$

as a function of the input size n.

3.2 Linear Time Algorithm

As discussed earlier in Section 2, DNA sequences can be extremely long. An algorithm with a time complexity of $O(n^3)$ with respect to the input size n makes the naive approach impractical for large input sizes. To efficiently analyze large DNA sequences, an algorithm with linear time complexity is preferable.
In 1999, Kolpakov and Kucherov introduced a linear time algorithm capable of computing all PMRs in a string [KK99]. This section provides a detailed description of how all PMRs in a string can be computed using this algorithm. For clarity, we divide the algorithm into four phases.

S-factorization. To find all PMRs using the linear algorithm, the input string S must first be decomposed using s-factorization, also called Lempel-Ziv factorization and LZ77 factorization. Let S be a string, then the s-factorization of S is:

$$S = u_1 u_2 \dots u_k,$$

where the factors u_i are defined as follows:

- If the next symbol directly after $u_1 u_2 \dots u_{i-1}$ does not occur in $u_1 u_2 \dots u_{i-1}$, then we define $u_i = \mathit{symbol}$. In other words, whenever a completly new symbol appears in the string, this symbol forms a new factor.
- Otherwise, u_i is the longest prefix of $u_i \dots u_k$ that has another occurence to the left in S.

This s-factorization plays a central role in efficiently locating PMRs. This s-factorization can be computed in lineair time using suffix trees.

3.2.1 Phase 1 — Constructing Suffix Tree

A suffix tree is a tree that contains all suffixes of a string S as paths from the root to the leaves. Figure 4 illustrates an example of a suffix tree for $S =$ `xabxa$`. The structure of a suffix tree is determined by the following key properties [Gus97b]:

- The root node may have zero or more children.
- Each internal node has at least two children.
- Every edge is labeled with a non-empty substring of S.
- No two edges leaving the same node may begin with the same character.

Constructing a suffix tree using a naive approach requires $O(n^2)$ time for a string S of length n [Gus97b]. Therefore, a linear time algorithm is used to construct the suffix tree. There are several algorithm that construct a suffix tree in lineair time. However, in this work we focus on Ukkonen's algorithm. Ukkonen's algorithm offers a space-saving improvement over Weiner's algorithm [Gus97b]. Hence, Ukkonen's algorithm is the method of choice for most problems requiring the construction of a suffix tree [Gus97a]. In the following, we explain how this algorithm works [Gus97b, Ukk95].

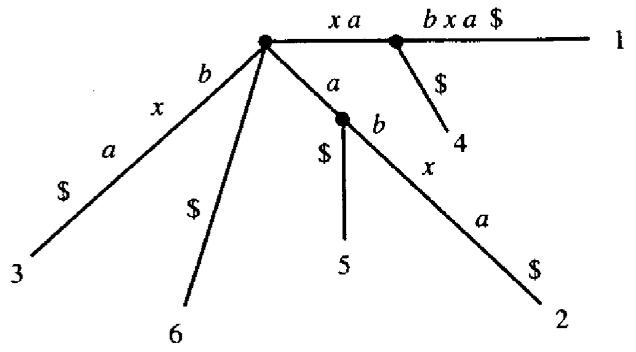


Figure 4: Suffix tree for the string $S =$ `xabxa$`. The labels on the edges are indicated by substrings of S, and the numbers on the leaves indicate the corresponding suffixes.[Gus97b]

This algorithm incrementally builds an implicit suffix tree for every prefix of S. For example, implicit suffix tree T_{j+1} is constructed by extending T_j with the new character $S[j+1]$. An implicit suffix tree differs from an (explicit) suffix tree. For example, not all suffixes necessarily end in leaf nodes. Some suffixes may end in the middle of an edge. In addition, the tree does not contain the unique termination symbol that forces every suffix to be represented explicitly. The final suffix tree is obtained from T_n (for $n = |S|$) by appending a termination symbol to ensure that all suffixes end in leaves.
The construction process consists of n phases, one for each prefix $S[1 \dots j]$. Within each phase, every suffix $S[i \dots j]$ of the current prefix is inserted into the implicit suffix tree according to one of the following rules:

- **Rule 1:** If the path from the root labbelled $S[i \dots j-1]$ ends at the leaf edge and $S[j-1]$ is the last character on the leaf edge. Then character $S[j]$ is just added to the end of the label on that lead edge.
- **Rule 2a:** Add new leaf edge if there is no path from the root that begins with $S[j]$.
- **Rule 2b:** If there is a path from the root that matches $S[i \dots j-1]$, but $S[j]$ does not match after it and the mismatch occurs in the middle of an edge, then the edge is split at the mismatch position and a new internal node is created. The first child of the new internal node is the remaining part of the original edge after the mismatch position, and the second child is a new leaf edge labeled with $S[j]$.
- **Rule 3:** If there is a path from the root that matches $S[i \dots j]$, then do nothing.

For each of the n phases, all suffixes $S[i \dots j]$ of the current prefix $S[1 \dots j]$ must be extended according to one of the three extension rules described above. For every suffix, we must first locate the endpoint of the path labeled $S[i \dots j-1]$ in the current implicit suffix tree. A naive implementation would start this traversal from the root for every suffix and match characters one by one. This will take $O(n^3)$ time to build the suffix tree. To achieve linear time construction, Ukkonen's algorithm introduces several key optimizations to avoid repeated traversal from the root. These optimizations are described below.

Active Point To efficiently determine the endpoint of the path $S[i \dots j]$ in the current implicit suffix tree, Ukkonen's algorithm maintains an active point. The activepoint stores the exact location where the previous suffix or phases ended and consists of three components: `activeNode`, `activeEdge`, and `activeLength`. Here, `activeNode` denotes the node from which traversal begins, `activeEdge` identifies the outgoing edge from `activeNode`, and `activeLength` indicates how many symbols along that edge have already been matched. The key idea is that the location where phase j or suffix i ends is precisely the location where phase $j+1$ or suffix $i+1$ must begin.

Suffix Links A second mechanism used to accelerate traversal is the suffix link. A suffix link is a pointer from one internal node to another. Let the path label of an internal node v be xA, where x is a single symbol and A is a (possibly empty) substring of S. If there exists another internal node w whose path label is exactly A, the algorithm creates a suffix link from v to w. If A is the empty string, the suffix link points to the root.

Skip/Count Trick The third optimisation is the skip/count trick, which speeds up downward traversal over long edge labels. Suppose we must descend y symbols starting from a node w. If the current edge contains fewer than y symbols, the entire edge can be skipped in one step. If the edge contains more than y symbols, we can jump directly to the appropriate symbol position on that edge without inspecting the symbols one by one. With this trick, the traversal time depends on the number of edges, rather than the number of symbols on those edges. This trick works because whenever a node v has a suffix link to w, any path labelled with a string y reachable from node v is also reachable from node w.

By combining the active point, suffix links, and the skip/count trick, $S[i \dots j-1]$ can be located much more efficiently. The active point ensures that each new search starts exactly where the previous one ended, avoiding repeated traversal from the root. From this position, the algorithm moves up to the parent node, and from there it follows the suffix link to the node. Starting from that node, the skip/count trick allows the algorithm to walk down the required path in only a few steps, skipping entire edges rather than comparing symbols one by one. Together, these techniques drastically reduce the amount of work per extension.

Show Stopper This optimisation concerns the behaviour of rule 3. rule 3 applies when the substring $S[i \dots j]$ is already present in the current implicit suffix tree, in which case no further action is required. A key observation is that once rule 3 applies for a particular suffix $S[i \dots j]$ during phase j, it will also apply for all remaining suffixes in that phase $S[i+1 \dots j], S[i+2 \dots j], \dots, S[j \dots j]$. This means that no further modifications to the tree are needed. As a result, the algorithm can immediately terminate the current phase as soon as rule 3 is triggered. This show stopper behaviour significantly reduces the number of suffixes that must be processed.

Once a leaf, always a leaf If phase j and suffix i create a leaf, this leaf will remain a leaf in all later phases. In a later phases, such a leaf is simply extended with the newly read symbol, which corresponds to rule 1. Because a leaf never becomes an internal node, it will never be revisited for splitting. However, in a naive implementation, extending a leaf in each phase would require explicitly updating the end index in the edge label of every leaf created so far. Since rule 1 may apply to many leaves in every phase, repeatedly updating all of their end indices would be expensive. To avoid this, Ukkonen introduced a global variable. Instead of storing a fixed end index in each leaf edge, all leaves simply store the global variable. Incrementing the global variable in each phase automatically updates the end index of every leaf edge in the tree. This mechanism allows all leaves affected by rule 1 to be extended in constant time, without the need to update each leaf individually.

Edge-Label Compression If every edge in the suffix tree stored its full substring explicitly, the total memory usage would be $O(n^2)$, since many substrings appear repeatedly along different paths. To avoid this, the tree stores only the start and end indices of each edge label. This compression reduces the total space requirement from $O(n^2)$ to $O(n)$.

All these implementation techniques together form Ukkonen's algorithm, which enables the construction of a suffix tree for a string in linear time.

3.2.2 Phase 2 — Computing the S-Factorization Using a Suffix Tree

To compute the s-factorization from a suffix tree, the Lempel-Ziv (LZ) algorithm can be used.[Smy03] Before the factorization can be computed, the suffix tree must satisfy the following property. Each internal node v is labeled with the smallest starting position among all suffixes represented by the leaves in the subtree rooted at v. We denote this value by $\mathit{Label}(v)$. The root node is assigned the label 0.

Each factor u_i in the s-factorization is represented by a pair (i_L, ℓ), where i_L denotes the starting position of the leftmost occurrence of u_i in $u_1 \cdots u_{i-1}$, and $\ell = |u_i|$. Let i_0 denote the first position in S immediately following the prefix $u_1 \cdots u_{i-1}$.
For each position i_0, the algorithm searches the suffix tree for the longest prefix of the suffix $S[i_0 \ldots n]$, this will always lead to the leaf with $\mathit{Label} = i_0$. However, the suffix is not the factor. Instead, the factor may only correspond to a substring that already occurs in $u_1 \cdots u_{i-1}$. Therefore, the search proceeds down the path only as long as the current node v satisfies $\mathit{Label}(v) < i_0$. As soon as this condition is violated, the search is terminated. The last node v for which $\mathit{Label}(v) < i_0$ holds determines the factor. We set $i_L = \mathit{Label}(v)$ and ℓ equal to the length of the substring represented by v. If the last node v is the root, then a new symbol has been identified, and we set $(i_L, \ell) = (i_0, 0)$.

If the suffix tree is implemented with the Ukkonen's algorithm, then the s-factorization can be computed at the same time as the suffix tree is created.[Smy03]

3.2.3 Phase 3 — Finding Leftmost PMRs

This s-factorization of the string can now be used to find PMRs. The usefullness of the s-factorization can be explained by dividing the repeats in two classes.

- Type 1 repeat: repeat r such that $\mathit{start}(r) \leq \mathit{start}(u_i)$ and $\mathit{start}(u_i) \leq \mathit{end}(r) < \mathit{end}(u_i)$ for some s-factor u_i.

- Type 2 repeat: repeat r such that $start(u_i) < start(r) < end(r) < end(u_i)$. So, this type of repeat lies completely in u_i

Because a Type 2 repeat lies entirely within u_i and u_i has another occurrence to the left in the string, Type 2 repeats also have another occurrence to the left in the string. Consequently, finding all Type 1 runs guarantees finding all leftmost occurrences of distinct PMRs.
To find all Type 2 repeats, assume we are given the S-factorization $S = u_1u_2 \dots u_k$. For each factor u_i with $2 \leq i \leq k$, we consider the substring t_iu_i. Here, $t_i = t_i[1 \dots m]$ denotes the suffix of $u_1 \dots u_{i-1}$ of length $|u_{i-1}| + 2|u_i|$, and $u_i = u_i[1 \dots n]$. Consequently, we define $v = t_iu_i = v[1 \dots m + n]$. For each possible period p, there exists at most one PMR of period p in v that starts in t_i, ends in u_i, and whose periodicity is visible within u_i.
PMRs that span t_iu_i may have their period expressed either within t_i or within u_i. For algorithmic purposes, however, the detection of such PMRs is divided into two symmetric cases: the algorithm searches for PMRs by considering the periodic structure in either t_i or u_i, but never both simultaneously. In this section, we focus on PMRs whose period lies in u_i; PMRs whose period lies in t_i can be found symmetrically.
These PMRs can be found with the following arrays:

- **LongestPrefix** ($LP[p]$), for $2 \leq p \leq n$: the length of the longest prefix of u_i that is also a prefix of $u_i[p..n]$. For example, let $t_i =$ `TCTATATA` and $u_i =$ `TATC`. Then $LP[3] = |b| = 1$, since the longest prefix of $u_i =$ `babc` is also a prefix of $u_i[3..n] =$ `TC` has length 1.

- **LongestSuffix** ($LS[p]$), for $1 \leq p \leq n$: the length of the longest suffix of t_i that is also a suffix of $v[1..m + p] = t_iu_i[1..p]$. For example, let $t_i =$ `TCTATATA` and $u_i =$ `TATC`. Then $LS[2] = |baba| = 4$, since the substring $v[1..m + 2] =$ `TCTATATATA` ends with `baba`, which is the longest suffix of $t_i =$ `TCTATATA` occurring as a suffix of $v[1..m + 2]$.

Note by Theorem 6 of [KK99], for $1 \leq p \leq n$, there is at most one PMR of period p starting in t_i, ending in u_i, and having a period in u_i iff $LS[p] + LP[p + 1] \geq p$. If the equality holds, this repeat is $t_iu_i[m - LS[p] + 1 \dots m + p + LP[p + 1]]$.

Following [Mai89], the two arrays LP and LS can be computed in linear time using a variation of the Knuth–Morris–Pratt algorithm. In [ML84], it is shown how the arrays *lppattern* and *lptext* can be computed. The value `lppattern[i]` denotes the length of the longest substring of the string *pattern* that begins at position i and is a prefix of *pattern*. Thus, by taking *pattern* $= u_i$, `lppattern[i]` corresponds to $LP[i]$.
Similarly, `lptext[i]` denotes the length of the longest substring of the string *text* that begins at position i and is a prefix of the string *pattern*. In this case, if both strings are reversed and *text* corresponds to v while *pattern* corresponds to t_i, then LS can be computed using the algorithm for calculating `lptext[i]`.

Constructing the LP Array To obtain the values of $LP[p]$ for $2 \leq p \leq n$, we construct an array $LP[1 \dots n]$. This can be achieved using Algorithm 1 introduced in [ML84] for computing `lppattern[i]`. This algorithm reuses previously computed values to avoid unnecessary symbol comparisons.
Suppose we want to compute $LP[p]$ and have already calculated $LP[2] \dots LP[p - 1]$. Suppose we have remembered the value of k in range $2 \leq k < p$ which maximaizes the sum $k + LP[k]$. If

the overlap at position k extends beyond position p (that is, if $p < k + LP(k)$), then position p lies inside the region where a prefix match is already known to hold. In that case, the value of $LP[p-k+1]$ gives some information on what $LP[p]$ is. This holds because, if p lies inside the region this holds $u_i[p \dots k + LP[k] - 1] = u_i[p - k + 1 \dots LP[k]] = x$. Therefore, if $LP[p - k + 1] < |x|$, then $LP[p] = LP[p - k + 1]$. Otherwise, if $LP[p - k + 1] \geq |x|$, then $LP[p] \geq |x|$. In this case, additional symbols are compared until a mismatch occurs. As a result, the entire LP array can be constructed in linear time.
For example, let $u_i =$ `CACACCAC` and consider the computation of $LP[5]$. Table 2 shows the LP array for the positions that have already been computed. Let $k = 3$. Then $k+LP[k] = 3+LP[3] = 6$, which implies that the condition $p < k + LP[k]$ holds. Therefore, the value $LP[p-k+1] = LP[3]$ provides information about $LP[5]$. Let $x = u_i[p \dots k + LP[k] - 1] = u_i[5 \dots 5] =$ `C`. Since $LP[p-k+1] \geq |x|$, it follows that $LP[5] \geq 1$.

	1	2	3	4	5	6	7	8
u_i	C	A	C	A	C	C	A	C
$LP[p]$	-	0	3	0	?			

Table 2: Example of the string $S =$ `CACACCAC` with the LP array, showing the values that have already been computed.

Constructing the LS Array To obtain the values of $LS[p]$ for $1 \leq p \leq n$, we construct an array $LS[1..n]$. This can be achieved using Algorithm 2 introduced in [ML84] for computing `lptext[i]`. This algorithm is structurally identical to the algorithm for computing `lppattern[i]`. The only difference is that Algorithm 2 matches the string *pattern* against the string *text* rather than against itself.

3.2.4 Phase 4 — Finding Remaining PMRs

After all Type 1 maximal repetitions have been identified, the remaining Type 2 maximal repetitions must be found. As discussed earlier, every Type 2 maximal repetition lies entirely within some s-factor u_i, and each u_i has an earlier occurrence in the string S. Consequently, previously identified Type 1 repetitions can be used to derive the Type 2 repetitions. Before this can be done, all Type 1 repetitions are sorted.
First, all Type 1 maximal repetitions are sorted by their end position. To achieve this, the repetitions are distributed into $n = |S|$ lists according to their end positions, such that list j contains all repetitions that end at position j. As a result, all repetitions are ordered by increasing end position. Next, the repetitions are traversed in this order and sorted again into n lists using bucket sort according to their starting positions. After this second sorting step, repetitions with the same starting position are stored in the same list and are ordered by increasing end position.
Let v_i denote the earlier occurrence of u_i, and let $\Delta_i = \mathit{start}(u_i) - \mathit{start}(v_i)$.
For each s-factor u_i with $1 \leq i \leq k$ and for each position j, we consider all PMRs that start at position $j - \Delta_i$ in v_i and end within v_i. Each such PMR is shifted by Δ_i to the right to obtain a corresponding PMR in u_i. This is done by iterating over the list containing PMRs that start at position $j - \Delta_i$ and selecting those whose end positions lie within v_i. Each selected PMR is then

shifted by Δ_i and inserted at the head of the list corresponding to position j. This insertion order is valid because all existing PMRs in list j end inside u_i and therefore have larger end positions than the newly inserted repetitions.

In conclusion, the algorithm introduced by Kolpakov and Kucherov [KK99] is a linear time algorithm that can be used to compute all PMRs in a string. This algorithm achieves linear time complexity by combining s-factorization, suffix trees, and variants of the Knuth–Morris–Pratt algorithm. In this research, the algorithm of Kolpakov and Kucherov is not implemented. However, this remains an interesting direction for future work.
A limitation of the current linear time approach is that it is based on suffix trees. Suffix trees require a significant amount of memory to store the complete tree structure. While this is not problematic for short strings, the associated memory consumption becomes a limiting factor for long DNA sequences.
A possible way to address this issue is to replace suffix trees with suffix arrays, which are considerably more space-efficient [CIS08]. Using suffix arrays would require changes to the algorithm. In particular, the suffix tree construction in the first phase would need to be replaced, and the computation of the s-factorization would have to be adapted to work with a suffix array.
How these changes can be implemented, or whether alternative data structures or approaches may offer better performances. Is not explored furher in this work and is left for further research.

4 Maximal Cover

After all PMRs have been identified in the string S, we want to describe the string using its repeats. We therefore introduce the *maximal cover* approach, which selects PMRs such that they cover the largest possible number of symbols in the string. A *cover* is a selection of PMRs whose occurrences cover positions in S. Each cover corresponds to one specific combination of selected PMRs. We define the *cover size* as the total number of symbols in S covered by a cover. A *maximal cover* is a cover that maximizes the cover size.

All detected PMRs may be used for the cover, but several rules must be followed. PMRs in the cover may not overlap, although they may be placed directly next to each other. Each PMR may be used only once in the same cover. A PMR does not have to be used in its full length. For every PMR, a shorter run u^c with $c > 1$ is allowed. It is also possible to use shifted versions of a PMR, as long as the shift remains within the PMRs allowed shift range. A *run* of a PMR is a valid occurrence of that PMR, obtained by choosing an allowed shift and count $c > 1$. Each run represents a possible way in which the PMR can be included in a cover. c is an integer, which implies that runs consist exclusively of whole periods.

For example, Figure 5 shows the PMR with annotation $a = (0, 3, 3, 2)$. The figure illustrates all possible runs of this PMR that can be used in a cover. Runs 1, 4, and 7 correspond to the full run, while the remaining runs use only two periods instead of three. Runs 4–9 represent shifted variants of the PMR, with runs 4–6 shifted one symbol to the right and runs 7–9 shifted two symbols to the right.

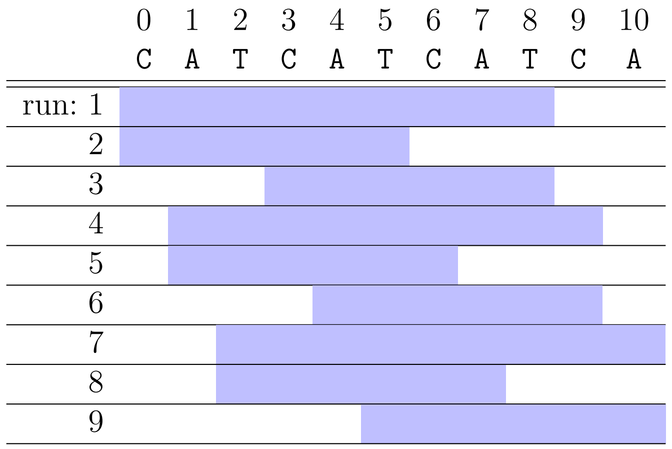


Figure 5: Illustration of a PMR with annotation $(0, 3, 3, 2)$ showing the nine possible runs that can be used in a cover.

Covers can be represented as descriptions. In these descriptions, the number in square brackets indicates the number of repetitions of the preceding repeat unit, and semicolons are used to separate runs and uncovered symbols. This notation differs from others, such as the dbSNP notation, which does not use separators and represents uncovered symbols by appending `[1]` to the end of the region. Throughout this research, we adopt a single, consistent notation, as it makes comparisons between different descriptions easier and clearly distinguishes covered regions from uncovered symbols. The number in round brackets at the end of a description indicates the cover size of the corresponding

cover.
For example, consider the string $S =$ `CATCATCATCA` shown in Figure 5. If we have the cover that includes run 1, then the corresponding description is `CAT[3];CA` (9).

4.1 Brute-Force Algorithm

As an initial approach, a brute-force algorithm is implemented to compute the maximal cover. The algorithm first determines all possible runs of each PMR. Subsequently, it iterates over all possible covers and computes the cover size. For each new cover, it checks whether the cover size is larger than the best result found so far.
Consider the example $S =$ `CATCATACATACTACTA`. Figure 6 illustrates all covers evaluated by the algorithm. The rightmost column indicates the cover size. As shown, cover 5 yields the largest cover size and is therefore selected as the maximal cover. If multiple covers result in the same largest cover size, all such covers are returned as maximal cover.

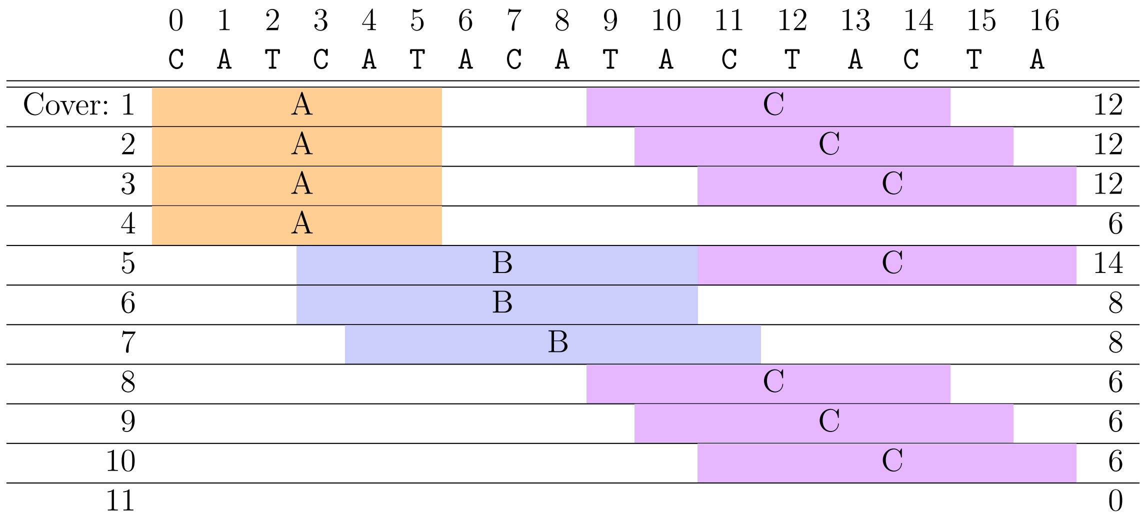


Figure 6: Example of the string $S =$ `CATCATACATACTACTA` containing three PMRs: $\mathrm{PMR_A} = (0, 3, 2, 0)$, $\mathrm{PMR_B} = (3, 4, 2, 1)$, and $\mathrm{PMR_C} = (9, 3, 2, 2)$. The figure illustrates all covers evaluated by the brute-force algorithm. The rightmost column indicates the cover size.

4.1.1 Computational Complexity

Let $n = |S|$. To determine the computational complexity of the algorithm, we derive a coarse upper bound. The algorithm first computes all possible runs induced by the PMRs. It iterates over all PMRs in the string. Since the maximal number of PMRs in a string S of length n is linearly bounded in n [KK99], this step takes $O(n)$ time.
For each PMR, the algorithm enumerates all possible runs that can be used in a cover. This is done using a nested loop over the repetition count. Since the repetition count is bounded by $n/2$, this takes $O(n^2)$ time. For each segment, the algorithm iterates over all shifts of the PMR. The shift is also bounded by $n/2$ and therefore this takes $O(n)$ time. Consequently, the generation of all

possible runs has a computational complexity of:

$$O(n \cdot n^2 \cdot n) = O(n^4)$$

as a function of the input size n.

In the second phase, the algorithm computes the cover size of every possible cover by considering all combinations of runs. In the worst case, each PMR induces $O(n^3)$ candidate runs. Since a cover can contain at most one run per PMR, calculating the cover size of every cover has a computational complexity of:

$$\prod_{i=1}^{n} O(n^3) = (O(n^3))^n = O\big((n^3)^n\big).$$

as a function of the input size n.

Consequently, the overall time complexity of the brute-force algorithm is:

$$O\big(O(n^4) + O\big((n^3)^n\big)\big) = O\big((n^3)^n\big)$$

as a function of the input size n.

4.2 Greedy Algorithm

The maximal cover problem considered in this research shares several similarities with the activity-selection problem.[CLRS09] In the activity-selection problem, a set of activities is given, each with a start time and a finish time, and the goal is to select a maximum number of non-overlapping activities.
Similarly, in our problem the input consists of PMRs, each described by an annotation $(start, period, count, shift)$. From each PMR we derive a set of *runs*, where each run can be viewed as an interval in string S with a start position and end position. As in the activity-selection problem, overlapping runs are not permitted. Moreover, both problems aim to maximize a specific value. In the activity-selection problem, the goal is to maximize the number of activities, whereas in our problem the goal is to maximize the cover size.
An optimal solution to the activity-selection problem is obtained by a greedy algorithm that repeatedly selects the activity with the earliest finishing time. Assuming the activities are initially sorted by their finishing times, the algorithm runs in linear time.[CLRS09]

4.2.1 Greedy Algorithm Based on End Position

Based on the similarities with the activity-selection problem, we develop a greedy algorithm that constructs a cover by selecting the run with the smallest end position. As in the brute-force algorithm, the greedy algorithm first determines all runs of each PMR. Once all runs have been identified, they are sorted in increasingly order of their end positions.
After sorting, the greedy algorithm constructs a cover by repeatedly selecting the run with the smallest end position that does not overlap with the already selected runs. Figure 7 illustrates the construction of the cover for the string $S =$ `CATCATACATACTACTA`. As shown earlier, this example contains three PMRs. $\text{PMR}_\text{A} = (0, 3, 2, 0)$, $\text{PMR}_\text{B} = (3, 4, 2, 1)$, and $\text{PMR}_\text{C} = (9, 3, 2, 2)$.

As shown in Figure 7, the algorithm first selects a run derived from PMR_A, followed by a run derived from PMR_C. In this case, the unshifted version is selected, as it yield the smallest end positions.

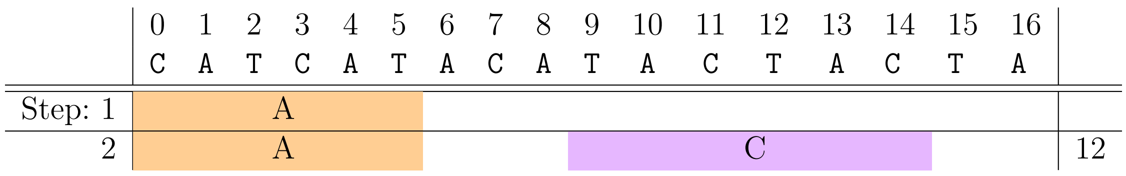


Figure 7: Example of the string S = `CATCATACATACTACTA` containing three PMRs: $\mathrm{PMR}_\mathrm{A} = (0, 3, 2, 0)$, $\mathrm{PMR}_\mathrm{B} = (3, 4, 2, 1)$, and $\mathrm{PMR}_\mathrm{C} = (9, 3, 2, 2)$. The figure illustrates how the greedy algorithm based on end position constructs the cover for the string S.

Greedy algorithms make locally optimal choices at each step in the hope of obtaining a globally optimal solution. However, greedy algorithms do not always yield optimal results. As illustrated in Figure 7, the cover constructed by the greedy algorithm has a cover size of 12, whereas Figure 6 shows that the maximal cover for this string has a cover size of 14. This demonstrates that the greedy algorithm does not obtain a globally optimal solution in this case.
To evaluate how often this greedy algorithm produces an optimal solution, we computed the cover size for 612 strings of length 10 and compared the results to the maximal cover obtained by the brute-force algorithm. The string considered of non-isomorphic binary words, Fibonacci words, and Lyndon words, which are described in more detail in Section 5.1 Out of the 612 strings considered, the greedy algorithm does not obtain a maximal cover for 532 strings. In these cases, the cover size is on average 2.98 smaller than that of the maximal cover. For instance, consider the string S = `AAAAAAAAAA`. The brute-force covers this string as `A[10]`, whereas the greedy algorithm produces the cover `A[2];AAAAAAAA`. This difference arises because the greedy algorithm always selects the run with the smallest end position first. Once this run is selected, the remaining symbols cannot be covered, as no additional runs are available: the only PMR has already been used.

String S:	`AAAAAAAAAA`
Greedy end position cover:	`A[2];AAAAAAAA` (2)
Maximal cover:	`A[10]` (10)

4.2.2 Greedy Algorithm Based on Length

To explore whether this can be improved, we introduce a second greedy algorithm. This algorithm has the same structure as the previous greedy algorithm, but differs in the selection criterion. Instead of selecting the run with the smallest end position, this algorithm selects always the run with the largest covered length. As before, the algorithm first determines all runs of each PMR. These runs are then sorted in decreasing order of their length, and the cover is constructed by selecting the longest run at each step.
Figure 8 illustrates the construction of the cover for the string S = `CATCATACATACTACTA`. As shown, a run derived from $\mathrm{PMR}_\mathrm{B} = (3, 4, 2, 1)$ is selected first, followed by a run derived from

$\text{PMR}_\text{C} = (9, 3, 2, 2)$. In this case, the algorithm selects a shifted version of PMR_C, as the other possible runs overlap with the previously selected run.

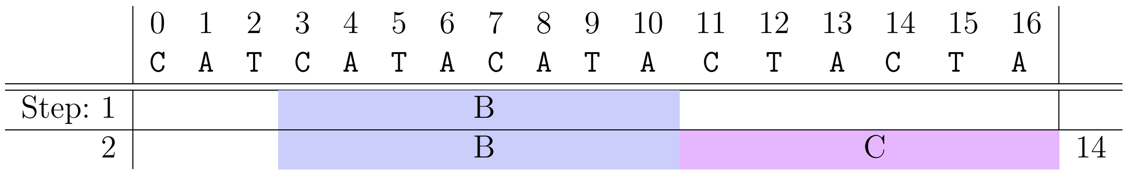


Figure 8: Example of the string S = `CATCATACATACTACTA` containing three PMRs: $\text{PMR}_\text{A} = (0, 3, 2, 0)$, $\text{PMR}_\text{B} = (4, 4, 2, 1)$, and $\text{PMR}_\text{C} = (9, 3, 2, 2)$. The figure illustrates how the greedy algorithm based on run length constructs the cover for the string S.

As shown in Figure 8, the greedy algorithm based on run length obtains the maximal cover for this example. To evaluate how often this greedy algorithm produces an optimal solution, we again computed the cover size for the same 612 strings and compared the results with the maximal covers obtained by the brute-force algorithm.
Out of the 612 strings considered, the greedy algorithm fails to obtain a maximal cover for 140 strings. In these cases, the resulting cover size is on average 1.56 smaller than that of the maximal cover. For example, consider the string S = `AACAACAACC`. The brute-force algorithm covers this string as `A[2];CAA[2];C[2]`, whereas the greedy algorithm produces the cover `AAC[3];C`. This difference arises because the greedy algorithm prioritizes the run with the largest covered length. Once this run is selected, only a single symbol remains uncovered, which cannot be covered by any additional run.

String S:	`AACAACAACC`
Greedy length cover:	`AAC[3];C` (9)
Maximal cover:	`A[2];CAA[2];C[2]` (10)

4.2.3 Computational Complexity

Let $n = |S|$. To determine the computational complexity of the algorithm, we derive a coarse upper bound. The algorithm first computes all possible runs induced by the PMRs. It iterates over all PMRs in the string. Since the maximal number of PMRs in a string S of length n is linearly bounded in n [KK99], this step takes $O(n)$ time.
For each PMR, the algorithm enumerates all possible segments of the repeat that can be used in a cover. This is done using a nested loop over the repetition count. Since the repetition count is bounded by $n/2$, this takes $O(n^2)$ time. For each segment, the algorithm iterates over all shifts of the PMR. The shift is also bounded by $n/2$ and therefore this takes $O(n)$ time. Consequently, the generation of all possible runs has a computational complexity of:

$$O(n \cdot n^2 \cdot n) = O(n^4),$$

as a function of the input size n.

To apply the greedy selection, the 2D structure of runs is flattened into a 1D vector in $O(n^4)$ time. The runs are then sorted, which has worst-case time complexity $O(n^4 \log n^4)$[sor25]. After sorting, the algorithm performs a iteration over the run vector, this step takes $O(n^4)$ time. Finally, the resulting cover produced by the greedy algorithm is sorted once more, which incurs a worst-case time complexity of $O(n^4 \log n^4)$. This additional sorting step is necessary because a run with a larger run length may also have a larger end position and therefore appear earlier in the cover than a run with both a smaller length and a smaller end position.
Consequently, the overall time complexity of the greedy algorithm based on end position is:

$$O\big(n^3 + n^4 + n^4 \log n^4 + n^4\big) = O(n^4 \log n^4).$$

The overall time complexity of the greedy algorithm based on end position is:

$$O\big(n^3 + n^4 + n^4 \log n^4 + n^4 + n \log n\big) = O(n^4 \log n^4).$$

as a function of the input size n.

4.3 Dynamic Programming Algorithm

The brute-force algorithm runs in $O((n^3)^n)$ time and therefore becomes impractical for long DNA sequences. Moreover, the greedy algorithm does not always produce an optimal solution. To address these limitations, we introduce a dynamic programming algorithm.
The dynamic programming algorithm computes the maximal cover for every prefix of the string S. Formally, for each position i in S, the algorithm computes the maximal cover for the prefix $S_{0,i}$. At each position i, the algorithm considers whether including the symbol $S[i]$ in the cover leads to a larger cover size than excluding it. If $S[i]$ is excluded, the cover size equals the maximal cover size computed for the prefix $S_{0,i-1}$. If $S[i]$ is included, the algorithm determines which runs end at position i. These runs are referred to as *valid runs*. This determination is based on the values ρ and σ, defined as:

$$\rho = (i - \mathit{start} + 1) \bmod \mathit{period},$$
$$\sigma = \frac{i - \mathit{start} + 1}{\mathit{period}}.$$

Here, ρ denotes the offset within the period at position i, and σ represents the number of full periods of the PMR that have been completed at position i. For example, consider the string $S =$ `CATCATCATCA` with annotation $(0, 3, 3, 2)$. Let $i = 8$. Then $\sigma = \frac{8-1+1}{3} = 3$ which means that three full periods are completed at position 8. Since a run must consist of at least two periods, there are $\sigma - 1 = 2$ valid runs at position 8. These runs are marked blue in Figure 9.
The length of a valid run at position i is denoted by ℓ_c, where c is the count of the run and satisfies $2 \leq c \leq \sigma$. For example, run 1 has length ℓ_3, while the run 3 has length ℓ_2.
To compute the maximal cover at position i, the algorithm evaluates the cover size obtained by including each valid run at position i. For a valid run of length ℓ_c, the corresponding cover size is given by:

$$\mathit{Size}[i - \ell_c] + \ell_c.$$

Here, $Size$ is an array that stores the maximal cover for each prefix of the string, where $Size[i]$ denotes the maximal cover size of the prefix $S_{0,i}$.
Among all valid runs ending at position i, the algorithm selects the run that yields the largest cover size. To determine the maximal cover at position i, the algorithm then takes the maximum between excluding the symbol $S[i]$ and including the best valid run:

$$Size[i] = \max\bigl(Size[i-1],\ Size[i-\ell_c] + \ell_c\bigr),$$

where ℓ_c is the length of the valid run with the highest cover size.
For example, consider again the previous example. The two valid runs marked in blue yield the following cover sizes: run 1 gives $Size[8-9]+9 = Size[-1]+9 = 9$, and run 3 gives $Size[8-6]+6 = Size[2]+6 = 6$. Thus, run 1 yields the larger cover size. The maximal cover at position 8 is therefore computed as:

$$Size[8] = \max\bigl(Size[7],\ Size[8-9]+9\bigr) = \max(6,\ 9) = 9.$$

Once the maximal cover has been computed for each position i in the string S, the value $Size[n-1]$, where $n = |S|$, represents the maximal cover of the entire string.

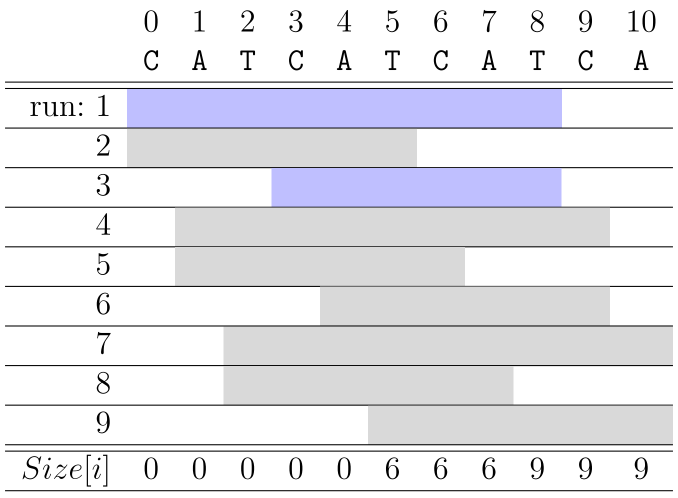


Figure 9: Illustration of a PMR with annotation $(0, 3, 3, 2)$ showing the nine possible runs that can be used in a cover. The runs marked in blue are the valid runs for $i = 8$.

Constructing a Cover To reconstruct a cover that leads to the maximal cover, the algorithm iterates over the $Size$ array from the last index to the first. Let i denote the current position. If there exists a run of length ℓ_c ending at position i such that:

$$\text{Condition 1:}\ \ Size[i] = Size[i-\ell_c] + \ell_c,$$

then this run is included in the cover, and the algorithm continues at position $i - \ell_c$. The algorithm always considers runs in decreasing order of c, thereby preferring longer runs. If Condition 1 does not hold, the value of c is decreased by one, and Condition 1 is evaluated again using the new value of ℓ_c. If no valid run satisfies Condition 1, then:

$$\text{Condition 2: } Size[i] = Size[i-1],$$

must hold, and the algorithm moves one position backward to $i-1$. If $Size[i] = 0$, the algorithm terminates, since no valid runs remain in the prefix $S[0 \ldots i]$.

As an example, consider the string $S =$ `AGAAAGAAAGAAAGAGA`, which contains two PMRs with annotations $(0, 4, 3, 3)$ and $(12, 2, 2, 1)$. Figure 10 shows the corresponding $Size$ array for this string. To reconstruct the cover, the algorithm starts at position $i = 16$. At this position, there exists a valid run derived from the PMR with annotation $(12, 2, 2, 1)$. This run has length $\ell_2 = 4$ and satisfies the Condition $Size[16] = Size[12] + 4$. The algorithm therefore moves to position $i = 12$. At $i = 12$, a valid run derived from the PMR with annotation $(0, 4, 3, 3)$ exists. This run has length $\ell_3 = 12$ and satisfies $Size[12] = Size[0] + 12$. The algorithm then moves to $i = 0$, where it terminates since $Size[0] = 0$. Figure 10 also illustrates the resulting cover.
This cover construction always selects the longest valid run first. An interesting direction for future research would be to explore alternative strategies, such as prioritizing shorter runs or reconstructing the cover by traversing the $Size$ array from left to right.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	G	A	A	A	G	A	A	A	G	A	A	A	G	A	G	A
Size[i]	0	0	0	2	3	3	3	8	8	8	8	12	12	12	12	16	16
Cover:		🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟦	🟦	🟦	🟦

Figure 10: Illustration of $S =$ `AGAAAGAAAGAAAGAGA`, which contains two PMRs with annotations $(0, 4, 3, 3)$ and $(12, 2, 2, 1)$. This Figure shows the constructed cover. The blue and orange marked intervals are the runs used in the cover.

Constructing All Covers For some strings, multiple covers can lead to the maximal cover. We therefore also developed an algorithm that constructs all there covers. This algorithm follows the same general approach as the one described above. However, instead of terminating after finding a single cover, it backtracks to explore whether other valid runs satisfy Condition 1 or whether position i satisfies Condition 2. In this way, all distinct covers are generated.

As an example, consider again the string $S =$ `AGAAAGAAAGAAAGAGA`, which contains two PMRs with annotations $(0, 4, 3, 3)$ and $(12, 2, 2, 1)$. The algorithm first constructs the initial cover, shown as cover 1 in Figure 11. It then begins to backtrack.
The algorithm first returns to position $i = 12$. At this position, it checks whether there exists an alternative valid run for which Condition 1 holds. However, no such run exists. Condition 2 does hold, and the algorithm therefore moves to $i = 11$. At position $i = 11$, a valid run satisfies Condition 1, which results in the construction of cover 2.
The algorithm then moves to position $i = 15$, since at $i = 11$ neither Condition 1 nor Condition 2 holds. From position $i = 15$, the algorithm constructs the third and final cover.

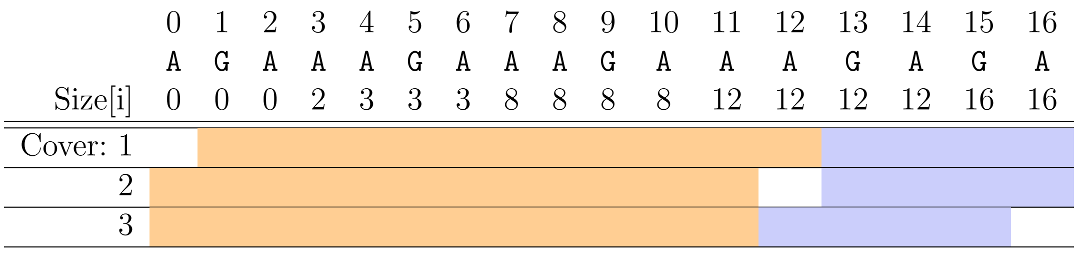

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
A G A A A G A A A G A A A G A G A
Size[i] 0 0 0 2 3 3 3 8 8 8 8 12 12 12 12 16 16
Cover: 1
2
3

Figure 11: Illustration of $S = \texttt{AGAAAGAAAGAAAGAGA}$, which contains two PMRs with annotations $(0, 4, 3, 3)$ and $(12, 2, 2, 1)$. This Figure shows all the possible constructed covers. The blue and orange marked intervals are the runs used in the covers.

4.3.1 Computational Complexity

Let $n = |S|$. To determine the computational complexity of the algorithm, we derive a coarse upper bound. Iterating over all positions in the string S takes $O(n)$ time. Since the maximal number of PMRs in a string S of length n is linearly bounded in n [KK99], this step takes $O(n)$ time. For each position and each PMR whose interval contains that position, the algorithm iterates over the repetitions of the corresponding repeat unit. The repetition count is bounded by $n/2$ and therefore takes $O(n)$ time.
Consequently, the overall computational complexity of this dynamic programming algorithm is:

$$O(n \cdot n) = O(n^2),$$

as a function of the input size n.

4.4 Linear Algorithm

When a PMR has a large count, the dynamic programming algorithm must iterate over a large number of valid runs determine the best one a given position. For example, consider Figure 12 and the string $S =$ `ACTCTCTCTCT`, which contains a single PMR with annotation $a = (0, 2, 6, 0)$. Let $i = 12$. Then $\sigma = \frac{12-1+1}{2} = 6$ and position 13 has $\sigma - 1 = 5$ valid runs. Consequently, the dynamic programming algorithm must compute the cover size for all five valid runs at this position.

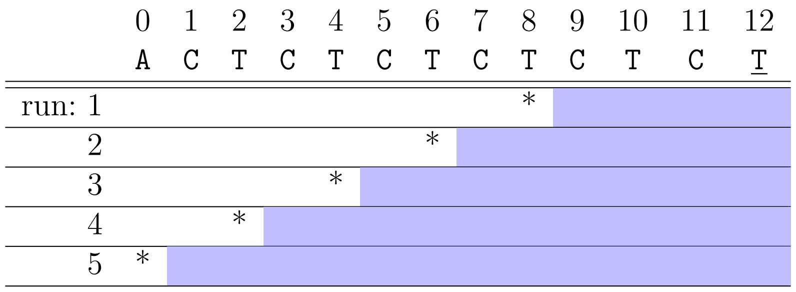


Figure 12: String $S =$ `ACTCTCTCTCTCT` with annotation $a = (0, 2, 6, 0)$, and let $i = 12$. This figure illustrates all runs that need to be evaluated at position $i = 12$. The symbol $*$ denotes a lookup in the $Size$ array at the indicated index. For instance, for run 2, the algorithm performs an array lookup of $Size[6]$.

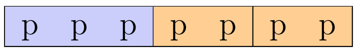
Since STRs can have lengths of up to several hundred base pairs with periods ranging from 1 to 6 base pairs, their counts can be very large. In such cases, the dynamic programming algorithm becomes inefficient.
Every valid run length can be defined as $\ell_c = p^c$, where p is the period. Since $c > 1$, c can be expressed as:

$$c = 2a + 3b,$$

with non-negative integers a and b, at least one of which is non-zero. This can be proven by considering two cases.

- Case 1 — c is even: In this case, c can be written as $c = 2k$ with $k \geq 1$. Choosing $a = k$ and $b = 0$ yields $c = 2a + 3b = 2k + 3 \cdot 0 = 2k$. Since $k \geq 1$, a is non-zero.
- Case 2 — c is odd: In this case, c can be written as $c = 3 + (c - 3)$. Since $c - 3$ is even, we have $c - 3 = 2k$ for some $k \geq 0$. Choosing $a = k$ and $b = 1$ yields $c = 2a + 3b = 2k + 3 \cdot 1 = 2k + 3$. Since $k \geq 0$, b is non-zero.

Consider the example below: $\ell_7 = p^c = p^{3\cdot 1} \cdot p^{2\cdot 2}$.



Let i denote the current position in the string S. Then there are three possible situations.
If $\sigma = 2$, then there is one valid run at position i. The maximal cover at position i is computed as:

$$Size[i] = \max\bigl(Size[i-1],\ Size[i-\ell_2] + \ell_2\bigr).$$

If $\sigma = 3$, then there are two valid runs at position i. The maximal cover at position i is computed as:

$$Size[i] = \max\bigl(Size[i-1],\ Size[i-\ell_2]+\ell_2,\ Size[i-\ell_3]+\ell_3\bigr).$$

If $\sigma > 3$, then there are more than two valid runs at position i. Each ℓ_c can be decomposed into $p^{2a} \cdot p^{3b}$, and for each ℓ_c with $c > 3$ a part as already been computed. Therefore, the maximal cover at position i can be computed in exactly the same way as for $\sigma = 3$.
Figure 13 shows an example for the string $S =$ `CTCTCTCTCTA` with annotation $a = (0, 2, 5, 0)$. The figure shows all valid runs that are evaluated at positions 3, 5, 7, and 9 in the dynamic programming algorithm. It also shows how each ℓ_c with $c > 3$ can be decomposed into smaller runs (ℓ_2 and ℓ_3). For example, run A evaluated at $i = 3$ can be combined with run D evaluated at $i = 7$ to form a valid run with ℓ_4 at position $i = 7$.

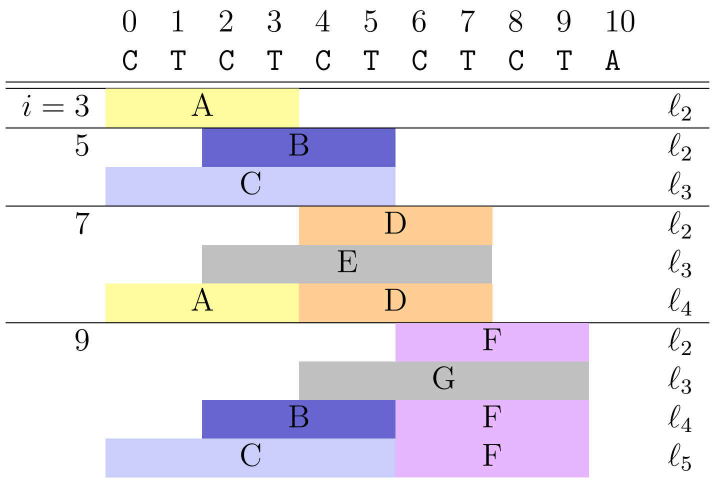


Figure 13: String $S =$ `CTCTCTCTCTA` contains a single PMR with annotation $(0, 2, 5, 0)$. This figure illustrates how all valid runs at positions $i = 3, 5, 7$, and 9 can be decomposed into ℓ_2 and ℓ_3 runs.

4.4.1 Computational Complexity

Let $n = |S|$. To determine the computational complexity of the algorithm, we derive a coarse upper bound. Iterating over all positions in the string S takes $O(n)$ time. Since the maximal number of PMRs in a string S of length n is linearly bounded in n [KK99], this step takes $O(n)$ time. For each position and each PMR whose interval contains that position, the algorithm computes the maximal cover for at most three segment lengths, namely ℓ_0, ℓ_2, and ℓ_3. This step therefore takes constant time, $O(3) = O(1)$.
Consequently, the overall computational complexity of this lineair algorithm is:

$$O(n \cdot 1) = O(n),$$

as a function of the input size n.

5 Experiments and Discussion

To evaluate the introduced algorithms, we conducted a series of experiments. The experiments focus on two main aspects: the runtime performance of the algorithms for constructing maximal covers and the quality of the resulting descriptions. We conducted experiments on both synthetic input sequences and real genomic data. Synthetic sequences are used to analyze runtime performance, while real DNA sequences from dbSNP are used to evaluate the descriptions. All experiments were conducted on a modern PC equipped with a 12th-generation Intel Core i7 processor with a maximum clock frequency of 4.7 GHz.

5.1 Synthetic Sequences

In this experiment, we execute all algorithms that determine the maximal cover in order to evaluate and compare their runtime performance. To ensure a fair comparison, only input strings with a fixed length are used. Three different types of input strings are considered:

- **Non-isomorphic binary words:** All non-isomorphic words of a fixed length over a binary alphabet $\Sigma = \{A, C\}$. Non-isomorphic words are words that differ in structure. For example, the non-isomorphic words of length 4 are `AAAA`, `AAAC`, `AACA`, `AACC`, `ACAA`, `ACAC`, `ACCA`, and `ACCC`.
- **Fibonacci word:** A prefix of a given length of the infinite Fibonacci word. A Fibonacci word is defined over a binary alphabet and is constructed by concatenating the two preceding words in the sequence. For example, the Fibonacci word of length 4 is `ACAA`.
- **Lyndon words:** All Lyndon words of a specified length over an ordered alphabet, generated using Duval's algorithm. [BP94] A Lyndon word is strictly smaller in lexicographic order than all of its circular rotations. For instance, for length 4, the Lyndon words are `AAAC`, `AACC`, and `ACCC`.

As the string length increases, the number of input strings also increases. This is because the number of non-isomorphic binary words and Lyndon words grows rapidly with the length of the string. In contrast, the Fibonacci word always contributes a single input string. For example, at a string length of $n = 20$, the number of input strings increases to 576,666. Consequently, the runtime results are reported as the average runtime per input string.

Short Input Strings First, we ran all algorithms on strings of lengths $\{8, 10, 14, 16, 18, 20\}$. The results are shown in Figure 14. We observed that the two greedy algorithms, the dynamic programming algorithm, and the linear algorithm show nearly identical runtimes. Therefore, only the linear time algorithm is included in the figure. For strings of length 8, 10, and 14, the average runtime is almost identical across all algorithms. However, as the string length increases, the average runtime of the brute-force algorithm grows significantly, while the remaining algorithms continue to show similar runtimes.

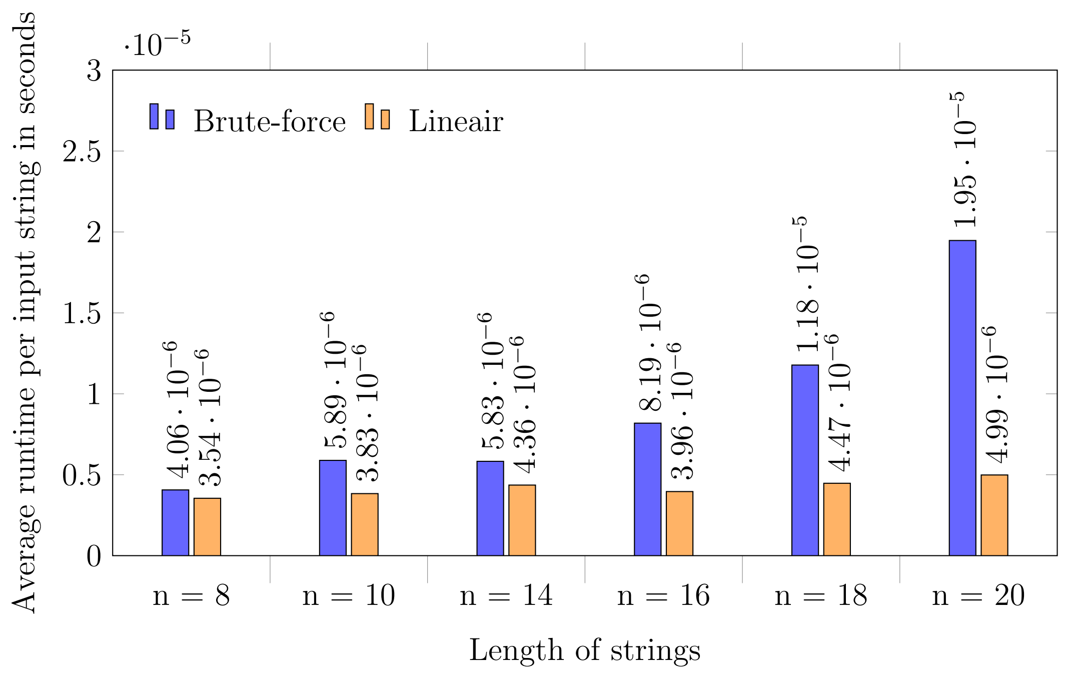


Figure 14: Results of the brute-force and linear algorithms for input strings of lengths $8, 10, 14, 16, 18,$ and 20. For each length, multiple input strings were evaluated, and the reported values represent the average running time per input string in seconds.

Long Input Strings Since the number of input strings grows prohibitively large for $n > 20$, we evaluate the algorithms on larger input sizes using only the Fibonacci word. Because the brute-force algorithm already shows substantially higher runtimes for string lengths $\{16, 18, 20\}$, it is excluded from this evaluation.

The results are shown in Figure 15. All runtimes remain low, even for input strings of length up to 40,000. A notable observation is that the linear algorithm shows the highest runtime among the four algorithms. This can be explained by the fact that the linear algorithm only outperforms the dynamic programming algorithm when PMRs have very high repetition counts. In Fibonacci words, however, the average repetition count remains between 2 and 3. As a result, the dynamic programming algorithm performs comparably to the linear algorithm in this setting. This performance difference can be explained by the structure of the implementations. The linear variant relies on additional function calls and conditional checks, which introduce a small overhead when repetition counts are low.

Finally, we observe that the greedy algorithm based on run length is significantly faster than the greedy algorithm based on end position. Although both greedy algorithms are identical except for the sorting step, one sorts runs by end position whereas the other sorts runs by covered length. This suggests that the observed performance difference is primarily caused by the sorting of the runs.

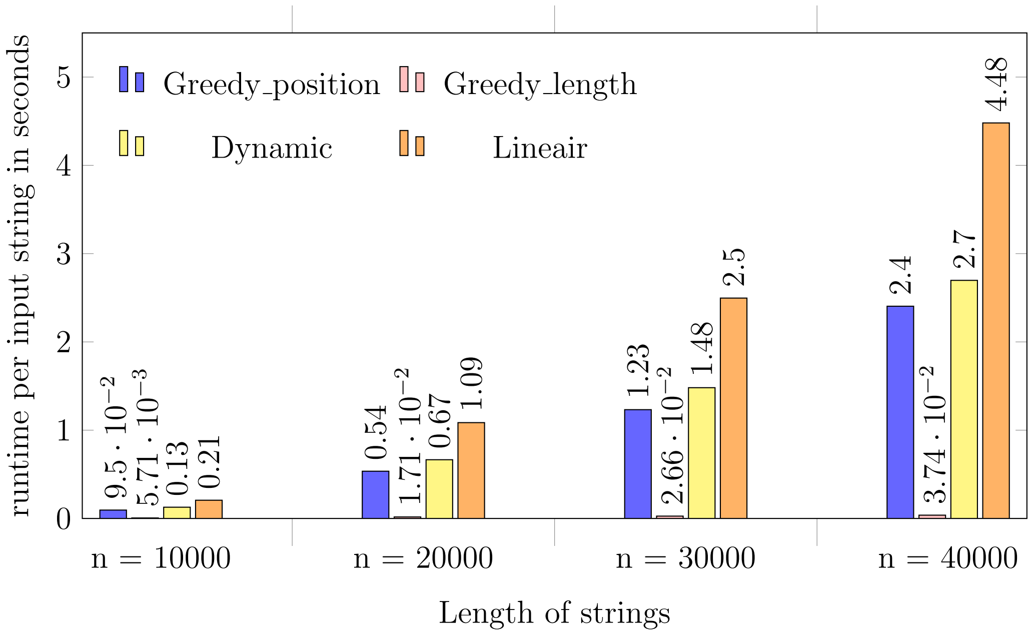


Figure 15: Results of the two greedy algorithms, the dynamic programming algorithm, and the linear algorithm for input strings of lengths 10,000, 20,000, 30,000, and 40,000.

5.2 DNA Sequences from dbSNP

In this experiment, we compare our constructed maximal cover with the variant descriptions available in the Single Nucleotide Polymorphism Database (dbSNP). dbSNP is a public database that catalogs single nucleotide polymorphisms (SNPs) as well as other small-scale genetic variations. Submissions to dbSNP originate from a wide range of sources, including individual research laboratories, large-scale genome sequencing projects, and private industry [PZW+24].
We analyze the same set of 338,582 mixed repeat variants that was previously studied in [SKL+25]. In that work, the proposed method was unable to produce mixed repeat descriptions. We therefore use the same dataset in this experiment, as it provides a large and representative collection of mixed repeat variants, relying on the variant descriptions provided in dbSNP Build 156. For all variants on the genomic reference sequence NC_000001.11 (chromosome 1 of GRCh38), both the SPDI and dbSNP descriptions are retrieved. The SPDI descriptions are used as input sequences for our algorithms, while the corresponding HGVS descriptions are used as a reference for comparison with the constructed maximal covers. All maximal covers in this experiment were computed using the linear algorithm. The results of this experiment are presented in Table 3.

We observed that dbSNP descriptions are structured such that they always start with a repeat. If multiple repeats are present, they occur consecutively in the dbSNP description. Consequently, any uncovered symbols in the description always appear at the end of the dbSNP description. As a result, repeat structures that consist of multiple distinct repeats with uncovered symbols occurring in between, such as the complex repeats described in Section 2.1, cannot be represented by the dbSNP description.
Our results show that, out of 338,582 descriptions, there are 31,139 cases in which the dbSNP description exactly matches the maximal cover constructed by our algorithm. Furthermore, there

are 61,951 cases in which both descriptions cover the same number of symbols but do not match syntactically. In these cases, the algorithm for constructing all covers, described in Section 4.3, can be used to obtain a description that exactly matches the dbSNP description.
Consequently, for 245,492 cases, the maximal cover differs from the dbSNP description. An analysis of these mismatches reveals two distinct types of differences.

Table 3: The categorized counts of the comparison between dbSNP description in dbSNP and our generated description.

	One cover	All covers
Identical cover size		
Identical description	31139	93090
Distinct description	61951	-
Type 1		
Single repeat	89178	140085
Short period	37951	57028
Long period	9599	26749
Other		
Type 2	108764	21630
Total	338582	338582

5.2.1 Type 1

The first type of difference occurs in 136,728 cases. In these cases, all repeats in the dbSNP description occur at the exact same positions as in our description. This means that the prefixes of both descriptions are identical. However, the suffixes of the two descriptions differ. Specifically, the suffix of the dbSNP description contains only uncovered symbols, whereas the suffix of our description contains one or more repeats. This type is futher divided into the following classes; *suffix with a single repeat*, *suffix with small periods*, and *suffix with long period*.

Suffix With a Single Repeat In the class *suffix with a single repeat*, the suffix of our description contains exactly one repeat. An example of this case is shown below. This case occurs in 89,178 instances. Our method describes the suffix `AAAAAAAAAAAAAAAA` as `A[16]`, whereas the dbSNP description represents this suffix without using any repeat notation. In this class, our description is more compact than the dbSNP description, particularly for longer sequences, where this compact representation is more space-efficient. Moreover, our method improves readability by explicitly indicating the length of the repeat in the suffix.

Selected substring:	`AAAAAAAAAAAAAAAAAAAAAAAAATAAAAAAAAAAAAAAAA`
dbSNP description:	`A[25];TAAAAAAAAAAAAAAAA` (25)
Maximal cover:	`A[25];T;A[16]` (41)

Suffix with Small Periods In the class *suffix with small periods*, the suffix of our description contains multiple repeats, none of which has a period longer than 3. This case occurs in 37,951 instances. An example is shown below. Our method describes the suffix `CCAAGCGCCGGAGCGCG` as `C[2];A[2];GC[2];C;G[2];AG;CG[2]`, whereas the dbSNP description represents this suffix without using any repeat notation.
As illustrated by this example, the description produced by our method is longer than the dbSNP description due to the presence of many short repeats. To quantify how often this situation occurs, we examined how frequently our method yields a shorter description than the dbSNP description within this class. Out of the 37,951 cases, there are 14,441 strings for which the description produced by our method is shorter than the corresponding dbSNP description. This indicates that in the majority of cases within this class, our method produces a longer description, probably due to the presence of many short repeats.
As the example demonstrates, the presence of numerous short repeats causes the description to become cluttered and difficult to read. Unless the descripted short repeats are biologically relevant, this representation reduces readability without providing meaningful additional information.

Selected substring:	`GCGCGGGGCCAAGCGCCGGAGCGCG`
dbSNP description:	`GC[2];G[4];CCAAGCGCCGGAGCGCG` (8)
Maximal cover:	`GC[2];G[4];C[2];A[2];GC[2];C;G[2];AG;CG[2]` (22)

Suffix With Long Period In the class *suffix with long period*, the suffix of our description contains multiple repeats, of which one or more have a period longer than 4. This case occurs in 9,599 instances. An example is shown below. In this example, the suffix produced by our method contains four repeats with a period of 5. Such longer repeats can be biologically relevant and may be overlooked in the dbSNP descriptions that do not explicitly represent these repetitions.

Selected substring:	`AGCCCAGCCCAGCCCAGCCCGGCCCGGGCCGGGCCGGGCCCGGCCCGGCCCAGCCCAGC CCAGCCCAGCCCAGCCCAGCCCAGC`
dbSNP description:	`AGCCC[4];GGCCCGGGCCGGGCCGGGCCCGGCCCGGCCCAGCCCAGCCCAGCCCAGCC CAGCCCAGCCCAGC` (20)
Maximal cover:	`AGCCC[4];G[2];C[2];CGGGC[3];CCGGC[2];CCAGC[7]` (84)

5.2.2 Type 2

The second type of difference occurs in 108,764 cases. In this category, not all repeats present in the dbSNP description also occur in the description produced by our method. An example is shown below. This example shows that the repeats in the dbSNP description does not match with the repeats in the description produced by our method.
As explained earlier, we have implemented an additional algorithm that enumerates all covers yielding the maximal cover. For these 108,764 cases, we examined whether an alternative cover exists that more closely matches the dbSNP description. Due to extensive overlap between repeats, some strings have an extremely large number of covers that yield the maximal cover. For this

Selected substring:	`AAAAGAAAGAAAGAAAGAAAGAAAGAAAAGAAAGAAA`
dbSNP description:	`A[4];GAAA[6];AGAA[2];A` (36)
Maximal cover:	`A[3];AGAA[4];AGAAAGAAA[2]` (37)

reason, we decided to terminate the process once 10,000 distinct covers had been generated. In 1,005 cases, the algorithm was terminated early. As a result, not all possible covers were obtained, and it is therefore possible that the best-matching cover was not included.

As shown in Table 3, there are 50,981 cases for which an alternative cover leading to the maximal cover can be classified as a Type 1 *suffix with single repeat*. Furthermore, based on the analysis of alternative covers, 19,104 cases are now classified as Type 1 *suffix with small periods*, and 17,092 cases as Type 1 *suffix with long period*.
There are 21,630 cases in which the description cannot yet be classified as Type 1. We therefore analyze these cases in more detail. For all such cases, we consider two descriptions. First, we use the description obtained from the algorithm that returns a single cover, which we refer to as the *main cover*. Second, we use the algorithm that enumerates all covers that lead to the maximal cover. From these covers, we select the shortest description, which we refer to as the *shortest description*.

Shorter Description We examined how often our algorithm produces a shorter description compared to the dbSNP description. Out of the 21,630 cases, there are 6,328 cases in which the main cover yields a shorter description than the dbSNP description. A small improvement is observed when considering all covers. In this setting, there are 6,771 cases in which the shortest description among all constructed covers is shorter than the corresponding dbSNP description. This behavior can potentially be explained by the lengths of the repeats and their associated periods.

Difference in Repeat Periods We also examined the differences in repeat periods across the various descriptions. Specifically, we analyzed how often repeats of a given period occur in each description. Since the frequency with which a period appears does not capture the lengths of the corresponding repeats, we additionally considered the total counts associated with each period. The results are presented in Table 4.

The example below illustrates how this works. In this example, the main maximal cover differs from the shortest maximal cover. This difference arises because the main maximal cover is constructed from right to left and prioritizes longer runs over shorter ones. For this example, the dbSNP description contains one repeat with period > 6 (total count $= 2$). The main cover contains two repeats with period 1 (total count $= 2 + 2 = 4$) and five repeats with period 2 (total count $= 3 + 4 + 2 + 3 + 4 = 16$). In contrast, the shortest cover contains one repeat with period 1 (total count $= 2$), three repeats with period 2 (total count $= 3 + 4 + 3 = 10$), and one repeat with period > 6 (total count $= 2$).
As shown in Table 4, the dbSNP descriptions contain fewer repeats for every period length compared to both the main cover and the shortest description. The observed difference is especially noticeable for repeats with short periods. Moveover, for short periods, the total count is only slightly larger than the number of occurrences. For example, for period $p = 1$ in the main cover, this period occurs

Selected substring:	`GGTGTGTGGTGTGTGTCTGTGTGGTGTGTGGTGTGTGT`
dbSNP description:	`GGTGTGT[2];GTCTGTGTGGTGTGTGGTGTGTGT` (14)
Main maximal cover:	`G[2];TG[3];GT[4];CT;GT[2];G[2];TG[3];GT[4]` (36)
Shortest maximal cover:	`G[2];TG[3];GT[4];CT;GTGTGGT[2];GT[3]` (36)

39,668 times, while the sum of counts is 137,529. This implies that repeats with period 1 have an average count of approximately 3. Such short repeats contribute little to the overall coverage. Differences are also observed for larger repeat periods. In particular, for period 6, the description produced by our method contains approximately twice as many repeats as the dbSNP description. Repeats with longer periods can be biologically relevant, which indicates that the dbSNP description may not capture some of these important repeats.
The differences between the main cover and the shortest description are relatively small. However, as shown in Table 4, the shortest description contains fewer repeats with periods 1, 2, and 3, and more repeats with periods 4, 5, and 6. While the main cover contains more repeats with periods greater than 6 overall, the shortest description has a larger total sum of counts, indicating that its repeats are longer on average. Descriptions consisting of fewer but longer repeats tend to be shorter and are generally more readable.

Table 4: Total numbers of periods and total counts for repeat periods 1, 2, 3, 4, 5, 6 and > 6 for the main cover description, shortest cover description, and dbSNP description. The leftmost column indicates the period. The next three columns report the total number of repeats for each description, and the three rightmost columns report the corresponding total counts.

	Total periods			Total counts		
	Main	Shortest	dbSNP	Main	Shortest	dbSNP
p = 1	39668	37476	5662	137529	131991	47658
2	32583	31837	10841	111854	111164	48379
3	8972	8898	2854	21446	21247	8122
4	15653	15963	9746	52666	54496	40060
5	4735	4934	3137	15056	15799	11296
6	2568	2605	1126	5563	5642	2652
> 6	9068	9038	6669	19368	19483	14471

Difference in Uncovered Symbols We also examined the differences in the number of uncovered symbols across the descriptions. A substantial difference is observed. The maximal cover contains a total of 46,015 uncovered symbols, corresponding to an average of approximately two uncovered symbols per string. In contrast, the dbSNP descriptions contain a total of 440,889 uncovered symbols, which corresponds to an average of about 20 uncovered symbols per string.
This difference is not necessarily a sconcern if the uncovered symbols do not form biologically relevant repeats. However, as shown in Table 4, the dbSNP descriptions also miss a considerable number of repeats with longer periods, which can be biologically important.

6 Conclusion

In this research, we addressed the problem of describing repeats in DNA sequences. Repeats play an important role in several domains, including forensic and biomedical science. However, existing standards such as the HGVS nomenclature lack sufficient rules to consistently describe complex forms of repeats.
We first studied the problem of identifying all repeats in a sequence. We introduced a naive algorithm. In addition, we provided a detailed description of an existing linear time algorithm for finding all repeats, although this algorithm was not implemented.
Subsequently, we introduced five different algorithms to describe a sequence using repeats by selecting combinations of primitive maximal repeats that maximize the number of covered symbols. These algorithms were examined with respect to their runtime performance.
The experimental results show that the brute-force algorithm becomes impractical for larger input sizes. In contrast, the remaining four algorithms efficiently process sequences of length up to 40,000. However, the two greedy algorithms do not always produce a maximal cover. Both the dynamic programming algorithm and the linear time algorithm consistently yield a maximal cover.
We also examined the quality of the descriptions produced by our method in comparison to descriptions from dbSNP. The results show that our method is able to identify more repeats than the dbSNP descriptions and cover a larger number of symbols in the sequence. However, our method may sometimes result in longer descriptions.

6.1 Further Research

Although the experiments clearly demonstrate the effectiveness of our methods, several directions for further research remain, beyond those discussed earlier in this thesis.
In this work, the objective is to describe a DNA sequence by constructing a maximal cover, thereby maximizing the number of covered symbols. While this criterion ensures maximal coverage, it does not necessarily result in the most compact description. Future research could therefore explore alternative optimization objectives, such as minimizing the total length of the resulting description.
Since our methods always prioritize maximal coverage, biologically meaningful repeats, such as specific short tandem repeats (STRs), may be overlooked when overlapping repeats yield a higher overall cover. An interesting direction for future research would be to incorporate biological relevance into the selection process, for example by prioritizing biologically meaningful repeat.
A limitation of our approach is that repeats are detected only when they occur as exact repetitions. If mutations are present within a repeat, such as in the case of simple repeats with non-consensus repeat described in Section 2.1, the repeat may be interrupted or not detected at all. Future research could investigate methods for detecting imperfect repeats by allowing mutations within repeat units.

6.2 Implementation

The implementation used in this thesis is publicly available on GitHub at `https://github.com/elinerodoe/repeats.git`. All algorithms are implemented in `C++`. The experimental framework is implemented in `Python`.

References

[BP94] Jean Berstel and Michel Pocchiola. Average cost of Duval's algorithm for generating Lyndon words. *Theoretical computer science*, 132(1-2):415–425, 1994.

[But07] John M. Butler. Short tandem repeat typing technologies used in human identity testing. *Biotechniques*, 43(4):Sii–Sv, 2007.

[CIS08] Maxime Crochemore, Lucian Ilie, and William F. Smyth. A simple algorithm for computing the Lempel Ziv factorization. In *Data Compression Conference*, pages 482–488, 2008.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms.* MIT Press, 3rd edition, 2009.

[GLH11] William Goodwin, Adrian Linacre, and Sibte Hadi. *An introduction to forensic genetics.* John Wiley & Sons, 2nd edition, 2011.

[Gus97a] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[Gus97b] Dan Gusfield. Linear-time construction of suffix trees. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*, 1997.

[KK99] Roman Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science*, pages 596–604, 1999.

[Mai89] Michael G. Main. Detecting leftmost maximal periodicities. *Discrete Applied Mathematics*, 25(1-2):145–153, 1989.

[ML84] Michael G. Main and Richard J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984.

[OZ07] Harry T. Orr and Huda Y. Zoghbi. Trinucleotide repeat disorders. *Annual Review of Neuroscience*, 30(1):575–621, 2007.

[Pau18] Henry Paulson. Repeat expansion diseases. *Handbook of clinical neurology*, 147:105–123, 2018.

[PZW$^+$24] Lon Phan, Hua Zhang, Qiang Wang, Ricardo Villamarin, Tim Hefferon, Aravinthan Ramanathan, and Brandi Kattman. The evolution of dbSNP: 25 years of impact in genomic research. *Nucleic Acids Research*, 53(D1):D925–D931, 2024.

[SKL$^+$25] Mark A. Santcroos, Walter A. Kosters, Mihai Lefter, Jeroen F. J. Laros, and Jonathan K. Vis. A graph-based approach to variant extraction from sequences. *NAR Genomics and Bioinformatics*, 7(4), 2025.

[Smy03] Bill Smyth. *Computing patterns in strings.* Pearson Education, 1st edition, 2003.

[sor25] std::sort. https://en.cppreference.com/w/cpp/algorithm/sort.html, 2025. Accessed: 2026-01-03.

[STR24] Tandem repeats in the long-read sequencing era. *Nature Reviews Genetics*, 25(7):449–449, 2024.

[SuRM+24] Sanaullah Sajid, Sajjad ur Rahman, Shahid Mahmood, Shayan Bashir, and Mudasser Habib. *Fundamentals of cellular and molecular biology*. Bentham Science Publishers, 1st edition, 2024.

[Ukk95] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[vd15] Kristiaan J. van der Gaag and Peter de Knijff. Forensic nomenclature for short tandem repeats updated for sequencing. *Forensic Science International: Genetics Supplement Series*, 5:e542–e544, 2015.

[vvv+25] Kristiaan J. van der Gaag, Vincent van Marion, Redmar R. van den Berg, Natalie E.C. Weiler, Jerry Hoogenboom, Arnoud Kal, Manfred Kayser, Peter de Knijff, Jeroen F.J. Laros, Titia Sijen, and Klaas Slooten. Identifying a monozygotic twin brother as a donor of dna in minimal, mixed forensic stains – a case example. *Forensic Science International: Genetics*, 78, 2025.

[Xu24] Isaac Xu. *Tandem repeat variation in human genomes and its role in health and disease: insights from long-read sequencing*. PhD thesis, University of Miami, 2024.