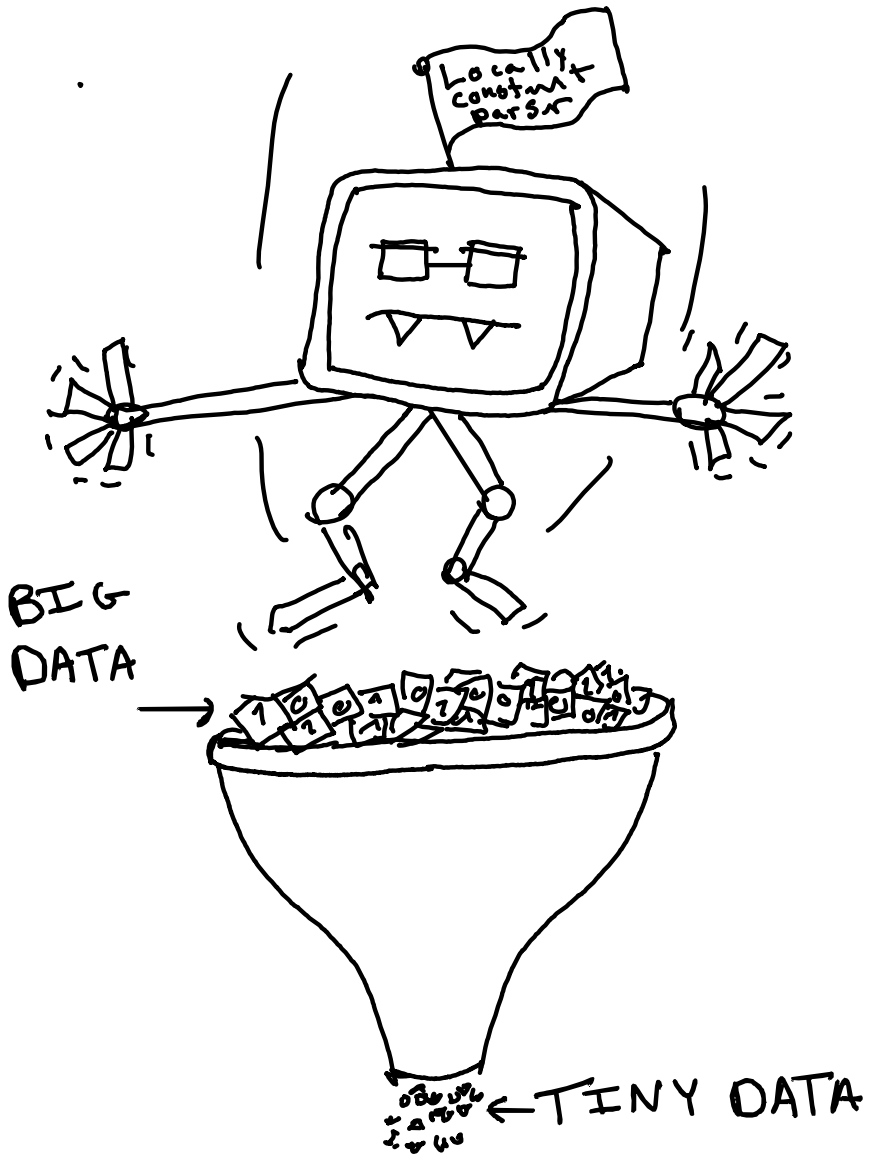
DTU Compute
Department of Applied Mathematics and Computer Science

Locally Consistent Parsing

A survey on different locally consistent data structures and the problems they solve

Johanne Müller Vistisen



Preface

This thesis has been prepared over five months at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark, DTU, specifically in the Section for Algorithms, Logic and Graphs. This thesis is the last project of a Master of Science in Engineering, MSc Eng.

I want to thank my supervisors, Eva Rotenberg and Inge Li Gørtz, for their guidance, ideas, and insights, and furthermore for allowing me to experience being a part of the AlgoLog section. On that note, thank you to my patient "office mates" who have listened, given me advice, and answered the question: "Does this figure make sense?" a million times. Also, many thanks to Frederikke Uldahl, Johanne Falk, and Bodil Kjeldgaard Vistisen for proofreading. A special thanks to Carl Georg Müller Vistisen for creating the front page of this thesis.

On a final personal note, I would like to thank my very supportive family, friends, and girlfriend for helping me finish the project to my own satisfaction.

Johanne Müller Vistisen

Abstract

This thesis surveys different data structures, all using the concept of *locally consistent parsing*. Locally consistent parsing is a way of process a string, S, of length n so that all identical substrings of a certain length, no matter their position in S, are treated the same way. Many problems related to string indexing can be solved within sublinear space using locally consistent data structures.

We present several constructions of locally consistent partitionings. Two of these are based on prior work: the (τ, δ)-partitioning set and the τ-synchronising set. Additionally, we introduce the $(\alpha, \beta, \tau, \rho)$-locally consistent set as a unifying framework to highlight the parallels and distinctions between the two existing partitionings.

We demonstrate that given a τ-synchronising set, it is possible to construct a (τ, τ)-partitioning set in $\mathcal{O}\left(\frac{n}{\tau}\right)$ time. Furthermore, in the absence of periodic substrings, such a (τ, τ)-partitioning set can be transformed into a $(\tau + 1)$-synchronising set. This implies that one can use a deterministic $\mathcal{O}(n)$-time construction of a τ-synchronising set, transform it into a (τ, τ)-partitioning set in $\mathcal{O}\left(\frac{n}{\tau}\right)$, and thereby obtain a faster construction than the previously suggested deterministic $\mathcal{O}(n \log \tau)$ approach.

Continuing the literature survey of local consistency, we also cover the *locally consistent grammar* known as the signature grammar, a structure introduced in earlier literature. Using the main idea behind the signature grammars construction, we propose a new partitioning method called τ-local minimum set.

Furthermore, two concepts of locality are introduced: *index-based* and *block-based* locality, and these are applied to the previously described partitionings. They are also used to clarify the difference between the problems which can be solved by the locally consistent partitionings with index-based locality properties and those solved by the signature grammar with block-based properties. Index-based locality ensures that when solving, for instance, problems similar to The Longest Common Extension Problem, it is possible to make very few look-ups in the original string and instead work on a much smaller string (of size $\frac{n}{\tau}$) for most of the query. Block-based locality makes the original string superfluous and, in fact, works as a compression of the original string. It is possible to make string indexing queries on the compressed string.

Finally, this thesis points to an inadequate part of the original correctness proof of the signature grammar's string indexing query. We describe an example where the description in the paper would overlook an occurrence. We provide an updated query algorithm and prove that it works, while still preserving the runtime of the original algorithm.

Contents

Chapter 1

Introduction

1.1 Introduction and motivation

We collect and store data like never before. New technologies make creating data easier than ever, while others enable us to save vast quantities of data. But even though the amount of data is rising, the information it conveys might not be. Consider, for instance, the number of pixels in an average digital photo today compared to five years ago. Although the image may be sharper and more detailed, it also contains a lot of repetitive information, like areas of similar colour or texture.

This phenomenon is not limited to images, as the same can be observed with text data. Extensive collections of text data, such as sensor logs, server records, large version-controlled text files, and biological databases storing data like DNA, are growing due to more precise, fine-grained data collection tools and better storage availability. However, much of this increase comes from repetitive, recurring patterns rather than new information. Sensor logs may include frequent status updates that differ little over time, version-controlled text might have some parts which are rarely changed, and DNA datasets often consist of overlapping or repeated sequences.

Recognising this, we can exploit repetition and redundancy to compress data effectively. Instead of working with massive original datasets, we can identify patterns and encode them, preserving essential information while significantly reducing the data sizes we are working with. This is the core idea behind locally consistent parsing. It is a method that enables us to compress data in a way that allows minimal access to the original dataset, performing most operations on a compact, pattern-based representation.

1.1.1 Locally consistent parsing

Given a string, a parsing (also called a partitioning) is a way of splitting the string into non-overlapping blocks. What makes a parsing locally consistent is that two "long enough" and identical substrings will be made out of the same blocks, except for "border" elements (placed amongst the leftmost and rightmost indices of the substrings).

Some of the intuition behind what makes a parsing *locally consistent* is that it can be used to construct a new, shorter version of the original string, where each block of the parsing is assigned a unique symbol, which it shares only with blocks with exactly the same elements. Then, if we want to compare two substrings of a big string or perform some other string query on it, we can do most of the work on the new, smaller string created from the blocks because the blocks of the substrings will (for the most part) be completely identical.

1.1.2 Starting point of the thesis

The foundation of this thesis is four publications by Birenzwige, Golan, and Porat (2020, 5), Kempa and Kociumaka (2019, 22), Christiansen and Ettienne (2017, 8), and Christiansen, Ettienne, Kociumaka, Navarro, and Prezza (2021, 9). They each build a data structure with local properties and all

work towards the goal of a sub-linear working space. The (τ, δ)-partitioning set is introduced for the first time in 2018 by Birenzwige et al. [5], later came the τ-synchronising set by Kempa and Kociumaka [22] in 2019. The signature grammar was introduced by Christiansen and Ettienne [8] in 2017 and revisited in greater detail by Christiansen et al. [9] in 2018.

1.1.3 Related works

Searching for patterns in compressed strings has been studied intensely, but we will in this related works section restrict ourselves to publications using locally consistent methods that have been published after the four previously mentioned papers [5, 8, 9, 22].

On the practical side, Dinklage, Fischer, Herlez, Kociumaka, and Kurpicz (2020, 13) evaluates the practical performance of synchronising sets [22] and (and indirectly partitioning sets [5]) for LCE queries, showing that these theoretically motivated structures outperform naive approaches even for moderate-length queries on real-world repetitive data. Claude, Navarro, and Pacheco (2020, 10) propose a grammar-compressed self-index with space $O(G \log n)$ bits, where G is the size of the grammar, and search time $O((m^2 + occ) \log G)$. They also implemented their results in practice and tested them in highly repetitive text collections.

Another development has been made by Ayad, Loukides, and Pissis (2024, 1), who introduce the concept of sample-based *locally consistent anchors*. They use them for efficient text indexing when the queried patterns exceed a known lower bound in length. These locally consistent anchors have average-case guarantees, and the contribution also offers a practical implementation of the data structure.

Kociumaka, Navarro, and Prezza (2020, 24) propose a unified framework for measuring repetitiveness, $\delta \leq \gamma$ (where γ is the size of the smallest attractor), in strings by introducing and relating several compression measures, including attractors, LZ77, and grammar-based size, thereby offering a basis for comparing compressed indexes. Using this δ and combining two distinct locally consistent parsing techniques[1] Kociumaka, Navarro, and Olivares (2022, 25) obtain search time $O(m + (occ + 1) \log^{\varepsilon} n)$ in tight attractor-bounded space $O(\delta \log(n/\delta))$.

Kempa and Kociumaka (2022, 23) solves the dynamic suffix array problem using a new type of dynamic locally consistent parsing. They also offer a dynamic construction of string synchronising sets [22]. Lastly, the survey by Navarro (2021, 26) is worth mentioning as it provides a comprehensive portrayal of compressed text indexing. However, his focus is largely on classical data structures and does not focus very much on recent developments in locally consistent parsing.

1.1.4 The organisation of the thesis

In the remaining of Chapter 1, we introduce the notation and algorithmic concepts used throughout the thesis.

Chapter 2 examine the partitioning of a string such that it has some *local* properties. We will look at a general index-based locally consistent partitioning called $(\alpha, \beta, \tau, \rho)$-locally consistent set. This partitioning is a generalisation of two other partitionings: (τ, δ)-partitioning set [5] and τ-synchronising set [22]. The idea of keeping local properties has also been introduced in a slightly different setting, namely in the signature grammar by Christiansen and Ettienne [8]. In their paper [8], a grammar with local properties is created, and the construction strongly resembles that of the (τ, δ)-partitioning set. We will show how the construction from the signature grammar can also be turned into partitioning and how it behaves in relation to the other partitionings.

In Chapter 3, two different forms of locality are introduced. In order to highlight the parallels and differences between the different partitionings. The two localities are *index-based* and *block-based* locality. These different locality concepts are applied to the partitionings' (and their constructions), and

[1] Based on the grammar introduced by Christiansen et al. [9] and another type of locally consistent parsing called *recompression* by Jez (2014, 20).

in Chapter 4 we uncover the relationship between the (τ, δ)-partitioning set and the τ-synchronising set.

In Chapter 5, we look at the signature grammar and turning the construction of a partitioning set into a grammar construction. This grammar is investigated. Chapter 6 focuses on some of the problems local data structures can solve and uses their runtimes to argue for similarities between the different locally consistent data structures.

Chapter 7 contains the final remarks and is followed by the appendix and the references.

1.1.5 The contribution of this thesis

This thesis introduces a more general construction, the $(\alpha, \beta, \tau, \rho)$-locally consistent set, and uses it to show the similarities between the (τ, δ)-partitioning set and the τ-synchronising set.

Furthermore, the two concepts of locality, *index-based* and *block-based* locality, are introduced to compare the different locally consistent partitionings and grammars. Moreover, this thesis shows that when working with non-periodic strings, a partitioning and synchronising set can be reduced to one another. In the general case, there exists a reduction from a τ-synchronising set to a (τ, τ)-partitioning set.

Lastly, in the problem section, this thesis shows an example of a pattern occurrence, which could be missed by the string indexing query by Christiansen and Ettienne [8]. We propose a small alteration that does not influence the query's complexity and we prove that with this change the query solves the text indexing problem in sub-linear space.

1.2 Notation, definitions, and general concepts

In this section, we introduce the general notation and concepts used throughout the thesis.

We operate within the standard word RAM model, where a machine word has size $w \geq \log n$, with $n = |S|$. We denote the set of integers $\{1, \ldots, n\}$ by $[n]$. We consider strings over a finite ordered alphabet $\Sigma = [0 \ldots \sigma - 1]$, where the alphabet size σ satisfies $\sigma = n^{\mathcal{O}(1)}$. A *string* S of length n is a sequence of symbols from Σ, that is, $S = S[1] \ldots S[n] \in \Sigma^n$.

$S[i \ldots j]$ denotes the substring of S that starts at index i and ends at index j (both inclusive), while $S[i \ldots j)$ denotes the substring starting at i and ending at $j - 1$. We define the *prefix* of a string S of length ℓ as the substring $S[1 \ldots \ell]$, and the *suffix* of length ℓ as the substring $S[n - \ell + 1 \ldots n]$.

An integer $\rho \in [1 \ldots |S|]$ is called a *period* of the string S if $S[i] = S[i + \rho]$ for all $i \in [1 \ldots |S| - \rho]$. The smallest such ρ is called the *principal period* of S, and is denoted by $\mathrm{per}(S)$. For a substring to be *periodic*, its principal period must occur at least twice. A substring S' of S is called a (d,ρ)-run if $|S'| \geq d$ and $\mathrm{per}(S') \leq \rho$.

Definition 1. The succeeding element of element i in a set $\mathcal{X}$ is denoted by $\mathrm{succ}_{\mathcal{X}}(i)$.

We introduce the Karp-Rabin fingerprint, a random rolling hash function. It has the property that the fingerprints of all the prefixes of a string can be precomputed in $\mathcal{O}(n)$, and afterwards the fingerprint of any substring can be computed in constant time. Using a definition by Bille et al. [4] and the result introduced by Karp and Rabin [21] we get the following definition:

Definition 2. Let S be a length n string, p is a prime number, and $r \in \mathbb{Z}_p$. Then we can define the Karp-Rabin fingerprint function is defined in the following way

$$\phi(S) = \sum_{i=1}^{n} S[i] \cdot r^{i-1} \mod p.$$

1.2.1 The id-function

In the randomised constructions of both the (τ, δ)-partitioning set and τ-synchronising set, the id-function is used to determine which indices should be in the partitionings. It is a function which maps a substring to a positive integer. We will define this function by the properties it needs to have, and then briefly introduce how such a function can be implemented in practice.

Definition 3 (The id-function)**.** Given a string S of length n, then $\texttt{id} : [1 \dots n - \tau + 1] \to \mathbb{R}$ The following property defines the id-function:

- $\texttt{id}(i) = \texttt{id}(j)$ if and only if $S[i \dots i + \tau) = S[j \dots j + \tau)$.
- Given a length τ interval $[i \dots i + \tau)$ where $S[i \dots i + \tau)$ contains no (τ,τ)-runs then the probability that any element in $[i \dots i + \tau)$ has minimal id-value is $\mathcal{O}\left(\frac{1}{\tau}\right)$. An index $j \in [i \dots i + \tau]$ has minimal value if $\texttt{id}(j)$ is smaller than $\texttt{id}(k)$ for any $k \in [i \dots i + \tau]$ where $k \neq j$.
- Computing $\texttt{id}(i)$ for all $i \in [1 \dots n]$ in a string of length n takes $\mathcal{O}(n)$.

One way to implement the id-function in practice is by using a fingerprint function, like the Karp-Rabin function ϕ, and afterwards hashing the fingerprint.

Birenzwige et al. [5] hashes their fingerprint with a min-wise hash-function h from a family of $\left(\frac{1}{2}, \tau\right)$-min-wise hash functions. This family of hash functions has the property that, given a set of fingerprints $\mathcal{X}$ (a subset of all possible fingerprints) with $|\mathcal{X}| < \tau$, the behaviour of hash values can be characterized probabilistically. Specifically, the probability that any element $x \notin \mathcal{X}$ has a hash value smaller than any value in the interval is $\frac{3/2}{|\mathcal{X}|+1}$.

This is a brief explanation on one possible way to create an id function with the properties of Definition 3, and it is more thoroughly described in section 2 (preliminaries) of [5]. The following definition is used for comparing the similarity of two substrings' parsing.

Definition 4 ($B_{\mathcal{X}}(i, j)$)**.** Let S be a string and let $\mathcal{X}$ be a set of indices of this string. Then $B_{\mathcal{X}}(i, j) = \{x - i : i \leq x \leq j \text{ and } x \in \mathcal{X}\}$. The set $B_{\mathcal{X}}(i, j)$ is the set of indices of a substring $S[i \dots j]$ which are part of $\mathcal{X}$ and then subtracted by the first index of the substring.

An example of the set $B_{\mathcal{X}}(i, j)$ is illustrated in Figure 1.

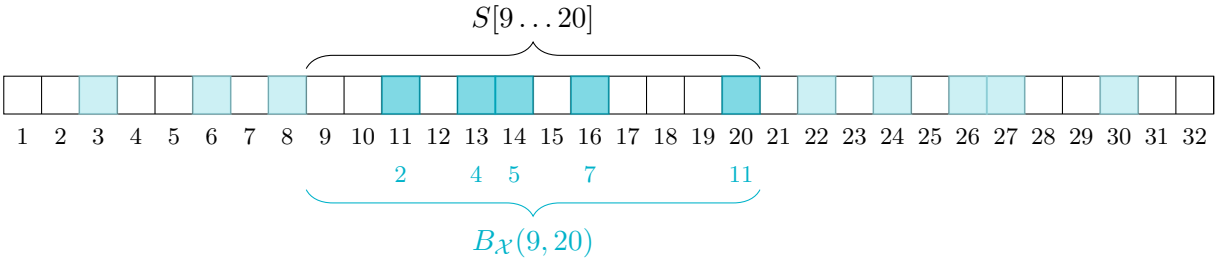


Figure 1: An example with a string S of length 32. Every cell indicates a character, and all the marked cells have indices which are elements of the set $\mathcal{X}$. In this case $\mathcal{X} = \{3, 6, 8, 11, 13, 14, 16, 20, 22, 24, 26, 27, 30\}$. The darker cells are the elements of $\mathcal{X}$ which are also elements of the interval $[9 \dots 20]$. The set $\mathcal{B}_{\mathcal{X}}(9, 20) = \{2, 4, 5, 7, 11\}$ consists of the distance between the indices of those elements and the beginning of the substring we are looking at. In this case, the distance between the indices $\{11, 13, 14, 16, 20\}$ and index 9.

1.2.2 Tries

This brief introduction to tries is inspired by the lecture notes on the subject of Gørtz [17]. Given a collection of strings $\mathbf{S} = \{S_1, \ldots, S_s\}$ over an alphabet Σ of size σ, a *trie* is a rooted, ordered tree that represents all strings in $\mathbf{S}$ such that each root-to-leaf path corresponds to one of the strings in the collection. Common prefixes are shared among the paths, and edges from each node are labelled by distinct characters and sorted alphabetically.

A pattern of length m can be searched for in the trie by traversing the pattern character by character. This search takes $\mathcal{O}(m)$ time if the alphabet size is constant. If the alphabet is large, a balanced binary search tree or hash table can be used at each node to guide the search. In this thesis we will use a trie to do a deterministic search and therefore we only need the following Lemma on search using a balance binary tree.

Lemma 1. *Given a trie over a collection of strings and a balanced binary search tree at each node, a pattern of length m can be searched in $\mathcal{O}(m \log \sigma)$ time, where σ is the alphabet size.*

If we were using hash tables to store edges at each node (with expected constant-time access), then the expected time to search for a pattern of length m is $\mathcal{O}(m)$.

1.2.3 Lempel-Ziv 77

This description of the LZ77 parse is based on the one made by Bille, Ettienne, Gørtz, and Vildhøj (2018, 4).

The Lempel-Ziv77 parse is a compression algorithm from 1977, introduced by Ziv and Lempel (1977, 29), and it plays a vital role in the field of string compression as well as in this thesis. It is a way of compressing a text using repetition in the string. An LZ77 parse is a partition of a string, S, of length n, into a sequence Z of z succeeding substrings, called phrases. We say that $S = Z[1]Z[2]\ldots Z[z]$. Every phrase is usually depicted as a tuple with a start, a length, and a border symbol, c.

For the ith phrase, $Z[i]$ we will write the tuple as (s_i, l_i, c_i), where s_i is the start index of the phrase's source, l_i is the length of the source, and c_i is the border position of $Z[i]$. The string S can in this way be described by $(s_1, l_1, c_1)\ldots(s_z, l_z, c_z) \in ([n], [n], \Sigma)^z$.

The following provides an informal overview of the encoding process. Imagine that we have already partitioned $S[1\ldots j]$ into phrases $Z[1]\ldots Z[i]$. Let j' be defined such that $S[j+1\ldots j'-1]$ is the longest prefix of the substring $S[j+1\ldots n]$ that is also a substring of $S[1\ldots j'-2]$[2]. Then s_{i+1} will be the index of the substring in $S[1\ldots j'-2]$, and $l_i = j'-1-j+1 = j'-j$ is the length of the substring. The border point is $S[j']$. Then the $(i+1)$th phrase describes $S[j+1\ldots j']$ and $Z[i+1] = (s_i, j'-j, S[j'])$.

1.2.4 Weak prefix search with z-fast tries

Moving forward, we discuss the Weak Prefix Search Problem, which will be visited in Chapter 6, when doing pattern matching using a locally consistent grammar. The problem is defined in the following way

The Weak Prefix Problem
Input: A lexicographically sorted set of k strings denoted D. A pattern P of length m.
Query: Report the ranks of those strings in D of which P is a prefix. If there is no strings are prefixes of the pattern, then return arbitrary (and incorrect) ranks.

The weak prefix search is as the name implies a weaker version of prefix search in the sense that if no string with the pattern, we are searching for, the search might return something completely arbitrary.

[2]Note that the interval only goes to $j'-2$ because the source need to be place in the interval $[1\ldots j]$.

In this thesis, we will use a data structure, denoted a z-fast trie, which is described in the following Lemma by Djamal Belazzougui and Vigna (2010,[14, Appendix H.3]).

Lemma 2 (Djamal Belazzougui and Vigna [14],Appendix H.3)**.** *Given a set D of k strings with average length l, from an alphabet of size σ, we can build a data structure using $\mathcal{O}(k(\log l + \log\log\sigma))$ bits of space supporting weak prefix search for a pattern P of length m in $\mathcal{O}(m\log\sigma/w + \log m)$ time where w is the word size.*

When looking something up in the z-fast trie, we must hash it with the Rabin-Karp fingerprint function and then do the query, because the data structure contains Rabin-Karp fingerprints rather than full-length substrings.

1.2.5 2D range reporting

The 2D range reporting problem is defined as follows

2D Range Reporting
Input: A set P of k points in $\mathbb{R}^2$.
Query: Given a query rectangle $Q = [x_1, x_2] \times [y_1, y_2]$, report all points in $P \cap Q$.

There are standard solutions solving this problem in $\mathcal{O}(\log n + occ)$ time and $\mathcal{O}(n\log n)$ space, and another one in $\mathcal{O}(\sqrt{n} + occ)$ time and $\mathcal{O}(n)$ space (covered by Schulz [27] using the results of [3, 7, 12, 28]).

Chan, Larsen, and Pătraşcu (2011, 6) are responsible for the complexity results used for 2D range reporting in this thesis, which Christiansen and Ettienne [8] use to get a data structure of size $\mathcal{O}(z\log(n/z))$ and with a query time of $\mathcal{O}(\log^{\epsilon}(z\log(n/z)))$ [8, 9].

1.2.6 Distributed 6-colouring a path in $\mathcal{O}(\log^* n)$

The algorithm, which in this thesis is referred to as the *colouring algorithm*, is a version of the Cole-Vishkin algorithm [11].

Introduced in 1986, the Cole-Vishkin algorithm can be used to compute a proper colouring of a path using only 6 colours in $\mathcal{O}(\log^* n)$ rounds in the LOCAL model. It starts from unique node identifiers and iteratively compresses colours using bitwise string operations. In each round, a node gets a new colour on the index of the first bit that differs from the colour of its successor, ensuring that adjacent nodes receive different colours. This process rapidly reduces the colour space from 2^{128} (or larger) to 6, after which a simple greedy reduction yields a proper 3-colouring if needed.

In this thesis, a very similar algorithm is used to transform a string over a large alphabet Σ into a string over a constant-size alphabet of size 6. The character at each position i becomes the *label* (or colour) of i, and the transformation preserves local uniqueness. If each character is distinct from its immediate neighbours in the input, then the corresponding labels will also be distinct. The transformation runs in $c\log^* n$ rounds for a string of length n, and is local; that is, the label at index i depends only on the original characters in the $c\log^* n$ positions to the right of i.

While the original Cole-Vishkin algorithm is defined in the LOCAL model (a model used when working with distributed computing), the transformation used in this thesis also works naturally in the word RAM model. Since the label at position i depends only on a window of $c\log^* n$ characters to the right, the transformation can be implemented sequentially in linear time by scanning the input from right to left, maintaining a sliding window of size $\mathcal{O}(\log^* n)$. This locality ensures that the core structure of the algorithm remains efficient even outside the distributed setting.

1.3 A note on the figures

Many of this thesis's figures illustrate abstract strings of length n (or parts of strings of length n). In these figures, the string is represented by a rectangle (which we will call an array). In most cases, this array has some highlighted cells (squares), each corresponding to a single position in the string. Text beneath a cell denotes the index of that particular cell. Brackets over a range of positions mark the number of elements in the range, except if there is an $\in$ sign. This means that all elements in this range belong to a specific set. This is all shown in Figure 2.

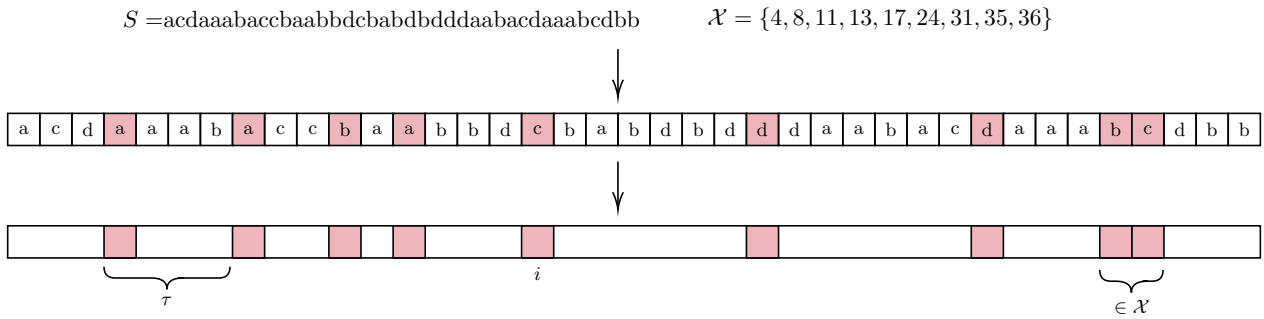


Figure 2: This figure shows how an abstract version a string, S, and a highlighted set of indices, $\mathcal{X}$, are depicted throughout the thesis (in this example S = acdaaabaccbaabbdcbabdbdddaabacdaaabcdbb and $\mathcal{X} = \{4, 8, 11, 13, 17, 24, 31, 35, 36\}$). The bottom array in this figure is the one which is used the most. Text beneath a cell denotes the index of that particular cell, in this example $i = 17$. The number of positions between cell 4 and cell 8 is marked by τ, which is 4 in this figure. Index 35 and 36 both belong to the set $\mathcal{X}$, and the rightmost bracket in the bottom array marks this.

Chapter 2

Locally Consistent Partitionings

In this thesis, a *partitioning* (which will sometimes be called a *parsing*) is a set of splitting positions or indices that partition a string into substrings, called *blocks*. Every character of the string will be in exactly one block.

A partitioning can be viewed as a set of splitting points (also called border points or representatives), as illustrated in Figure 3.

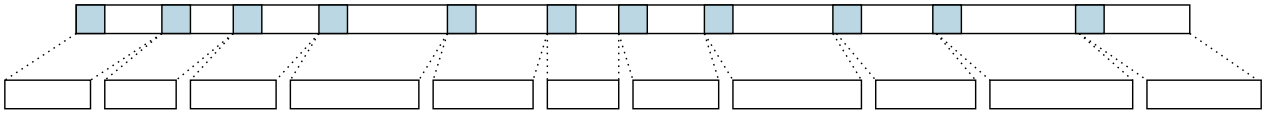


Figure 3: This figure shows the string, S, and the splitting positions of the partitioning set marked in light blue. At every index of the partitioning set, a block starts.

noindent It can also be viewed as a new string where each symbol (from a different alphabet than the alphabet of S) corresponds to a block in the old string, and the new symbol depends on the block's content, such that all blocks with identical elements get the same symbol. Figure 4 shows this.

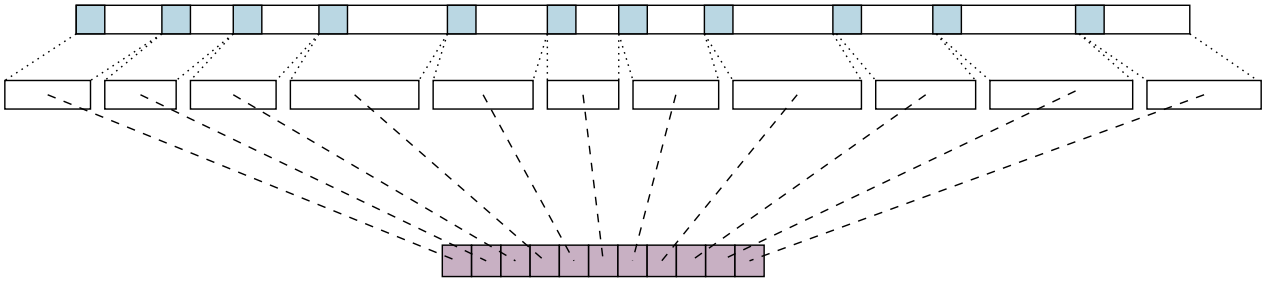


Figure 4: The string, S, is depicted as an array, and the splitting positions of the partitioning set are the marked positions in light blue. They each have a corresponding block which, depending on its content, gets a new symbol from another alphabet. All blocks with the same content get the same symbol. These new symbols (marked with purple in the bottom array) represent a new string.

A locally consistent parsing is a way of parsing identical substrings in the same way, no matter how they are placed in the original string. This results in certain local properties of the string that the partitioning outputs. These properties will be covered in great detail throughout the following chapters.

2.1 $(\alpha, \beta, \tau, \delta)$-locally consistent set

We now introduce a partitioning called a $(\alpha, \beta, \tau, \delta)$-locally consistent set. It is a locally consistent partitioning, unifying the (τ, δ)-partitioning set of Birenzwige et al. [5] and the τ-synchronising set of

Kempa and Kociumaka [22]. This partitioning is defined by two conditions, the locality condition, which is the one that gives the "local" behaviour previously mentioned, and the density condition, which ensures that the positions of the locally consistent set are not too far apart. The only time we allow positions from the locally consistent partitioning to be far apart is when we have substrings with highly periodic behaviour[1].

Definition 5. Let f be a function such that $f : S[i-\alpha \dots i+\beta] \to \{0,1\}$. We denote the set of positions for which the function returns 1 for $\mathcal{L}$. Now the following has to hold for f and $\mathcal{L}$:

- *The locality condition:* $f(S[i-\alpha \dots i+\beta]) = f(S[j-\alpha \dots j+\beta])$ if and only if $S[i-\alpha \dots i+\beta] = S[j-\alpha \dots j+\beta]$ for $i, j \in [1+\alpha \dots n-\beta]$.
- *The density condition:* (a locally consistent set follows one of the two following conditions):
  - Density condition a: For $i \in [1 \dots n-\tau]$ it holds that $\mathcal{L} \cap [i+1 \dots i+\tau] = \emptyset \Rightarrow i \in \mathcal{R}_a$, where $\mathcal{R}_a$ is a set of indices which are beginnings of periodic substrings with period smaller than or equal to ρ.
  - Density condition b: For $i \in [1 \dots n-\tau]$ it holds that $\mathcal{L} \cap [i \dots i+\tau] = \emptyset \Leftrightarrow i \in \mathcal{R}_b$, where $\mathcal{R}_b$ is a set of indices which are beginnings of periodic substrings with period smaller than or equal to ρ.

We call a $(\alpha, \beta, \tau, \rho)$-locally consistent set that fulfils the density condition a for $\mathcal{L}_a$, and one that fulfils condition b for $\mathcal{L}_b$.

Density condition a can also be viewed in another way. It states that if two consecutive elements, l_i and l_{i+1} in $\mathcal{L}$ are more than τ apart then the substring $S[l_i \dots l_{i+1}-1]$ is periodic with period less than or equal to ρ.

2.2 Construction locally consistent partitioning sets

The next step is to look at different constructions of locally consistent partitionings. All the partitionings explored in this work are of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ with different construction algorithms, local behaviour, and use cases. The constructions of the locally consistent partitionings of this thesis can roughly be split into two categories.

2.2.1 The sliding window construction

The first type of construction scans through the string S using a sliding window of size τ (or $\tau+1$), which moves from left to right. At every step, the construction processes the current window in constant time and afterwards shifts the window one position to the right; if the window was, for instance, the interval $[k \dots k+\tau-1]$, then the next window that will be processed is $[k+1 \dots k+\tau]$.

2.2.2 The hierarchical construction

The other type of construction is hierarchical, meaning it is constructed of levels built on top of each other, bottom-up. At the first level (the bottom one), all characters start as single blocks. Then the construction is built of different levels, and blocks are merged with neighbouring blocks at each level to form new blocks.

We often call the current level for level $\mu-1$, and the blocks at this level are denoted *sub-blocks*. The blocks at the level above, level μ, are simply referred to as *blocks*. This structure can also be viewed as an ordered forest, where each block corresponds to a vertex. The children of a vertex

[1] With highly periodic we refer to substrings with a small principal period.

represent the sub-blocks that were merged to form that block at the previous level. We call this *the tree view* or *the forest view*. This gives rise to a couple of definitions related to viewing the hierarchical construction as an ordered forest.

Definition 6. Given a vertex v, then $str(v)$ denotes the string one gets by concatenating all the leaves for which v is an ancestor from left to right. This is shown in Figure 5.

Definition 7. Given a forest view representation of a partitioning, the relevant nodes of a substring $S[i \dots j]$ are all the ancestors of the leaves with indices in the range $[i \dots j]$ (in the hierarchical construction of the partitioning). The relevant nodes of $S[i \dots j]$ are denoted $T(i, j)$.

An example of the relevant nodes of a substring given some hierarchical construction (with forest-view) is given in Figure 6.

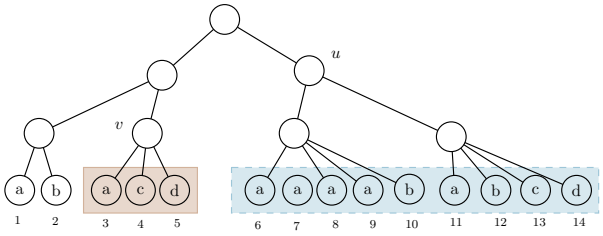


Figure 5: $str(v)$ is acd and $str(u)$ is aaaababcd.

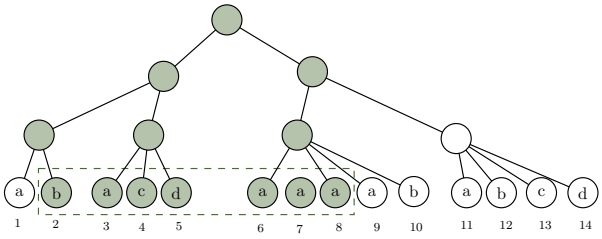


Figure 6: The relevant nodes of $S[2 \dots 8]$ (also denoted by $T(2, 8)$) are the coloured (green) nodes of the graph.

2.3 (τ, δ)-partitioning set

The partitioning set introduced by Birenzwige et al. [5] is referred to as a (τ, δ)-partitioning set and will be denoted by $\mathcal{P}$ throughout this thesis. We assume throughout that $\tau \leq \delta$. It corresponds to a $(\delta, \delta, \tau, \tau)$-locally consistent set with density condition a. This means that if $S[i - \delta \dots i + \delta] = S[j - \delta \dots j + \delta]$ then $i \in \mathcal{P}$ if and only if $j \in \mathcal{P}$, and also that if two consecutive elements in $\mathcal{P}$ are more than τ apart they mark the start and end of a periodic substring with period smaller than or equal to τ[2]. The two conditions are illustrated in Figure 7 and Figure 8.

Figure 7: *The locality condition of a (τ, δ)-partitioning set*: If the blue (dotted) areas in the figure are two identical substrings, then $i \in \mathcal{P}$ if and only if $j \in \mathcal{P}$.

[2] A (τ, δ)-partitioning set has a small extra density condition in [5]: there should at most be a distance of τ between the last element of $\mathcal{P}$ and index $n + 1$. This can be easily implemented in all our constructions by always adding index $n - \tau + 1$ (since $\tau \leq \delta$ this index has no locality requirements) to the (τ, δ)-partitioning set and therefore we still claim that a (τ, δ)-partitioning set is a $(\delta, \delta, \tau, \rho)$-locally consistent set.

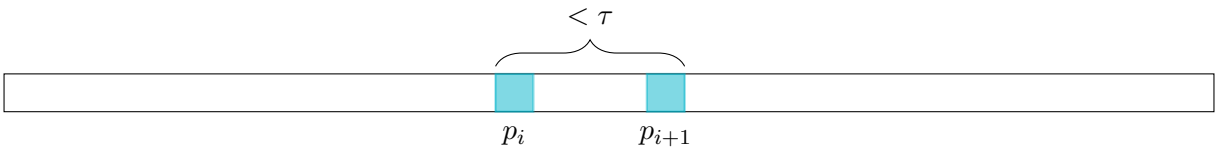


Figure 8: *The density condition of a (τ, δ)-partitioning set*: There is at most τ between two consecutive elements p_i and p_{i+1} unless the substring $S[p_i \dots p_{i+1}]$ is periodic with a period smaller than or equal to τ.

For the (τ, δ)-partitioning set there exists both a randomised and a deterministic construction introduced by Birenzwige et al. [5]. The following subsections will go through the constructions, explaining why they live up to the requirements for locally consistent sets (defined in definition 5), and their time and space complexity.

2.3.1 Randomised construction of a (τ, τ)-partitioning set

Birenzwige et al. [5] introduce a randomised construction, which creates a (τ, τ)-partitioning set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ in expected $\mathcal{O}(n)$ time. We will introduce the construction, then argue why it fulfils the locality and density conditions of Definition 5, and lastly look at the complexity of the construction.

Construction 1 ((τ, τ)-partitioning set randomised construction)**.** Let `id` be a function of the type introduced in Definition 3. Let S be a non-periodic string, which in this context means it contains no $(\tau, \frac{\tau}{6})$-runs. Then $\mathcal{P}$ for the string, S, is constructed as the following set

$$\mathcal{P} = \left\{ j \in [1..n-\tau] \mid \exists \ell \in [j-\tau+1..j] : \texttt{id}(j) = \min_{k \in [\ell \dots \ell+\tau-1]} \{\texttt{id}(k)\} \right\} \cup \{n-\tau+1\}. \quad (2.1)$$

We use a sliding window of size τ. At every position, the index with the smallest `id`-value is identified and added to $\mathcal{P}$ (such that $\mathcal{P}$ is as defined in 2.1).

Periodic construction: Given a S with periodic substrings (which means they contain $(\tau, \frac{\tau}{6})$-runs), the construction begins with identifying all $(\tau, \frac{\tau}{6})$-runs, adding the first and last index of each run to $\mathcal{P}$, and then removing the runs. Subsequently, the remaining string is treated like the non-periodic construction and partitioned like in 2.1.

Having established Construction 1, we move on to prove that this construction builds a (τ, τ)-partitioning set. This is done in the following lemma.

Lemma 3. *Construction 1 yields a (τ, τ)-partitioning set, $\mathcal{P}$.*

The proof of density condition a is based on the proof of Lemma 4.1 [5].

Proof. First, we will show the locality condition and then density condition a of Construction 1.

Locality condition: Assume that there exist $i, j \in [1 + \tau \dots n - \tau]$ such that $S[i - \tau \dots i + \tau] = S[j - \tau \dots j + \tau]$ and additionally that $i \in \mathcal{P}$. Then there must exist an interval $[k \dots k + \tau - 1]$ for $k \in [i - \tau + 1 \dots i]$ where $\texttt{id}(i)$ is minimal. Note that the interval $[k \dots k + \tau - 1]$ will always contain i when k is defined as it is. In Figure 9, an example of such an interval is marked in dark blue (stripes).

Because of the relation between index i and index j, any interval $[k \dots k + \tau - 1]$ (containing i) will have a corresponding interval $[k - i + j \dots k - i + j + \tau - 1]$ which contains j. We consider

the extreme values k can take, namely $i - \tau + 1$ and i, and examine the corresponding intervals $[k - i + j \dots k - i + j + \tau - 1]$. For these values, we obtain the intervals $S[j - \tau + 1 \dots j]$ and $S[j \dots j + \tau - 1]$, respectively.

By our initial assumption we know that $S[j - \tau + 1 \dots j] = S[i - \tau + 1 \dots i]$ and $S[j \dots j + \tau - 1] = S[i \dots i + \tau - 1]$ and we therefore conclude that for any $k \in [i - \tau + 1 \dots i]$ the intervals $S[k \dots k + \tau - 1] = S[k - i + j \dots k - i + j + \tau]$ will be identical. Therefore, if an interval exists where $\texttt{id}(i)$ is minimal, this interval will also exist for $\texttt{id}(j)$. We have shown $i \in \mathcal{P} \Rightarrow j \in \mathcal{P}$. The other way is shown by symmetry.

Figure 9: An example of i and j where the light blue (dotted) areas indicate that $S[i - \tau \dots i + \tau] = S[j - \tau \dots j + \tau]$. The darker blue (striped) area corresponds to an interval of length τ where $\texttt{id}(i)$ is minimal. The idea of this proof is to show that an interval of length τ with similar values will exist in the interval $S[j - \tau \dots j + \tau]$. This area is marked with beige (grid).

Density condition a: We begin by looking at the density condition of the first element of $\mathcal{P}$, p_1, and the last element here denoted p_l. The distance between 1 and p_1 is at most τ, since we begin by finding a minimum $\texttt{id}$-value in the interval $S[1 \dots \tau]$, and this means $p_1 \in [1 \dots \tau]$. The last element will be $n - \tau + 1$ because of (2.1), and this element will have distance τ to index $n + 1$. The distance between p_l and $n - \tau + 1$ is also at most τ because p_l must be from the interval $[n - 2\tau + 1 \dots n - \tau]$. To get the biggest distance between p_l and index $n - \tau + 1$, let us assume that $p_l = n - 2\tau + 1$. Then the distance between the two indices is $n - \tau + 1 - (n - 2\tau + 1) = \tau$.

Now let us look at the more general distance between p_i and p_{i+1}. If the string S is non-periodic, then a distance greater than τ would mean that the interval $[p_i + 1 \dots p_{i+1} - 1]$ is greater than τ. But Construction 1 picks a minimum $\texttt{id}$-value in every interval of length τ, so this can never happen.

If the string contains $(\tau, \frac{\tau}{6})$-runs these are handled by making the first and last index of the run part of $\mathcal{P}$ and the argument from before for non-periodic substring still holds even if the string is split into smaller chunks, when removing the $(\tau, \frac{\tau}{6})$-runs.

□

Having shown that Construction 1 is a partitioning set, leads us to the next topic: determining the size of this partitioning.

Lemma 4. *The (τ, τ)-partitioning set yielded by Construction 1 has expected size $\mathcal{O}\left(\frac{n}{\tau}\right)$.*

The following proof is based on the proof of Lemma 4.3 in [5].

Proof. Let A_i be the event that index i is chosen as part of the (τ, τ)-partitioning set, $\mathcal{P}$. Furthermore, assume that the interval $S[i - \tau \dots i + \tau]$ is without a $(\tau, \frac{\tau}{6})$-run. This means that there will be at least $\frac{\tau}{6}$ different labels in the interval $S[i - \tau \dots i + \tau]$.

We assume that $i \in \mathcal{P}$ and since i is not part of a $(\tau, \frac{\tau}{6})$-run this means that $\texttt{id}(i)$ needs to be the smallest in some interval of length τ. This interval would either overlap the interval $[i - \frac{\tau}{2} \dots i]$ or the interval $[i \dots i + \frac{\tau}{2}]$ (or both). Since the $\texttt{id}$-function is designed to label indices completely at random and there are at least $\frac{\tau}{6}$ different labels, the chance that $\texttt{id}(i)$ is minimal in either of those two intervals is $2 \cdot (\frac{6}{\tau}) = \mathcal{O}\left(\frac{1}{\tau}\right)$. Then, to get the expected number of elements in $\mathcal{P}$, we use linearity

of expectation and get

$$\mathbb{E}(|\mathcal{P}|) = \sum_{i=1}^{n} A_i = \sum_{i=1}^{n} \mathcal{O}\left(\frac{1}{\tau}\right) = \mathcal{O}\left(\frac{n}{\tau}\right) \tag{2.2}$$

Now we have shown that if S is a string without $(\tau, \frac{\tau}{6})$-runs, then the expected size of $\mathcal{P}$ is $\mathcal{O}\left(\frac{n}{\tau}\right)$. If S contained $(\tau, \frac{\tau}{6})$-runs, these runs would be partitioned into blocks of size greater than τ per the definition of locally consistent sets Definition 5.

Conclusively, we can therefore say that the non-periodic parts of S will be partitioned into blocks of expected size $\mathcal{O}(\tau)$, and the periodic parts are guaranteed to be partitioned into blocks of size $\Omega(\tau)$. Hence the expected size of $\mathcal{P}$ remains $\mathcal{O}\left(\frac{n}{\tau}\right)$. □

We have shown that the set constructed by Construction 1 has expected size $\mathcal{O}\left(\frac{n}{\tau}\right)$, and now we want to show that the construction is expected to be linear in the size of the string S.

Lemma 5. *The (τ, τ)-partitioning set yielded by Construction 1 of expected size $\mathcal{O}\left(\frac{n}{\tau}\right)$ can be constructed in expected $\mathcal{O}(n)$.*

The following proof is based on lemma 4.3 in [5].

Proof. We assume that the `id`-function returns the `id`-value of an index in $\mathcal{O}(1)$ time. Let us assume that the string S is traversed from left to right and p_i is the latest value added to $\mathcal{P}$. During Construction 1 we are constantly looking at some interval (the sliding window) of size τ, and since p_i is in $\mathcal{P}$, $\texttt{id}(p_i)$ must be a minimum in some interval of length τ. After processing this particular interval (and adding p_i to $\mathcal{P}$), the window, $[k \dots k+\tau-1]$, is shifted one position to the right. Then the value of $\texttt{id}(k+\tau)$ is compared to that of $\texttt{id}(p_i)$. If $\texttt{id}(k+\tau)$ is smaller than $\texttt{id}(p_i)$ the index $k+\tau$ is added to $\mathcal{P}$ otherwise $\texttt{id}(p_i)$ is also the minimum in the window $[k+1 \dots k+\tau]$. In this way, we are moving linearly through the string.

Nevertheless, there might be a problem if the window is moved so much that p_i is no longer in the interval, but no new value has been added to $\mathcal{P}$ since p_i was added. Then all the values of the interval $[p_i+1 \dots p_i+\tau]$ must be recomputed and a minimum in this interval is then found in $\mathcal{O}(\tau)$.

Now let the cost of this recomputation be "paid" by p_i. This can only happen once for p_i because after the recomputation, a new minimum is found and p_i is no longer the last added element of $\mathcal{P}$. This means that the total runtime is $\mathcal{O}(n + \tau \cdot |\mathcal{P}|) = \mathcal{O}(2n) = \mathcal{O}(n)$. □

Combining *Lemma* 4 and Lemma 5, and using Markov's inequality, we get the following corollary, which shows that only the time needs to be expected.

Corollary 5.1. *A (τ, τ)-partitioning set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ can be constructed in expected linear time.*

Proof. From Lemma 4 and Lemma 5, we know that we can create a (τ, τ)-partitioning set using Construction 1, with expected size $\mathcal{O}\left(\frac{n}{\tau}\right)$. This mean that the expected size of $\mathcal{P}$ is $c \cdot \frac{n}{\tau}$ for some constant c. Using Markov's inequality[3] we get that the change of $|\mathcal{P}|$ being larger than $3c\frac{n}{\tau}$ is

$$\mathbb{P}\left[|\mathcal{P}| > \frac{3cn}{\tau}\right] < \frac{cn/\tau}{3cn/\tau} = \frac{1}{3}.$$

We can draw a new `id`-function (remember that part of the `id`-function is a hash-function from a family of hash-functions with the property we wanted). This new function will also have more than $\frac{1}{3}$ change of creating a $\mathcal{P}$ of size less than $\frac{3cn}{\tau}$. Drawing `id`-functions can be viewed as a geometric

[3] Markov's inequality is given by $\mathbb{P}[X > a] < \frac{\mathbb{E}[X]}{a}$.

variable with more than $\frac{1}{3}$ chance of success. Thus, the expected number of `id`-functions we need to try is three. We expect to use a constant number of tries before we have a (τ, τ)-partitioning set of the wished size. Each try takes $\mathcal{O}(n)$ according to Lemma 5 (and if we start over as soon as the $\mathcal{P}$ gets bigger than $\frac{3cn}{\tau}$). □

2.3.1.1 Conclusion on the randomised construction of a (τ, τ)-partitioning set

We have shown that the randomised construction introduced by Construction 1 yields a valid (τ, τ)-partitioning set by satisfying both the locality and density conditions required by Definition 5. Further, we demonstrated in Lemma 4 that the expected size of the constructed set is $\mathcal{O}\left(\frac{n}{\tau}\right)$, and in Lemma 5 that the construction runs in expected linear time with respect to the input size. Combining these results, Corollary 5.1 shows that the construction yields a (τ, τ)-partitioning set of optimal expected size in expected $\mathcal{O}(n)$ time, with high probability.

2.3.2 Deterministic construction $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set

This constructions returns a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set and was also presented by Birenzwige et al. [5]. We will go through the construction (which is quite technical), then show that it yields a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set, and then lastly we will show that this partitioning set has size $\mathcal{O}\left(\frac{n}{\tau}\right)$ and is constructed in $\mathcal{O}(n \log \tau)$ using only $\mathcal{O}\left(\frac{n}{\tau} + \log \tau\right)$ construction space (or alternatively $\mathcal{O}(n \log \sigma)$ using $\mathcal{O}(n)$).

This construction is hierarchical, as described previously in Section 2.2, which means it partitions each level into blocks using the sub-blocks from the previous level. The idea behind the construction is to look at all the sub-blocks in each level $\mu - 1$, categorize the blocks into four types, and then merge (or leave alone) the sub-blocks such that they have the right size for level μ while still living up to the criteria of Definition 5 for a $(\delta, \delta, \tau, \tau)$-locally consistent set.

There are four types of sub-blocks:

- *Type 1:* A sub-block with size at least $(3/2)^\mu$.
- *Type 2:* A sub-block that is part of a sequence of identical sub-blocks all smaller than $(3/2)^\mu$.
- *Type 3:* A sub-block that is part of a sequence of $c \log^* n$ or more sub-blocks[4] which are neither type 1 nor type 2.
- *Type 4:* A sub-block that is part of a sequence of less than $c \log^*$ sub-blocks which are not type 1 nor type 2.

When a sub-block or a position is referred to as *chosen*, it means that it will mark the beginning of a new block at the level above.

Construction 2. At the bottom layer of the partitioning set, every character of the string is considered a block. Now, at every level $\mu - 1$, the sub-blocks are categorised as one of the four block types and then treated in the following way:

- *Type 1*: A sub-block of this type is moved directly to level μ without any alteration.
- *Type 2*: A sub-block of type 2 is merged with all the identical sub-blocks to the right and left of the block, creating one big block, which is moved to level μ.
- *Type 3*: A sequence of type 3 sub-blocks is merged in the following way. The first (leftmost)

[4]The c comes from the colouring algorithm Section 1.2.6, c is the constant which is hidden in $\mathcal{O}$-notation of $\mathcal{O}(\log^* n)$.

sub-block is always chosen; subsequently, the second leftmost sub-block is not chosen.

Moving on to the general case, every sub-block gets a label depending on the elements it contains. Using the colouring algorithm labelling method from Section 1.2.6, we use $\mathcal{O}(\log^* n)$ time to reduce the labels of each sub-block such that all the labels are from $\{0,1,2,3,4,5\}$. Then, since no sub-block has the same label as its neighbours, it will be possible to find valleys (sub-blocks with labels strictly smaller than both of its neighbours) and then merge the "valley" sub-block with the neighbours to its right (until another locally minimal label is found). Now all the new blocks created by this method are moved to level μ.

- *Type 4*: If a sub-block of type 4 is in a sequence with one or more other sub-blocks of its type, the blocks are merged pairwise from right to left, and each new sub-block is moved to level μ. If there is an uneven number of blocks in the sequence, the leftmost three sub-blocks of type 4 are merged. If a sub-block of type 4 has no neighbours of its own type, it is moved to level μ regardless of its size.

We will move on to show that Construction 2 indeed produces a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set. To show the locality condition (of Definition 5), we will need some definitions and an additional lemma. We will begin by introducing a shorthand for a specific range, which will be used repeatedly throughout this section.

Definition 8. Let $S_{i,\mu} = S[i - 6(3/2)^\mu c \log^* n \dots i + 6(3/2)^\mu c \log^* n]$ (illustrated in Figure 10).

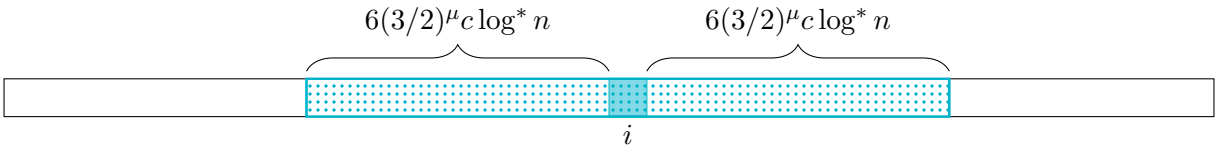


Figure 10: The blue (dotted) area corresponds to $S[i - 6(3/2)^\mu c \log^* n \dots i + 6(3/2)^\mu c \log^* n]$.

With this definition in place, we define the partitioning at a given level in the following way.

Definition 9. Let $\mathcal{P}_\mu$ be the partitioning at level μ, and let $\mathcal{P} = \mathcal{P}_{\log_{3/2}\tau}$.

Now we introduce a rather technical Lemma, which states that if $S_{i,\mu} = S_{j,\mu}$. Then for any δ in the interval $[-2(3/2)^\mu c \log^* n \dots 2(3/2)^\mu c \log^* n]$, it holds that $S_{i,\mu-1} = S_{j,\mu-1}$.

Lemma 6. *Let $S_{i,\mu} = S_{j,\mu}$ for level $1 \le \mu \le \log_{3/2}\tau$ of the hierarchical construction then for any $\delta \in [-2(3/2)^\mu c \log^* n \dots 2(3/2)^\mu c \log^* n]$ we have $S_{i+\delta,\mu-1} = S_{j+\delta,\mu-1}$.*

Proof. Recall that

$$S_{i,\mu} = S[i - 6(3/2)^\mu c \log^* n \dots i + 6(3/2)^\mu c \log^* n],$$

and let $S_{i,\mu} = S_{j,\mu}$. Now, looking at $\delta = -2(3/2)^\mu c \log^* n$ we get

$$\begin{aligned} S_{i+\delta,\mu-1} &= [i - 2(3/2)^\mu c\log^* n - 6(3/2)^{\mu-1} c\log^* n \dots i - 2(3/2)^\mu c\log^* n + 6(3/2)^{\mu-1} c\log^* n] \\ &= [i + (-2-4)(3/2)^\mu c\log^* n \dots i + (-2+4)(3/2)^\mu c\log^* n] \\ &= [i - 6(3/2)^\mu c\log^* n \dots i + 2(3/2)^\mu c\log^* n] \subseteq S_{i,\mu} \end{aligned}$$

and since $S_{i,\mu} = S_{j,\mu}$ then $S_{i+\delta,\mu-1} = S_{j+\delta,\mu-1}$ for $\delta = -2(3/2)^\mu c\log^* n$.

Now, we move on to $\delta = 2(3/2)^\mu c\log^* n$, and get the almost similar result that

$$\begin{aligned} S_{i+\delta,\mu-1} &= [i + 2(3/2)^\mu c\log^* n - 6(3/2)^{\mu-1} c\log^* n \dots i + 2(3/2)^\mu c\log^* n + 6(3/2)^{\mu-1} c\log^* n] \\ &= [i + (2-4)(3/2)^\mu c\log^* n \dots i + (2+4)(3/2)^\mu c\log^* n] \\ &= [i - 2(3/2)^\mu c\log^* n \dots i + 6(3/2)^\mu c\log^* n] \subseteq S_{i,\mu}. \end{aligned}$$

The same argument holds that because $S_{i+\delta,\mu-1} \subseteq S_{i,\mu}$ then $S_{i+\delta,\mu-1} = S_{j+\delta,\mu-1}$ for $\delta = 2(3/2)^\mu c\log^* n$. It is evident that for $-2(3/2)^\mu c\log^* n < \delta < 2(3/2)^\mu c\log^* n$, this claim also holds, since it holds for the two most extreme values of δ and the value of δ is used to move the interval $S_{i+\delta,\mu-1}$ either to the left or the right (between the two extremes). □

Using Lemma 6, we continue to show the locality condition for the partitioning set built by Construction 2.

Lemma 7. *[Lemma D.2. [5]] For any level $0 \le \mu \le \log_{3/2}\tau$ of the hierarchical construction, if $i, j \in [n]$ are two indices such that $S_{i,\mu} = S_{j,\mu}$ then $i \in \mathcal{P}_\mu \Leftrightarrow j \in \mathcal{P}_\mu$.*

The following proof is based on the proof of Lemma D.2. by Birenzwige et al. [5].

Proof. We will prove this by induction on the level of the hierarchical construction. The base case is level $\mu = 0$ where $\mathcal{P}_0 = [n]$, and the Lemma trivially holds because all positions are selected for $\mathcal{P}_0$. We assume that the Lemma holds for level $\mu - 1$, and we will now show that it holds for level μ.

Let $S_{i,\mu} = S_{j,\mu}$ and $i \in \mathcal{P}_\mu$. We begin by showing that because $S_{i,\mu} = S_{j,\mu}$ then $i \in \mathcal{P}_{\mu-1}$ and $j \in \mathcal{P}_{\mu-1}$. The reason $i \in \mathcal{P}_{\mu-1}$ is that $\mathcal{P}_\mu \subseteq \mathcal{P}_{\mu-1}$ because the indices chosen to be in $\mathcal{P}_\mu$ are all taken from $\mathcal{P}_{\mu-1}$. We have assumed that $S_{i,\mu} = S_{j,\mu}$, and since $S_{i,\mu-1} \subset S_{i,\mu}$ and $S_{j,\mu-1} \subset S_{j,\mu}$, $S_{i,\mu-1} = S_{j,\mu-1}$. From the induction hypothesis we have the fact that if $S_{i,\mu-1} = S_{j,\mu-1}$ then $i \in \mathcal{P}_{\mu-1} \Leftrightarrow j \in \mathcal{P}_{\mu-1}$. Thus $j \in \mathcal{P}_{\mu-1}$.

A more general observation can be made using Lemma 6. The Lemma tells us that if $S_{i,\mu} = S_{j,\mu}$ then for any $\delta \in [-2(3/2)^\mu c\log^* n \dots 2(3/2)^\mu c\log^* n]$ then $S_{i+\delta,\mu-1} = S_{j+\delta,\mu-1}$, which given the induction hypothesis of this proof means that $i + \delta \in \mathcal{P}_{\mu-1}$ if and only if $j + \delta \in \mathcal{P}_{\mu-1}$.

The rest of the proof goes through the four possible sequences of sub-blocks that i can be the leftmost index of at level μ, and argues why we also have $j \in \mathcal{P}_\mu$.

Type 1: If i is the leftmost index of a sub-block of type 1 at level $\mu - 1$, this would mean that at level μ the interval $[i \dots i + (3/2)^\mu - 1]$ would only contain i as an element from $\mathcal{P}_\mu$. Because $j \in \mathcal{P}_{\mu-1}$ and $S[i \dots i + (3/2)^\mu - 1] = S[j \dots j + (3/2)^\mu - 1]$ we know that the sub-block starting at j in level $\mu - 1$ will also be of type 1, which means that $j \in \mathcal{P}_\mu$.

For all remaining sequence types, the block for which i is the leftmost index consists of sub-blocks, each of size less than $(3/2)^\mu$. We denote the length of the sub-block at level $\mu - 1$ that i is the leftmost index of for δ.

Type 2: The block beginning with i at level μ consists of a sequence of sub-blocks which are all identical. This sequence consists of at least two sub-blocks, and they have all length $\delta < (3/2)^\mu$.

This means that a sub-block begins at i, $i+\delta$ and $i+2\delta$. Also, since $i \in \mathcal{P}_\mu$, the sub-block before $[i \dots i+\delta-1]$ cannot be identical to it, because otherwise i would not be in $\mathcal{P}_\mu$.

From Lemma 6 we know that if both $i+\delta$ and $i+2\delta$ are in $\mathcal{P}_{\mu-1}$ then $j+\delta$ and $j+2\delta$ are also in $\mathcal{P}_{\mu-1}$.

Due to the induction hypothesis and Lemma 6, we know that $S[i-\delta \dots i-1] = S[j-\delta \dots j-1]$, and because the block to the left of $[i \dots i+\delta-1]$ was not of type 2, this means that the sub-block before $[j \dots j+\delta-1]$ is also not identical to the sub-block $[j \dots j+\delta-1]$. Therefore, we know that the index j is the first index in a sequence of sub-blocks of type 2, which means that $j \in \mathcal{P}_\mu$.

Type 3: Assume that the sub-block of i at level $\mu-1$ is not among the rightmost $c\log^* n$ sub-blocks. If it were one of those sub-blocks, it would be treated like a sub-block of type 4. Now let $p_0, p_1, \dots, p_h$ be the picked positions of level $\mu-1$ in ascending order. Then let the positions corresponding to i and j be $p_{i'}$ and $p_{j'}$. There are two cases: either the sub-block of $p_{i'}$ is the leftmost sub-block of a type 3 sequence, or the label $p_{i'}$'s block is a local minimum.

Let us begin with case one. If the sub-block of $p_{i'}$ is the leftmost sub-block of the sequence, there must be some block of another type to the left of i. If this is the case, there will be two identical blocks in the interval $[i-2(3/2)^\mu \dots i-1]$ of size $< (3/2)^\mu$ or a single large block which is $\geq (3/2)^\mu$. Since $S[p_{j'}-2(3/2)^\mu \dots p_{j'}-1] = S[p_{j'}-2(3/2)^\mu \dots p_{j'}-1]$ due to our original assumption, the sub-block of j at level $\mu-1$ will also be the leftmost sub-block of a sequence of type 3 sub-blocks. And thus $p_{j'} = j \in \mathcal{P}_\mu$.

In the second case, block $p_{i'}$ has a label that is minimal compared to its neighbouring blocks. From Lemma 6, we know that the blocks $p_{i'}$ and $p_{j'}$ are identical. Furthermore, for every $\delta \in [-2(3/2)^\mu c\log^* n,\ 2(3/2)^\mu c\log^* n]$, the positions $p_{i'}+\delta$ and $p_{j'}+\delta$ must both satisfy the locality constraint. This implies that if a position $p_{i'}+\delta$ is included in $\mathcal{P}$, then the corresponding position $p_{j'}+\delta$ must also be included, for the same values of δ. In other words, the $2c\log^* n$ blocks to the left and right of $p_{i'}$ and $p_{j'}$ are identical.

This also means that their initial labels will be identical. The colouring algorithm depends only on $c\log^* n$ blocks to the right of a block, and if these are completely identical for $p_{i'}$ and $p_{j'}$, these blocks will get the same label. Even more importantly, the blocks $p_{i'-1}$ and $p_{i'+1}$ will also have exactly the same property, and thus will get the same label as $p_{j'-1}$ and $p_{j'+1}$. Hence if $p_{i'} \in \mathcal{P}$ then will the label of $p_{j'}$ also be a local minimum and $p_{j'} = j \in \mathcal{P}$.

Type 4: As in the previous sequence type, we let $p_0, p_1, \dots, p_h$ be the picked positions of level $\mu-1$ in ascending order. Then let the positions corresponding to i and j be $p_{i'}$ and $p_{j'}$. There are two cases to consider. The sub-block starting at p_i is either the first block in a type 4 sequence or it has even parity within the sequence. Here, even parity means that if the sub-blocks of the type 4 sequence were enumerated from right to left (such that the rightmost block has index one), then the sub-block at p_i would have an even index.

A type 4 sequence is so small (it can only consist of less than $c\log^* n$ sub-blocks) that it will be squeezed in between sub-blocks of either type 1 or 2. This means that the entire sequence will be inside the interval of $S_{p_{i'},\mu}$ as well and therefore also inside $S_{p_{j'},\mu}$. The type 1 (or 2) sub-block left (and right) of the type 4 sequence will also end (or start) at the same relative position as they are also contained in the substring $S_{p_{i'},\mu}$. This means that if the sub-block of $p_{i'}$ is the first in the type 4 sequence, pj' will also be the first (because they both have either a sub-block of type 1 or two or more sub-blocks of type 2 to their left).

If the sub-block of $p_{i'}$ is not the first block in the sequence, it has to have even parity. But the end of the type 4 sequence is contained in $S_{p_{i'},\mu}$ which means it is also contained in $S_{p_{j'},\mu}$ and thus they must have the same parity (with regards to the type 4 sequence). And thus $p_{j'} = j \in \mathcal{P}$.

This shows that no matter which type block i is, then if $S_{i,\mu} = S_{j,\mu}$ and $i \in \mathcal{P}_\mu$, then j will also be in $\mathcal{P}_\mu$. □

The Lemma which was just proved (Lemma 7), can be used at any level $0 \geq \mu \geq \log_{3/2}\tau$. If we use it at level $\log_{3/2}\tau$ we see that if $S_{i,\log_{3/2}\tau} = S_{j,\log_{3/2}\tau}$ then $i \in \mathcal{P}$ if and only if $j \in \mathcal{P}$. The interval $S_{i,\log_{3/2}\tau}$ is equal to

$$S[i - 6(3/2)^{\log_{3/2}\tau} c\log^* n \dots i + 6(3/2)^{\log_{3/2}\tau} c\log^* n] = [i - 6\tau c\log^* n \dots i + 6\tau c\log^* n].$$

Therefore, since $6\tau c\log^* n = \mathcal{O}(\tau\log^* n)$, it follows that Lemma 7 implicitly demonstrates that the $(\mathcal{O}(\tau), \mathcal{O}(\tau\log^* n))$-partitioning set satisfies the locality condition. The next step is to prove that density condition a holds, in other words, that there is at most $\mathcal{O}(\tau)$ between two consecutive elements of $\mathcal{P}$ unless the substring between those elements is periodic. To do so, we again introduce an assisting lemma.

Lemma 8. *For any level $0 \leq \mu \leq \log_{3/2}\tau$ of the hierarchical construction, a block can only consist of more than 12 sub-blocks if those sub-blocks are all of type 2.*

Proof. A sub-block of type 1 at level $\mu - 1$ always corresponds to a single block at level μ. A sequence of sub-blocks of type 4 is merged into new blocks in a way that at most merges 3 sub-blocks simultaneously.

Now we look at how many sub-blocks of type 3 a block can consist of. Given a sequence of type 3 sub-blocks, the colouring (labelling) algorithm assigns a label from $\{0, 1, 2, 3, 4, 5\}$ to each block, and a block begins every time a label is a local minimum among its neighbours. Since the labelling in reality is a colouring, no two consecutive sub-blocks can have the same label, and thus the longest sequence without a local minimum is $\dots, 0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0, \dots$. A new block would begin at the sub-block with label 1, and hence there would be a block made out of 10 sub-blocks.

In Construction 2, the first sub-block of a sequence of type 3 blocks is always chosen, and consequently, the second sub-block can never be chosen. Therefore, if the maximal sequence appears at the second position in the sequence of type 3 sub-blocks, there can be a block which consists of 11 sub-blocks.

Now we can conclude that if there exists a block consisting of more than 12 sub-blocks, those sub-blocks must be of type 2. □

We have just proven that a block at level μ can only be made out of 12 or more sub-blocks if those sub-blocks are of type 2. This fact is essential when proving the following Lemma, from which we can derive the density condition of the $(\mathcal{O}(\tau), \mathcal{O}(\tau\log^* n))$-partitioning set.

Lemma 9 (Lemma D.3. [5])**.** *For any level $0 \leq \mu \leq \log_{3/2}\tau$ of the hierarchical construction, if $p_i < p_{i+1}$ are two consecutive positions in P_μ such that $p_{i+1} - p_i > 12 \cdot (3/2)^\mu$ then $S[p_i \dots p_{i+1} - 1]$ has period length smaller than $(3/2)^\mu$.*

The following proof is based on the proof of Lemma D.3. from [5].

Proof. This is proven by induction on the level of the hierarchical construction. We will assume that the Lemma holds for level $\mu - 1$ and then show that it is also true for level μ. The base case is $\mu = 0$ and there we have $\mathcal{P}_0 = [n]$, meaning that all blocks are precisely of length 1.

Let us assume that we have some block at level μ which is bigger than $12 \cdot (3/2)^\mu$. Now if this block only consists of a single sub-block we are done due to the induction hypothesis which tells us that any sub-block at level $\mu - 1$ has period length of at most $(3/2)^{\mu-1}$ if it is bigger than $12 \cdot (3/2)^{\mu-1}$ which is strictly smaller than $12 \cdot (3/2)^\mu$.

Let us therefore assume that this block consists of several sub-blocks. Since it consists of more than one sub-block, none of these can be of type 1, and must hence all be strictly smaller than $(3/2)^\mu$. From Lemma 8 it is known that only blocks made out of type 2 sub-blocks can consist of more than 12 sub-blocks. Therefore, all blocks bigger than $12 \cdot (3/2)^\mu$ will be periodic with period strictly smaller than $(3/2)^\mu$. □

Lemma 9 shows that at the final level of Construction 2, namely level $\log_{3/2}\tau$, the distance between two consecutive elements in $\mathcal{P}$ exceeds $12\cdot(3/2)^{\log_{3/2}\tau} = 12\tau = \mathcal{O}(\tau)$ only if the substring between them has principal period $\leq (3/2)^{\log_{3/2}\tau} = \tau$. This is exactly density condition a from Definition 5 applied to the $(\mathcal{O}(\tau), \mathcal{O}(\tau\log^* n))$-partitioning set.

The last thing we need to settle is the complexity of Construction 2. We begin by looking at the size of the partitioning.

Lemma 10. *[Lemma D.1. [5]] For any level $0 \leq \mu \leq \log_{3/2}\tau$ of the hierarchical construction, the length of any pair of two consecutive blocks is at least $(3/2)^\mu$.*

The following proof is based on the proof of Lemma D.1 from [5].

Proof. This proof is done by induction on the level, denoted μ, of the hierarchical construction. The base case is $\mu = 0$, meaning every block should have at least size $(3/2)^0 = 1$. This is trivially correct, since at level 0, all characters form individual blocks.

Now we assume that at level $\mu-1$ the length of any pair of two consecutive sub-blocks is at least $(3/2)^{\mu-1}$. We begin the proof by looking at an arbitrary pair of blocks at level μ. We will go through the different sub-block types that can form the pair. If one or both of these two blocks are of type 1, the proof is done because this block has on its own size $(3/2)^\mu$. If both blocks in the pair originate from two or more sub-blocks the proof is also done, because we know from the induction hypothesis that all consecutive pairs at level $\mu-1$ in total has size $(3/2)^{\mu-1}$ and hence the two blocks we are looking, at will together have size $2\cdot(3/2)^{\mu-1} \geq (3/2)(3/2)^{\mu-1} = (3/2)^\mu$.

There is only a problem if one of the two blocks originates from a single sub-block of size less than $(3/2)^{\mu-1}$. Let us denote the sub-blocks of this pair A and B such that A is the leftmost block of the two. Assume, without loss of generality, that it is block A which originates from a single "small" sub-block. This sub-block would have to be type 4 since all other sub-blocks are merged with at least one other sub-block. Now, if block B is of type 1, we are again done with the proof. Block B cannot be type 3 or type 4 because then the sub-block of block A will not be a stand-alone block. This means block B must be of type 2.

This tells us that block B consists of two or more identical sub-blocks. Let us denote the size of these sub-blocks x (remember that all the sub-blocks of type 2 have the same size). From the induction hypothesis we know that $2x \geq (3/2)^{\mu-1}$ which means that $x \geq \frac{(3/2)^{\mu-1}}{2}$. Let us denote the size of block A's single sub-block y. We know that $x + y \geq (3/2)^{\mu-1}$ from the induction hypothesis (this is also illustrated in Figure 11).

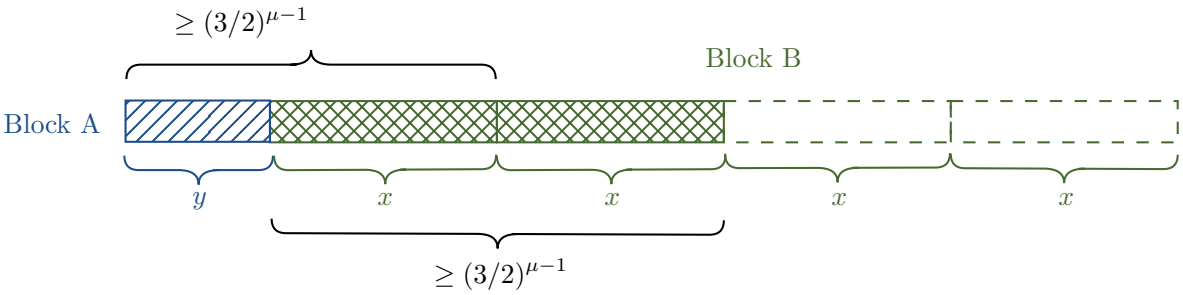


Figure 11: Illustration shows block A (in blue cross-lines) and block B (in green double cross-lines) and what is known about the size of sub-blocks x and y from the induction hypothesis.

The combined lengths of block A and B are at least $2x + y$. Using what we know about x and y we get:

$$2x + y = (x + y) + x \geq (3/2)^{\mu-1} + \frac{(3/2)^{\mu-1}}{2} = (3/2)^{\mu-1}\left(1 + \frac{1}{2}\right) = (3/2)^{\mu-1}\cdot(3/2) = (3/2)^\mu,$$

and hence we conclude that for any consecutive pairs of blocks their combined length will be $(3/2)^\mu$ or greater. □

From Lemma 10 we learn that at level $\log 3/2\tau$ any pair of consecutive blocks has at least size $(3/2)^{\log 3/2\tau} = \tau$. This means that the number of blocks at the last level is $\leq \frac{n}{\tau}$, which can be upper-bounded by $\mathcal{O}\left(\frac{n}{\tau}\right)$. In conclusion is $\mathcal{P}$ of size $\mathcal{O}\left(\frac{n}{\tau}\right)$.

In the next section, we will explain how to build Construction 2 using only $\mathcal{O}\left(\frac{n}{\tau} + \log \tau\right)$ working space. This is possible using label lists and efficiently computing those (both ideas presented by Birenzwige et al. [5]).

2.3.2.1 Label lists and efficiently computing them

This construction can be used recursively and in such a way that we only need six active blocks in each level to compute the whole partitioning. Essential to this are the label lists. Intuitively, these lists are empty for type 1 blocks and almost all type 2 blocks (except the rightmost block in a sequence of type 2 blocks). Otherwise, they contain all label/colour computations performed by the colouring algorithm on some block B at some level μ of the hierarchical construction.

More formally, let every block, $B = S[i \dots j]$, at every layer have a label list, denoted L_B. Let B' denote the neighbour block to the right of B. If B is a block of type 1 $L_B = \emptyset$, and if block B is identical to B', it also get an empty label list. If the list is non-empty then its size is $|L_B| = \min\{|L_{B'}| + 1, c \log^* n\}$.

Otherwise, the label list consists of all the iterations of the colouring algorithm. We define $L_B[k]$ as the kth label of block B when running the colouring algorithm. $L_B[0] = S[i \dots j]$ and this is stored implicitly (as the only label of the list) by just storing the indices i and j. The label $L_B[1]$ is computed linearly in the length of block B and its right neighbour B'. This label has size $\log(n)$ bits. The size of the labels decreases logarithmically (due to the colouring algorithm defined in Section 1.2.6). The label list is therefore dominated by the first explicitly stored label $L_B[1]$, and in total, the label list takes up $\mathcal{O}(\log n)$ bits, which is $\mathcal{O}(1)$ machine words.

The next step is to show how to efficiently compute the label lists. We only need to do three iterations of the colouring algorithm, because given the entire label list of a block to the right of the current block, it is possible to calculate the label list of the current block. After the first three iterations, the label $L_B[3]$ takes up at most $\mathcal{O}(\log \log \log n)$ bits. When the first three elements are removed, the label list of $L_{B'}$ also takes up $\mathcal{O}(\log \log \log n)$. Now we want to create pairs of possible "third labels" of block B (labels of block B after three iterations of the colouring algorithm) and lists of B' without the first three elements.

The possible number of pairs is at most $2^{\mathcal{O}(\log \log \log n)} = (\log \log n)^{\mathcal{O}(1)} = \mathcal{O}(\log^a \log n)$ for some constant a. All such pairs are precomputed and stored in a look-up table. Since a pair takes up $\mathcal{O}(\log \log \log n)$ bits, the table will have size $\mathcal{O}(\log^a \cdot \log n \cdot \log \log \log n)$ bits and a constant look-up time.

This means that to compute the label list of a block B, it is only necessary to first look at its size, then compare it to the block to its right, running the first three iterations of colouring algorithm (using the three first labels of $L_{B'}$) and then look up in the table using the the label after the third iteration and the unused part of $L_{B'}$.

Lemma 11. *Total space of the construction is* $\mathcal{O}\left(\frac{n}{\tau} + \log \tau\right)$

The following proof sketch is based on section D.2 in [5].

Proof sketch. With the aid of the label list and the look-up table described earlier, the entire label list of a sub-block B can be computed using only the label list of the neighbouring sub-block to the right. Let us imagine that we have done this for six sub-blocks. Now we argue that if we are at level

$\mu - 1$ and have the label list for these six sub-blocks, it is enough to determine whether or not the second rightmost sub-block (let it be denoted B_5) is going to be the first sub-block in a new block at level μ. The idea is to figure out the type of sub-block B_5 and its relation to the three sub-blocks to its left and the sub-block to its right. This process is shown in Figure 12, and the figure is explained in Appendix A.1.

The hierarchical construction is done recursively from the bottom and up while simultaneously going from right to left. The only level where we keep more than $\mathcal{O}(1)$ blocks is the top level, where we need to keep all the indices of the partitioning set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$. This means that the space used by this construction is $\mathcal{O}(1)$ at each of the $\mathcal{O}(\log \tau)$ levels plus $\mathcal{O}\left(\frac{n}{\tau}\right)$, which makes the final working space $\mathcal{O}\left(\log \tau + \frac{n}{\tau}\right)$. □

Lemma 12. *The runtime of Construction 2 using is* $\mathcal{O}(n \log \tau)$.

Proof. At every level, finding all the periodic blocks and making the first iteration of the colouring algorithm requires a comparison between a block and the block to its right. This takes $\mathcal{O}(n)$. The rest of the colouring algorithm's rounds make $\log n + \log \log n + \cdots + \log^* n$ comparisons, which is also $\mathcal{O}(n)$. The last part of the algorithm is to find the local minimal labels (in the sequences of type 3 sub-blocks). At level μ, this takes constant time for each block, which means the total time is $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$. This is also $\mathcal{O}(n)$. There are $\mathcal{O}(\log \tau)$ levels in Construction 2 which means the time spent in total is $\mathcal{O}(n \log \tau)$. □

2.3.2.2 Alternative construction time

If we are allowed to use $\mathcal{O}(n)$ space while constructing the $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set, then we can reduce the runtime to $\mathcal{O}(n \log \sigma)$ instead of $\mathcal{O}(n \log \tau)$ where σ is the size of the alphabet Σ.

It requires that Construction 2 is changed slightly. At every level $\mu - 1$ after creating the new blocks for level μ we use $\mathcal{O}\left(\frac{n}{(3/2)^{\mu-1}} \cdot \log \sigma\right)$ time on labelling all the new blocks uniquely such that all identical blocks gets the same label. We do this by inserting all the sub-block labels from level $\mu - 1$ in a trie, T. Every time we get to a new block (in level μ), we check the string made out of the labels of the characters of the block's sub-blocks from level $\mu - 1$. If this string is already in T, we give the block the same label as the already inserted string (this can be stored as satellite data); otherwise, we label the block with an unused character.

At level μ there are at most $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$ blocks which means there will at most be needed $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$ labels. This means that identifying all periodic sequences at level $\mu + 1$ can be done in $\mathcal{O}\left(\frac{n}{(3/2)^\mu}\right)$ instead of $\mathcal{O}(n)$ from before. We claim that this runtime can be improved by partitioning at level μ, then using $\mathcal{O}\left(\frac{n}{(3/2)^\mu} \cdot \log \sigma\right)$ time before moving to level $\mu+1$. The problem with this new construction is that it is impossible to keep the working space at $\mathcal{O}\left(\log \tau + \frac{n}{\tau}\right)$. Even if we discard the trie of each level once it is no longer needed, the tries built at the lower levels are too large to stay within the desired space bound of $\mathcal{O}\left(\log \tau + \frac{n}{\tau}\right)$.

2.3.2.3 Conclusion on the deterministic construction of a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set

We have showed how to build a deterministic $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set which fulfils both the locality condition and density condition a for a $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set, which is in fact a $(\mathcal{O}(\tau \log^* n), \mathcal{O}(\tau \log^* n), \mathcal{O}(\tau), \tau)$-locally consistent set. Furthermore, this set has been shown to have of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ and that Construction 2 run in $\mathcal{O}(n \log n)$ time using $\mathcal{O}\left(\frac{n}{\tau} + \log \tau\right)$ space. Lastly, we presented an alternative construction uses $\mathcal{O}(n)$ space during construction but takes only $\mathcal{O}(n \log \sigma)$ time.

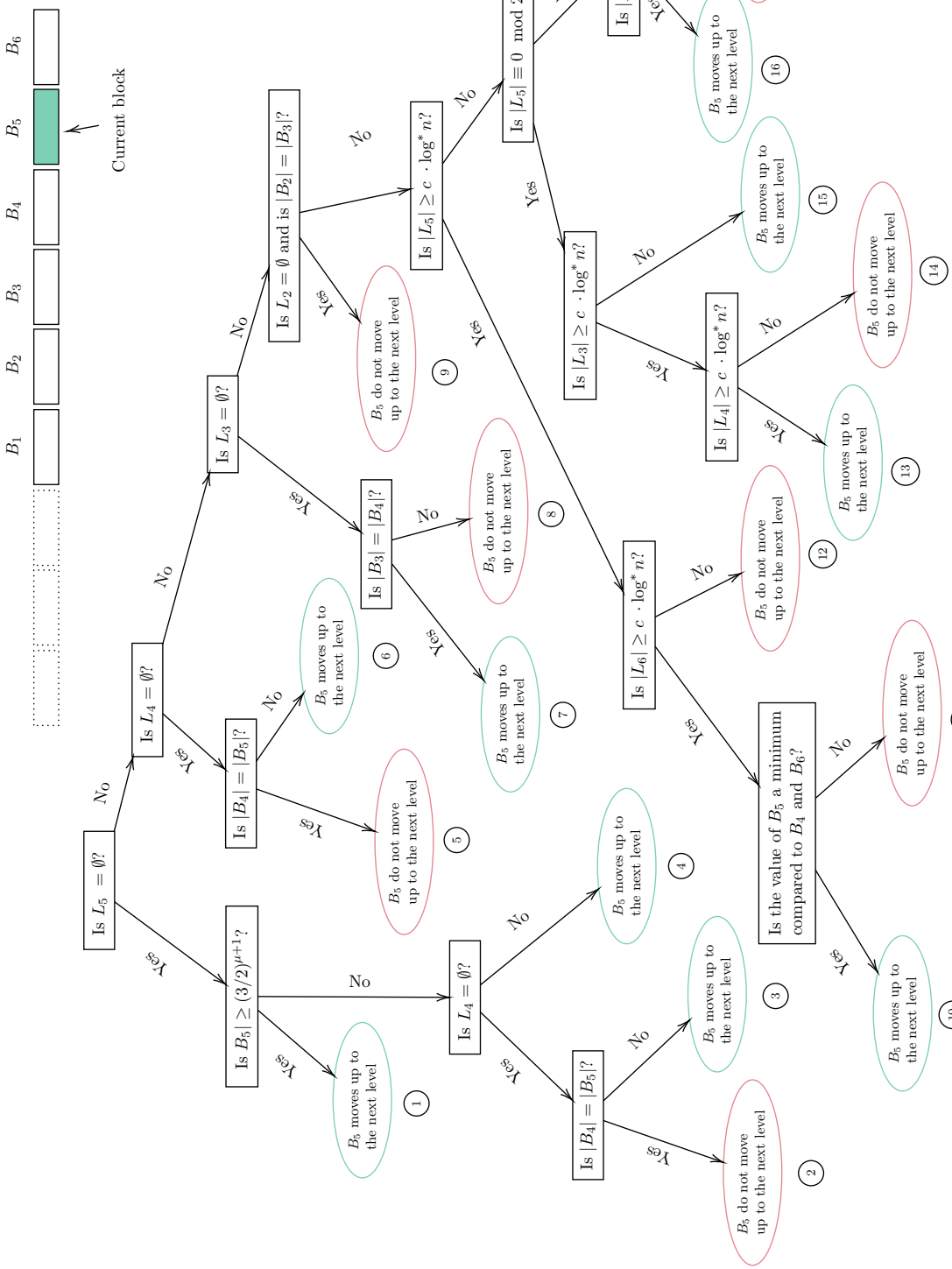


Figure 12: B_i denotes block $i \in [6]$ and is one of the six active blocks at some level $\mu - 1$. All the label lists of these 6 blocks (L_i refers to the label list of block i) have been computed already. $|B_i|$ means the size of block i.. Now, deciding whether block 5 should move up to level $\mu + 1$ can be done by starting at the root of the directed tree (the question *Is* $L_5 = \emptyset$ *?*), moving through the tree. The number underneath each leaf corresponds to an explanation of the statement of the leaf, which can be found in Appendix A.1.

2.4 τ-synchronising set

We now turn our attention to another locally consistent partitioning. It was introduced by Kempa and Kociumaka [22] and called a τ-synchronising set. We sometimes refer to it as $\mathcal{S}$. In this section, we will begin by defining this partitioning, then define a general construction, which can be used to make both a randomised and a deterministic construction of a τ-synchronising set. We then prove the locality and density condition of this general construction and move on to show the complexities of the randomised and then the deterministic implementations of the general construction.

This partitioning corresponds to a $(0, 2\tau-1, \tau, \frac{1}{3}\tau)$-locally consistent set with density condition b and the set $\mathcal{R}_b$ is called $\mathcal{R}$ and defined as follows

$$\mathcal{R} = \{i \in [1 \dots n-3\tau+2] : \mathrm{per}(T[i \dots i+3\tau-2]) \leq \frac{1}{3}\tau\}. \tag{2.3}$$

The locality constraint is illustrated in Figure 13, and the density condition b is illustrated in Figure 14.

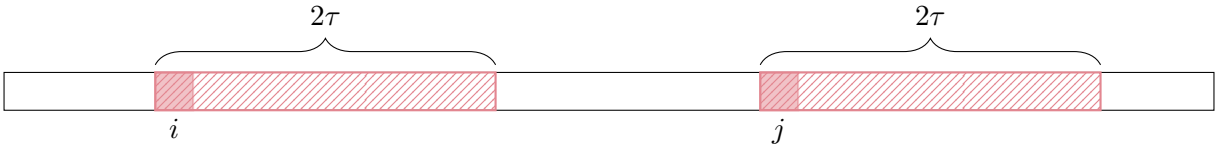


Figure 13: *The locality condition of τ-synchronising set*: If the substrings $S[i \dots i+2\tau)$ and $S[j \dots j+2\tau)$ are identical they should either both be in $\mathcal{S}$ or both not be in the set, according to the definition of a τ-synchronising set (Definition 5 and $(0, 2\tau-1, \tau, \frac{\tau}{3})$-locally consistent set with density condition b) .

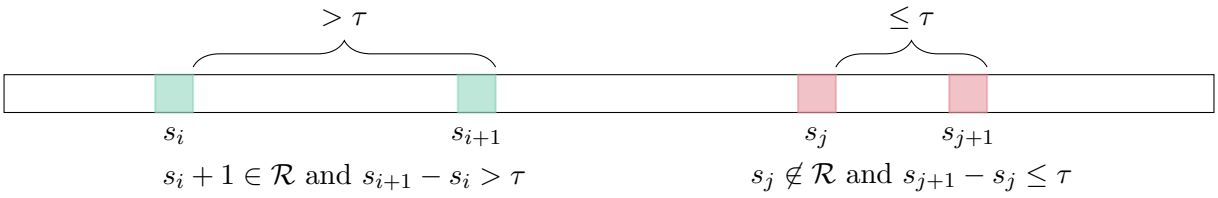


Figure 14: *The density condition of τ-synchronising set*: Since τ-synchronising set is $(0, 2\tau-1, \tau, \frac{\tau}{3})$-locally consistent set with density condition b, the distance between two consecutive elements, s_i and s_{i+1}, of $\mathcal{S}$ is $> \tau$ if and only if $s_i + 1 \in \mathcal{R}$.

Both in the randomized and the deterministic construction of an τ-synchronising set we need the axillary sets $\mathcal{Q}$ and $\mathcal{U}$ defined the following way,

$$\mathcal{Q} = \left\{ i \in [1 \dots n-\tau+1] : \mathrm{per}\left(S[i \dots i+\tau) \leq \frac{1}{3}\tau\right)\right\} \tag{2.4}$$

and

$$\mathcal{U} = \left\{ i \in [1 \dots n-\tau+1] \backslash \mathcal{Q} : \mathrm{per}(S[i \dots i+\tau-1)) \leq \frac{1}{3}\tau \text{ or } \mathrm{per}(S[i+1 \dots i+\tau)) \leq \frac{1}{3}\tau \right\}. \tag{2.5}$$

An index i is in $\mathcal{Q}$ if at i there is a substring of length τ with a period smaller than or equal to $\frac{1}{3}\tau$. The set $\mathcal{U}$ marks the borders of $\mathcal{Q}$, meaning that for every interval of indices belonging to $\mathcal{Q}$, there will be one index belonging to $\mathcal{U}$ on every side of this interval. This is shown in Figure 15.

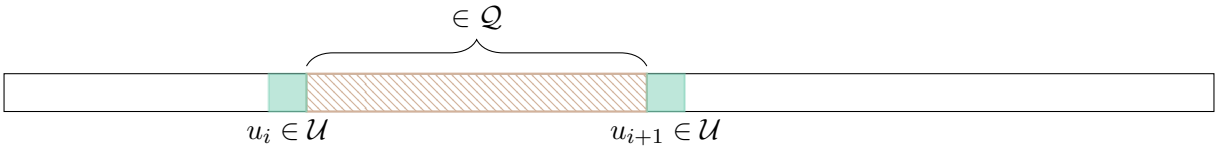


Figure 15: This figure shows how at any interval with elements from $\mathcal{Q}$ the element before and after will belong to $\mathcal{U}$.

Now we can define a more abstract construction (which is in reality a definition of a set) used by both the randomised and the deterministic constructions. Once again, we use the `id`-function (Definition 3) and the construction works by adding an index i to the τ-synchronising set if either $\texttt{id}(i)$ or $\texttt{id}(i+\tau)$ is a local minimum in the range from i to $i+\tau$. It is furthermore required that the local minimum is not an element of $\mathcal{Q}$ defined in (2.4). The more formal (but identical) definition of the general construction is as follows:

Construction 3. Let S be a string of length n. Then we build a τ-synchronising set by creating the set

$$\mathcal{S} = \{i \in [1 \dots n - 2\tau + 1] : \min\{\texttt{id}(j) : j \in [i \dots i + \tau] \setminus \mathcal{Q}\} \in \{\texttt{id}(i), \texttt{id}(i + \tau)\}\}. \tag{2.6}$$

To prove Construction 3 yields a τ-synchronising set, we will introduce a second definition of $\mathcal{R}$ which is equivalent to the first but uses the definition of $\mathcal{Q}$, (2.4). This definition shows very nicely the relationship between the two sets.

Definition 10. Let the set $\mathcal{Q}$ be defined as in (2.4) then the set $\mathcal{R}$ can be defined as

$$\mathcal{R} = \{i \in [1 \dots n - 3\tau + 2] : [i \dots i + 2\tau) \subseteq \mathcal{Q}\} \tag{2.7}$$

The next step is then to prove that a set, $\mathcal{S}$, made with Construction 3 will be a τ-synchronising set.

Lemma 13. *Construction 3 will always return a τ-synchronising set.*

Proof. Let $\mathcal{S}$ denote the τ-synchronising set created by Construction 3. Assume that $S[i \dots i + 2\tau) = S[j \dots j + 2\tau)$. We will show that $i \in \mathcal{S}$ if and only if $j \in \mathcal{S}$. We need to argue two things.

If $i + k \in \mathcal{Q}$ for some $k \in [0 \dots \tau]$ then $i + k$ is a periodic substring of length τ with principal period smaller than or equal to $\frac{\tau}{3}$. Therefore, looking at the substring $S[i + k \dots i + k + \tau - 1]$ is enough to determine whether $i + k$ is in $\mathcal{Q}$ or not. This substring is, due to our assumptions, identical to the substring $S[j + k \dots j + k + \tau - 1]$ (still for $k \in [0 \dots \tau]$). Thus, we can conclude that if any $i + k \in Q$ for $k \in [0 \dots \tau]$ then for the same particular k we will find $j + k \in \mathcal{Q}$. The sets of $i + k$ values and $j + k$ values are illustrated in Figure 16.

The second argument is that if i or $i + \tau$ has minimal `id`-values in the interval $S[i \dots i + \tau]$ then since those values are identical to $S[j \dots j + \tau]$ either the `id`-value of j or that of $j + \tau$ will also be minimal. Therefore, $i \in \mathcal{S}$ if and only if $j \in \mathcal{S}$.

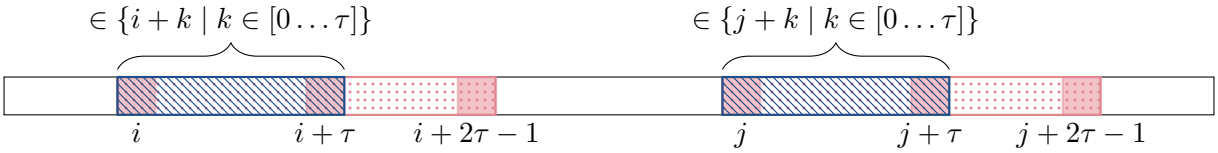


Figure 16: The string S with the identical substrings $S[i\dots i+2\tau-1] = S[j\dots j+2\tau-1]$ (marked with pink (dots)), and it illustrates how for any $k \in [0\dots\tau]$ then $S[i+k] = S[j+k]$.

□

We now show density condition b from Definition 5, which is $i \in \mathcal{R}$ if and only if $[i\dots i+\tau] \cap \mathcal{S} = \emptyset$. Firstly, we will show that if $i \notin \mathcal{R}$ then there exists at least one element $s \in \mathcal{S}$ where $s \in S[i\dots i+2\tau)$. Since $i \notin \mathcal{R}$, we know that not all elements of $i\dots i+2\tau$ can be in $\mathcal{Q}$. This is evident from the second definition of $\mathcal{R}$ (Definition 10) which states that if a range $[i\dots i+2\tau)$ is fully contained in $\mathcal{Q}$, then element i is in $\mathcal{R}$. Since $i \notin \mathcal{R}$ we know that some element (or elements) in the range $[i\dots i+2\tau)$ is not in $\mathcal{Q}$. Looking at all the elements in range $[i\dots i+2\tau)$ which are not in $\mathcal{Q}$, one of them must have minimal `id`-value, and we denote this element k^*.

Now there are two different cases: $k^* < i+\tau$ and $k^* \geq i+\tau$. If $k^* < i+\tau$, it is not part of $\mathcal{Q}$ and also the smallest element in the whole $[i\dots i+2\tau)$, which includes the interval $[k^*\dots k^*+\tau]$, it will be chosen to be in $\mathcal{S}$ because of Construction 3. Since $k^* < i+\tau$, it will also be in the interval $[i\dots i+\tau)$. On the other hand, if $k^* \geq i+\tau$, then the element $k^*-\tau$ is chosen to be in $\mathcal{S}$ because k^* will have minimal `id`-value in the interval $[k^*-\tau\dots k^*]$. The element $k^*-\tau$ will (because $k^* \geq i+\tau$) be in the interval $[i+\tau-\tau\dots i+2\tau-\tau) = [i\dots i+\tau)$.

Either way, an k^* will be chosen to be in $\mathcal{S}$ from the interval $[i\dots i+\tau)$; thus, the claim has been proven.

Now for the second and last part of the proof. If $i \in R$, then the interval $[i\dots i+\tau)$ and $\mathcal{S}$ will have no overlapping indices. This comes directly from the second definition of $\mathcal{R}$ (Definition 10), because if $i \in R$ then every element in the interval $[i\dots i+2\tau)$ will be in $\mathcal{Q}$ and thus no elements can be chosen to be in $\mathcal{S}$ due to the definition of Construction 3 (as seen in Figure 17).

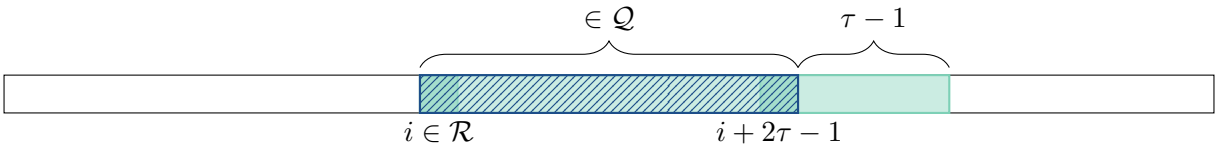


Figure 17: If $i \in \mathcal{R}$ then it means that the substring $S[i\cdots+3\tau-2]$ is periodic with period $\leq \frac{\tau}{3}$ and that all the indices of the interval $[i\dots i+2\tau)$ belong to $\mathcal{Q}$.

2.4.1 Randomised construction of a τ-synchronising set

In this section, we implement the general construction (Construction 3) in a randomised fashion. This is done in the following construction.

Construction 3a. This construction is different whether we are working with a string S with $(\frac{\tau}{3},\tau)$-runs or not. We call strings without $(\frac{\tau}{3},\tau)$-runs for *non-periodic*.

Non-periodic case: We use an identifier function as described in Definition 3, where $\texttt{id}(i) =$

$\texttt{id}(j)$ if and only if $S[i \dots i+\tau) = S[j \dots j+\tau)$, and a sliding window of size $\tau+1$. Adding a position i to $\mathcal{S}$ depends on whether $\texttt{id}(i)$ or $\texttt{id}(i+\tau)$ are minimal in the interval $[i \dots i+\tau]$.

General case: When the string contains $(\frac{\tau}{3},\tau)$-runs, we first identify the sets $\mathcal{Q}$ and $\mathcal{U}$, where $\mathcal{U}$ marks the boundaries of the runs in $\mathcal{Q}$. We then assign $\texttt{id}$-values so that all indices in $\mathcal{U}$ receive $\texttt{id}$-values smaller than any index not in $\mathcal{U}$. This ensures that the indices in $\mathcal{U}$ are always selected. After this, the selection procedure proceeds as in the non-periodic case: for each position $i \notin \mathcal{Q}$, i is added to $\mathcal{S}$ if either $\texttt{id}(i)$ or $\texttt{id}(i+\tau)$ is minimal within the interval $[i \dots i+\tau]$.

We now explore the complexity of Construction 3a. For the non-periodic case, the proof of why we expect a τ-synchronising set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ is very similar to the proof of Lemma 4. If there are no substrings with periods smaller than or equal to $\frac{\tau}{3}$, then any length τ interval has at least $\frac{\tau}{3}$ different $\texttt{id}$s. This means that the chance of any index i to have $\texttt{id}(i)$ or $\texttt{id}(i+\tau)$ as the smallest value in the set $\{\texttt{id}(k) \mid k \in [i \dots i+\tau]\}$ is less than or equal to $2 \cdot \frac{\tau}{3} = \frac{6}{\tau}$. This is also more or less how Kempa and Kociumaka [22] show in their proof of Fact 8.4.

For the general case, the periodic parts of the string will by Construction 3 only contribute to creating more than τ between indices in $\mathcal{S}$. This means that the expected size of $\mathcal{S}$ is still $\mathcal{O}(n)$.

Now we move on to the runtime of the construction. By a similar argument to the one of Lemma 5 we can scan through S in $\mathcal{O}(n)$ and keep track of the current minimal element and constantly compare it to the next element every time the window is moved one space to the right. The only problem would be if the current minimum disappeared out of the window without a new one being found. Then it takes $\mathcal{O}(\tau)$ to find the new minimum in the window. But every time the minimum disappears, it is because the previous leftmost position has been chosen as a minimum and thus was chosen to be in $\mathcal{S}$. Therefore, each rescan of the length τ interval can be paid by a position in $\mathcal{S}$. Since $\mathcal{S}$ has expected size $\mathcal{O}\left(\frac{n}{\tau}\right)$ this makes the run time $\mathcal{O}\left(n + \tau \cdot \frac{n}{\tau}\right) = \mathcal{O}(n)$ expected.

For the general case, as shown in Lemma 8.8 of [22], the sets $\mathcal{Q}$ and $\mathcal{U}$ can be constructed in $\mathcal{O}(n)$ time. This means that for the general case, the expected runtime is still $\mathcal{O}(n)$, since identifying $\mathcal{Q}$ and $\mathcal{U}$ also only takes $\mathcal{O}(n)$. With these two sets, we can give each index of $\mathcal{U}$ an additional marker, which marks it as a minimum element compared with elements not in $\mathcal{U}$. This takes at most $\mathcal{O}(n)$. Therefore, Construction 3a takes only expected $\mathcal{O}(n)$ in the general case as well.

2.4.1.1 Conclusion on the randomised construction of a τ-synchronising set

This means that we can construct a τ-synchronising set of expected size $\mathcal{O}\left(\frac{n}{\tau}\right)$ in expected $\mathcal{O}(n)$ time. Using Markov's inequality as we did in the randomised construction of the partitioning set, we could show that only the runtime would be expected $\mathcal{O}(n)$ (and not the size).

2.4.2 Deterministic construction of a τ-synchronising set

In their paper, Kempa and Kociumaka [22] use the randomised construction of Construction 3a to make a deterministic construction. The key in their approach is the introduction of a *scoring function* which aids in constructing a deterministic $\texttt{id}$-function (with the properties of Definition 3) where $\texttt{id}(i) = \texttt{id}(j)$ if and only if $S[i \dots i+\tau) = S[j \dots j+\tau)$, and where only $\mathcal{O}\left(\frac{n}{\tau}\right)$ elements are picked to be part of $\mathcal{S}$.

Construction 3b. This construction also distinguishes between whether the input string S contains $(\frac{\tau}{3},\tau)$-runs or not. As before, we refer to strings without such runs as *non-periodic*.

Non-periodic case: The non-periodic construction consists of the following steps.

1. Partition all substrings of length τ into equivalence classes such that all indices in an equivalence class mark the beginning of the same length τ substring (an example of this is given in Figure 19). Each class consists of indices where identical substrings begin. All indices initially have undefined `id`-values.

2. Identify *active blocks*, which are maximal contiguous regions of undefined indices. For blocks of length at least τ, assign score -1 to the first and last $\lfloor \frac{\tau}{3} \rfloor$ indices, and score $+2$ to all others. Blocks smaller than τ receive a score of 0.

3. Compute the sum of scores for each class over its member indices. This is the aggregated score of this class. Maintain a set $\mathcal{C}^+$ of unprocessed classes with non-negative aggregated score.

4. Iteratively assign increasing `id`-values to classes from $\mathcal{C}^+$: At each step, assign the current `id`-value to one class, update all scores, recompute aggregated class scores, and update $\mathcal{C}^+$. Repeat until $\mathcal{C}^+$ is empty.

5. Apply the sliding window selection rule from Construction 3a: For each position i, add i to $\mathcal{S}$ if either $\texttt{id}(i)$ or $\texttt{id}(i+\tau)$ is minimal in the interval $[i \dots i+\tau]$.

General case: When the string contains $(\frac{\tau}{3},\tau)$-runs, we first identify the sets $\mathcal{Q}$ and $\mathcal{U}$, where $\mathcal{U}$ marks the run boundaries. Classes whose indices lie entirely within $\mathcal{U}$ are assigned the smallest `id`-values. Next, we assign `id`-values to classes fully contained within $\mathcal{Q}$. The remaining classes are handled using the same scoring-based method described in the non-periodic case. Finally, we perform the sliding window selection as before.

In Figure 18 and Figure 19 we see a (non-periodic) example of the scoring function described in Construction 3b. The two figures show how the deterministic `id`-function is created and not how the synchronising set is computed.

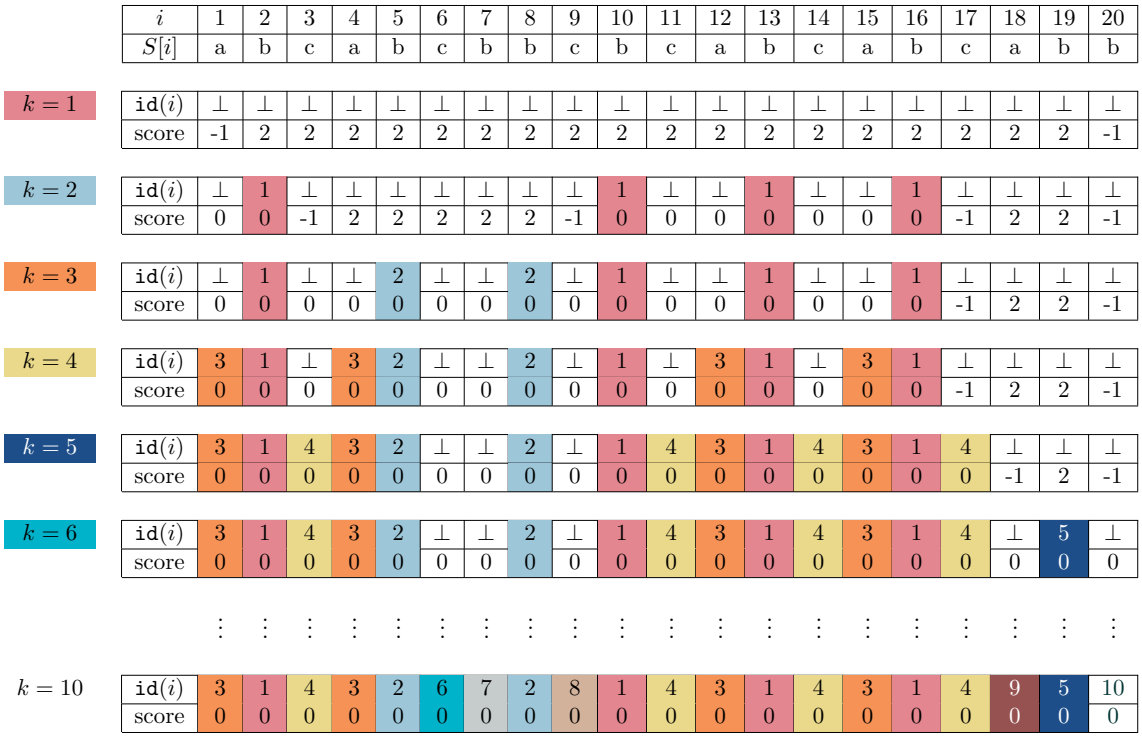


Figure 18: The string $S = abcabcbbcbcabcabcabb$ is partitioned into classes for $\tau = 3$ in Figure 19 and using these we find the aggregated score at $k = 1 - 10$. For every value of k, we calculate the score of all active positions (positions in an active block).

Classes in $\mathcal{C}$		Aggregated scores						
Substring	Indices	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$		$k=10$
abc	$\{1,4,12,15\}$	5	2	0	0	0	...	0
bca	$\{2,10,13,16\}$	8	0	0	0	0	...	0
cab	$\{3,11,14,17\}$	8	-2	0	0	0	...	0
bcb	$\{5,8\}$	4	4	0	0	0	...	0
cbb	$\{6\}$	2	2	0	0	0	...	0
bbc	$\{7\}$	2	2	0	0	0	...	0
cbc	$\{9\}$	2	-1	-1	0	0	...	0
abb	$\{18\}$	2	2	2	-1	0	...	0
bb	$\{19\}$	2	2	2	2	0	...	0
b	$\{20\}$	-1	-1	-1	-1	0	...	0

Figure 19: The classes of the string $S = abcabcbbcbcabcabcabb$ for $\tau = 3$. The set of classes $\mathcal{C} = \{\{1,4,12,15\},\{2,10,13,16\},\{3,11,14,17\},\{5,8\},\{6\},\{7\},\{9\},\{19\},\{20\}\}$ and their aggregated scores for $k = 1 - 10$.

In section 8.1.2 and proposition 8.10 of [22] it is shown that the deterministic construction of a τ-synchronising set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ takes $\mathcal{O}(n)$ for a text $S \in [0 \dots \sigma)^n$ for $\sigma = n^{\mathcal{O}(1)}$ and $\tau \leq \frac{1}{2}n$, where τ is a positive integer.

2.4.2.1 Conclusion on the deterministic construction of a τ-synchronising set

Aided by the complexity results of Kempa and Kociumaka [22], we can conclude that using Construction 3b a τ-synchronising set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ can be constructed in $\mathcal{O}(n)$ time.

2.5 The local minimum parsing (of the signature grammar)

The last partitioning is based on a signature grammar introduced by Christiansen and Ettienne [8] and revisited by Christiansen et al. [9]. We will denote this partitioning, $\mathcal{M}$, and call it a τ-local minimum set. There is only one construction of the partitioning, and it is randomised.

2.5.1 Randomised construction of a τ-local minimum set

This construction is hierarchical and built bottom up. To begin with, all characters of the string are considered an individual block. At every level, blocks are merged to form new blocks, and the new blocks are all represented by a character. This representative is just the character itself in the first level (the bottom level).

Like in the deterministic construction of (τ, δ)-partitioning set this construction distinguished between two types of sub-blocks at level $\mu - 1$.

- *Type 1:* A sub-block has a label that differs from the labels of both its neighbours.
- *Type 2:* A sub-block where at least one of its neighbours has a label identical to the sub-block's.

Then the construction consists of two rounds, which are repeated until a level where the number of blocks is $\mathcal{O}\left(\frac{n}{\tau}\right)$ is reached. Every time the two rounds run, a new level is created.

Construction 4. Let S be a string of length n and $\pi_{\mu-1}$ be a random permutation of the labels at level $\mu-1$ (such that every distinct label at level $\mu-1$ has a unique rank). Then, the following two rounds will create a level.

1. All sub-blocks of type 2 are merged with the surrounding identically labelled neighbours. Those new blocks receive new labels that depend on the merged sub-blocks and their number.
2. Now using $\pi_{\mu-1}$ all the sub-blocks, which are local minima (meaning $\pi_{\mu-1}$ of their label is strictly smaller than the value of both its neighbours), are found. Then blocks of level μ are created by letting the first block start at position 1 and make it consist of all the following sub-blocks until a sub-block with a label, which is a local minimum, is reached. Then, a new block is created starting at this local minimum, and this continues to create new blocks starting at every local minimum.

Repeat these two steps until level $\log_3 \tau$.

A very important property of this construction is that for two identical substrings (with different offsets), their parsing only differs by the first and last two blocks, no matter which level we look at. This is shown in the following lemma.

Lemma 14. *[Lemma 5 [8]] Whenever $S[i \dots i'] = S[j \dots j']$ the blocks $S[i \dots i']$ is partitioned into are identical to the blocks of $S[j \dots j']$ except for possibly the two first and two last blocks.*

The following proof is based on the proof of Lemma 5 in [8].

Proof. Assume that the substrings $S[i \dots i']$ and $S[j \dots j']$ are identical. We now view the hierarchical construction like an ordered forest[5] and look at the blocks as nodes in an ordered forest (as described

[5] A forest where all vertices have an ordering.

in Section 2.2). Then we look at the relevant nodes, $T(i, i')$ and $T(j, j')$, for both sub-strings, and we will do the proof by induction on the level of the hierarchical construction.

The leaves of two ordered forests $T(i, i')$ and $T(j, j')$ are, by definition, identical, since they are the relevant nodes of two identical substrings, and all the characters of the substrings are found in the leaves. We assume that the Lemma holds for level $\mu - 1$ and then show that it also holds for level μ. We begin by looking at only the leftmost nodes of the two forests. Let $v_1, v_2, v_3 \dots$ be the nodes of $T(i, i')$, with v_1 as the leftmost node, at level $\mu - 1$, and let $u_1, u_2, u_3 \dots$ be the leftmost nodes of $T(j, j')$. The induction hypothesis tells us that at level $\mu - 1$ there are at most four nodes which are different for $T(i, i')$ and $T(j, j')$. Focusing only on the two leftmost nodes at level $\mu - 1$ this means that for some $a, b \in [1, 2, 3]$ then $v_a = u_b$, $v_{a+1} = u_{b+1}$ and so on.

Now we construct level μ by merging blocks and in this way creating new nodes. Let v_{a+k} be the first block that starts after v_a. An identical block will start at u_{b+k} because of the induction hypothesis. So the following blocks are identical (until we reach the right side of the ordered forests, but this comes later).

The only thing left to argue is that the nodes $v_1, \dots, v_{a+k}$ (and $u_1, \dots, u_{b+k}$) is spanned by at most 2 blocks. To get a contradiction, we will look at $v_1, \dots, v_{a+k-1}$ and try to span these sub-blocks by more than two blocks. Let us begin by assuming that the number of different sub-blocks is maximal such that v_1 and v_2 differ from u_1 and u_2. The first block could contain v_1 alone (this is possible due to Construction 4), then the second block would need to contain at least two sub-blocks and would thus certainly contain v_2 and $v_a = v_3$. Now the third block would begin after v_a, but by definition this block starts at v_{a+k}, which means that $v_1, \dots, v_{a+k-1}$ are spanned by at most two blocks.

This means that at level μ only the two leftmost blocks for $T(i, i')$ and $T(j, j')$ might be different, but the blocks starting at v_{a+k} and u_{b+k} are completely identical (and the following blocks will also be). The argument for the two rightmost blocks is symmetric. □

Let us now examine the size of the τ-local minimum set and the runtime of the construction. Christiansen and Ettienne [8] show in their Lemma 3 that the expected number of sub-blocks from level $\mu - 1$ to form a block at level μ is 3. Using this fact, we will compute the expected size of the τ-local minimum set. Let $\mathbb{E}[b_\mu]$ denote the expected size in characters of a block at level μ. At level 0, each block consists of a single character, so $\mathbb{E}[b_0] = 1$. Since each block at level μ is formed by grouping expected 3 sub-blocks from the previous level, we have the recurrence

$$\mathbb{E}[b_\mu] = 3 \cdot \mathbb{E}[b_{\mu-1}].$$

implies that $\mathbb{E}[b_\mu] = 3^\mu$. Since the total string has length n, the expected number of blocks at level μ is $\frac{n}{3^\mu}$. We stop the construction at level $\mu = \log_3(\tau)$. At this point, the expected block size is $\mathbb{E}[b_{\log_3 \tau}] = 3^{\log_3 \tau} = \tau$ and thus the expected size of $\mathcal{M}$ is $\mathcal{O}\left(\frac{n}{\tau}\right)$.

The runtime can be bounded by the fact that at every level, the number of blocks is at least halved. At every level, the construction uses constant time deciding whether the label of a block is a local minimum, and labelling each new block afterwards can be done with, for instance, perfect hashing, such that identical blocks get the same new label. Therefore the runtime is

$$\sum_{i=0}^{\log_3 \tau} \frac{1}{2^i} n \leq 2n = \mathcal{O}(n).$$

2.5.1.1 Conclusion on the randomised construction of a τ-local minimum set

The τ-local minimum set is constructed in $\mathcal{O}(n)$, and has expected size $\mathcal{O}\left(\frac{n}{\tau}\right)$. Lemma 14 shows that given two identical substrings, they will at most differ by four blocks (the two leftmost and two rightmost blocks).

2.6 Overview of the constructions

This section presents an overview of the complexities of the five constructions presented in this chapter. The table below (Table 1) shows the space of each partitioning, the construction time, the construction space, the construction number, and the paper which first presented the construction.

Randomised constructions

Partitioning type	*Space*	*Constr. time*	*Constr. space*	*Reference*
(τ, τ)-partitioning set	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}(n)$ exp.	$\mathcal{O}\left(\frac{n}{\tau}\right)$	constr. 1, [5]
τ-synchronising set	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}(n)$ exp.	$\mathcal{O}\left(\frac{n}{\tau}\right)$	constr. 3a,[22]
τ-local minimum set	$\mathcal{O}\left(\frac{n}{\tau}\right)$ exp.	$\mathcal{O}(n)$	$\mathcal{O}\left(\frac{n}{\tau}\right)$	constr. 4, ([8])

Deterministic constructions

Partitioning type	*Space*	*Constr. time*	*Constr. space*	*Reference*
$(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}(n \log n)$	$\mathcal{O}\left(\frac{n}{\tau} + \log \tau\right)$	constr. 2, [5]
τ-synchronising set	$\mathcal{O}\left(\frac{n}{\tau}\right)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	constr. 3b, [22]

Table 1: This table shows space, construction time, and construction space for the five different partitioning constructions, covered in Chapter 2. In this table, construction is abbreviated to constr.

We expect the randomised construction times to be $\mathcal{O}(n)$. Birenzwige et al. [5] even show that their construction time is $\mathcal{O}(n)$ with high probability in Section 8 of their work. The sliding window construction which was used by all the three randomised constructions ensure that the working space is very small, because we only need to save the positions picked for the partitioning set, and one minimum value at the time and compare it to a new `id` as the window shifts to the right.

Looking at the deterministic constructions, the $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set uses more time during construction than τ-synchronising set, while the τ-synchronising set uses more space during construction.

Chapter 3

Locality of the partitionings

In this chapter, we introduce two different types of locality and see how the different partitionings perform according to these definitions. The two types are called *index-based* and *block-based* locality.

Index-based locality is based on the locality concepts introduced by both Birenzwige et al. [5] and Kempa and Kociumaka [22]. This form of locality guarantees that given two identical substrings, their parsing will at most differ by the first α elements and the last β elements. Unless there is some periodic behaviour in the string, but this can be shown not to be a problem. This gives the nice property that we can compare the first α elements of two different substrings, then compare the substrings using their parsing, and then, when the parsing no longer matches, only compare additional β elements.

Block-based locality is built on the notion of locality used by Christiansen et al. [9] and Christiansen et al. [9]. In contrast to index-based locality, this type does not say how many elements may be parsed differently. It only states something about the number of blocks that might be different. Here, it is guaranteed that only the γ first and the γ last blocks will differ for two identical substrings.

The two types of localities are illustrated in the figure below (Figure 20).

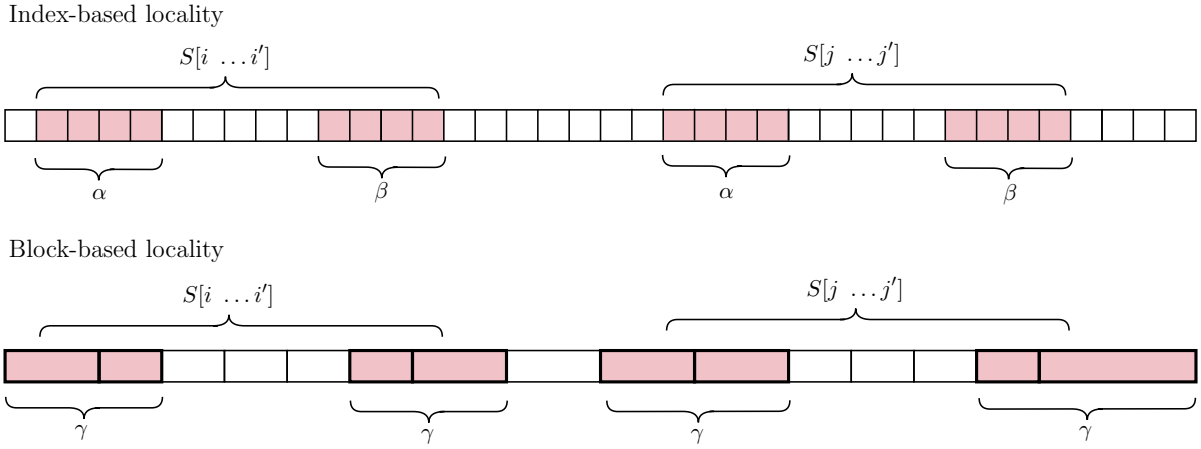


Figure 20: In this figure, the two arrays illustrate the same string with two identical substrings, $S[i \ldots i']$ and $S[j \ldots j']$, marked with cross-stripes. The red (marked) areas are the ones where there are no guarantees given by index-based and block-based locality, respectively.

3.1 Locality of (τ, δ)-partitioning set

In terms of index-based locality, the randomised construction (Construction 1) at most the 2τ first and the 2τ last elements differ. Construction 2 ensures that at most the $\mathcal{O}(\tau \log^* n)$ first and the

$\mathcal{O}(\tau \log^* n)$ last elements differ.

For block-based locality, the number of blocks which might differ for the randomised Construction 1 is τ. This means that the $\tau - 1$ first blocks and the $\tau - 1$ last blocks could differ for two identical substrings. Some interval τ can consist of τ blocks (not for all intervals, but there might exist some). Since nothing is guaranteed about these τ first (or last elements), it means that two identical substrings of S might differ by the τ first (or last) blocks (if one substring has all the $\tau - 1$ first or last elements picked as border points and the other substring only has one element picked).

In the deterministic Construction 2, some interval $\tau \log^* n$ can have every second element being a chosen position. Since nothing is guaranteed for the $\mathcal{O}(\tau \log^* n)$ first and last positions these blocks can look completely different and therefore two identical substrings of S might differ with $\frac{\tau \log^* n}{2}$ blocks which again is just $\mathcal{O}(\tau \log^* n)$.

3.2 Locality of τ-synchronising set

The index-based locality for both Construction 3a and Construction 3b is exactly the same as the locality property of the τ-synchronising set, so at most τ indices will differ on each side.

The block-based locality acts similarly to the constructions of the partitioning sets. We have no guarantees about the first or the last τ elements, which means it is possible that an entire interval of τ positions can be chosen except one position, and the corresponding interval could have only a single position chosen (the one which was not selected in the first interval).

3.3 Locality of τ-local minimum set

For the τ-local minimum set, the index-based locality is the size of the two largest consecutive blocks in $\mathcal{M}$. Since two substrings can at most differ by their two leftmost and two rightmost blocks, the "unknown" parts (or those without any local properties) of the two substrings are those blocks. There is no bound on the block size of the τ-local minimum set; thus, the index-based locality of this partitioning is $\mathcal{O}(n)$.

Nevertheless, we showed that the expected size of a block in $\mathcal{M}$ is τ, and since two substrings differ by at most two blocks on each side, the expected index-based locality is bounded by 2τ. Although this holds in expectation, the block size has no worst-case guarantee. In particular, Christiansen and Ettienne [8], building on results by Ferragina and Grossi [15], show that the expected worst-case block size for non-periodic strings is $\mathcal{O}(\log n)$.

The block-based locality of the τ-local minimum set is 2, which we showed in Lemma 14. This means that at most the first two and the last two blocks differ for any two identical substrings.

Chapter 4

Connections between partitionings

This chapter consists of two reductions. The first one is a reduction from a τ-synchronising set to (τ, τ)-partitioning set for any general string S. Then we move on to a reduction from a (τ, τ)-partitioning set to a $(\tau + 1)$-synchronising set but only for strings with no periodic substrings. Lastly, look at why it is not possible to make a partitioning set into a synchronising set for a general string.

4.1 A τ-synchronising set can be reduced to a (τ, τ)-partitioning set in linear time

Given a string S and its corresponding τ-synchronising set, $\mathcal{S}$, we can create a (τ, τ)-partitioning set, denoted $\mathcal{P}_{\text{new}}$, using the following procedure. Beginning with the first index of the partition, we add this first index to the new $\mathcal{P}_{\text{new}}$, which we are building. Moving on, we keep adding all the elements of the τ-synchronising set but shifted $\tau - 1$ to the right until we get to indices of the partition larger than n.

Formally, $\mathcal{P}_{\text{new}}$ can be described as

$$\mathcal{P}_{\text{new}} := \{p \in [1 \dots n - \tau] \mid \exists\ s \in \mathcal{S} : p = s + \tau - 1 \leq n\} \cup \{s_1\}, \tag{4.1}$$

where s_1 is the smallest index in $\mathcal{S}$. This procedure is illustrated in Figure 21.

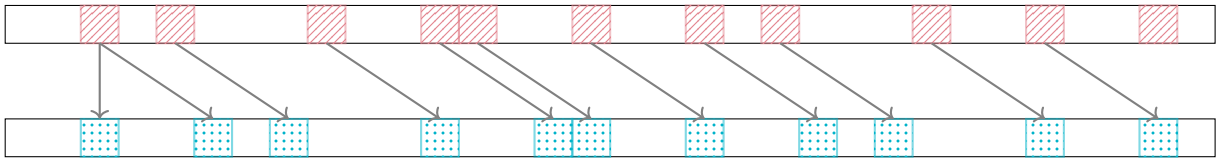


Figure 21: The two arrays picture the same string, S, and how we turn a τ-synchronising set marked by pink (cross-striped) positions into a (τ, τ)-partitioning set marked by the blue (dotted) positions, starting with the τ-synchronising set in the top. The first element is kept, and all the other elements are shifted by $\tau - 1$ to the right.

With the following Lemma, we show that the locality condition of Definition 5 holds for $\mathcal{P}_{\text{new}}$.

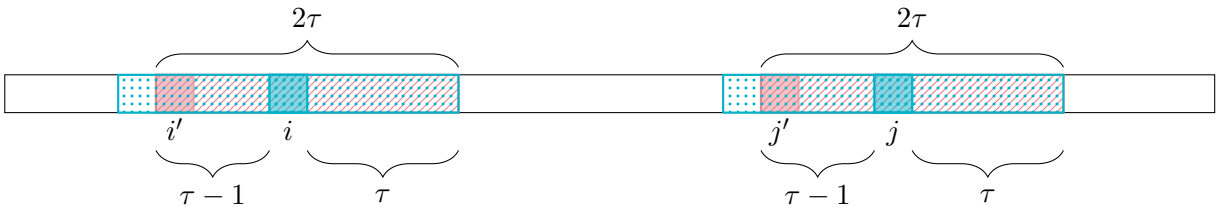
Lemma 15. *Let S be a string of length n. Using the procedure explained above, it will hold that for any $i, j \in [n]$ where $S[i - \tau \dots i + \tau] = S[j - \tau \dots j + \tau]$ then $i \in \mathcal{P}_{new}$ if and only if $j \in \mathcal{P}_{new}$.*

Proof. Let i, j be indices in $\mathcal{P}_{\text{new}}$ created by moving all elements of $\mathcal{S}$ forwards by τ. Let us further assume that $S[i - \tau \dots i + \tau] = S[j - \tau \dots j + \tau]$. Then we need to show $i \in \mathcal{P}_{\text{new}}$ if and only if $j \in \mathcal{P}_{\text{new}}$. We will do this by showing that it can never happen that $i \in \mathcal{P}_{\text{new}}$ and $j \notin \mathcal{P}_{\text{new}}$ (and then concluding it the other way around by a symmetry argument).

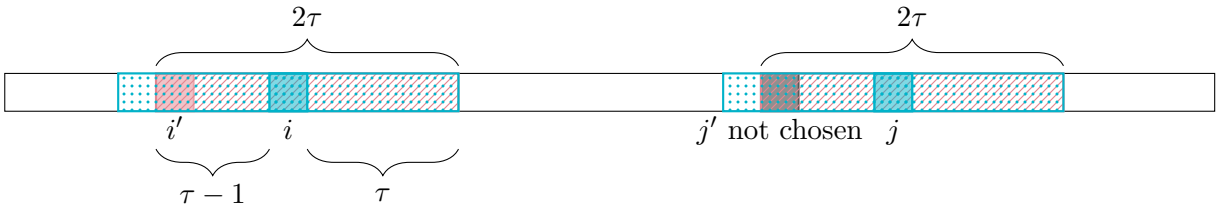
We will show this by using a contradiction. Assume that it is indeed the case that $i \in \mathcal{P}_{\text{new}}$ and $j \notin \mathcal{P}_{\text{new}}$. Index i can only be in $\mathcal{P}_{\text{new}}$ if index $i' := i - \tau + 1$ was in $\mathcal{S}$, and the only

way for j not to have been chosen is for $j' := j - \tau + 1$ not to have been in $\mathcal{S}$. But because $S[i-\tau \dots i+\tau] \supseteq S[i-\tau+1 \dots i+\tau] = S[i' \dots i'+2\tau] = S[j' \dots j'+2\tau] = S[j-\tau+1 \dots j+\tau]$ we get that $S[i' \dots i'+2\tau] = S[j' \dots j'+2\tau]$. Moreover, $\mathcal{S}$ is a τ-synchronising set and therefore if $i' \in \mathcal{S}$ then $j' \in \mathcal{S}$, which means that $j \in \mathcal{P}_{\text{new}}$ and thus we reach a contradiction. □

The idea behind the proof is illustrated in Figure 22.



(a) If we assume that the blue (dotted) areas are identical substrings, then the two (smaller) pink (cross-striped) areas are also exactly the same.



(b) If the blue (dotted) areas are the same and i' is in $\mathcal{P}_{\text{new}}$ then j' has to be in $\mathcal{P}_{\text{new}}$ as well. Is it possible that i' is chosen but j' is not? No, because this means that $j \notin \mathcal{S}$ would make $\mathcal{S}$ invalid.

Figure 22: The two figures illustrates the idea behind proof of locality (Lemma 15).

Now we look at the density condition of $\mathcal{P}_{\text{new}}$. For the non-periodic case, this is quickly done. Since all indices are moved the same number of positions to the right, they will remain within τ positions from each other. Because we keep the first index of $\mathcal{S}$, we ensure that the first index of $\mathcal{P}_{\text{new}}$ is not moved too far away from index 1.

This means that for a non-periodic string S, $\mathcal{S}$ can be turned into $\mathcal{P}_{\text{new}}$, which does fulfil the requirements of a (τ, τ)-partitioning set.

The only thing left is to see how this reduction handles periodic behaviour. The key idea is that if there are more than τ between s_i and s_{i+1} in $\mathcal{S}$, then it is because there exists at least one index $i \in \mathcal{R}$ between s_i and s_{i+1}.

Lemma 16. *Let S be a string. Using the procedure explained above, then for any $p_i, p_{i+1} \in \mathcal{P}_{new}$ where $p_{i+1} - p_i > \tau$ the substring $S[p_i \dots p_{i+1} - 1]$ will be periodic with principal period $\leq \frac{\tau}{3}$.*

Proof. Let us look at the case where $i := s_i + 1$ and $i \in \mathcal{R}$. Then, because of the definition of $\mathcal{R}$, we know that the substring $S[i + 3\tau - 2]$ is periodic with period smaller than or equal to $\frac{\tau}{3}$. Let us define $p_{i'} := s_i + \tau - 1$ and $p_{i'+1} = s_{i+1}$. Now, $p_{i'}$ will mark the beginning of a periodic string (which is the requirement of Definition 5). The question is if $S[p_{i'} \dots p_{i'+1} - 1]$ will be a periodic string (this is required by Definition 5. This means we must look at $p_{i'+1}$.

When a periodic string ends, then due to (2.3) (definition of $\mathcal{R}$) and Definition 5, there will be an element s_{i+1} of $\mathcal{S}$ such that the substring $S[s_{i+1} \dots s_{i+1} + \tau - 2]$ is periodic (with period smaller than $\frac{\tau}{3}$) but the substring $S[s_{i+1} \dots s_{i+1} + \tau - 1]$ is not.

This means that $p_{i'+1}$ will be placed at $s_{i+1} + \tau - 1$, which is the first position between i (the first periodic position) and $s_{i+1} + \tau - 2$ (the last periodic position).

This means that if $p_{i'}$ and $p_{i'+1}$ are more than τ apart then the substring $S[p_{i'} \dots p_{i'+1} - 1]$ is periodic with period smaller than or equal to $\frac{\tau}{3}$. This proves the lemma. □

Theorem 1. *Given a τ-synchronising set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ and in a form that can be scanned in linear time, this partition can be transformed into a (τ, τ)-partitioning set in $\mathcal{O}\left(\frac{n}{\tau}\right)$ time.*

Proof. Using Lemma 15 and Lemma 16, we see that using the earlier described, procedure $\mathcal{P}_{\text{new}}$ will be a (τ, τ)-partitioning set. Because we only increment the indices of the $\mathcal{S}$ (which had size $\mathcal{O}\left(\frac{n}{\tau}\right)$), this is a linear scan using constant time at each position in $\mathcal{S}$, thus construction $\mathcal{P}_{\text{new}}$ takes $\mathcal{O}\left(\frac{n}{\tau}\right)$. □

4.2 A (τ, τ)-partitioning set can be reduced to a $(\tau + 1)$-synchronising set in linear time if the string S is without periodic substrings

This reduction can be performed by scanning (backwards) through the (τ, τ)-partitioning set. Beginning with the last index of the (τ, τ)-partitioning set, $\mathcal{P}$, we add this index to a new synchronising set, $\mathcal{S}_{\text{new}}$.

Moving on, we keep shifting elements of $\mathcal{P}$ to the left until we get to indices of the partition which are smaller than τ, like in the example depicted by Figure 23.

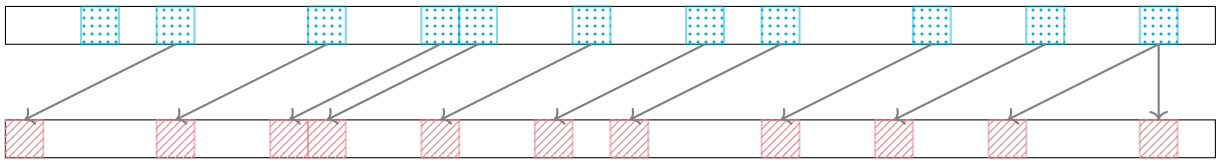


Figure 23: The two arrays picture the same string, and how we turn a (τ, τ)-partitioning set marked by blue (dotted) positions into a $(\tau+1)$-synchronising set marked by the pink (cross-striped) positions, starting with the (τ, τ)-partitioning set in the top.

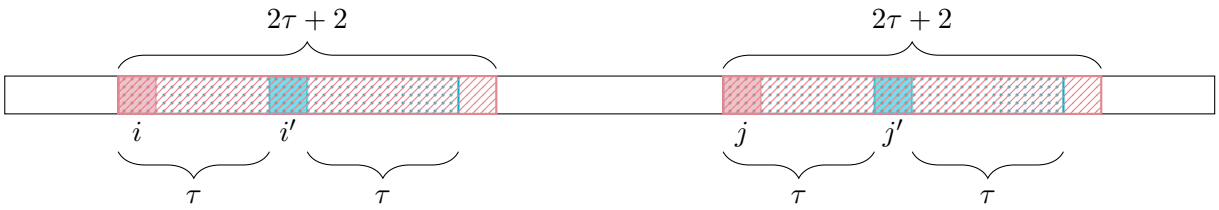
We now prove that the locality condition for a $(\tau + 1)$-synchronising set holds for $\mathcal{S}_{\text{new}}$.

Lemma 17. *Using the procedure explained above, it will hold that for any $i, j \in [n]$ where $S[i \dots i + 2\tau + 2) = S[j \dots j + 2\tau + 2)$ then $i \in \mathcal{S}_{new}$ if and only if $j \in \mathcal{S}_{new}$.*

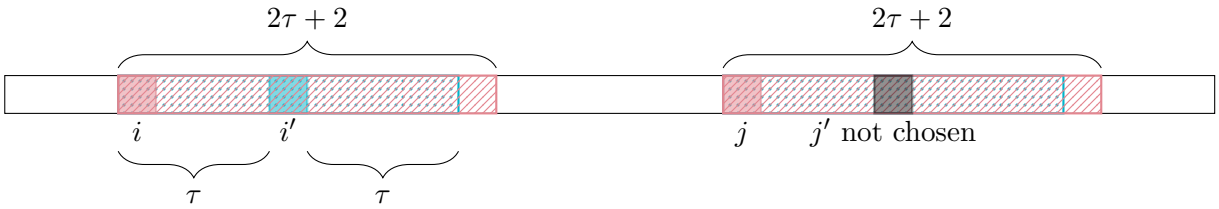
Proof. Let $\mathcal{S}_{\text{new}}$ be the $(\tau+1)$-synchronising set created by pushing indices from the (τ, τ)-partitioning set backwards. Now let $i, j \in [n]$ where $S[i \dots i + 2\tau + 2) = S[j \dots j + 2\tau + 2)$. Trivially, this means that $S[i \dots i+2\tau] = S[j \dots j+2\tau]$. Then, we only need to show that it can never happen that $i \in \mathcal{S}_{\text{new}}$ and $j \notin \mathcal{S}_{\text{new}}$ and vice versa.

Assume that $i \in \mathcal{S}_{\text{new}}$, then position $i' := i + \tau$ is in $\mathcal{P}$. If j is also in $\mathcal{S}_{\text{new}}$, then we would be done, so let us assume that $j \notin \mathcal{S}_{\text{new}}$. That means $j' := j + \tau$ is not in $\mathcal{P}$. If $i' \in \mathcal{P}$ this means that all indices equal to $S[i' - \tau \dots i' + \tau]$ must also be in $\mathcal{P}$. Since $S[i' - \tau \dots i' + \tau] = S[i \dots i + 2\tau]$ and we know that $S[i \dots i + 2\tau] = S[j \dots j + 2\tau] = S[j' - \tau \dots j' + \tau]$, and thus $i \in \mathcal{S}_{\text{new}}$ if and only if $j \in \mathcal{S}_{\text{new}}$. □

The concept of the proof is also portrayed by Figure 24.



(a) The two areas marked with pink (cross-striped) have the exact same values (meaning $S[i \dots i+2\tau+2) = S[j \dots j+2\tau+2)$). This means that the blue (dotted) areas are also exactly the same.



(b) The pink (cross-striped) areas of the figure have exactly the same values. If $i \in \mathcal{S}_{\text{new}}$ then i' is in $\mathcal{P}$. If $j \notin \mathcal{S}_{\text{new}}$ then j' is not in $\mathcal{P}$, but the blue (dotted) areas are exactly the same, which means that if $i' \in \mathcal{P}$ then j' also has to have been in $\mathcal{P}$.

Figure 24: Illustration for proof of Lemma 17 by showing what it means for two indices to be in $\mathcal{S}_{\text{new}}$ if $S[i \dots i+2\tau+2) = S[j \dots j+2\tau+2)$.

Theorem 2. *Given a string S, which contains no (τ,τ)-runs and a (τ,τ)-partitioning set built on this string of size $\mathcal{O}\left(\frac{n}{\tau}\right)$. Then this partition can be transformed into a $(\tau+1)$-synchronising set in $\mathcal{O}\left(\frac{n}{\tau}\right)$ time.*

Proof. Creating each element of the synchronising set takes constant time. The method we are using transform the partition of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ into a synchronising set which is at most the size of the partition plus one (because both p_{last} and $p_{\text{last}} - \tau$ is added). Therefore, the synchronising set has size $\mathcal{O}\left(\frac{n}{\tau}\right)$, and it takes at most $\mathcal{O}\left(\frac{n}{\tau}\right)$ to create it.

We argue that the density condition still holds because all elements of the synchronising set have been moved τ positions but have, in this way, kept their distance to both their predecessor and successor. The only problem could be the last τ positions of the string but because both p_{last} and $p_{\text{last}} - \tau$ are added to $\mathcal{S}_{\text{new}}$ we are guaranteed that all elements always have predecessor and successor at most τ away.

We must argue that $\mathcal{S}_{\text{new}}$ still fulfils the locality requirement of Definition 5. We do this by referring to Lemma 17 where we see that for any $i, j \in [n]$ where $S[i \dots i+2\tau+2) = S[j \dots j+2\tau+2)$ then $i \in \mathcal{S}_{\text{new}}$ if and only if $j \in \mathcal{S}_{\text{new}}$. □

Having shown Theorem 2 we can conclude that as long as S is a non-periodic string a (τ,τ)-partitioning set can always be transformed into a $(\tau+1)$-synchronising set.

4.3 Conclusion on the two reductions

The fact that (τ,δ)-partitioning set does not reduce to a τ-synchronising set can be understood by examining the two distinct density conditions outlined in Definition 5. Density condition a requires that if there is more than τ between a consecutive pair of elements in $\mathcal{L}_a$, then the substring between them must be periodic.

Density condition b is much stricter since it is a bi-implication. This means that it has to live up to similar requirements as density condition a and that periodic behaviour of a certain length with a specific period will require the distance between two elements to be larger than τ.

This also nicely shows why the two partitionings behave similarly when S has no periodic structure, then they are both locally consistent sets, just slightly shifted versions.

An interesting result of the reduction from a τ-synchronising set to a (τ, τ)-partitioning set is that this makes it possible to deterministically construct a τ-synchronising set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ in $\mathcal{O}(n)$ time, then using $\mathcal{O}\left(\frac{n}{\tau}\right)$ to transform it into a (τ, τ)-partitioning set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$ in $\mathcal{O}\left(\frac{n}{\tau}\right)$. This takes in total $\mathcal{O}\left(n + \frac{n}{\tau}\right) = \mathcal{O}(n)$ which means a (τ, τ)-partitioning set can be constructed deterministically in $\mathcal{O}(n)$ instead of the $\mathcal{O}(n \log \tau)$ used by Birenzwige et al. [5].

Chapter 5

Locally Consistent Grammars

This chapter covers locally consistent grammars and how they relate to the locally consistent partitionings. We begin by introducing what we, in this thesis, refer to as a (context-free) grammar.

5.1 Context-free grammar producing a string S (and only S)

We use the same grammar framework as the one described by Christiansen et al. [9]. A context-free grammar is defined by a finite alphabet Σ, a set of non-terminals, a set of production rules (each non-terminal having exactly one rule), and a start symbol. The grammar we consider is acyclic, meaning that no non-terminal can eventually derive itself, and it generates exactly one string, namely S.

Following Christiansen et al. [9], we rely on the concept of a *locally consistent grammar*. Informally, this means that if two substrings of S are equal, the way the grammar parses them is also locally similar. The grammar reuses the same non-terminals for equal substrings, up to a small, bounded difference near the boundaries.

Christiansen et al. [9] also introduce the variable c_g, which measures a grammar's local consistency. The following is the definition of a locally consistent grammar from [9].

Definition 11 (Based on Definition 4.6 [9])**.** A grammar is *locally consistent* if there is a constant c_g such that the partitioning at each level of the grammar has block-based locality at most c_g.

In other words, two identical substrings are parsed the same way throughout, except possibly within c_g blocks from either end.

5.2 The signature grammar

The grammar introduced by Christiansen and Ettienne [8] is called *the signature grammar*. Constructing this grammar is a hierarchical process, almost identical to building τ-local minimum set in Chapter 2. The only difference is that we stop with the steps of Construction 4 when only one vertex remains and not when the partitioning has some specifically bounded size. In this way (continuing until we only have one block at the top), we construct the *signature tree* of S, which will be denoted $sig(S)$. The construction still takes $\mathcal{O}(n)$ because the runtime did not depend on the level Construction 4 stopped at. Using the fact that the expected number of children of a node in the tree is 3, Christiansen and Ettienne [8] repeats the construction until the average degree of a node in the signature tree is 3. Using Markov's inequality, the average number of tries before the construction succeeds is $\mathcal{O}(1)$, which means that the expected construction time is $\mathcal{O}(n)$.

A very important thing to notice is that Lemma 14 also holds for the signature tree. The induction proof does not depend on which level we stop the algorithm; thus, we get the following corollary.

Corollary 17.1. *Let $S[i \ldots' i] = S[j \ldots j']$ be two identical substring in S. Then let $T(i, i')$ and $T(j, j')$ be the relevant nodes of those two substrings with regards to $sig(S)$. Then $T(i, i')$ and $T(j, j')$ will have the same vertices except for each level's first or last two vertices.*

The proof of this corollary is identical to that of Lemma 14.

5.2.1 The signature DAG

The signature tree of S, $sig(S)$, as described in Construction 4, can be turned into the signature DAG, $dag(S)$, by merging all nodes with the same labels and directing the edge such that a parent points towards its children. This means that a run node can be stored in constant space since it will only point to one child (and the number of times this child is repeated in the run-node is stored explicitly in the label of the run-node).

The signature DAG of S can also be viewed as an acyclic run-length grammar which produces the string S as introduced earlier in this chapter. The vertices with out-degree 0 will be the terminals, and the non-terminals are all the remaining vertices. The left side of a rule is then the label of a vertex, and the right side of the rule is the label of its children concatenated from left to right. When discussing the signature grammar, we refer to the signature DAG.

Figure 25 shows an example of a signature tree and grammar.

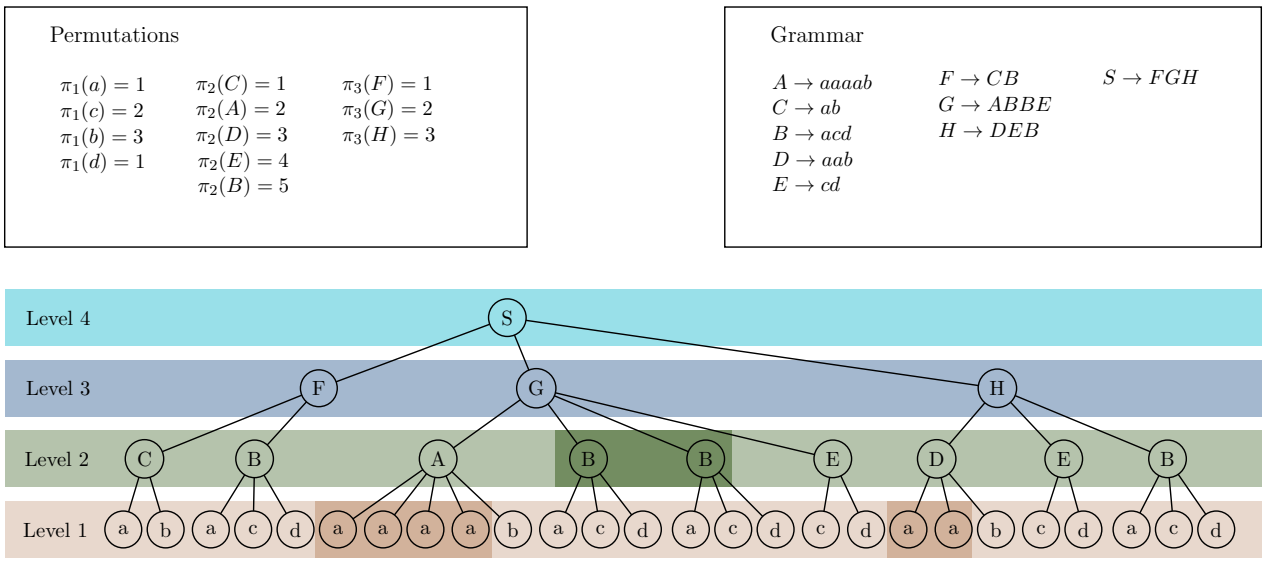


Figure 25: Example of a signature tree (and grammar) built on top of the string S = abacdaaaabacdacdcdaabcdacd. At both level 1 and level 2, we observe run-nodes; these are merged before the next level of the signature tree is constructed (indicated by a darker colour). Parts of this figure are inspired by Figure 2 in [9].

5.2.2 Compression of the signature grammar

One of the key properties of the signature grammar is that, as a compression of S, it is almost as good as the Lempel-Ziv77 parse. We will show that the size of the signature DAG (which is also the size of the signature grammar) is close to z, the size of the LZ77 compression of S. There have also been used *attractors* which give an even better bound of the size [9]. We now turn our attention back to the Lempel-Ziv77 parse and proving that the size of signature grammar is $\mathcal{O}(z \log(n/z))$.

Lemma 18 (Lemma 6 in [8])**.** *The expected size of the signature DAG $dag(S)$ is $\mathcal{O}(z \log(n/z))$ where z is the size of the Lempel-Ziv77 parse.*

The proof sketch is based on the proof of Lemma 6 in [22].

Proof sketch. The idea is to bound the number of unique vertices in $sig(S)$, because that is the number of vertices in $dag(S)$. The proof is split into two parts. First, we bound the number of unique nodes on levels lower than $\log(n/z)$ in $sig(S)$, and afterwards, we bound the number of nodes on levels higher than $\log(n/z)$ in $sig(S)$.

The number of nodes at levels lower than $\log(n/z)$ are counted by looking at the Lempel-Ziv77 parse phrases and borders, $S[u_1 \dots u_1 + l_1], S[u_1 + l_1 + 1] \dots S[u_z \dots u_z + l_1], S[u_z + l + 1]$, and creating a set of relevant nodes. Let us denote this set R, and define $R = \{T(u_1, u_1 + l_1), T(u_1 + l_1 + 1, u_1 + l_1 + 1), \dots, T(u_z, u_z + l_z), T(u_z + l_1 + 1, u_z + l_z + 1)\}$.

First, let us look at some border i. The relevant nodes of i, $T(i, i)$, is a path, since it corresponds to the relevant nodes of a single character in S. This means that $T(i, i)$ will only have $\mathcal{O}(log(n/z))$ nodes at levels lower than $\log(n/z)$.

A phrase $S[i \dots i']$ can have many more relevant nodes on levels lower than $\log(n/z)$, but here the local consistency of Corollary 17.1 comes into play. We use the fact that all phrases have a source (a first occurrence of the phrase), and that all the relevant nodes of a phrase only differ by four nodes at each level from the relevant nodes of its source. This means that at levels lower than $\log(z/n)$, the number of unique nodes of a phrase is $\mathcal{O}(\log(z/n))$. Therefore, for each of the z phrases of the Lempel-Ziv77 parse, only $\mathcal{O}(\log(z/n))$ are possibly unique. Thus the number of unique nodes at levels lower than $\log(n/z)$ is $\mathcal{O}(z \cdot \log(n/z))$.

There are at most $\log z$ levels left to look at after we have dealt with those at $\log(n/z) = \log n - \log z$. At level $\log(n/z)$ there are at most $\mathcal{O}(z)$ vertices[1]. Every vertex in these last $\log z$ levels has at least two children, meaning there can be at most $\mathcal{O}(z)$ nodes. Consequently the number of nodes in $dag(S)$ is $\mathcal{O}(z \log(n/z))$, and since the expected number of edges of a vertex in $sig(S)$ is constant (and a node in $dag(S)$ can never get more children than it had in $sig(S)$), the expected size of $dag(S)$ will be $\mathcal{O}(z \log(n/z))$. □

5.2.3 Conclusion on the signature grammar

The signature grammar has locally consistent $c_g = 2$ according to Definition 11 because it is shown in Corollary 17.1 that the parsing at each level at most has block-based locality 2 (meaning that at most the two leftmost and the two rightmost blocks differ for any pair of identical substrings). In addition, the signature grammar can be constructed in expected $\mathcal{O}(n)$ time, and its size is bounded by $\mathcal{O}(z \log(n/z))$, making it nearly as compact as the Lempel-Ziv77 parse.

5.3 The grammar derived from a (τ, δ)-partitioning set

The construction of both the τ-local minimum set partitioning and the signature grammar looks very similar to the deterministic constructions of the $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set, which introduces the natural thought of what would happen if Construction 2 were continued until only one block remained instead of stopping at level $\log_\sigma \tau$.

This would not be a grammar (of the type described earlier). It is, nevertheless, possible to turn it into one using the proposed alternative construction of Construction 2, where every block at every level is given a unique label, such that only identical blocks have identical labels. Thinking of the hierarchical structure of the continued Construction 2 as a tree, and then transforming it into

[1] This comes from the fact that at every level of the tree the number of nodes are at least halved, which means there are at most $n \cdot 2^{-x}$ nodes at level x. This means that there are at most z nodes at level $\log(n/z)$, because $n \cdot 2^{-\log(n/z)} = n \cdot (z/n) = z$.

an acyclic context-free grammar producing the string S (and only S) by merging all vertices with identical labels.

Lemma 19. *The grammar derived from a (τ, δ)-partitioning set has height at most $\mathcal{O}(\log n)$.*

Proof. From Lemma 10 we know that at level μ every pair of consecutive blocks is of size at least $(3/2)^\mu$. This means that there can at most be $2 \cdot \frac{n}{(3/2)^\mu}$ blocks at level μ, and at level $\mu = \log_{3/2}(2n)$ there will be at most $2 \cdot \frac{n}{(3/2)^{\log_{3/2}(2n)}} = 2\frac{n}{2n} = 1$. □

This new grammar has a block-locality which depends on the level of the grammar (this has been shown in Lemma 7). This means it is not possible to argue that the grammar is locally consistent even though at each layer of the construction two substrings can at most differ by $6 \cdot (3/2)^\mu c \log^* n$ blocks ($\mathcal{O}((3/2)^\mu \log^* n)$ blocks). The problem is that the locality depends on the current level, μ, and thus cannot be said to be the same for every layer of the grammar.

Chapter 6

Problems solved by locally consistent parsing

We move on to present some problems, which can be solved with the data structures introduced in the previous chapters. For each problem, we present the data structures and algorithms involved, along with proofs of correctness and runtime analyses. The goal is to demonstrate how the different kinds of locality are useful in different scenarios. The problems, which will be covered, are The Longest Common Extension Problem and The Text Indexing Problem. Before moving on to the problems and their solutions, a few definitions are needed.

Definition 12. Let S be a string of length n and $\mathcal{X}$ a partitioning of size $\mathcal{O}\left(\frac{n}{\tau}\right)$, then the partitioning string $S_{\mathcal{X}}$ is a string where each block induced by the partitioning gets a unique symbol, such that all identical blocks get the same symbol. $S_{\mathcal{X}}[i]$ is then the unique symbol of block i.

Lemma 20. *Let S be a string of length n and $\mathcal{X}$ a $(\alpha, \beta, \tau, \rho)$-locally consistent set of size $\mathcal{O}\left(\frac{n}{\tau}\right)$, then the partitioning string can be built in $\mathcal{O}(n)$ and takes up $\mathcal{O}\left(\frac{n}{\tau}\right)$ space.*

Proof. This proof is mostly citing Lemma F.5 from [5] which states that a set $\mathcal{O}\left(\frac{n}{\tau}\right)$ strings (all of size $\mathcal{O}(\tau)$) can be sorted in $\mathcal{O}(n)$ time . Now with this in hand, we can sort the $\mathcal{O}\left(\frac{n}{\tau}\right)$ string that the partitioning induces, and then every time we encounter a block which is not identical to the previous block, we give it an unused symbol (otherwise we give it the symbol of the previous block). The only case where a block (substring induced by the partitioning) can be larger than τ is if the block is periodic, with period $\rho < \tau$ which means that it is enough to keep the 2τ first characters of this block and describe them with a symbol and additionally append the length of the periodic block to this symbol as well. Then, to be identical, two periodic blocks must match on the first 2τ characters and have the same length.

The space consumption is derived trivially from the fact that there are $\mathcal{O}\left(\frac{n}{\tau}\right)$ blocks and each block is described with at most two characters from an alphabet of size $\mathcal{O}\left(\frac{n}{\tau}\right)$, which means that the length of $S_{\mathcal{X}}$ is $\mathcal{O}\left(\frac{n}{\tau}\right)$. □

6.1 Longest Common Extension with (τ, δ)-partitioning set

The Longest Common Extension Problem is defined the following way:

Longest Common Extension (LCE)
Input: A string S of length n and two indices $i, j \in [n]$.
Query: Given i and j find the longest common prefix of the substrings $T[i \dots n]$ and $T[j \dots n]$.

In the following sections, we explore how to solve the LCE problem using both a (τ, δ)-partitioning set and a τ-synchronising set.

6.1.1 Data structure

We begin by computing the *partitioning string* of the input string S, denoted as $S_{\mathcal{P}}$. Once this string is obtained, we construct a *suffix tree*, denoted $T_{\mathcal{P}}$, on the partitioning string $S_{\mathcal{P}}$. On top of the suffix tree we build a LCA-structure. The partitioning itself is stored in an ordered linked list, where each p_i in the list has a link to the leaf in the suffix tree that corresponds to the block for which p_i is a representative. Lastly, a data structure to answer $\text{succ}_{\mathcal{P}}(i)$ queries is created, which can answer queries in $\mathcal{O}(\tau)$[1].

6.1.2 LCE-query using partitioning sets (non-periodic case)

In the non-periodic case, answering an LCE (Longest Common Extension) query involves the following steps:

1. **Check the first 3δ positions:** We begin by directly comparing the first 3δ characters of the substrings starting at positions i and j in the original string S. If there is a mismatch, we terminate and return l (the length of the longest common extension).
2. **Find $\text{succ}_{\mathcal{P}}(i+\delta)$ and $\text{succ}_{\mathcal{P}}(j+\delta)$:** This is done by taking $SUC[\lfloor \frac{(i+\delta)}{\tau} \rfloor]$ and then doing a sequential search until an index larger than $i+\delta$ is found. The same thing is done for $j+\delta$.
3. **Compare blocks using LCA on the partitioned string:** We compare the blocks beginning at positions $\text{succ}_{\mathcal{P}}(i+\delta)$ and $\text{succ}_{\mathcal{P}}(j+\delta)$ and continue comparing them until a mismatch is found. To perform this efficiently, we use a Lowest Common Ancestor (LCA) data structure built on top of the suffix tree. This allows us to determine, in $O(1)$ time, how many partition blocks agree between the two substrings.
4. **Check the last $\delta + \tau$ positions:** After the LCA-based comparison, we check the last $\delta + \tau$ characters of the remaining substrings.

6.1.2.1 Correctness (non-periodic case)

To show correctness we will need the following Lemma.

Lemma 21. *Let S be a string with no periodic substrings with period $< \tau$. Let $i, j \in [1 \dots n]$ be two indices and let $l = LCE(i, j)$. If $l > 3\delta$, then the intervals $[i+\delta \dots i+l-\delta-1]$ and $[j+\delta \dots j+l-\delta-1]$ will be partitioned into blocks in exactly the same way, formally*

$$B_{\mathcal{P}}(i+\delta, i+l-\delta-1) = B_{\mathcal{P}}(j+\delta, j+l-\delta-1)$$

where $B_{\mathcal{P}}(i, j)$ is defined as in Definition 4.

The following proof is based on some observations from Lemma 3.2 in [5].

[1] This is done with the array SUC of length $\lfloor n/\tau \rfloor$ which for every $j \in [\lfloor n/\tau \rfloor]$ stores a pointer to the node of $\text{succ}_{\mathcal{P}}(j \cdot \tau)$ (in the suffix tree)[5].

Proof. Recall that the set $B_{\mathcal{P}}(i,j)$ is the set of distances to index i for all $p \in \mathcal{P}$ in the interval $[i \dots j]$. Now, let us assume that the Lemma does not hold. This means that there would be some $p'_i \in [i+\delta \dots i+l-\delta-1] \cap \mathcal{P}$ with the distance to i denoted γ ($\gamma := p_{i'} - i$ and per definition is $\gamma \geq \delta$), and the position $j+\gamma \notin \mathcal{P}$.

But we know that $S[i+\gamma-\delta \dots i+\gamma+\delta] = S[p_{i'}-\delta \dots p_{i'}+\delta] \subseteq S[i \dots i+l-1]$ and also that $S[j+\gamma-\delta \dots j+\gamma+\delta] \subseteq S[j \dots j+l-1]$. This means that $S[i+\gamma-\delta \dots i+\gamma+\delta] = S[j+\gamma-\delta \dots j+\gamma+\delta]$, and since $p_{i'} = i+\gamma \in \mathcal{P}$ and $\mathcal{P}$ being a (τ,δ)-partitioning set the locality condition of Definition 5 requires that $j+\gamma \in \mathcal{P}$ which contradicts $j+\gamma \notin \mathcal{P}$.

This means that if all $p \in \mathcal{P} \cap [i+\delta \dots i+l-\delta-1]$ will be distributed the same way as all $p \in \mathcal{P} \cap [j+\delta \dots j+l-\delta-1]$, which is the same as $B_{\mathcal{P}}(i+\delta, i+l-\delta-1) = B_{\mathcal{P}}(j+\delta, j+l-\delta-1)$, then it means that all blocks starting and ending in $[i+\delta \dots i+l-\delta-1]$ will be identical in $[j+\delta \dots j+l-\delta-1]$. □

Lemma 21 tells us that when the string S is non-periodic and $LCE(i,j) \geq 3\delta$ then $B_{\mathcal{P}}(i+\delta, i+l-\delta-1) = B_{\mathcal{P}}(j+\delta, j+l-\delta-1)$, which mean that $\text{succ}_{\mathcal{P}}(i+\delta) = \text{succ}_{\mathcal{P}}(j+\delta)$ (note that $\text{succ}_{\mathcal{P}}(i+\delta)$ is equal to i added to the first element (distance) of $B_{\mathcal{P}}(i+\delta, i+l-\delta-1)$).

It can be shown that with the previous mentioned conditions $B_{\mathcal{P}}(i+\delta, i+l-\delta-1)$ is non-empty. The intuition is that the if $l \geq 3\delta$ then the size of the interval $[i+\delta \dots i+l-\delta-1]$ is $(i+l-\delta-1)-(i+\delta)+1 = l-2\delta \geq 3\delta-2\delta = \delta$, and per the definition of the (τ,δ)-partitioning set $\tau \leq \delta$, and thus is $[i+\delta \dots i+l-\delta-1] \cap \mathcal{P} \neq \emptyset$, which means that $B_{\mathcal{P}}(i+\delta, i+l-\delta-1)$ is non-empty as well.

This shows that we only need to compare the first 3δ positions, $S[i \dots i+3\delta]$ and $S[j \dots j+3\delta]$, to ensure that we can find $\text{succ}_{\mathcal{P}}(i+\delta)$ and $\text{succ}_{\mathcal{P}}(j+\delta)$. Now these two indices will be identically placed, and we can move on to compare the blocks of the string induced by $\mathcal{P}$. Lemma 21 showed that as long the blocks starts and ends in $[i+\delta \dots i+l-\delta-1]$ and $[j+\delta \dots j+l-\delta-1]$ they are identical.

All that is left is to argue what happens when the first block mismatch occurs. Let $p_{i^*} \in \mathcal{P}$ be the element just after the first mismatched block. Hence, p_{i^*} marks the beginning of the block just after the first mismatched block. This tells us that $p_{i^*} \notin [i+\delta \dots i+l-\delta-1]$, because otherwise the previous blocks would have to match. Therefore, at most δ positions need to be compared after p_{i^*}. The last question is how many positions we must check before p_{i^*}. The previous block begin at p_{i^*-1} which is at most distance τ from p_{i^*}. The blocks before p_{i^*-1} matched and therefore the only positions we need to check are the ones between p_{i^*-1} and p_{i^*}, and the ones between p_{i^*} and $i+l-1$, and these distances are guaranteed to be at most $\tau+\delta$.

In this way, we have shown that the algorithm outputs that correct results and that it only checks $4\delta+\tau = \mathcal{O}(\delta)$ positions and make on call to the LCA data structure which takes $\mathcal{O}(1)$ which means that a query takes $\mathcal{O}(\delta)$.

6.1.3 LCE-query using partitioning sets (general case)

The procedure for answering an LCE (Longest Common Extension) query in the general case is quite similar to the non-periodic case, with adjustments to handle potential periodic blocks. The steps of the general query are:

1. **Check the first 3δ positions:** We begin by comparing the first 3δ characters of the substrings starting at positions i and j in S. If there is a mismatch, we terminate and return l (the length of the longest common extension).

2. **Check for common offset:** We examine the difference between $\text{succ}_{\mathcal{P}}(i+\delta)-i$ and $\text{succ}_{\mathcal{P}}(j+\delta)-j$. If these values are not equal, we find $\gamma = \min\{\text{succ}_{\mathcal{P}}(i+\delta)-i, \text{succ}_{\mathcal{P}}(j+\delta)-j\}$. Then we compare the substrings $S[i+\gamma \dots i+\gamma+\delta)$ and $S[j+\gamma \dots j+\gamma+\delta)$. From this comparison, the value l can be determined, and the query ends.

3. **Equal common offset case:** If $\text{succ}_{\mathcal{P}}(i+\delta) - i = \text{succ}_{\mathcal{P}}(j+\delta) - j$, we use the common offset to perform an LCA query. Each periodic block contains both the first 2τ characters and the length of the block. Therefore, for two periodic blocks to be equal in the context of the LCA query, their initial 2τ characters must match, and their lengths must be identical.

4. **Mismatch in the induced string:** If a mismatch is detected in the induced partitioning string $S_{\mathcal{P}}$, we proceed by checking the next 3δ elements in the original string S following the mismatch. If no mismatch has been found, then let p_{i^*} and p_{j^*} be the indices at the beginning of the first mismatched blocks. Then we compute $\gamma^* = \min\{\text{succ}_{\mathcal{P}}(p_{i^*}) - i, \text{succ}_{\mathcal{P}}(p_{j^*}) - j\}$. We then compare the substrings $S[i+\gamma \dots i+\gamma+\delta]$ and $S[j+\gamma \dots j+\gamma+\delta]$. This comparison will reveal a mismatch, concluding the query process.

6.1.3.1 Correctness (general case)

The following Lemma is essential for proving the correctness of the periodic query.

Lemma 22 (Lemma B.1 [5])**.** *Let $S[i \dots i+3\delta] = S[j \dots j+3\delta]$, $p_{i'} := succ_{\mathcal{P}}(i+\delta)$ and $p_{j'} := succ_{\mathcal{P}}(j+\delta)$. If $p_{i'} - i \neq p_{j'} - j$ then $LCE(i,j) \leq \gamma + \delta$, where $\gamma = \min\{p_{i'} - i, p_{j'} - j\}$.*

The following proof is based on the proof of Lemma B.1 in [5].

Proof. Let us assume that $p_{i'} - i \neq p_{j'} - j$ but $LCE(i,j) > \gamma + \delta$ in order to get a contradiction. Without loss of generality assume that $\gamma = \min\{p_{i'} - i, p_{j'} - j\} = p_{i'} - i$. From the assumptions and from the fact that $p_{i'} = \text{succ}_{\mathcal{P}}(i+\delta)$, we know that $\delta \leq p_{i'} - i = \gamma$ and $\gamma < LCE(i,j) - \delta$. This means that $0 \leq \gamma - \delta$. The interval $[i+\gamma-\delta \dots i+\gamma+\delta]$ is thus a subset of $[i \dots i+3\delta]$. Therefore is $S[i+\gamma-\delta \dots i+\gamma+\delta] = S[p_{i'}-\delta \dots p_{i'}+\delta] = S[p_{j'}-\delta \dots p_{j'}+\delta]$ but then according to the definition of a (τ,δ)-partitioning set $p_{i'} \in \mathcal{P}$ if and only if $j + (p_{i'} - i) \in \mathcal{P}$ but this contradicts our original assumptions that $p_{i'} - i$ was minimal and $p_{i'} - i \neq p_{j'} - j$. □

Going through each of the three phases and using Lemma 22 we will show that the periodic query is almost similar to the non-periodic one. When checking the first 3δ elements, we risk $\text{succ}_{\mathcal{P}}(i+\delta) - i \neq \text{succ}_{\mathcal{P}}(j+\delta) - j$ if we encounter a period substring. But then using Lemma 22 we know that the blocks before $p_{i'}$ and $p_{j'}$ must be periodic (this is the only reason for $p_{i'} - i \neq p_{j'} - j$) with period smaller than or equal to τ. But we know the first 3δ characters are identical, which means that the two periodic blocks have the same period but not the same length.

Now, let $\gamma = \min\{p_{i'} - i, p_{j'} - j\}$ then we know that $S[i \dots i+\gamma-1] = S[j \dots j+\gamma-1]$. Due to Lemma 22, the value of $LCE(i,j)$ is at most $\gamma + \delta$. This means we only have to compare the δ position of the substrings $S[i+\gamma \dots i+\gamma+\delta)$ and $S[j+\gamma \dots j+\gamma+\delta)$.

6.1.4 Runtime

We start by comparing 3δ elements, then finding the common offset in $\mathcal{O}(\tau)$ using the ordered linked list, then use the LCA-data structure to query in constant time, and lastly we check at most 3δ positions (in the case of periodic behaviour). This means that the runtime of a query will be $\mathcal{O}(\delta)$.

6.2 Longest Common Extension with τ-synchronising set

When Kempa and Kociumaka [22] created the τ-synchronising set, they focused on answering LCE-queries in constant time. This was possible if τ was $\mathcal{O}(\log_\sigma n)$ for a string of length n and an alphabet of size σ. Therefore, the algorithm and data structure are designed with this purpose in mind, even though it can be shown that it runs in $\mathcal{O}(\tau)$ if τ is not bounded and a LCE data structure is built on top of the partitioning string $S_{\mathcal{S}}$. We will focus on explaining the bounded LCE-queries.

6.2.1 Data structure

The data structure consists of three main components. First, we define a string S', where each character corresponds to the first 3τ characters of a block in the set τ-synchronising set. This string can be interpreted as a partitioning string in which each unique symbol represents a (possibly compressed) encoding of the first 3τ characters of the corresponding block. The only reason we are not using $S_{\mathcal{S}}$ is to keep the query time down. When $\tau = \mathcal{O}(\log_\sigma n)$, it is possible to compare 3τ characters of two different substrings in constant time. In the periodic case, each block also stores additional information, denoted d_i. This constant will not be explained in depth, but in simplified terms, it stores the length of the periodic string.

Additionally, we will need a bit-vector, B, of length n with a 1 at every index of $s_i \in \mathcal{S}$ and 0 otherwise. B is extended to answer efficient rank queries, meaning that given any index $i \in [n]$ we can get $\text{succ}_{\mathcal{S}}(i)$ in constant time. It has been shown how to do it by Jacobson [19](and it has later been shown how to implement it efficiently [18] [2]).

Finally, an LCE data structure is built on top of the string S', enabling constant-time queries for the longest common prefix between substrings in the partitioned representation.

6.2.2 LCE-query using synchronising sets (non-periodic case)

To answer an LCE query in the non-periodic case, the following steps are performed:

1. **Compare the first 3τ symbols:** We begin by directly comparing the first 3τ characters of the two substrings starting at positions i and j. If there is a mismatch, we terminate and return l (the length of the longest common extension).

2. **Compute the common offsets:** We determine the positions where the partitioned representations begin to align by computing successor positions after the initial 3τ positions.

3. **Use the LCE data structure:** We use the LCE structure built over S' to find the first pair of blocks that mismatch. If such a mismatch is found, we conclude that the two substrings differ within the next $< 3\tau$ characters.

4. **Compare the last 3τ characters:** After identifying the potential mismatch via the partitioned representation, we verify the result by comparing the final 3τ characters directly in the original string.

6.2.2.1 Correctness (non-periodic case)

Correctness is very similar to that of (τ, δ)-partitioning set in Lemma 21 expect instead of finding $\text{succ}_{\mathcal{P}}(i + \delta)$ and $\text{succ}_{\mathcal{P}}(j + \delta)$ we find $\text{succ}_{\mathcal{S}}(i)$ and $\text{succ}_{\mathcal{S}}(j)$. This is due to the locality of the τ-synchronising set compared to that of (τ, δ)-partitioning set, where with τ-synchronising set we only look to the right in the string, which means that we do not need to skip δ or τ steps before finding the common offset.

As it is very similar to the proof of (τ, δ)-partitioning set, we will reference the proof of Kempa and Kociumaka [22] and instead talk more about correctness for LCE-queries with τ-synchronising set in the general case later on.

6.2.3 LCE-query using synchronising sets (general case)

The general-case LCE query is an adjusted version of the non-periodic procedure. The process consists of the following steps:

1. **Compare the first 3τ symbols:** As in the non-periodic case, we compare the first 3τ characters of the substrings starting at positions i and j. If there is a mismatch, we terminate and return l (the length of the longest common extension).

2. **Compute the common offsets:** We then compute the offsets from i and j to their respective successor positions in the partitioning. If the computed offsets differ, it indicates that the substrings begin in periodic blocks of different lengths. In this case, the substrings can only match up to the length of the shorter block plus an additional $2\tau - 1$ characters.
3. **Use the LCE structure to find the first mismatching blocks:** If the offsets are equal or periodic blocks of the same length are suspected, we proceed by using the LCE structure on S' to identify the first pair of mismatching blocks. If a mismatch is found, the longest common extension is less than 3τ, unless the blocks are both periodic. In the periodic case, they may still agree for up to the length of the shorter block plus $2\tau - 1$ characters.
4. **Compare the last 3τ characters:** After the LCE comparison, we examine the final 3τ characters of the current matching region. If a mismatch is found, the query terminates. Otherwise, we report the result as $\min\{s_{i+1} - s_i, s_{j+1} - s_j\} + 2\tau - 1$, where s_i and s_j are the partition boundaries for the substrings starting at i and j, respectively.

6.2.3.1 Correctness (general case)

The essential part is to show that computing the $LCE(i, j)$ in the string S', which we from now on will denote $LCE_{S'}(i, j)$ is the same as computing it for S given that $LCE(i, j) \geq 3\tau - 1$ and expect for some "border behaviour". This behaviour can be dealt with by manually checking 3τ characters, which is shown by Kempa and Kociumaka [22] in the following Lemmas taken directly from their paper [22].

Lemma 23 (Lemma 5.2, [22]). *For positions i, j in S such that $LCE(i, j) \geq 3\tau - 1$, let us define $s_{i'} = succ_{\mathcal{S}}(i)$ and $s_{j'} = succ_{\mathcal{S}}(j)$. Then*

$$LCE(i,j) = \begin{cases} \min\left\{s_{i'} - i, s_{j'} - j\right\} + 2\tau - 1 & \text{if } s_{i'} - i \neq s_{j'} - j, \\ s_{i'} - i + LCE\left(s_{i'}, s_{j'}\right) & \text{if } s_{i'} - i = s_{j'} - j. \end{cases}$$

Lemma 24 (Lemma 5.3, [22]). *If $l = LCE_{S'}(i, j)$ for positions $i, j \in [1 \dots n' + 1]$, then $LCE(s_i, s_j) = s_{i+l} - s_i + LCE(s_{i+l}, s_{j+l})$. Moreover, $LCE(s_i, s_j) < 3\tau$ or $LCE(s_i, s_j) = \min\{s_{i+1} - s_i, s_{j+1} - s_j\} + 2\tau - 1$ holds if $l = 0$.*

Lemma 23 demonstrates that if the two substrings are the same for more than 3τ positions then the either have a common offset (an offset with in the same distance as both i and j), or the substrings exhibit periodic behaviour and the LCE query can be resolved by jumping to the closest (in relative distance to i and j) of the elements in $\mathcal{S}$ and adding additionally $2\tau - 1$ (because of the way $\mathcal{R}$ is defined (Equation (2.3))).

Lemma 24 exhibits that it is possible to do most of the work on the compressed string, S'. The lemma also states that once the blocks no longer match (if $l = 0$), then either the remaining number of matching characters is $< 3\tau$ or there is periodic behaviour. Once again, the remaining overlap of the substrings is the smallest distance of $s_{i+1} - s_i$ and $s_{j+1} - s_j$, and an additional $2\tau - 1$ positions.

Together, these two Lemmas show almost the same properties as those of Lemma 21 and Lemma 22.

6.2.4 Runtime

Since $\tau \in \mathcal{O}(\log_\sigma n)$ we can compare 3τ characters from the alphabet $\Sigma = \{0, \dots, \sigma - 1\}$ in $\mathcal{O}(1)$ time. This requires the concept Kempa and Kociumaka [22] introduces as *the compact representation*[2] of S.

[2]This representation uses the fact that the RAM model assumes every characters takes up $w = \mathcal{O}(\log n)$ bits, but in reality can we use $\log \sigma$, where σ is the size of the alphabet. Using this fact we can store S in $\mathcal{O}\left(\lceil \frac{n \log \sigma}{w} \rceil\right)$. This is done by storing the first character in the first $\lceil \log \sigma \rceil$ bits, the next character in the following $\lceil \log \sigma \rceil$ bits and so on.

We can find the successor using rank queries on the bit-vector B, this also takes constant time. The LCE data structure for the $LCE_{S'}(i', j')$-query takes $\mathcal{O}(1)$ time.

Then according to Lemma 24 we can do this by comparing at most 3τ characters and making one additional computation. This takes $\mathcal{O}(1)$ time.

Since there are a constant number of steps which all takes $\mathcal{O}(1)$ we end up having a runtime of $\mathcal{O}(1)$.

6.3 Comparing the two different LCE solutions

Using the tricks from τ-synchronising set [22] and bounding τ (or δ) to be $\mathcal{O}(\log_\sigma n)$ it is possible to make the an LCE-query run in constant time for (τ, δ)-partitioning set as well.

If we use the rank-queries and use the compact representation of S such that we can compare δ elements in $\mathcal{O}(1)$ time, then will the LCE-query with (τ, δ)-partitioning set also take constant time. The only time spent in the algorithm is comparing the 3δ first elements, computing $\text{succ}_{\mathcal{P}}(i + \delta)$ and $\text{succ}_{\mathcal{P}}(j + \delta)$, and lastly possibly checking the 3δ elements. If all these operations can be done in constant time, then the algorithm will indeed also be constant. The creators of τ-synchronising set, Kempa and Kociumaka [22], made a small remark about this in their introduction [22].

This means that even though (τ, δ)-partitioning set and τ-synchronising set might look slightly different, they still solve at least this problem very similarly.

6.4 Pattern matching using the signature grammar

We now go on to solve the text indexing problem using the locally consistent grammar introduced in Chapter 5.

> The Text Indexing Problem
> **Input:** A string S of length n and a pattern P of length m.
> **Query:** The indices of all occurrences of P in S.

In their paper, Christiansen and Ettienne [8] distinguishes between different pattern lengths. If $m \geq \log^\epsilon n$, then we say the P is a *long* pattern. If $m \leq \log\log n$ then P is *short* and if $\log^\epsilon n > m > \log\log n$ it is a *semi-short* pattern.

In this thesis, we will describe how to handle long patterns in depth, and only briefly mention the results for short and semi-short patterns.

6.4.1 Data structure

First, we compute the *signature DAG* of the string S, denoted as $dag(S)$. Let v be a vertex in the signature DAG $dag(S)$, and let $u_1, \ldots, u_k$ be the children of v. We define two functions: $pre(v, i)$ and $suf(v, i)$. The function $pre(v, i)$ returns the string corresponding to the first i children of v (from left to right), and $suf(v, i)$ returns the string corresponding to the last $k - i$ children. These functions are defined such that the concatenation $pre(v, i)suf(v, i)$ equals $str(v)$, the string represented by vertex v. The functions are illustrated by an example in Figure 26.

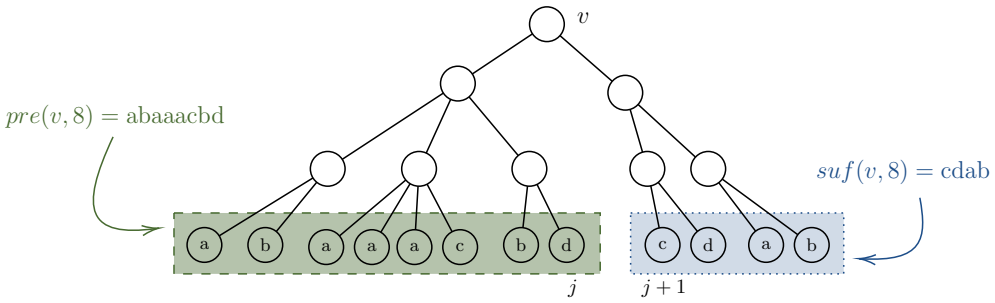


Figure 26: The figure shows an example of how the functions pre and suf works. If the characters of the leaves in the green (dashed) box are concatenated, then we get exactly $pre(v, 8)$. Thus in this example $pre(v, 8) = \text{abaaacbd}$ and $suf(v, 8) = \text{cdab}$.

We use two *z-fast tries*, as defined in Section 1.2.4, which we denote T_1 and T_2. When searching for a prefix in either trie, we use the Rabin-Karp fingerprint of the query prefix (see Definition 2). A query to a trie returns the rank interval of all strings with the query pattern, denoted P', as a prefix. Specifically, it returns two values x and y, where x is the rank of the smallest and y is the rank of the largest string (in lexicographical order) having P' as a prefix. Therefore, all strings with ranks in the interval $[x \dots y]$ share this prefix. Recall that if no prefix is found, the query returns some arbitrary ranks. Then we also use 2D range reporting structure (as introduced in Section 1.2.5), denoted R.

For every run-free, non-leaf node $v \in dag(S)$, we define k as the number of children of v. If v is a run node, we set $k = 2$. Then, for every $v \in dag(S)$, we perform the following three steps. First, for each $i \in [1, \dots, k-1]$, we insert the *reverse* of $pre(v, i)$ into the trie T_1. Second, for the same range of i, we insert $suf(v, i)$ into the trie T_2. Finally, we insert a point (x, y) into the data structure R, where x is the rank of the reverse of $pre(v, i)$ in T_1, and y is the rank of $suf(v, i)$ in T_2. Each point (x, y) additionally stores a reference to the vertex v and the length of the string $pre(v, i)$ as associated information.

6.4.2 Text indexing queries using the signature grammar (for long patterns)

Before introducing the query, a couple of definitions are needed.

Definition 13. A vertex v *stabs* a position i if v is the lowest common ancestor to the ith and the $(i+1)$th leaf (as seen Figure 27).

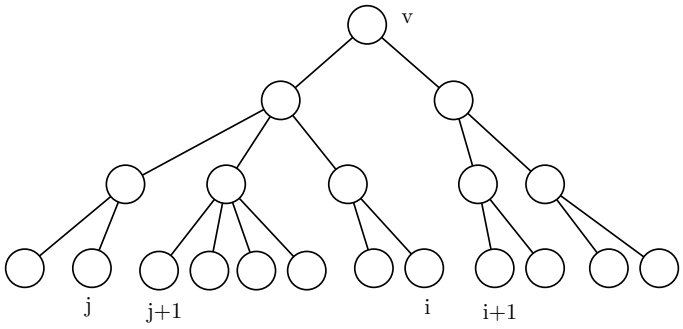


Figure 27: The vertex v stabs position i but not position j.

If a vertex v is the lowest common ancestor of a substring $S[i \dots i']$, we also say that v *stabs the occurrence*. Note that if v stabs the occurrence $S[i \dots i']$ then it must also stab some index in $[i \dots i']$. This is because it must have at least two children; otherwise, it would not be the lowest common

ancestor. Then if we pick two consecutive children, the index of the rightmost leaf of the left child will be in $[i \dots i']$ and v will stab this position.

Definition 14. Given a pattern P and its signature tree $sig(P)$, let l_i^k denote the kth leftmost vertex on level i, and let r_i^k denote the kth rightmost vertex at level i.

A very important concept we need to have in place it the concept of *consistency*. Given two substrings $S[i \dots i'] = S[j \dots j']$ of S then the consistent nodes of $T(i, i')$ (with respect to $T(j, j')$) are those nodes which are also guaranteed to be in $T(j, j')$. From Lemma 14 we know that only the two leftmost or rightmost nodes in each level of $T(i, i')$ can be inconsistent. Lastly, we introduce the set P_S^* defined as follows.

Definition 15. Given a pattern P and its signature tree $sig(P)$ where j is the last level then define

$$P_L^* = \{ \left|str\left(l_1^1\right)\right|, \left|str\left(l_1^1\right)\right| + \left|str\left(l_1^2\right)\right|, \left|str\left(l_1^1\right)\right| + \left|str\left(l_1^2\right)\right| + \left|str\left(l_1^3\right)\right|, \dots,$$
$$\left|str\left(l_j^1\right)\right|, \left|str\left(l_j^1\right)\right| + \left|str\left(l_j^2\right)\right|, \left|str\left(l_j^1\right)\right| + \left|str\left(l_j^2\right)\right| + \left|str\left(l_j^3\right)\right| \}$$

and define

$$P_R^* = \{ \left|str\left(r_1^1\right)\right|, \left|str\left(r_1^1\right)\right| + \left|str\left(r_1^2\right)\right|, \left|str\left(r_1^1\right)\right| + \left|str\left(r_1^2\right)\right| + \left|str\left(r_1^3\right)\right|, \dots,$$
$$\left|str\left(r_j^1\right)\right|, \left|str\left(r_j^1\right)\right| + \left|str\left(r_j^2\right)\right|, \left|str\left(r_j^1\right)\right| + \left|str\left(r_j^2\right)\right| + \left|str\left(r_j^3\right)\right| \}$$

then $P_S^* = P_L^* \cup P_R^*$.

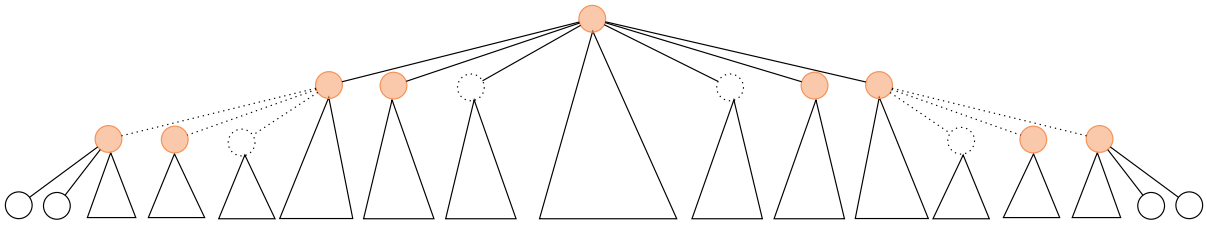


Figure 28: An example of a signature tree on some pattern P where the coloured vertices are the inconsistent ones. The uncoloured triangles indicate subtrees purely made out of consistent vertices. It is also easy to imagine where the set of indices P_S^* is located in the pattern P by looking at the marked (orange) vertices. There is an index $i \in P_S^*$ at every position after the three leftmost vertices and before the three rightmost vertices.

The procedure for reporting all occurrences of a pattern P of length m in the string S proceeds as follows:

1. **Compute Rabin-Karp fingerprints:** We begin by computing the Rabin-Karp fingerprints of all prefixes of P. These fingerprints are $\phi(P[1]), \phi(P[1 \dots 2]), \dots, \phi(P[1 \dots m])$, and can be computed in $\mathcal{O}(m)$ time. With these precomputed fingerprints, we can compute any substring fingerprint $\phi(P[i \dots j])$ in constant time, as established in Lemma 1 of [4].

2. **Construct the signature grammar:** Next, we construct the signature grammar of P, denoted $sig(P)$.

3. **Compute the set P_S^*:** We then compute the set P_S^*, as defined in Definition 15, which identifies valid split points of the pattern for searching.

4. **Search using z-fast tries.** For every position $p \in P_S^*$, we perform the following:
   - Search for the reverse of the prefix $P[1 \dots p]$ in the z-fast trie T_1.
   - Search for the suffix $P[p+1 \dots m]$ in the z-fast trie T_2.

   Both searches are performed using the precomputed Rabin-Karp fingerprints. The reverse prefix search in T_1 returns a range of ranks (a, b) corresponding to all strings in T_1 that have the reversed prefix as a prefix. Similarly, the search in T_2 returns a range (c, d).

5. **Perform 2D range query:** A 2D range query is executed over the range $(a, b) \times (c, d)$ in the 2D range reporting data structure. Each point returned corresponds to a pair (v, i), where v is a node in the signature DAG of S and i is a position such that the pattern P occurs in $S(v)$ starting at position i.

6. **Handle run-nodes:** If the node v is a run-node, then there may be additional occurrences of P at positions $i + q \cdot |S(\text{child}(v))|$ for values of $q \in [1, \dots, j]$, where j is the largest integer such that $j \cdot |S(\text{child}(v))| + m \leq |S(v)|$. These additional occurrences account for repeated patterns inside the run-node.

7. **Propagate matches up the DAG:** To report all pattern occurrences in S, we traverse the DAG upward from each matching node v. For each such node, we follow its ancestors:
   - If v has more than one parent, we follow each of them to continue reporting occurrences.
   - If v has only one parent, we follow the chain upward until reaching an ancestor with two or more parents, or until the root is reached.

   This process can be performed efficiently by storing, during the construction of the DAG, a pointer from each node to its nearest ancestor with more than one parent. These pointers are computed in $\mathcal{O}(n)$ time via a top-down traversal of the DAG.

Steps 4 and 5 of the query are illustrated in Figure 29 below.

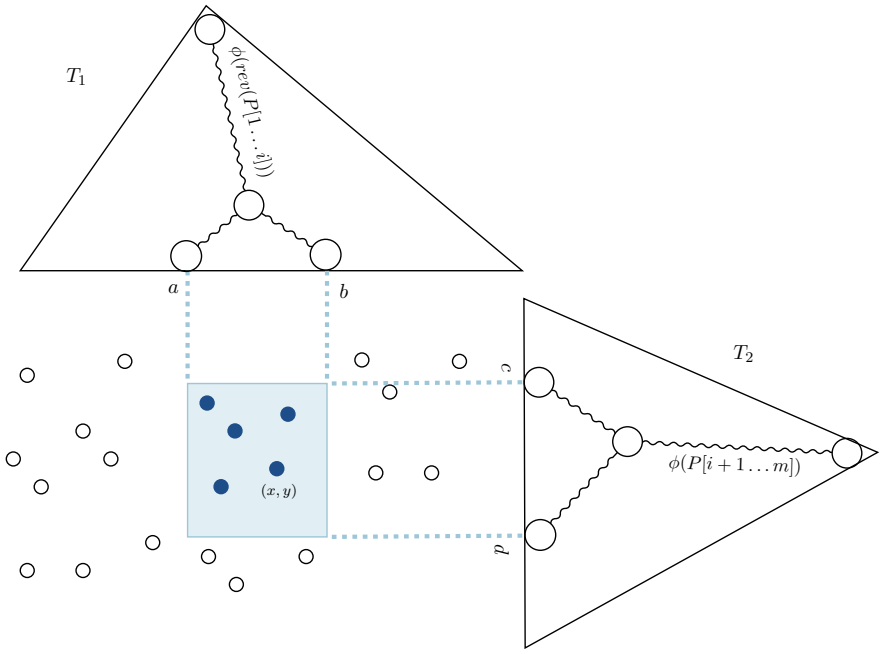


Figure 29: The figure illustrates a text indexing query given a splitting point $i \in [1 \cdots - 1]$. First is the range $\phi(rev(P[1 \dots i]))$ found in T_1 and then the range for $\phi(P[i+1 \dots m])$ in T_2. These ranges are used in a 2D-query and blue (marked) dots represent vertices $v \in sig(S)$ where $str(v) \supseteq P$. It can also be noted that these vertices stab at least one occurrence of P in S, meaning that for each v there is an occurrence $S[l \dots r]$ for which v stabs $l + i$.

6.4.2.1 Correctness

Let $S[l \dots r]$ be an occurrence of the pattern P in S. There will always exist some node v' in $sig(S)$ which is the lowest common ancestor to leaf l and leaf r, and therefore, every occurrence $S[l \dots r]$ of P will have a corresponding node v' and $str(v')$ contains all of P.

We say that v' which stabs $S[l \dots r]$ which means that some suffix of $pre(v', k)$ equals $P[1 \dots j]$ and some prefix of $suf(v', k)$ which equals $P[j+1 \dots m]$ for some $k \in [1 \dots child(v') - 1]$ with $child(v)$ being the number of children of v' and $j \in [1 \dots m-1]$.

Due to our data structure, any occurrence of P in S will be found if a correct splitting point $i \in [1 \dots m-1]$ is identified. We could check all splitting points in P, meaning the number of queries would be $\mathcal{O}(m)$. Let $T_S = T(l, r)$ that is; let T_S be the relevant nodes for the substring $S[l \dots r]$ with regards to $sig(S)$. Additionally, let $T_P = sig(P)$, and v' be a node that stabs the occurrence $S[l \dots r]$.

Christiansen and Ettienne [8] proposed that it was only needed to check four positions per layer in $sig(P)$, which is of height $\mathcal{O}(\log m)$. This would reduce the number of queries to $\mathcal{O}(\log m)$. This thesis shows that it is not enough to check only four positions per layer, but if six positions per layer are checked, then the statement holds. Note that checking six positions per layer in $sig(P)$ still yields $\mathcal{O}(\log m)$ queries.

The reason that only checking four points does not work is because the following assumption does not hold: if a vertex v in T_P is inconsistent but has two consistent children u_l and u_r (with respect to T_P) then their corresponding vertices u'_l and u'_r in T_S must be children of the same vertex (the counterexample to this statement is given in Figure 30).

$$\pi(\alpha) < \pi(\beta) < \pi(\gamma) < \pi(\delta) < \pi(\epsilon) < \pi(\zeta) < \pi(\eta)$$

Figure 30: This is a counterexample to the claim that if two consistent vertices have the same parent in T_P, they also have the same parent in T_S. Imagine that the ranking function π has ranked the Greek letters from α to η in their natural order ($\pi(\alpha) < \pi(\beta) < \pi(\gamma) < \pi(\delta) < \pi(\epsilon) < \pi(\zeta) < \pi(\eta)$). Then let us focus on the two consistent nodes with the labels ϵ and γ. The node with the label γ is not a local minimum in T_P because $\pi(\gamma) > \pi(\beta)$, but because the node to the right of the node with label γ is inconsistent it can have another label in T_S (because the label is defined by which children the node has and they may differ in T_P and T_S). This other label could be ζ such that the consistent node with label γ suddenly has a local minimum label and thus begins a new block (represented as a new parent node) in the next level. This results in two consistent nodes with the same parent in T_P, which have different parents in T_S.

Showing that only checking four positions at every level is insufficient is done with a counterexample, which arises from the earlier discussed assumption. The counterexample is illustrated in Figure 31 and will also be explained using this figure. Assume that the vertex v' that spans all of P has only inconsistent children and only stabs position $i' := i + l$. Now, if the label of u'_r is a local minimum in T_S, but the corresponding (and identical) node u_r in T_P does not have a locally minimal label then position i' is never considered as a splitting point since the corresponding index in T_P, i, will lie between consistent vertices in every level of T_P.

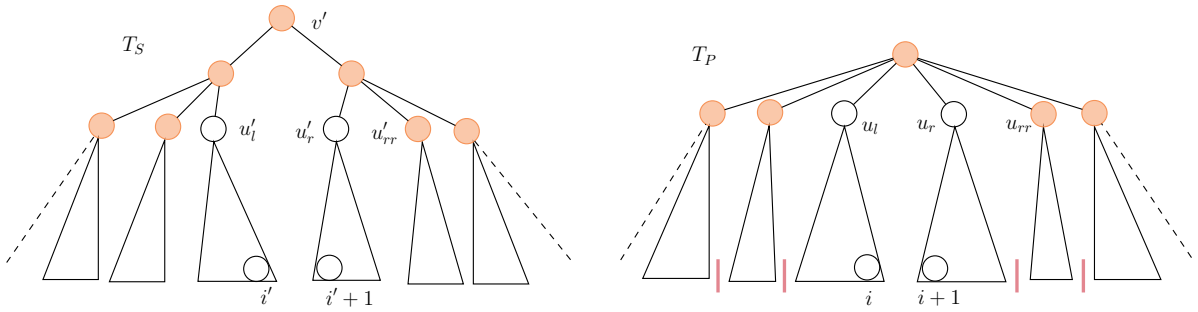


Figure 31: This shows a possible scenario where $str(v')$ contains P but the only splitting point for v' is index $i' := l + i$ and the corresponding split point could be between two consistent subtrees in T_P. The vertical lines at the bottom of T_P mark those indices considered possible split points for v' (at the second highest level of T_P). None of these will find the actual split point i.

The set of possible split points from Christiansen and Ettienne [8] is called P_S. We will show that every occurrence of P in S will be found using only positions from the superset P_S^* defined in Definition 15. The following Lemma will show that only $\mathcal{O}(\log m)$ queries are needed.

Lemma 25. *The only relevant split points of pattern P are contained in P_S^*.*

Let $S[l \dots r]$ be an occurrence of P in S, and let the node $v' \in sig(S)$ be the node that stabs this occurrence. T_S are the relevant nodes of the occurrences with regards to the signature tree of S, $sig(S)$, and T_P is the signature tree of P, $sig(P)$.

v' is going to an inconsistent node (with respect to T_P). A consistent node cannot have inconsistent children, and since v' spans all of P, this includes the inconsistent nodes. Assume that there were some inconsistent nodes in T_P called u. Then $str(u)$ could never be spanned by v' in T_S because it would only have consistent children. Furthermore, v' will have two or more children in $T(l, r)$. Both the lth leaf and the rth leaf are in $T(l, r)$, since v' is their lowest common split point, $i \in [1, m-1]$ will always exist which meets for the first time in v'.

v' is going to have at least one inconsistent child. This is the same argument as before. Assume that v' only had consistent children, then it would never be able to span all of P, since there would always exist some inconsistent u in T_P, where $str(u)$ would not be spanned by v'. Now we will argue that there will always exist a split point $i \in [1, m-1]$, which corresponds to a position in P_S^*. We will look at two cases. Either v' has at least one consistent child or only inconsistent children. Let $i' := l+i$.

Case 1 (v' has at least one consistent child): Let us, without loss of generality, assume that the consistent child of v' has an inconsistent node to its left. Let the consistent node be denoted u'_r and the inconsistent node u'_l (see Figure 32). The leftmost leaf of u'_r has index $i' + 1$.

Because u'_r is a consistent node, there will be an identical node in T_p, which we will denote u_r. The leftmost leaf of this node will be $i + 1$. Now, the node to the left of u_r must be inconsistent, because if it were consistent, it would have to be consistent in T_S also, and there, the node to the left of u'_r is inconsistent.

Let the inconsistent node to the left of u_r be denoted u_l. Note that the u_l and u'_l need not be identical since both are inconsistent, but they will have the ith (or the i'th) leaf as their rightmost index. This comes from the fact that the subtree of both u_r and u'_r stops at $i+1$ (or $i'+1$ respectively), and thus the previous blocks represented by node u_l and u'_l need to end at index i (or i'). This corresponds to u_l containing the ith leaf as its rightmost leaf and u'_l containing the i'th leaf as its rightmost leaf.

Therefore, no matter how u'_l and u_l look, index i will be checked as a splitting point because it is in T_P and will be situated between an inconsistent and a consistent node, which means it will be in P_S^*.

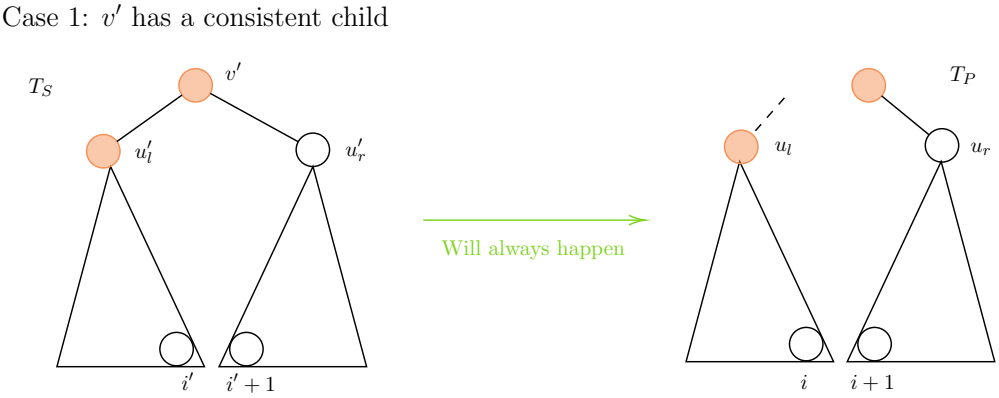


Figure 32: If v' has at least one consistent child, T_P will always have the split point i between a consistent and an inconsistent vertex.

Case 2 (v' has no consistent children): The second case is proven by contradiction. We assume that v' has no consistent children, and that the index i which v' stabs will have a corresponding (but not identical) vertex v which stabs index i in T_P, and that the ith and $i+1$th leaves would be in consistent subtrees meeting in v. The roots of these subtrees are denoted u_l and u_r, and we assume that neither of these two will have an inconsistent vertex as their neighbouring nodes. This would be the only time the split point i is not found.

Per our assumption, the neighbours of u_l and u_r are consistent vertices. Furthermore, u_l and u_r have the same parent v in T_P. But then there exists a corresponding pair of vertices u'_l and u'_r in T_S, and because they are also in T_S and have no inconsistent neighbours, they will also have the same parent. The only way they could have a different parent in T_S was if u_r became a local minimum (which it was not in T_P), but for this to happen the vertex to the right of u_r would have to be inconsistent such that it could be responsible for making the label of u'_r a local minimum in T_S and stopping the label of u_r from becoming one in T_P.

This means that if u_l and u_r have no inconsistent nodes as neighbours in T_P, the children of v' containing the i'th and $i'+1$th leaf should both be consistent, which is a contradiction to our original assumption.

Thus, we conclude that if v' only has inconsistent children, there will be a splitting point i which does not lie between two consistent nodes with no inconsistent neighbours and therefore will be in the set P^*_S (shown in Figure 33).

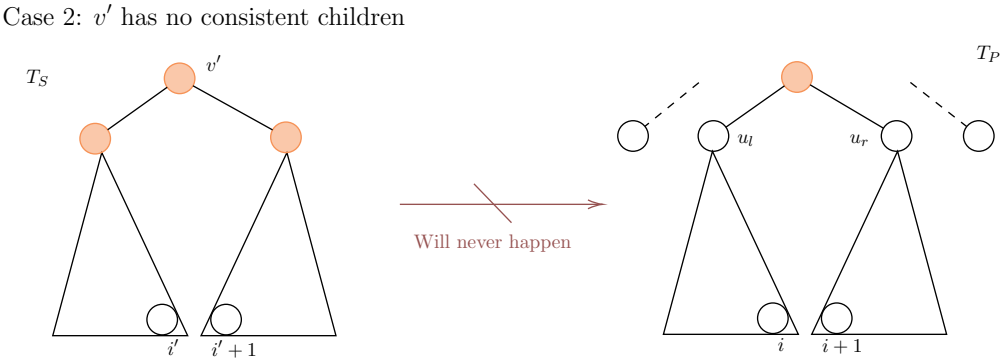


Figure 33: If v' has no consistent children, then the consistent nodes u_l and u_r which contains the ith and $i+1$th leaf and have no inconsistent neighbours cannot exist in T_P.

Now we can conclude that an occurrence of the pattern P will always have a vertex v' in T_S, where $str(v')$ spans the whole pattern, and this vertex v' will stab some index i', which will correspond to a splitting point i contained in the set P^*_S.

6.4.3 Complexity

The queries on the z-fast tries T_1 and T_2 takes $\mathcal{O}(\log m)$ time [14]. A query on the 2D-range structure R takes $\mathcal{O}(\log^{\epsilon}(z\log(n/z)))$ this was corrected by Christiansen et al. [9] because Christiansen and Ettienne [8] claimed that it was only $\mathcal{O}(\log^{\epsilon} z)$. $\mathcal{O}(\log m)$ queries are done, since P_S^* is size $\mathcal{O}(\log m)$[3]. Verification of the $\mathcal{O}(\log m)$ weak prefix queries are done in $\mathcal{O}\left(\log^2 m + m\right)$ time ([8] by citing Gagie et al. [16]). Building the signature DAG of P takes $\mathcal{O}(m)$. The runtime with reporting is therefore:

$$\mathcal{O}(m + (occ+1)(\log m \cdot (\log m + \log^{\epsilon}(z\log(n/z))) = \mathcal{O}\left(m + (1+occ)\log^2 m \cdot \log^{\epsilon}(z\log(n/z)\right).$$

Christiansen and Ettienne [8] bound reduces the expression further by differentiating between $m \leq \log^{2\epsilon} z$ and $m > \log^{2\epsilon} z$. They show that it is possible to reduce the runtime expression to:

$$\mathcal{O}(m + (1+occ)\log^{\epsilon'}(z\log(n/z)) \text{ where } \epsilon' > \epsilon.$$

Let us now examine the space of the data structure. We showed in Lemma 18 that the size of $dag(S)$ is $\mathcal{O}(z\log(n/z))$, and this is also the number of nodes in the DAG. Therefore and because of Lemma 2 is the size of T_1 and T_2 at most $\mathcal{O}(z\log(n/z))$ (this is elaborated in [8]). Lastly, the 2D-range data structure uses space linearly in the number of points it contains, and since we store a constant number of points (on average) per vertex in $dag(S)$, the size of this structure is also $\mathcal{O}(z\log(n/z))$. In total, this data structure uses $\mathcal{O}(z\log(n/z))$ space.

6.4.4 Queries for short and semi patterns

The solution for short and the one for semi-short patterns found by Christiansen and Ettienne [8] are based on the LZ77-parse of S. This results in a runtime of $\mathcal{O}(m + occ)$ for short patterns and $\mathcal{O}(m + occ(\log\log n + \log^{\epsilon} z)))$ for semi-short patterns, with z being the size of the LZ77 parse of S and ϵ is any positive constant $< \frac{1}{2}$. Their overall result (independent of the pattern size) is that the runtime with reporting is $\mathcal{O}(m + \log^{\epsilon}(z\log(n/z)) + occ(\log\log n + \log^{\epsilon} z))$ [8, Theorem 2(3)] (revised by Christiansen et al. [9]).

[3]The signature grammar has height $\mathcal{O}(\log m)$ and we only include a constant number of position in P_S^* per level.

Chapter 7

Conclusion and future work

7.1 Conclusion on the partitionings

Throughout this thesis, we have drawn many parallels between the (τ, δ)-partitioning set and the τ-synchronising set. They have very similar randomised constructions, they can both be defined as locally consistent sets (Definition 5), when working with a string, which has no periodic behaviour. They can be reduced to each other, and lastly they solve The Longest Common Extension Problem using a similar setup and query with the same runtimes.

The (τ, δ)-partitioning set and the τ-synchronising set have so much in common that they only differentiate from each other on the subject of how they treat highly periodic behaviour.

In addition, we have τ-local minimum set, which only has expected index-based locality. It cannot compete with the two other partitionings when solving, for instance, The Longest Common Extension Problem, but since this thesis is only theoretical, it might work well in practice.

7.2 Conclusion on the grammars

In Chapter 5 we saw that the signature grammar and the grammar derived from the deterministic $(\mathcal{O}(\tau), \mathcal{O}(\tau \log^* n))$-partitioning set have very similar constructions. Despite this fact and the fact that some properties (like their height) are common to both grammars, the derived grammar does not prove to be locally consistent (at least not according to Definition 11). This implies that the derived grammar will not be useful for solving The String Indexing Problem, at least not using the methods presented in Chapter 6. It might be useful for solving other problems, but this would require further investigation into these problems and into the compression rate of this derived grammar.

7.3 Conclusions on locality

Having examined the two kinds of locality, index-based and block-based, we observe that each serves different purposes and are useful in different settings. Index-based locality allows us to work primarily on the compressed structure, requiring only a limited number of character lookups near the boundaries of a query. This is particularly useful when storing the entire string externally, and accessing arbitrary characters is computationally costly.

In contrast, block-based locality enables pattern matching directly on a compressed representation, even without access to the original string. Although the properties of block-based locality may appear more subtle or less intuitive than those of index-based locality, they are crucial to the proof of correctness for the signature grammar pattern matching algorithm, as illustrated in Lemma 25.

7.4 Future work

There are other interesting areas within locally consistent parsing, which are not covered by this thesis, some of them are:

- **Is there something a τ-synchronising set can do that a (τ, δ)-partitioning set cannot?** In this thesis and the literature of the two partitionings, it is still unclear whether the more strict density condition of the τ-synchronising set makes it capable of solving some problems more efficiently than a partitioning set. The example studied here, The Longest Common Extension Problem, is solved just as well by a partitioning set, and the same goes for the Sparse Suffix Tree (which is not described in this work but has been solved by both Birenzwige et al. [5] and Kempa and Kociumaka [22] using their respective partitionings). It could be interesting to try to locate some (maybe more advanced) problems that would get distinctive performance from the two partitionings.
- **An even more unifying theory.** This work has not been possible to make a theory that unites the locally consistent sets (or grammars) based on different locality measures. It could be interesting to continue working towards a more general description containing both locality forms.
- **Looking into other compression measures than the Lempel-Ziv77 parse.** This is not necessarily a new research topic, but Christiansen et al. [9] measured the performance of their grammar (which is almost identical to the signature grammar they introduced earlier [8]) using attractors. It could be very interesting to look more into those and how the compression rate of a locally consistent grammar relates to this concept.
- **Investigation of the grammar derived from the deterministic partitioning set construction.** It could also be very interesting to dive further into the properties of the grammar derived from Construction 2. Studying both the compression rate of the grammar and the problems, it could be used to provide even better grounds for a comparison between this grammar and the signature grammar of Christiansen and Ettienne [8].

7.5 Concluding thoughts

This thesis aims to map out the differences and parallels of the different types of locally consistent parsing and to emphasise the use case and upsides of using locally consistent structures.

In a world of ever-increasing data collection and a growing need to efficiently identify patterns within such data, the potential of locally consistent parsing seems substantial.

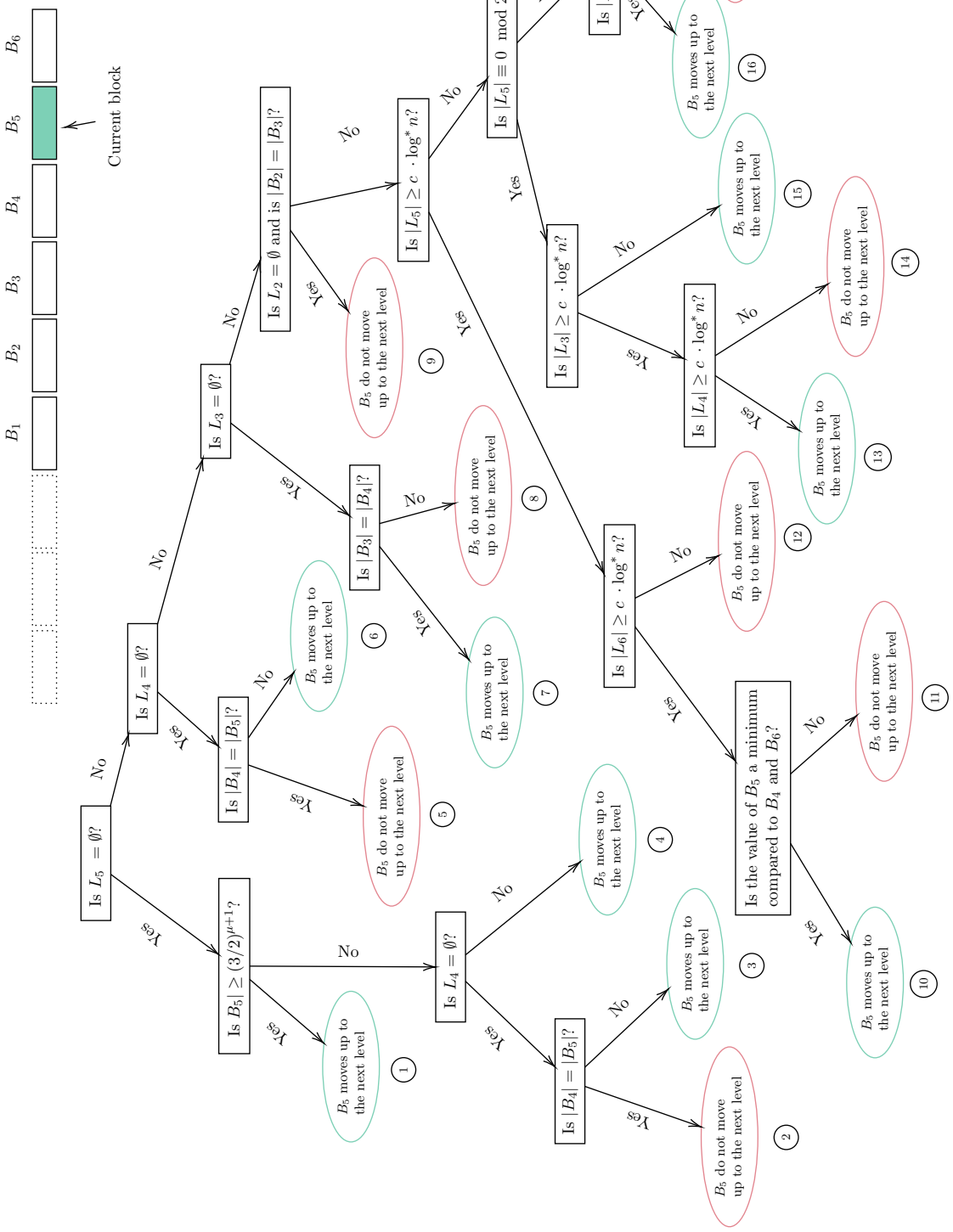
Appendix and References

A Appendix

A.1 Explaining Figure 12

1. B_5 moves up a level: B_5 is of type 1 and is therefore moved to the next level.
2. B_5 does not moves up a level: B_5 is of type 2 (part of a periodic sequence) but is not the first block.
3. B_5 moves up a level: B_5 is the first block in a type 2 sequence (B_4 is of type 1).
4. B_5 moves up a level: B_5 is the first block in a type 2 sequence.
5. B_5 does not moves up a level: B_5 is the last block of a type 2 sequence.
6. B_5 moves up a level: B_5 is the right neighbour of a type 1 block.
7. B_5 moves up a level: B_4 is the last block in a block 2 sequence thus B_5 moves up.
8. B_5 does not moves up a level: B_4 is the first block after a sequence of type 2 blocks which means B_5 does not move up.
9. B_5 does not moves up a level: B_3 is the last block in a sequence of type 2 blocks. This means B_4 is the first block after a sequence of type 2 blocks which means B_5 does not move up.
10. B_5 moves up a level: B_4, B_5, and B_6 are all blocks of type 3, and the label of B_5 is the smallest of the three. Therefore B_5 moves up.
11. B_5 does not moves up a level: B_4, B_5, and B_6 are all blocks of type 3, and the label of B_5 is not the smallest of the three. Therefore B_5 does not move up.
12. B_5 does not moves up a level: B_5 is the last block of type 3 with a complete label list. Therefore can the final label of B_5 and B_6 not be compared and B_6 is part of the $c \log^* n$ last blocks. Thus B_5 does not move up.
13. B_5 moves up a level: B_5 is the first of the $c \log^* n$ last blocks of a sequence if type 3 (which is treated like a sequence of type 4). Therefore it is moved up.
14. B_5 does not moves up a level: B_4 is the first of the $c \log^* n$ last blocks, which means B_5 should not be moved up.
15. B_5 moves up a level: B_5 is neither the first nor the second block in of of the $c \log^* n$ last blocks, $|L_5| \equiv 0 \mod 2$, and it is not the last (because $|L_5| \equiv 0 \mod 2$) thus it is moved to the next level.
16. B_5 moves up a level: B_5 is the first of the $c \log^* n$ last blocks of a sequence if type 3 (which is treated like a sequence of type 4). Therefore it is moved up.

17. B_5 does not moves up a level: B_5 is not the first of the $c \log^* n$ last blocks of a sequence if type 3 (which is treated like a sequence of type 4) and $|L_5| \not\equiv 0 \mod 2$ therefore it is not moved up.


B_1 B_2 B_3 B_4 B_5 B_6
Current block
Is $L_5 = \emptyset$?
Is $L_4 = \emptyset$?
Is $L_3 = \emptyset$?
Is $L_2 = \emptyset$ and is $|B_2| = |B_3|$?
Is $|L_5| \geq c \cdot \log^* n$?
Is $|L_5| \equiv 0 \mod 2$?
Is $|L_4| \geq c \cdot \log^* n$?
Is $|B_5| \geq (3/2)^{\mu+1}$?
Is $|B_4| = |B_5|$?
Is $|B_3| = |B_4|$?
Is $|L_3| \geq c \cdot \log^* n$?
Is $|L_4| \geq c \cdot \log^* n$?
Is $|L_6| \geq c \cdot \log^* n$?
Is $L_4 = \emptyset$?
Is $|B_4| = |B_5|$?
Is the value of B_5 a minimum compared to B_4 and B_6?
B_5 moves up to the next level
B_5 do not move up to the next level
Yes
No
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17

B References

[1] Lorraine A. K. Ayad, Grigorios Loukides, and Solon P. Pissis. Text indexing for long patterns using locally consistent anchors. *Arxiv (cornell University)*, 2024. doi: 10.48550/arXiv.2407.11819.

[2] Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. *Proceedings of the Twenty-sixth Annual Acm-siam Symposium on Discrete Algorithms*, pages 572–591, 2015. doi: 10.5555/2722129.2722168.

[3] J. L. Bently and D. F. Stanat. Analysis of range searches in quad trees. *Information Processing Letters*, 3(6):170–173, 1975. ISSN 00200190, 18726119. doi: 10.1016/0020-0190(75)90034-4.

[4] Philip Bille, Mikko Berggren Ettienne, Inge Li Gørtz, and Hjalte Wedel Vildhøj. Time–space trade-offs for lempel–ziv compressed indexing. *Theoretical Computer Science*, 713:66–77, February 2018. ISSN 0304-3975. doi: 10.1016/j.tcs.2017.12.021. URL `http://dx.doi.org/10.1016/j.tcs.2017.12.021`.

[5] Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. *Proceedings of the Annual Acm-siam Symposium on Discrete Algorithms*, 2020-:607–626, 2020.

[6] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. *Proceedings of the Annual Symposium on Computational Geometry*, pages 1–10, 2011. doi: 10.1145/1998196.1998198.

[7] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1(1-4):133–162, 1986. ISSN 01784617, 14320541. doi: 10.1007/BF01840440.

[8] Anders Roy Christiansen and Mikko Berggren Ettienne. Compressed indexing with signature grammars, 2017.

[9] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Optimal-time dictionary-compressed indexes. *Acm Transactions on Algorithms*, 17(1):3426473, 2021. ISSN 15496333, 15496325. doi: 10.1145/3426473.

[10] Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco. Grammar-compressed indexes with logarithmic search time. 2020.

[11] R Cole and U Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking universal algorithm for sequential data compression. *Information and Control*, 70(1):32–53, 1986. ISSN 18782981, 00199958. doi: 10.1016/S0019-9958(86)80023-7.

[12] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational geometry: Algorithms and applications*. Springer Berlin Heidelberg, 2008. ISBN 3540779736, 3540779744, 3642096816, 9783540779735, 9783540779742, 9783642096815. doi: 10.1007/978-3-540-77974-2.

[13] Patrick Dinklage, Johannes Fischer, Alexander Herlez, Tomasz Kociumaka, and Florian Kurpicz. Practical performance of space efficient data structures for longest common extensions. *Leibniz International Proceedings in Informatics, Lipics*, 173:39, 2020. ISSN 18688969. doi: 10.4230/LIPIcs.ESA.2020.39.

[14] Rasmus Pagh Djamal Belazzougui, Paolo Boldi and Sebastiano Vigna. Fast prefix search in little space, with applications. volume 6346 of *LNCS*, pages 427–438. Springer Berlin Heidelberg, 2010. (Appendix H.3 can be found at http://www.itu.dk/people/pagh/papers/prefix.pdf).

[15] Paolo Ferragina and Roberto Grossi. Improved dynamic text indexing. *Journal of Algorithms*, 31(2):291–319, 1999. ISSN 10902678, 01966774. doi: 10.1006/jagm.1998.0999.

[16] Travis Gagie, Paweł Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi. Lz77-based self-indexing with faster pattern matching. In Alberto Pardo and Alfredo Viola, editors, *LATIN 2014: Theoretical Informatics*, pages 731–742, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-642-54423-1.

[17] Inge Li Gørtz. Tries and suffix trees. `https://www2.compute.dtu.dk/courses/02105/tries-and-suffix-trees.pdf`, n.d. Lecture notes, Technical University of Denmark.

[18] J. Ian Munro, Yakov Nekrich, and Jeffrey S. Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. ISSN 18792294, 03043975. doi: 10.1016/j.tcs.2015.11.011.

[19] Guy Jacobson. Space-efficient static trees and graphs. *Annual Symposium on Foundations of Computer Science (proceedings)*, pages 549–554, 1989. ISSN 02725428.

[20] Artur Jez. A really simple approximation of smallest grammar. *Combinatorial Pattern Matching, Cpm 2014*, 8486:182–191, 2014. ISSN 16113349, 03029743.

[21] Richard M. Karp and Michael O. Rabin. efficient randomized pattern-matching algorithms. *Ibm Journal of Research and Development*, 31(2):249–260, 1987. ISSN 21518556, 00188646. doi: 10.1147/rd.312.0249.

[22] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time bwt construction and optimal lce data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC '19, page 756–767. ACM, June 2019. doi: 10.1145/3313276.3316368. URL `http://dx.doi.org/10.1145/3313276.3316368`.

[23] Dominik Kempa and Tomasz Kociumaka. Dynamic suffix array with polylogarithmic queries and updates. *Proceedings of the Annual Acm Symposium on Theory of Computing*, pages 1657–1670, 2022. ISSN 07378017. doi: 10.1145/3519935.3520061.

[24] Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza. Towards a definitive measure of repetitiveness. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12118:207–219, 2020. ISSN 16113349, 03029743. doi: 10.1007/978-3-030-61792-9_17.

[25] Tomasz Kociumaka, Gonzalo Navarro, and Francisco Olivares. Near-optimal search time in δ-optimal space. *Lecture Notes in Computer Science (including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13568:88–103, 2022. ISSN 16113349, 03029743. doi: 10.1007/978-3-031-20624-5_6.

[26] Gonzalo Navarro. Indexing highly repetitive string collections, part ii. *Acm Computing Surveys*, 54(2):26, 2021. ISSN 15577341, 03600300. doi: 10.1145/3432999.

[27] André Schulz. 6.851: Advanced data structures – lecture 3: Range reporting. `https://courses.csail.mit.edu/6.851/spring10/`, February 2010. Scribed by Jacob Steinhardt and Greg Brockman.

[28] R WILBER. Lower bounds for accessing binary search-trees with rotations. *Siam Journal on Computing*, 18(1):56–67, 1989. ISSN 10957111, 00975397. doi: 10.1137/0218004.

[29] J Ziv and A Lempel. Universal algorithm for sequential data compression. *Ieee Transactions on Information Theory*, 23(3):337–343, 1977. ISSN 15579654, 00189448. doi: 10.1109/TIT.1977.1055714.