



HAL
open science

Optimal Square Detection Over General Alphabets

Jonas Ellert, Pawel Gawrychowski, Garance Gourdel

► **To cite this version:**

Jonas Ellert, Pawel Gawrychowski, Garance Gourdel. Optimal Square Detection Over General Alphabets. SODA - 2023 Annual ACM-SIAM Symposium on Discrete Algorithms, Jan 2023, Florence, Italy. pp.5220-5242, 10.1137/1.9781611977554.ch189 . hal-03991054

HAL Id: hal-03991054

<https://hal.science/hal-03991054>

Submitted on 16 Feb 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimal Square Detection Over General Alphabets

Jonas Ellert*

Paweł Gawrychowski†

Garance Gourdel‡

Abstract

Squares (fragments of the form xx , for some string x) are arguably the most natural type of repetition in strings. The basic algorithmic question concerning squares is to check if a given string of length n is square-free, that is, does not contain a fragment of such form. Main and Lorentz [J. Algorithms 1984] designed an $\mathcal{O}(n \log n)$ time algorithm for this problem, and proved a matching lower bound assuming the so-called general alphabet, meaning that the algorithm is only allowed to check if two characters are equal. As an open question, they asked if there is a faster algorithm if one restricts the size of the alphabet. Crochemore [Theor. Comput. Sci. 1986] designed a linear-time algorithm for constant-size alphabets, and combined with the more recent results his approach in fact implies such an algorithm for linearly-sortable alphabets. Very recently, Ellert and Fischer [ICALP 2021] significantly relaxed this assumption by designing a linear-time algorithm for general ordered alphabets, that is, assuming a linear order on the characters. However, the open question of Main and Lorentz from 1984 remained unresolved for general (unordered) alphabets. In this paper, we show that testing square-freeness of a length- n string over general alphabet of size σ can be done with $\mathcal{O}(n \log \sigma)$ comparisons, and cannot be done with $o(n \log \sigma)$ comparisons. We complement this result with an $\mathcal{O}(n \log \sigma)$ time algorithm in the Word RAM model.

1 Introduction

The notion of repetition is a central concept in combinatorics on words and algorithms on strings. In this context, a word or a string is simply a sequence of characters from some finite alphabet Σ . In the most basic version, a repetition consists of two (or more) consecutive occurrences of the same fragment. Repetitions are interesting not only from a purely theoretical point of view, but are also very relevant in bioinformatics [52]. A repetition could be a square, defined as two consecutive occurrences of the same fragment, a higher power (for example, a cube), or a run. In this paper, we will be mostly working with squares.

The study of squares in strings goes back to the work of Thue published in 1906 [74], who considered the question of constructing an infinite word with no squares. It is easy to see that any sufficiently long binary word must contain a square, and Thue proved that there exists an infinite ternary word with no squares. His result has been rediscovered multiple times, and in 1979 Bean, Ehrenfeucht and McNulty [9] started a systematic study of the so-called avoidable repetitions, see for example the survey by Currie [27].

Combinatorics on words. The basic tool in the area of combinatorics on words is the so-called periodicity lemma. A period of a string $T[1..n]$ is an integer d such that $T[i] = T[i+d]$ for every $i \in [1, n-d]$, and the periodicity lemma states that if p and q are both such periods and $p + q \leq n + \gcd(p, q)$ then so is $\gcd(p, q)$ [32]. This was generalised in a myriad of ways, for strings [17, 49, 75], partial words (words with don't cares) [11–13, 47, 51, 70, 71], Abelian periods [14, 20], parametrized periods [6], order-preserving periods [42, 65], approximate periods [2–4]. Now, a square can be defined as a fragment of length twice its period. The string \mathbf{a}^n contains $\Omega(n^2)$ such fragments, thus from the combinatorial point of view it is natural to count only distinct squares. Fraenkel and Simpson [35] showed an upper bound of $2n$ and a lower bound of $n - \Theta(\sqrt{n})$ for the maximum number of distinct squares in a length- n string. After a sequence of improvements [28, 48, 73], the upper bound was very recently improved to n [16]. The last result was already generalised to higher powers [60]. Another way to avoid the trivial examples such as \mathbf{a}^n is to count only maximal periodic fragments, that is, fragments with period at most half of their length and that cannot be extended to the left or to the right without breaking the period. Such fragments are usually called runs. Kolpakov and Kucherov [53] showed an upper bound of $\mathcal{O}(n)$ on their number, and this started a

*Technical University of Dortmund, Germany. Partially supported by the German Research Association (DFG) within the Collaborative Research Center SFB 876, project A6.

†Institute of Computer Science, University of Wrocław, Poland. Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR).

‡DI/ENS, PSL Research University, IRISA Inria Rennes, France. Partially supported by the grant ANR-20-CE48-0001 from the French National Research Agency (ANR).

long line of work on determining the exact constant [24, 25, 40, 41, 68, 69], culminating in the paper of Bannai et al. [8] showing an upper bound of n , and followed by even better upper bounds for binary strings [33, 45]. This was complemented by a sequence of lower bounds [36, 63, 64, 72].

Algorithms on strings. In this paper, we are interested in the algorithmic aspects of detecting repetitions in strings. The most basic question in this direction is checking if a given length- n string contains at least one square. Testing square-freeness was first considered by Main and Lorentz [62], who designed an $\mathcal{O}(n \log n)$ time algorithm based on a divide-and-conquer approach and a linear-time procedure for finding all new squares obtained when concatenating two strings. In fact, their algorithm can be used to find (a compact representation of) all squares in a given string within the same time complexity. They also proved that any algorithm based on comparisons of characters needs $\Omega(n \log n)$ such operations to test square-freeness in the worst case. Here, comparisons of characters means checking if characters at two positions of the input string are equal. However, to obtain the lower bound they had to consider instances consisting of even up to n distinct characters, that is, over alphabet of size n . This is somewhat unsatisfactory, and motivates the following open question that was explicitly asked by Main and Lorentz [62]:

QUESTION 1.1. *Is there a faster algorithm to determine if a string is square-free if we restrict the size of the alphabet?*

Another $\mathcal{O}(n \log n)$ time algorithm for finding all repetitions was given by Crochemore [22], who also showed that for constant-size alphabets testing square-freeness can be done in $\mathcal{O}(n)$ time [23]. In fact, the latter algorithm works in $\mathcal{O}(n \log \sigma)$ time for alphabets of size σ with a linear order on the characters. That is, it needs to test if the character at some position is smaller than the character at another position. In the remaining part of the paper, we will refer to this model as general ordered alphabet, while the model in which we can only test equality of characters will be called general (unordered) alphabet. Later, Kosaraju [54] showed that in fact, assuming constant-size alphabet, $\mathcal{O}(n)$ time is enough to find the shortest square starting at each position of the input string. Apostolico and Preparata [7] provide another $\mathcal{O}(n \log n)$ time algorithm assuming a general ordered alphabet, based more on data structure considerations than combinatorial properties of words. Finally, a number of alternative $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log \sigma)$ time algorithms (respectively, for general unordered and general ordered alphabets) can be obtained from the work on online [46, 55, 57] and parallel [5] square detection (interestingly, this cannot be done efficiently in the related streaming model [66, 67]).

Faster algorithms for testing square-freeness of strings over general ordered alphabets were obtained as a byproduct of the more general results on finding all runs. Kolpakov and Kucherov [53] not only proved that any length- n string contains only $\mathcal{O}(n)$ runs, but also showed how to find them in the same time assuming linearly-sortable alphabet. Every square is contained in a run, and every run contains at least one square, thus this in particular implies a linear-time algorithm for testing square-freeness over such alphabets. For general ordered alphabets, Kosolobov [56] showed that the decision tree complexity of this problem is only $\mathcal{O}(n)$, and later complemented this with an efficient $\mathcal{O}(n(\log n)^{2/3})$ time algorithm [58] (still using only $\mathcal{O}(n)$ comparisons). The time complexity was then improved to $\mathcal{O}(n \log \log n)$ by providing a general mechanism for answering longest common extension (LCE) queries for general ordered alphabets [39], and next to $\mathcal{O}(n\alpha(n))$ by observing that the LCE queries have additional structure [26]. Finally, Ellert and Fischer provided an elegant $\mathcal{O}(n)$ time algorithm, thus fully resolving the complexity of square detection for general ordered alphabets. However, for general (unordered) alphabets the question of Main and Lorentz remains unresolved, with the best upper bound being $\mathcal{O}(n \log n)$, and only known to be asymptotically tight for alphabets of size $\Theta(n)$.

General alphabets. While in many applications one can without losing generality assume some ordering on the characters of the alphabet, no such ordering is necessary for defining what is a square. Thus, it is natural from the mathematical point of view to seek algorithms that do not require such an ordering to efficiently test square-freeness. Similar considerations have led to multiple beautiful results concerning the pattern matching problem, such as constant-space algorithms [15, 38], or the works on the exact number of required equality comparisons [18, 19]. More recent examples include the work of Duval, Lecroq, and Lefebvre [29] on computing the unbordered conjugate/rotation, and Kosolobov [59] on finding the leftmost critical point.

Main results. We consider the complexity of checking if a given string $T[1..n]$ containing σ distinct characters is square-free. The input string can be only accessed by issuing comparisons $T[i] \stackrel{?}{=} T[j]$, and the value of σ is not assumed to be known. We start by analysing the decision tree complexity of the problem. That is, we only

consider the required and necessary number of comparisons, without worrying about an efficient implementation. We show that, even if the value of σ is assumed to be known, $\Omega(n \log \sigma)$ comparisons are required.

THEOREM 1.1. *For any integers n and σ with $8 \leq \sigma \leq n$, there is no deterministic algorithm that performs at most $n \ln \sigma - 3.6n = \mathcal{O}(n \ln \sigma)$ comparisons in the worst case, and determines whether a length- n string with at most σ distinct symbols is square-free.*

Next, we show that in fact $\mathcal{O}(n \log \sigma)$ comparisons are sufficient. We stress that the value of σ is not assumed to be known. In fact, as a warm-up for the above theorem, we first prove that finding a sublinear multiplicative approximation of this value requires $\frac{n\sigma}{8}$ comparisons. This does not contradict the claimed upper bound, as we are only saying that the number of comparisons used on a particular input string is at most $\mathcal{O}(n \log \sigma)$, but might actually be smaller. Thus, it is not possible to extract any meaningful approximation of the value of σ from the number of used comparisons.

THEOREM 1.2. *Testing square-freeness of a length- n string that contains σ distinct symbols from a general unordered alphabet can be done with $\mathcal{O}(n \log \sigma)$ comparisons.*

The proof of the above result is not efficient in the sense that it only restricts the overall number of comparisons, and not the time to actually figure out which comparisons should be used. A direct implementation results in a quadratic time algorithm. We first show how to improve this to $\mathcal{O}(n \log \sigma + n \log^* n)$ (while still keeping the asymptotically optimal $\mathcal{O}(n \log \sigma)$ number of comparisons), and finally to $\mathcal{O}(n \log \sigma)$. In this part of the paper, we assume the Word RAM model with word of length $\Omega(\log n)$. We stress that the input string is still assumed to consist of characters that can be only tested for equality, that is, one should think that we are given oracle access to a functions that, given i and j , checks whether $T[i] = T[j]$.

THEOREM 1.3. *Testing square-freeness of a length- n string that contains σ distinct symbols from a general unordered alphabet can be implemented in $\mathcal{O}(n \log \sigma)$ comparisons and time.*

Altogether, our results fully resolve the open question of Main and Lorentz for the case of general unordered alphabets and deterministic algorithms. We leave extending our lowerbound to randomised algorithms as an open questions.

Overview of the methods. As mentioned before, Main and Lorentz [62] designed an $\mathcal{O}(n \log n)$ time algorithm for testing square-freeness of length- n strings over general alphabets. The high-level idea of their algorithm goes as follows. They first designed a procedure for checking, given two strings x and y , if their concatenation contains a square that is not fully contained in x nor y in $\mathcal{O}(|x| + |y|)$ time. Then, a divide-and-conquer approach can be used to detect a square in the whole input string in $\mathcal{O}(n \log n)$ total time. For general alphabets of unbounded size this cannot be improved, but Crochemore [23] showed that, for general ordered alphabets of size σ , a faster $\mathcal{O}(n \log \sigma)$ time algorithm exists. The gist of this approach is to first obtain the so-called f -factorisation of the input string (related to the well-known Lempel-Ziv factorisation), that in a certain sense “discovers” repetitive fragments. Then, this factorisation can be used to apply the procedure of Main and Lorentz on appropriately selected fragments of the input strings in such a way that the leftmost occurrence of every distinct square is detected, and the total length of the strings on which we apply the procedure is only $\mathcal{O}(n)$. The factorisation can be found in $\mathcal{O}(n \log \sigma)$ time for general ordered alphabets of size σ by, roughly speaking, constructing some kind of suffix structure (suffix array, suffix tree or suffix automaton). For general (unordered) alphabets, computing the f -factorisation (or anything similar) seems problematic, and in fact we show (as a corollary of our lower bound on approximating the alphabet size) that computing the f -factorisation or Lempel-Ziv-factorisation (LZ-factorisation) of a given length- n string containing σ distinct characters requires $\frac{n\sigma}{24}$ equality tests. Thus, we need another approach. Additionally, the $\mathcal{O}(n)$ time algorithm of Ellert and Fischer [30] hinges on the notion of Lyndon words, which is simply not defined for strings over general alphabets. Thus, at first glance it might seem that $\Theta(n\sigma)$ is the right time complexity for testing square-freeness over length- n strings over general alphabets of size σ . At the same time, due to the $\Omega(n \log n)$ lower bound of Main and Lorentz for testing square-freeness of length- n string consisting of up to n distinct characters, one might hope for an $\mathcal{O}(n \log \sigma)$ time algorithm when there are only σ distinct characters.

We begin our paper with a lower bound of $\Theta(n \log \sigma)$ for such strings. Intuitively, we show that testing square-freeness has the direct sum property: $\frac{n}{\sigma}$ instances over length- σ strings can be combined into a single

instance over length- n string. As in the proof of Main and Lorentz, we use the adversarial method. While the underlying calculation is essentially the same, we need to appropriately combine the smaller instances, which is done using the infinite square-free Prouhet-Thue-Morse sequence, and use significantly more complex rules for resolving the subsequent equality tests. As a warm-up for the adversarial method, we prove that computing any meaningful approximation of the number of distinct characters requires $\Omega(n\sigma)$ such tests, and that this implies the same lower bound on computing the f -factorisation and the Lempel-Ziv factorisation.

We then move to designing an approach that uses $\mathcal{O}(n \log \sigma)$ equality comparisons to test square-freeness. First, we introduce the notion of Δ -approximate LZ factorisation. Intuitively, its goal is to “capture” all sufficiently long squares, while the original LZ factorisation (or f -factorisation) captures all squares. Each phrase in a Δ -approximate LZ factorisation consists of a head of length at most Δ and a tail (possibly empty) that must occur at least once before, such that the whole phrase is at least as long as the classical LZ phrase starting at the same position. Contrary to the classical LZ factorisation, this factorisation is not unique. The advantage of our modification is that there are fewer phrases (and there is more flexibility as to what they should be), and hence one can hope to compute such factorisation more efficiently. Indeed, we provide such an efficient construction method using the so-called Δ -difference covers and sparse suffix trees. This method is then applied in $\log \log n$ phases, corresponding to smaller and smaller estimates on the size of the alphabet. Constructing the sparse suffix tree and then finding a Δ -approximate LZ factorisation uses sufficiently few equality tests, unless we notice that there are many distinct characters in the alphabet. In such case, by carefully adjusting the parameters, we can afford switching to the approach of Main and Lorentz on sufficiently short fragments of the input string. After computing a Δ -approximate LZ factorisation, we apply the procedure of Main and Lorentz on fragments around a sufficiently long tail of some phrase. The overall number of equality tests used in all such invocations is amortised over all phases. More precisely, during a given phase, we avoid looking for squares in a fragment fully contained in a tail from an earlier phase.

The above approach uses an asymptotically optimal number of equality tests in the worst case, but does not result in an efficient algorithm. The main bottleneck is constructing the sparse suffix trees. However, it is not hard to provide an efficient implementation using the general mechanism for answering LCE queries for strings over general alphabets [39]. Unfortunately, the best known approach for answering such queries incurs an additional $\mathcal{O}(n \log^* n)$ in the time complexity, even if the size of the alphabet is constant. We overcome this technical hurdle by applying the recent notion of string synchronizing sets introduced by Kempa and Kociumaka [50]. We note that constructing such sets efficiently requires assuming linearly-sortable alphabets, therefore we can only apply this mechanism on sufficiently short fragments of the input strings, which leads to some additional technical difficulties.

It is relatively straightforward to extend our intermediate $\mathcal{O}(n \log^* n)$ time algorithm to reporting all runs in the same time complexity. Adapting the final $\mathcal{O}(n)$ time algorithm to this end is left as an open question.

2 Preliminaries

Strings. A string of length n is a sequence $T[1] \dots T[n]$ of characters from a finite alphabet Σ of size σ . The substring $T[i..j]$ is the string $T[i] \dots T[j]$, whereas the fragment $T[i..j]$ refers to the specific occurrence of $T[i..j]$ starting at position i in T . If $i > j$, then $T[i..j]$ is the empty string. $T[i..j]$ is a shortcut for $T[i..j - 1]$. Similarly, we write $[i, j] = [i, j + 1)$ as a shortcut for the integer interval $\{i, \dots, j\}$. Given two positions $i \leq j$, their longest common extension (LCE) is the length of the longest common prefix between suffixes $T[i..n]$ and $T[j..n]$, formally defined as $\text{LCE}(i, j) = \text{LCE}(j, i) = \max\{\ell \mid \ell \in \{0, \dots, n - j + 1\} \wedge T[i..i + \ell] = T[j..j + \ell]\}$.

DEFINITION 2.1. *A positive integer p is a period of a string $T[1..n]$ if $T[i] = T[i + p]$ for every $i \in \{1, \dots, n - p\}$. The smallest such p is called the period of $T[1..n]$, and we call a string periodic if its period p is at most $\frac{n}{2}$.*

Computational model. For a general unordered alphabet Σ , the only allowed operation on the characters is comparing for equality. In particular, there is no linear order on the alphabet. Unless explicitly stated otherwise, we will only use such comparisons. A general ordered alphabet has a total order, such that comparisons of the type less-equals are possible.

In the algorithmic part of the paper, we assume the standard unit-cost Word RAM model with words of length $\Omega(\log n)$, but the algorithm is only allowed to access the input string $T[1..n]$ by comparisons $T[i] \stackrel{?}{=} T[j]$. We say that a string of length n is over a linearly-sortable alphabet, if we can sort the n symbols of the string in $\mathcal{O}(n)$ time. Note that whether or not an alphabet is linearly-sortable depends not only on the alphabet, but also

on the string. For example, the alphabet $\Sigma = \{1, \dots, m^{\mathcal{O}(1)}\}$ is linearly-sortable for strings of length $n = \Omega(m)$ (e.g., using radix sort), but it is unknown whether it is linearly-sortable for all strings of length $n = o(m)$ [44]. Our algorithm will internally use strings over linearly-sortable alphabets. We stress that in such strings the characters are not the characters from the input string, but simply integers calculated by the algorithm. Note that every linearly-sortable alphabet is also a general ordered alphabet.

Squares. A square is a length- 2ℓ fragment of period ℓ . The following theorem is a classical result by Main and Lorentz [62].

THEOREM 2.1. *Testing square-freeness of $T[1..n]$ over a general alphabet can be implemented in $\mathcal{O}(n \log n)$ time and comparisons.*

The proof of the above theorem is based on running a divide-and-conquer procedure using the following lemma.

LEMMA 2.1. *Given two strings x and y over a general alphabet, we can test if there is a square in xy that is not fully contained in x nor y in $\mathcal{O}(|x| + |y|)$ time and comparisons.*

Assuming a general ordered alphabet, it is possible to test square-freeness in linear time. This follows from the approach by Ellert and Fischer [30], who showed that all runs (maximal repetitions) of a string over a general ordered alphabet can be computed in $\mathcal{O}(n)$ time. As mentioned earlier, each run contains a square, and each square is contained in a run. Thus, the string contains a square if and only if it contains a run.

THEOREM 2.2. *Testing square-freeness of $T[1..n]$ over a general ordered alphabet can be implemented in $\mathcal{O}(n)$ time.*

Lempel-Ziv factorisation. The unique LZ phrase starting at position s is a fragment $T[s..e]$ such that $T[s..(e-1)]$ occurs at least twice in $T[1..(e-1)]$ and $T[s..e]$ occurs only once in $T[1..e]$. The Lempel-Ziv factorisation of $T[1..n]$ consists of z phrases f_1, \dots, f_z such that the concatenation $f_1 \dots f_z$ is equal to $T[1..n]$ and each f_i is the unique LZ phrase starting at position $1 + \sum_{j=1}^{i-1} |f_j|$.

Tries. Given a collection $\mathcal{S} = \{T_1, \dots, T_k\}$ of strings over some alphabet Σ , its trie is a rooted tree with edge labels from Σ . For any node v , the concatenation of the edge labels from the root to the node *spells* a string. The string-depth of a node is the length of the string that it spells. No two nodes spell the same string, i.e., for any node, the labels of the edges to its children are pairwise distinct. Each leaf spells one of the T_i , and each T_i is spelled by either an internal node or a leaf.

The compacted trie of \mathcal{S} can be obtained from its (non-compacted) trie by contracting each path between a leaf or a branching node and its closest branching ancestor into a single edge (i.e., by contraction we eliminate all non-branching internal nodes). The label of the new edge is the concatenation of the edge labels of the contracted path in root to leaf direction. Since there are at most k leaves and all internal nodes are branching, there are $\mathcal{O}(k)$ nodes in the compacted trie. Each edge label is some substring $T_i[s..e]$ of the string collection, and we can avoid explicitly storing the label by instead storing the reference $\langle i, s, e \rangle$. Thus $\mathcal{O}(k)$ words are sufficient for storing the compacted trie. Consider a string T' that is spelled by a node of the non-compacted trie. We say that T' is explicit, if and only if it is spelled by a node of the compacted trie. Otherwise T' is implicit.

The suffix tree of a string $T[1..n]$ is the compacted trie containing exactly its suffixes $\{T[i..n] \mid i \in \{1, \dots, n\}\}$. It is one of the most fundamental data structures in string algorithmics, and is widely used, e.g., for compression and indexing [43]. The suffix tree can be stored in $\mathcal{O}(n)$ words of memory, and for linearly-sortable alphabets it can be computed in $\mathcal{O}(n)$ time [31]. The sparse suffix tree of T for some set $B \subseteq \{1, \dots, n\}$ of sample positions is the compacted trie containing exactly the suffixes $\{T[i..n] \mid i \in B\}$. It can be stored in $\mathcal{O}(|B|)$ words of memory.

We assume that T is terminated by some special symbol $T[n] = \$$ that occurs nowhere else in T . This ensures that each suffix is spelled by a leaf, and we label the leaves with the respective starting positions of the suffixes. Note that for any two leaves $i \neq j$, their lowest common ancestor (i.e., the deepest node that is an ancestor of both i and j) spells a string of length $\text{LCE}(i, j)$.

3 Lower Bounds

In this section, we show lower bounds on the number of symbol comparisons required to compute a meaningful approximation of the alphabet size (Section 3.1) and to test square-freeness (Section 3.2). For both bounds we use an adversarial method, which we briefly outline now.

The present model of computation may be interpreted as follows. An algorithm working on a string over a general unordered alphabet has no access to the actual string. Instead, it can only ask an oracle whether or not there are identical symbols at two positions. The number of questions asked is exactly the number of performed comparisons. In order to show a lower bound on the number of comparisons required to solve some problem, we describe an adversary that takes over the role of the oracle, forcing the algorithm to perform as many symbol comparisons as possible.

We use a conflict graph $G = (V, E)$ with $V = \{1, \dots, n\}$ and $E \subseteq V^2$ to keep track of the answers given by the adversary. The nodes directly correspond to the positions of the string. Initially, it holds $E = \emptyset$ and all nodes are colorless, which formally means that they have color $\gamma(i) = \perp$. During the algorithm execution, the adversary may assign colors from the set $\Sigma = \{0, \dots, n-1\}$ to the nodes, which can be seen as permanently fixing the alphabet symbol at the corresponding position (i.e., each node gets colored at most once). The rule used for coloring nodes depends on the lower bound that we want to show (we describe this in detail in the respective sections). Apart from this coloring rule, the general behaviour of the adversary is as follows. Whenever the algorithm asks whether $T[i] = T[j]$ holds, the adversary answers “yes” if and only if $\gamma(i) = \gamma(j) \neq \perp$. Otherwise, it answers “no” and inserts an edge (i, j) into E . Whenever the adversary assigns the color of a node, it has to choose a color that is not used by any of the adjacent nodes in the conflict graph. This ensures that the coloring does not contradict the answers given in the past.

Let us define a set $\mathcal{T} \subseteq \Sigma^n$ of strings that is consistent with the answers given by the adversary. A string $T \in \Sigma^n$ is a member of \mathcal{T} if

$$\forall i \in V : \gamma(i) \in \{\perp, T[i]\} \quad \wedge \quad \forall i, j \in V : (T[i] = T[j]) \implies (i, j) \notin E.$$

Note that \mathcal{T} changes over time. Initially (before the algorithm starts), it holds $\mathcal{T} = \Sigma^n$. With every question asked, the algorithm might eliminate some strings from \mathcal{T} . However, there is always at least one string in \mathcal{T} , which can be obtained by coloring each colorless node in a previously entirely unused color.

3.1 Approximating the Alphabet Size Given a string $T[1..n]$ of unknown alphabet size $\sigma \geq 2$, assume that we want to compute an approximation of σ . We show that if an algorithm takes at most $\frac{n\sigma}{8}$ comparisons in the worst-case, then it cannot distinguish strings with at most σ distinct symbols from strings with at least $\frac{n}{2}$ distinct symbols. Thus, any meaningful approximation of σ requires $\Omega(n\sigma)$ comparisons.

For the sake of the proof, consider an algorithm that performs at most $\frac{n\sigma}{8}$ comparisons when given a length- n string with at most $\sigma \geq 2$ distinct symbols. We use an adversary as described at the beginning of Section 3, and ensure that the set \mathcal{T} of strings consistent with the adversary’s answers always contains a string with at most σ distinct symbols. Thus, the algorithm terminates after at most $\frac{n\sigma}{8}$ comparisons. At the same time, we ensure that \mathcal{T} also contains a string with at least $\frac{n}{2}$ distinct symbols, which yields the desired result. The adversary is equipped with the following coloring rule. All colors are from $\{1, \dots, \sigma\}$. Whenever the degree of a node in the conflict graph becomes $\sigma - 1$, we assign its color. We avoid the colors of the $\sigma - 1$ adjacent nodes in the conflict graph. At any moment in time, we could hypothetically complete the coloring by assigning one of the colors $\{1, \dots, \sigma\}$ to each colorless node, avoiding the colors of adjacent nodes. This way, each node gets assigned one of the σ colors, which means that \mathcal{T} contains a string with at most σ distinct symbols. It follows that the algorithm terminates after at most $\frac{n\sigma}{8}$ comparisons. Each comparison may increase the degree of two nodes by one. Thus, after $\frac{n\sigma}{8}$ comparisons, there are at most $\frac{n\sigma}{8} \cdot \frac{2}{\sigma-1} \leq \frac{n}{2}$ nodes with degree at least $\sigma - 1$. Therefore, at least $\frac{n}{2}$ nodes are colorless. We could hypothetically color them in $\frac{n}{2}$ distinct colors, which means that \mathcal{T} contains a string with at least $\frac{n}{2}$ distinct symbols. This leads to the following result.

THEOREM 3.1. *For any integers n and σ with $2 \leq \sigma < \frac{n}{2}$, there is no deterministic algorithm that performs at most $\frac{n\sigma}{8}$ equality-comparisons in the worst case, and is able to distinguish length- n strings with at most σ distinct symbols from length- n strings with at least $\frac{n}{2}$ distinct symbols.*

The theorem implies lower bounds on the number of comparisons needed to compute the LZ factorisation (as defined in Section 2) and the f -factorisation. In the unique f -factorisation $T = f_1 f_2 \dots f_z$, each factor f_i is either a single symbol that does not occur in $f_1 \dots f_{i-1}$, or it is the fragment of maximal length such that f_i occurs twice in $f_1 \dots f_i$.

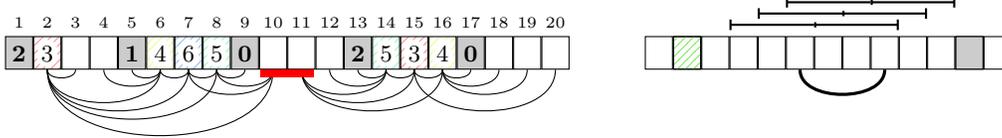


Figure 1: Example conflict graph of the adversary described in Section 3.2. The alphabet $\{0, \dots, 15\}$ is of size $\sigma = 16$. The blocks are of length $\frac{\sigma}{4} = 4$. The gray nodes are exactly the starting positions of the blocks and contain the symbols of the ternary Thue-Morse sequence $v = 2, 1, 0, 2, 0, 1, 2, \dots$, which is square-free. We assume that the colored nodes were colored in the following order: 2, 6, 8, 7, 15, 16, 14. At the time of coloring node 8, we had to avoid colors 0, 1, 2 (because they are reserved for the separator positions), 3 (because the adjacent node 2 already has color 2), and 4 (because node 6 is in the same block and already has color 4). The algorithm has not eliminated all squares yet. For example, nodes 10 and 11 with $(10, 11) \notin E$ are adjacent to nodes of colors $\{3, 6, 5\} \cup \{5, 3, 4\}$. Thus, any of the colors $\{0, 1, 2\} \cup \{7, \dots, 15\}$ can be assigned to both nodes, enforcing the square $T[10..11]$. As visualized on the right, an edge of length ℓ eliminates at most ℓ squares.

COROLLARY 3.1. *For any integers n and σ with $2 \leq \sigma < \frac{n}{4}$, there is no deterministic algorithm that performs at most $\frac{(n-1)\sigma}{16}$ equality-comparisons in the worst case, and computes the f -factorisation of a length- n string with at most σ distinct symbols.*

Proof. For some string $T = T[1]T[2] \dots T[\frac{n}{2}]$ with σ distinct symbols, let $T' = T[1]T[1]T[2]T[2] \dots T[\frac{n}{2}]T[\frac{n}{2}]$ be the length- n string with σ distinct symbols constructed by doubling each character of T . The alphabet size of T' is exactly the number of length-one phrases in the f -factorisation of T' starting at odd positions in T' . Thus, by Theorem 3.1, we need $\frac{n\sigma}{16} = \frac{|T|\sigma}{8}$ comparisons to find the f -factorisation of T' . We assumed that n is even, and account for odd n by adjusting the bound to $\frac{(n-1)\sigma}{16}$. \square

COROLLARY 3.2. *For any integers n and σ with $3 \leq \sigma < \frac{n}{6} + 1$, there is no deterministic algorithm that performs at most $\frac{(n-2)(\sigma-1)}{24}$ equality-comparisons in the worst case, and computes the Lempel-Ziv factorisation of a length- n string with at most σ distinct symbols.*

Proof. For some string $T = T[1]T[2] \dots T[\frac{n}{3}]$ with $\sigma - 1$ distinct symbols, let T' be the length- n string with σ distinct symbols constructed by doubling every character of T with a separator in between, i.e., $T' = T[1]T[1]\#T[2]T[2]\# \dots \#T[\frac{n}{3}]T[\frac{n}{3}]\#$. The first occurrence of character x in T corresponds to the first occurrence of $xx\#$ in T' , thus the preceding phrase (possibly of length one) ends at the first x in the first occurrence of $xx\#$, and the subsequent phrase must be $x\#$. Then, for the later occurrences of $xx\#$ we cannot have that $x\#$ is a phrase. Consequently, the alphabet size of T is exactly the number of length-two phrases in the Lempel-Ziv factorisation of T' starting at positions $i \equiv 2 \pmod{3}$ in T' . Thus, by Theorem 3.1, we need $\frac{n(\sigma-1)}{24} = \frac{|T|(\sigma-1)}{8}$ comparisons to find the LZ factorisation of T' . We assumed that n is divisible by 3, and account for this by adjusting the bound to $\frac{(n-2)(\sigma-1)}{24}$. \square

3.2 Testing Square-Freeness In this section, we prove that testing square-freeness requires at least $n \ln \sigma - 3n$ comparisons (even if σ is known). The proof combines the idea behind the original $\Omega(n \lg n)$ lower bound by Main and Lorentz [62] with the adversary described at the beginning of Section 3. This time, we ensure that \mathcal{T} always contains a square-free string with at most σ distinct symbols. At the same time, we try to ensure that \mathcal{T} also contains a string with at least one square. We will show that we can maintain this state until at least $n \ln \sigma - 3n$ comparisons have been performed.

The string (or rather family of strings) constructed by the adversary is organized in $\lceil \frac{4n}{\sigma} \rceil$ non-overlapping blocks of length $\frac{\sigma}{4}$ (we assume $\frac{\sigma}{4} \in \mathbb{N}$ and $8 \leq \sigma \leq n$). Each block begins with a special separator symbol. More precisely, the first symbol of the k -th block is the k -th symbol of a ternary square-free word over the alphabet $\{0, 1, 2\}$ (e.g., the distance between the k^{th} and $(k+1)^{\text{th}}$ occurrence of 0 in the Prouhet-Thue-Morse sequence, also known as the ternary Thue-Morse-Sequence, see [1, Corollary 1]). Initially, the adversary colors the nodes that correspond to the separator positions in their respective colors from $\{0, 1, 2\}$. All remaining nodes will later get colors other than $\{0, 1, 2\}$. Any fragment crossing a block boundary can be projected on the colors $\{0, 1, 2\}$,

and by construction the string cannot contain a square. Thus, the separator symbols ensure that there is no square crossed by a block boundary, which implies that the string is square-free if and only if each of its blocks is square-free.

During the algorithm execution, we use the following coloring rule. The available colors are $\{3, \dots, \sigma - 1\}$. Whenever the degree of a node becomes $\frac{\sigma}{4}$, we assign its color. We avoid not only the at most $\frac{\sigma}{4}$ colors of already colored neighbors in the conflict graph, but also the less than $\frac{\sigma}{4}$ colors of nodes within the same block (due to $\sigma \geq 8$, there are at least $\sigma - 3 - \frac{\sigma}{2} \geq 1$ colors available). An example of the conflict graph is provided in Fig. 1. At any moment in time, we could hypothetically complete the coloring by assigning one of the colors $\{3, \dots, \sigma - 1\}$ to each colorless node, avoiding colors of adjacent nodes and colors of nodes in the same block. Afterwards, each node holds one of the σ colors, but no two nodes within the same block have the same color. Thus, each block is square-free, and therefore \mathcal{T} always contains a square-free string with at most σ distinct symbols.

Now we consider the state of the conflict graph *after the algorithm has terminated*. We are particularly concerned with consecutive ranges of colorless nodes. The following lemma states that for each such range, the algorithm either performed many comparisons, or we can enforce a square within the range.

LEMMA 3.1. *Let $R = \{i, \dots, j\} \subset V$ be a consecutive range of $m = j - i + 1$ colorless nodes in the conflict graph. Then either $|E \cap R^2| \geq \sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m-2\ell+1}{\ell}$, or there is a string $T \in \mathcal{T}$ with at most σ distinct symbols such that $T[i..j]$ contains a square.*

Proof. We say that an integer interval $[x, x + 2\ell - 1]$ with $i \leq x < (x + 2\ell - 1) \leq j$ has been *eliminated*, if for some y with $x \leq y < x + \ell$ there is an edge $(y, y + \ell)$ in the conflict graph. If such an edge exists, then (by the definition of \mathcal{T}) all strings $T \in \mathcal{T}$ satisfy $T[y] \neq T[y + \ell]$. Thus $T[x..x + 2\ell - 1]$ is not a square for any of them.

Now we show that if $[x, x + 2\ell - 1]$ has not been eliminated, then there exists a string $T \in \mathcal{T}$ such that $T[x..x + 2\ell - 1]$ is a square. For this purpose, consider any position y with $x \leq y < x + \ell$, i.e., a position in the first half of the potential square. Since $[x, x + 2\ell - 1]$ has not been eliminated, $(y, y + \ell)$ is not an edge in the conflict graph. It follows that we could assign the same color to y and $y + \ell$. We only have to avoid the at most $2 \cdot (\frac{\sigma}{4} - 1)$ colors of adjacent nodes of both y and $y + \ell$ in the conflict graph. Thus there are $\frac{\sigma}{2} + 2$ appropriate colors that can be assigned to both nodes. Unlike during the algorithm execution, we do not need to avoid the special separator colors or the colors in the same block; since we are trying to enforce a square, we do not have to worry about accidentally creating one. By applying this coloring scheme for all possible choices of y , we enforce that all strings $T \in \mathcal{T}$ have a square $T[x..x + 2\ell - 1]$. Note that by coloring additional nodes after the algorithm terminated, we only remove elements from \mathcal{T} . Thus, the strings with square $T[x..x + 2\ell - 1]$ were already in \mathcal{T} when the algorithm terminated. It follows that, if the algorithm actually guarantees square-freeness, then it must have eliminated all possible intervals $[x, x + 2\ell - 1]$ with $i \leq x < (x + 2\ell - 1) \leq j$.

While each interval needs at least one edge to be eliminated, a single edge eliminates multiple intervals. However, all the intervals eliminated by an edge must be of the same length. Now we give a lower bound on the number of edges needed to eliminate all intervals of length 2ℓ . Any edge $(y, y + \ell)$ eliminates ℓ intervals, namely the intervals $[x, x + 2\ell - 1]$ that satisfy $x \leq y < x + \ell$. Within R , we have to eliminate $m - 2\ell + 1$ intervals of length 2ℓ , namely the intervals $[x, x + 2\ell - 1]$ that satisfy $i \leq x \leq j - 2\ell + 1$ (see right side of Fig. 1). Thus we need at least $\frac{m-2\ell+1}{\ell}$ edges to eliminate all squares of length 2ℓ . Finally, by summing over all possible values of ℓ , we need at least $\sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m-2\ell+1}{\ell}$ edges to eliminate all intervals in R . Note that the edges used for elimination have both endpoints in R , and are thus contained in $E \cap R^2$. Consequently, if $|E \cap R^2| < \sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m-2\ell+1}{\ell}$, then not all intervals have been eliminated, and there is a string in \mathcal{T} that contains a square. \square

Finally, we show that the algorithm either performed at least $\Omega(n \lg \sigma)$ comparisons, or there is a string $T \in \mathcal{T}$ that contains a square. Let c_1, c_2, \dots, c_k be exactly the colored nodes. Initially (before the algorithm execution), the adversary colored $\lceil \frac{4n}{\sigma} \rceil$ nodes. Thus $k \geq \lceil \frac{4n}{\sigma} \rceil$, and there are $k - \lceil \frac{4n}{\sigma} \rceil$ nodes that have been colored after their degree reached $\frac{\sigma}{4}$. Therefore, the sum of degrees of all colored nodes is at least $(k - \lceil \frac{4n}{\sigma} \rceil) \cdot \frac{\sigma}{4} \geq \frac{\sigma k - 4n - \sigma}{4} \geq \frac{\sigma k - 5n}{4}$. Each comparison may increase the degree of two nodes by one. Thus, the colored nodes account for at least $\frac{\sigma k - 5n}{8}$ comparisons. There are k non-overlapping maximal colorless ranges of nodes, namely $\{c_i + 1, \dots, c_{i+1} - 1\}$ for $1 \leq i \leq k$ with auxiliary value $c_{k+1} = n + 1$. According to Lemma 3.1, each respective range accounts for $e_i = \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} \frac{m_i - 2\ell + 1}{\ell}$ edges, where $m_i = c_{i+1} - c_i - 1$. (No edge gets counted more than once because the ranges are non-overlapping, and both endpoints of the respective edges are within the range.) Thus, in order to verify

square-freeness, the algorithm must have performed at least $\sum_{i=1}^k e_i + \frac{\sigma k - 5n}{8}$ comparisons. The remainder of the proof consists of simple algebra. First, we provide a convenient lower bound for e_i (explained below):

$$\begin{aligned}
e_i &= \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} \frac{m_i - 2\ell + 1}{\ell} = \sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{m_i - 2\ell + 1}{\ell} \geq (m_i + 1) \left(\left(\sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{1}{\ell} \right) - 1 \right) \\
&> (m_i + 1) \cdot \left(\ln \frac{m_i}{2} - \frac{1}{2} \right) \\
&= (m_i + 1) \cdot \ln \frac{m_i}{2\sqrt{e}} \\
&\geq (m_i + 1) \cdot \ln \frac{m_i + 1}{2.5\sqrt{e}}
\end{aligned}$$

We can replace $\lfloor m_i/2 \rfloor$ with $\lceil m_i/2 \rceil$ because if m_i is odd the additional summand equals zero. The first inequality uses simple arithmetic operations. The second inequality uses the classical lower bound $(\ln x + \frac{1}{2}) < H_x$ of harmonic numbers. The last inequality holds for $m_i \geq 4$. For $m_i < 4$ the result becomes negative and is thus still a correct lower bound for the number of comparisons. We obtain:

$$\begin{aligned}
\underbrace{\sum_{i=1}^k (m_i + 1) \cdot \ln \frac{m_i + 1}{2.5\sqrt{e}}}_{\text{comparisons within colorless ranges}} &+ \underbrace{\frac{\sigma k - 5n}{8}}_{\text{comparisons for colored nodes}} \geq n \cdot \ln \frac{n}{2.5\sqrt{e}k} + \frac{\sigma k - 5n}{8} \\
&= n \cdot \ln \frac{\sigma}{2.5\sqrt{e}x} + \frac{xn - 5n}{8} \\
&= n \cdot \ln \sigma + n \cdot \left(\frac{x - 5}{8} - \ln 2.5\sqrt{e}x \right) \\
&> n \cdot \ln \sigma - 3.12074n
\end{aligned}$$

The first step follows from $\sum_{i=1}^k (m_i + 1) = n$ and the log sum inequality (see [21, Theorem 2.7.1]). In the second step we replace k by using $x = \frac{\sigma k}{n}$. The third step uses simple arithmetic operations. The last step is reached by substituting $x = 8$, which minimizes the equation. Finally, we assumed that σ is divisible by 4. We account for this by adjusting the lower bound to $n \ln(\sigma - 3) - 3.12074n$, which is larger than $n \ln \sigma - 3.6n$ for $\sigma \geq 8$.

THEOREM 1.1. *For any integers n and σ with $8 \leq \sigma \leq n$, there is no deterministic algorithm that performs at most $n \ln \sigma - 3.6n = \mathcal{O}(n \ln \sigma)$ comparisons in the worst case, and determines whether a length- n string with at most σ distinct symbols is square-free.*

4 Upper Bound

In this section, we consider the problem of testing square-freeness of a given string. We introduce an algorithm that decides whether or not a string is square-free using only $\mathcal{O}(n \lg \sigma)$ comparisons, matching the lower bound from Section 3.2. Note that this algorithm is not yet time efficient because, apart from the performed symbol comparisons, it uses other operations that are expensive in the Word RAM model. A time efficient implementation of the algorithm will be presented in Section 5, where we first achieve $\mathcal{O}(n \lg \sigma + n \log^* n)$ time, and then improve this to $\mathcal{O}(n \lg \sigma)$ time.

High-Level Overview of the Upper Bound Algorithm As discussed earlier, one way of detecting squares uses the f -factorisation of the string, which is similar to its LZ factorisation. However, as shown in Corollaries 3.1 and 3.2, we cannot compute either of these factorisations over a general unordered alphabet in $o(n\sigma)$ comparisons. Therefore, we will instead use a novel type of factorisation that can be seen as an approximate version of the LZ factorisation. The high-level structure of our algorithm (and of the remainder of this section) is as follows:

- Before introducing the factorisation, we show how to compute a sparse suffix tree while trying to use as few symbol comparisons as possible. The sample positions will later be taken from a so-called difference cover with convenient synchronizing properties.

- The approximate factorisation is parametrized with some positive integer Δ . For larger Δ , we need fewer comparisons to compute the factorisation. However, with larger Δ we also lose the ability to find short squares. We show how to compute the factorisation from a sparse suffix tree, where the number of samples is inverse proportional to Δ .
- We show how to detect squares of length $\geq 8\Delta$ from the Δ -approximate factorisation.
- A first simple algorithm fixes Δ depending on n and σ (assuming that σ is known), and uses the approximate LZ factorisation to find all squares of length at least 8Δ . It then finds all the shorter squares by dividing the string in blocks of length 8Δ , and applying the original algorithm by Main & Lorentz (Theorem 2.1) on each block pair. Our choice of Δ leads to $\mathcal{O}(n(\lg \sigma + \lg \lg n))$ comparisons.
- A more sophisticated algorithm starts with a large $\Delta = \Omega(n)$, and then progressively decreases Δ in at most $\mathcal{O}(\lg \lg n)$ phases, where later phases detect shorter squares. Since we cannot afford $\Omega(n)$ comparisons per phase, we use a deactivation technique, where whenever we perform a large number of comparisons in a phase, we will discard a large part of the string in all following phases. This leads to optimal $\mathcal{O}(n \lg \sigma)$ comparisons.

4.1 Sparse Suffix Trees and Difference Covers

LEMMA 4.1. *The sparse suffix tree containing any b suffixes $T[i_1..n], \dots, T[i_b..n]$ of $T[1..n]$ can be constructed using only $\mathcal{O}(b\sigma \log b)$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

Proof. We maintain a union-find structure over the positions of $T[1..n]$. Initially, each position is in a separate component. Before issuing a query $T[x] \stackrel{?}{=} T[y]$, we check if x and y are in the same component of the union-find structure, and if so immediately return that $T[x] = T[y]$ without performing any comparisons. Otherwise, we issue the query and if it returns that $T[x] = T[y]$ we merge the components of x and y . Thus, the total number of issued queries with positive answer, over all invocations of the lemma, is less than n , and it remains to bound the number of issued queries with negative answer.

Considering inserting the suffixes $T[i_j..n]$ one-by-one into an initially empty sparse suffix tree \mathcal{T} . To insert the next suffix, we descend from the root of \mathcal{T} to identify the node u corresponding to the longest common prefix of $T[i_j..n]$ and the already inserted suffixes, make u explicit unless it is explicit already, and then add an edge from u to a new leaf corresponding to the whole $T[i_j..n]$. We say that the insertion procedure terminates at u . u can be identified with only $\mathcal{O}(\sigma \log b)$ comparisons with negative answers as follows. Let the current node (initially, the root of \mathcal{T}) be v , and its children v_1, \dots, v_d , where $d \leq \sigma$. Here, v can be either explicit or implicit, in the latter case $d = 1$. We arrange the children of v so that the number of leaves in the subtree rooted at v_1 is at least as large as the number of leaves in the subtree rooted at any other child of v . Then, we compare the character on the edge leading to v_1 with the corresponding character of the current suffix. If they are equal we continue with v_1 , otherwise we compare the characters on the edges leading to v_2, \dots, v_d with the corresponding character of the current suffix one-by-one. Then, we either continue with some v_j , $j \geq 2$, or terminate at v . To bound the number of comparisons with negative answer, observe that such comparisons only occur when we either terminate at v or continue with v_j , $j \geq 2$. Whenever we continue with v_j , $j \geq 2$, the number of leaves in the current subtree rooted at v_j decreases at least by a factor of 2 compared to subtree rooted at v (as the subtree rooted at v_1 had the largest number of leaves). Thus, during the whole descent from the root performed during an insertion this can happen only at most $1 + \log b$ times. Every time we do not continue in the subtree v_1 we might have up to $d \leq \sigma$ comparisons with negative answer, thus the total number of such comparisons is as claimed¹. \square

Now we describe the sample positions that we will later use to compute the approximate LZ factorisation. A set $\mathbf{S} \subseteq \mathbb{N}$ is called a t -cover of $\{1, \dots, n\}$ if there is a constant-time computable function h such that, for any $1 \leq i, j \leq n - t + 1$, we have $0 \leq h(i, j) < t$ and $i + h(i, j), j + h(i, j) \in \mathbf{S}$.

LEMMA 4.2. *For any n and $t \leq n$, there exists a t -cover $\mathbf{D}(t)$ of $\{1, \dots, n\}$ with size $\mathcal{O}(n/\sqrt{t})$. Furthermore, its elements can be enumerated in time proportional to their number.*

¹In the descent, if all children are sorted according to their subtree size, the number of comparisons decreases to $\mathcal{O}(b(\sigma/\log \sigma) \log b)$, but this appears irrelevant for our final algorithm.



Figure 2: Positions in a Δ -difference cover.

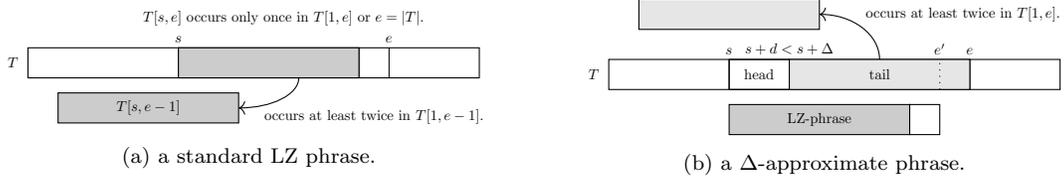


Figure 3: Illustration of the definition of a LZ-phrase and a Δ -approximate phrase.

Proof. We use the well-known combinatorial construction known as difference covers, see e.g. [61]. Let $r = \lfloor \sqrt{t} \rfloor$ and define $\mathbf{D}(t) = \{i \in \{1, \dots, n\} : i \bmod r = 0 \text{ or } i \bmod r^2 \in \{0, \dots, r-1\}\}$. By definition, $|\mathbf{D}(t)| \leq \lfloor n/r \rfloor + \lfloor n/r^2 \rfloor r = \mathcal{O}(n/\sqrt{t})$. The function $h(i, j)$ is defined as $a + b \cdot r$, where $a = (r - i) \bmod r$ and $b = (r - \lfloor (j + a)/r \rfloor) \bmod r$. Note that $i + h(i, j) \leq n$ and $j + h(i, j) \leq n$. Then, $i + (a + b \cdot r) = 0 \pmod{r}$, while $\lfloor (j + (a + b \cdot r))/r \rfloor = \lfloor (j + a)/r + b \rfloor = 0 \bmod r$ implies $j + h(i, j) \bmod r^2 \in \{0, \dots, r-1\}$, thus $i + h(i, j), j + h(i, j) \in \mathbf{D}(t)$ as required. \square

4.2 Detecting Squares with a Δ -Approximate LZ Factorisation A crucial notion in our algorithm is the following variation on the standard Lempel-Ziv factorisation:

DEFINITION 4.1. (Δ -APPROXIMATE LZ FACTORISATION) *For a positive integer parameter Δ , the fragment $T[s..e]$ is a Δ -approximate LZ phrase if it can be split into a head and a tail $T[s..e] = \text{head}(T[s..e])\text{tail}(T[s..e])$ such that $|\text{head}(T[s..e])| < \Delta$ and additionally*

- *$\text{tail}(T[s..e])$ is either empty or occurs at least twice in $T[1..e]$, and*
- *the unique (standard) LZ phrase $T[s..e']$ starting at position s satisfies $e' - 1 \leq e$.*

In a Δ -approximate LZ factorisation $T = b_1 b_2 \dots b_z$, each factor b_i is a Δ -approximate phrase $T[s..e]$ with $s = 1 + \sum_{j=1}^{i-1} |b_j|$ and $e = \sum_{j=1}^i |b_j|$.

Note that a standard LZ phrase is not a Δ -approximate phrase. Also, while the LZ phrase starting at each position (and thus also the LZ factorisation) is uniquely defined, there may be multiple different Δ -approximate phrases starting at each position. This also means that a single string can have multiple different Δ -approximate factorisations. The definitions of both standard and Δ -approximate LZ phrases are illustrated in Fig. 3.

The intuition behind the above definition is that constructing the Δ -approximate LZ factorisation becomes easier for larger values of Δ . In particular, for $\Delta = n$ one phrase is enough. We formalise this in the following lemma, which is made more general for the purpose of obtaining the final result in this section.

LEMMA 4.3. *For any parameter $\Delta \in [1, m]$, a Δ -approximate LZ factorisation of any fragment $T[x..y]$ of length m can be computed with $\mathcal{O}(m\sigma \log m/\sqrt{\Delta})$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

Proof. By Lemma 4.2, there exists a Δ -cover $\mathbf{D}(\Delta)$ of $\{1, \dots, n\}$ with size $\mathcal{O}(n/\sqrt{\Delta})$. Let $S = \mathbf{D}(\Delta) \cap \{x, x+1, \dots, y\}$. Let $S = \{i_1, i_2, \dots, i_b\}$. It is straightforward to verify that the construction additionally guarantees $b = \mathcal{O}(m/\sqrt{\Delta})$. We apply Lemma 4.1 on the suffixes $T[i_1..n], \dots, T[i_b..n]$ to obtain their sparse suffix tree T with $\mathcal{O}(b\sigma \log b)$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma. T allows us to obtain the longest common prefix of any two fragments $T[i..y]$ and $T[j..y]$, for $i, j \in S$, with no additional comparisons. By the properties of $\mathbf{D}(\Delta)$, for any $i, j \in \{x, x+1, \dots, y-\Delta+1\}$ we have $0 \leq h(i, j) < \Delta$ and $i+h(i, j), j+h(i, j) \in S$.

We compute the Δ -approximate LZ factorisation of $T[x..y]$ phrase-by-phrase. Denoting the remaining suffix of the whole $T[x..y]$ by $T[x'..y]$, we need to find $x' \leq y' \leq y$ such that $T[x'..y']$ is a Δ -approximate phrase. This is done as follows. We iterate over every $x' \leq x'' < x' + \Delta$ such that $x'' \in S$. For every such x'' , we consider every $x \leq a' < x'$ such that $a' \in S$, and compute the length ℓ of the longest common prefix of $T[x''..y]$ and $T[a'..y]$. Among all such x'', a' we choose the pair that results in the largest value of $x'' - x' + \ell - 1$ and choose the next phrase to be $T[x'..(x'' + \ell - 1)]$, with the head being $T[x'..(x'' - 1)]$ and the tail $T[x''..(x'' + \ell) - 1]$. Finally, if there is no such pair, or the value of $x'' - x' + \ell - 1$ corresponding to the found pair is less than $\Delta - 2$, we take the next phrase to be $T[x'..\min\{x' + \Delta - 1, y\}]$ (with empty tail). Selecting such a pair requires no extra comparisons, as for every $x'', a' \in S$ we can use the sparse suffix tree to compute ℓ . While it is clear that the generated Δ -approximate phrase has the required form, we need to establish that it is sufficiently long.

Let $T[x'..y'']$ be the (unique) phrase being a prefix of $T[x'..y]$. If $y'' < x' + \Delta - 1$ then we only need to ensure that the generated Δ -approximate phrase is of length at least $\min\{\Delta - 1, y - x' + 1\}$, which is indeed the case. Therefore, it remains to consider the situation when $y'' \geq x' + \Delta - 1$. Let $T[a..b]$ be the previous occurrence of $T[x'..(y'' - 1)]$ in $T[x..y]$ (because $T[x'..y'']$ is a phrase this is well defined). Thus, $T[a..b] = T[x'..(y'' - 1)]$ and $a < x'$. Because $y'' \geq x' + \Delta - 1$ and $y'' \leq y$, as explained above $0 \leq h(a, x') < \Delta$ and $a + h(a, x'), x' + h(a, x') \in S$. We will consider $x'' = x' + h(a, x')$ and $a' = a + h(a, x')$ in the above procedure. Next, $T[a'..b] = T[x''..(y'' - 1)]$, so when considering this pair we will obtain $\ell \geq |T[x''..(y'' - 1)]|$. Thus, for the found pair we will have $x'' + \ell - 1 \geq y'' - 1$ as required in the definition of a Δ -approximate phrase. \square

Next, we show that even though the Δ -approximate LZ factorisation does not capture all distinct squares, as it is the case for the standard LZ factorisation, it is still helpful in detecting all sufficiently long squares. A crucial component is the following property of the Δ -approximate LZ factorisation.

LEMMA 4.4. *Let $b_1 b_2 \dots b_z$ be a Δ -approximate LZ factorisation of a string T . For every square $T[s..s + 2\ell - 1]$ of length $2\ell \geq 8\Delta$, there is at least one phrase b_i with $|\text{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$ such that $\text{tail}(b_i)$ and the right-hand side $T[s + \ell..s + 2\ell - 1]$ of the square intersect.*

Proof. Assume that all tails that intersect $T[s + \ell..s + 2\ell - 1]$ are of length less than $\frac{\ell}{4}$, then the respective phrases of these tails are of length at most $\frac{\ell}{4} + \Delta - 1$ (because each head is of length less than Δ). This means that $T[s + \ell..s + 2\ell - 1]$ intersects at least $\lceil \ell / (\frac{\ell}{4} + \Delta - 1) \rceil \geq \lceil \ell / (\frac{\ell}{2} - 1) \rceil = 3$ phrases. Thus there is some phrase $b_i = T[x..y]$ properly contained in $T[s + \ell..s + 2\ell - 1]$, formally $s + \ell < x \leq y < s + 2\ell - 1$. However, this contradicts the definition of the Δ -approximate LZ factorisation because $T[x..s + 2\ell]$ is the prefix of a standard LZ phrase (due to $T[x..s + 2\ell - 1] = T[x - \ell..s + \ell - 1]$), and the Δ -approximate phrase $b_i = T[x..y]$ must satisfy $y \geq s + 2\ell - 1$. The contradiction implies that $T[s + \ell..s + 2\ell - 1]$ intersects a tail of length at least $\frac{\ell}{4} \geq \Delta$. \square

LEMMA 4.5. *Given a Δ -approximate LZ factorisation $T = b_1 b_2 \dots b_z$, we can detect a square of size $\geq 8\Delta$ in $\mathcal{O}\left(\sum_{|\text{tail}(b_i)| \geq \Delta} |\text{tail}(b_i)| + z\right)$ time and $\mathcal{O}\left(\sum_{|\text{tail}(b_i)| \geq \Delta} |\text{tail}(b_i)|\right)$ comparisons.*

Proof. We consider each phrase $b_i = T[a_1..a_3]$ with $\text{head}(b_i) = T[a_1..a_2 - 1]$ and $\text{tail}(b_i) = T[a_2..a_3]$ separately. Let $k = |\text{tail}(b_i)|$. If $k \geq \Delta$, we apply Lemma 2.1 to $x_1 = T[a_2 - 8k..a_2 - 1]$ and $y_1 = T[a_2..a_3 + 4k - 1]$, as well as $x_2 = T[a_2 - 8k..a_3 - 1]$ and $y_2 = T[a_3..a_3 + 4k - 1]$ trimmed to $T[1..n]$. This takes $\mathcal{O}(|\text{tail}(b_i)|)$ time and comparisons, or $\mathcal{O}\left(\sum_{|\text{tail}(b_i)| \geq \Delta} |\text{tail}(b_i)|\right)$ time and comparisons for all phrases. We need additional $\mathcal{O}(z)$ time to check if $k \geq \Delta$ for each phrase.

Now we show that the described strategy detects a square of size at least 8Δ . Let $T[s..s + 2\ell - 1]$ be any such square. Due to Lemma 4.4, the right-hand side $T[s + \ell..s + 2\ell - 1]$ of this square intersects some tail $\text{tail}(b_i) = T[a_2..a_3]$ of length $k = |\text{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$. Due to the intersection, it holds $a_2 \leq s + 2\ell - 1$ and $a_3 \geq s + \ell$. Thus, when processing b_i and applying Lemma 2.1, the starting position of x_1 and x_2 satisfies $a_2 - 8k \leq s + 2\ell - 1 - 8\frac{\ell}{4} = s - 1$, while the end position of y_1 and y_2 satisfies $a_3 + 4k - 1 \geq s + \ell + 4\frac{\ell}{4} - 1 = s + 2\ell - 1$. Therefore, the square is entirely contained in the respective fragments corresponding to $x_1 y_1$ and $x_2 y_2$. If $s < a_2 \leq s + 2\ell - 1$, we find the square with our choice of x_1 and y_1 . If $s < a_3 \leq s + 2\ell - 1$, we find the square with our choice of x_2 and y_2 . Otherwise, $T[s..s + 2\ell - 1]$ is entirely contained in tail $T[a_2..a_3]$, and we find another occurrence of the square further to the left. \square

4.3 Simple Algorithm for Detecting Squares Now we have all the tools to introduce our simple method for testing square-freeness of $T[1..n]$ using $\mathcal{O}(n(\log \sigma + \log \log n))$ comparisons, assuming that σ is known in advance. Let $\Delta = (\sigma \log n)^2$. We partition $T[1..n]$ into blocks of length 8Δ , and denote the k^{th} block by B_k . A square of length at most 8Δ can be found by invoking Theorem 2.1 on B_1B_2 , B_2B_3 , and so on. This takes $\mathcal{O}(\Delta \log \Delta) = \mathcal{O}(\Delta(\log \sigma + \log \log n))$ comparisons for each pair of adjacent blocks, or $\mathcal{O}(n(\log \sigma + \log \log n))$ comparisons in total. It remains to test for squares of length exceeding 8Δ . This is done by first invoking Lemma 4.3 to compute a Δ -approximate LZ factorisation of $T[1..n]$ with $\mathcal{O}(n\sigma \log n/\sqrt{\Delta}) = \mathcal{O}(n)$ comparisons, and then using Lemma 4.5, which adds another $\mathcal{O}(n)$ comparisons. The total number of comparisons is dominated by the $\mathcal{O}(n(\log \sigma + \log \log n))$ comparisons needed to apply Theorem 2.1 to the block pairs.

4.4 Improved Algorithm for Detecting Squares We are now ready to describe the algorithm that uses only $\mathcal{O}(n \log \sigma)$ comparisons without knowing the value of σ . Intuitively, we will proceed in phases, trying to “guess” the value of σ . We first observe that Lemma 4.3 can be extended to obtain the following.

LEMMA 4.6. *There is an algorithm that, given any parameter $\Delta \in [1, m]$, estimate $\tilde{\sigma}$ and fragment $T[x..y]$ of length m , uses $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$ comparisons plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma, and either computes a Δ -approximate LZ factorisation of $T[x..y]$ or determines that $\sigma > \tilde{\sigma}$.*

Proof. We run the procedure described in the proof of Lemma 4.3 and keep track of the number of comparisons with negative answer. As soon as it exceeds $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$ (where the constant follows from the complexity analysis) we know that necessarily $\sigma > \tilde{\sigma}$, so we can terminate. Otherwise, the algorithm obtains a Δ -approximate LZ factorisation with $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$ comparisons. Comparisons with positive answer are paid for globally. \square

Now we describe how to find any square using $\mathcal{O}(n \log \sigma)$ comparisons. We define the sequence $\sigma_t = 2^{2^{\lceil \log \log n \rceil - t}}$, for $t = 0, 1, \dots, \lceil \log \log n \rceil$. We observe that $\sigma_{t-1} = (\sigma_t)^2$, and proceed in phases corresponding to the values of t . In the t^{th} phase we are guaranteed that any square of length at least $(\sigma_t)^2$ has been already detected, thus it is enough to consider squares of length less than $(\sigma_t)^2$. We partition the whole $T[1..n]$ into blocks of length $(\sigma_t)^2$, and denote the k^{th} block by B_k . A square of length less than $(\sigma_t)^2$ is fully contained within some two consecutive blocks B_iB_{i+1} , hence we consider each such pair B_1B_2 , B_2B_3 , and so on. We first apply Lemma 4.6 with $\Delta = \sigma_t/8$ and $\tilde{\sigma} = (\sigma_t)^{1/4}/\log(\sigma_t)$ to find an $(\sigma_t/8)$ -approximate LZ factorisation of the corresponding fragment of $T[1..n]$, and then use Lemma 4.5 to detect squares of length at least σ_t . We cannot always afford to apply Lemma 4.5 to all block pairs. Thus, we have to deactivate some of the blocks, which we explain when analysing the number of comparisons performed by the algorithm. If any of the calls to Lemma 4.6 in the current phase detects that $\sigma > \tilde{\sigma}$, we switch to applying Theorem 2.1 on every pair of blocks B_iB_{i+1} of the current phase and then terminate the whole algorithm.

We now analyse the total number of comparisons, ignoring the $\mathcal{O}(n)$ comparisons shared by all invocations of Lemma 4.6. Throughout the t^{th} phase, we use $\mathcal{O}(n \cdot \tilde{\sigma} \log \sigma_t/\sqrt{\Delta}) = \mathcal{O}(n \cdot (\sigma_t)^{1/4}/\log(\sigma_t) \cdot \log(\sigma_t)/\sqrt{\sigma_t}) = \mathcal{O}(n/(\sigma_t)^{1/4})$ comparisons to construct the Δ -approximate factorisations (using Lemma 4.6) until we either process all pairs of blocks or detect that $\sigma > (\sigma_t)^{1/4}/\log(\sigma_t)$. In the latter case, we finish off the whole computation with $\mathcal{O}(n \log(\sigma_t))$ comparisons (using Theorem 2.1), and by assumption on σ this is $\mathcal{O}(n \log \sigma)$ as required. Until this happens (or until we reach phase $t = \lceil \log \log n \rceil - 3$ where $\sigma_t \leq 256$), we use $\mathcal{O}(\sum_{t=0}^t n/(\sigma_t)^{1/4})$ comparisons to construct the Δ -approximate factorisations, for some $0 \leq t' \leq \lceil \log \log n \rceil$. To analyse the sum, we need the following bound (made more general for the purpose of the next section).

LEMMA 4.7. *For any $0 \leq x \leq y$ and $c \geq 0$ we have $\sum_{i=x}^y 2^{ic}/2^{2^i} = \mathcal{O}(2^{xc}/2^{2^x})$.*

Proof. We observe that the sequence of exponents 2^i is strictly increasing from $i = 0$, hence

$$\sum_{i=x}^y \frac{2^{ic}}{2^{2^i}} \leq \sum_{i=2^x}^{2^y} \frac{i^c}{2^i} \leq \sum_{i=2^x}^{\infty} \frac{i^c}{2^i} = \sum_{i=0}^{\infty} \frac{(2^x + i)^c}{2^{(2^x+i)}} \leq \sum_{i=0}^{\infty} \frac{2^{xc} \cdot (i+1)^c}{2^{(2^x+i)}} = \frac{2^{xc}}{2^{2^x}} \cdot \sum_{i=0}^{\infty} \frac{(i+1)^c}{2^i}.$$

$\sum_{i=0}^{\infty} \frac{(i+1)^c}{2^i}$ is a series of positive terms, we thus use Alembert’s ratio test $\frac{(i+2)^c}{2^{i+1}} \cdot \frac{2^i}{(i+1)^c} = \frac{1}{2} \frac{(i+2)^c}{(i+1)^c}$ which tends to $\frac{1}{2}$ when i goes to the infinity, thus the series converges to a constant. \square

COROLLARY 4.1. For any $0 \leq t' \leq \lceil \log \log n \rceil$, it holds $\sum_{t=0}^{t'} n \cdot \text{polylog}(\sigma_t)/(\sigma_t)^{1/4} = \mathcal{O}(n)$.

Proof. We have to show that $\sum_{t=0}^{t'} n \log^c(\sigma_t)/(\sigma_t)^{1/4} = \mathcal{O}(n)$ for any constant $c \geq 0$. We achieve this by splitting the sum and applying Lemma 4.7.

$$\begin{aligned} \sum_{t=0}^{t'} \frac{n \log^c(\sigma_t)}{(\sigma_t)^{1/4}} &\leq \sum_{t=0}^{\lceil \log \log n \rceil} \frac{n \cdot (2^{\lceil \log \log n \rceil - t})^c}{(2^{2^{\lceil \log \log n \rceil - t}})^{1/4}} = \sum_{t=0}^{\lceil \log \log n \rceil} \frac{n \cdot (2^t)^c}{(2^{2^t})^{1/4}} \\ &= n \cdot \sum_{t=0}^{\lceil \log \log n \rceil} \frac{2^{tc}}{2^{2^{t-2}}} = n \cdot \left(\frac{1}{2^{2^{-2}}} + \frac{2^c}{2^{2^{-1}}} + 4 \cdot \sum_{t=0}^{\lceil \log \log n \rceil - 2} \frac{2^{tc}}{2^{2^t}} \right) = \mathcal{O}(n) \quad \square \end{aligned}$$

Thus, all invocations of Lemma 4.6 cause $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4}) = \mathcal{O}(n)$ comparisons.

Deactivating Block Pairs It remains to analyse the number of comparisons used by Lemma 4.5 throughout all phases. As mentioned earlier, we cannot actually afford to apply Lemma 4.5 to all block pairs. Thus, we introduce a mechanism that deactivates some of the pairs.

First, note that there are $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^2) \subseteq \mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4}) = \mathcal{O}(n)$ block pairs in all phases. For each pair, we store whether it has been deactivated or not, where being deactivated roughly means that we do not have to investigate the pair because it does not contain a leftmost distinct square. For each block pair $B_i B_{i+1}$ in the current phase t , we first check if it has been marked as deactivated. If not, we also check if it has been *implicitly* deactivated: if any of the two pairs from the previous phase that contain $B_i B_{i+1}$ are marked as deactivated. If $B_i B_{i+1}$ has been implicitly deactivated, then we mark it as deactivated and do not apply Lemma 4.6 and Lemma 4.5 (the implicit deactivation serves the purpose of propagating the deactivation to all later phases). Note that if some position of the string is not contained in any active block pairs in some phase, then it is also not contained in any active block pairs in all later phases. This is because it always holds $\sigma_{t-1} = (\sigma_t)^2$ (with no rounding required), which guarantees that block boundaries of earlier phases do not intersect blocks of later phases.

We only apply Lemma 4.6 and then Lemma 4.5 to $B_i B_{i+1}$ if the pair has neither explicitly nor implicitly been deactivated. When applying Lemma 4.5, a tail $T[a..a + \ell]$ contributes $\mathcal{O}(\ell)$ comparisons if $\ell \geq \Delta = \sigma_t/8$ (and otherwise it contributes no comparisons). Note that if we finish the phase without detecting a square, then $T[a..a + \ell]$ has a previous occurrence $T[a'..a' + \ell] = T[a..a + \ell]$ with $a' + \ell \leq a$, i.e., the occurrences do not overlap (if the only previous occurrence is overlapping, then $T[a'..2a - a']$ is a square that is detected when applying Lemma 4.5). Thus we can assume that there is a previous non-overlapping occurrence of $T[a..a + \ell]$. It follows that any block pair (of any phase) entirely contained in $T[a..a + \ell]$ occurs somewhere before $B_i B_{i+1}$ in the string, and thus such block pairs can be deactivated.

The mechanism used for deactivation works as follows. Let $T[a..a + \ell]$ be a tail contributing $\mathcal{O}(\ell)$ comparisons with $\ell \geq \Delta = \sigma_t/8$ in phase t . We mark all blocks of phase $t + 2$ that are entirely contained in $T[a..a + \ell]$ as deactivated. Note that blocks in phase $t + 2$ are of length $\sqrt{\sigma_t}$, and consider the fragment $T[a + 2\sqrt{\sigma_t}..a + \ell - 2\sqrt{\sigma_t}]$. In phase $t + 2$, and by implicit deactivation in all later phases, this fragment only contains or overlaps block pairs that have been deactivated. Thus, after phase $t + 1$, we will never inspect any of the symbols in $T[a + 2\sqrt{\sigma_t}..a + \ell - 2\sqrt{\sigma_t}]$ again. We say that tail $T[a..a + \ell]$ deactivated the fragment of length $\ell - 4\sqrt{\sigma_t} = \Omega(\ell)$, which is positive until phase $t = \lceil \log \log n \rceil - 3$ because $\sigma_t > 256$. Since the number of deactivated positions is linear in the number of comparisons that the tail contributes to Lemma 4.5, it suffices to show that each position gets deactivated at most a constant number of times. In a single phase, any position gets deactivated at most twice. This is because the tails of each factorisation do not overlap by definition, but the tails of the two factorisations of adjacent block pairs $B_i B_{i+1}$ and $B_{i+1} B_{i+2}$ can overlap. If a position gets deactivated for the first time in phase t , then (as explained earlier) we will not consider it in any of the phases $t' \geq t + 2$. Thus, it can only be that we deactivate the position again in phase $t + 1$, but not in any later phases. In total, each position gets deactivated at most four times.

We have shown:

THEOREM 1.2. *Testing square-freeness of a length- n string that contains σ distinct symbols from a general unordered alphabet can be done with $\mathcal{O}(n \log \sigma)$ comparisons.*

5 Algorithm

In this section we show how to implement the approach described in the previous section to work in $\mathcal{O}(n \log \sigma)$ time. The main difficulty is in efficiently implementing the sparse suffix tree construction algorithm, and then computing a Δ -approximate factorisation. We first show how to obtain an $\mathcal{O}(n \log \sigma + n \log^* n)$ time algorithm that still uses only $\mathcal{O}(n \log \sigma)$ comparisons, and then further improve its running time to $\mathcal{O}(n \log \sigma)$.

5.1 Constructing Suffix Tree and Δ -Approximate Factorisation To give an efficient algorithmic construction of the sparse suffix tree from Lemma 4.1, we will use a restricted version of LCEs, where a query $\text{ShortLCE}_x(i, j)$ (for any positive integer x) returns $\min(x, \text{LCE}(i, j))$. The following result was given by Gawrychowski et al. [39]:

LEMMA 5.1. (LEMMA 14 IN [39]) *For a length- n string over a general unordered alphabet², a sequence of q queries $\text{ShortLCE}_{4^{k_i}}$ for $i \in \{1, \dots, q\}$ can be answered online in total time $\mathcal{O}(n \log^* n + s)$ and $\mathcal{O}(n + q)$ comparisons³, where $s = \sum_{i=1}^q (k_i + 1)$.*

In the lemma, apart from the $\mathcal{O}(n \log^* n)$ time, each $\text{ShortLCE}_{4^{k_i}}$ query accounts for $\mathcal{O}(k_i + 1)$ time. Note that we can answer the queries online, without prior knowledge of the number and length of the queries. Also, computing an LCE in a fragment $T[x..y]$ of length m trivially reduces to a $\text{ShortLCE}_{4^{\lceil \log_4 m \rceil}}$ query on T . Thus, we have:

COROLLARY 5.1. *A sequence of q longest common extension queries on a fragment $T[x..y]$ of length m over a general unordered alphabet can be answered in $\mathcal{O}(q \log m)$ time plus $\mathcal{O}(n \log^* n)$ time shared by all invocations of the lemma. The number of comparisons is $\mathcal{O}(q)$, plus $\mathcal{O}(n)$ comparisons shared by all invocations of the lemma.*

While constructing the sparse suffix tree, we will maintain a heavy-light decomposition using a rebuilding scheme introduced by Gabow [37]. Let $L(u)$ denote the number of leaves in the subtree of a node u . We use the following recursive construction of a heavy-light decomposition: Starting from a node r (initially the root of the tree), we find the deepest descendant node e such that $L(e) \geq \frac{5}{6}L(r)$ (possibly $e = r$). The path p from the root $r(p) = r$ to $e(p) = e$ is a heavy path and any edge (u, v) on this path is such that $L(v) \geq \frac{5}{6}L(u)$, we call those edges *heavy*. As a consequence, a node u can have at most one child v such that (u, v) is heavy. For each edge (u, v) with u is on the heavy path and v is not, we recursively build a new heavy path construction starting from v .

When inserting a new suffix in our tree, we keep track of the insertion in the following way: for every root of a heavy path, we maintain $I(u)$ the number of insertions made in the subtree of u since we built the heavy-light decomposition of this subtree. When $I(u) \geq \frac{1}{6}L(u)$ we recalculate the values of $L(v)$ for all nodes v in the subtree of u and rebuild the heavy-light decomposition for the subtree of u .

This insures that, despite insertion, for any heavy path starting at node r and a node u on that heavy path, $L(e) \geq \frac{2}{3}L(r)$. When crossing a non heavy edge the number of nodes in the subtree reduces by a factor at least $\frac{5}{6}$ which leads to the following property:

OBSERVATION 5.1. *The path from any node to the root crosses at most $\mathcal{O}(\log m)$ heavy paths.*

Additionally, rebuilding a subtree of size s takes $\mathcal{O}(s)$ time and adding a suffix $T[i_j..y]$ to the tree increases $I(r)$ for each path p from the root r to the new leaf. Those are at most $\mathcal{O}(\log m)$ nodes and thus maintaining the heavy path decomposition takes amortized time $\mathcal{O}(\log n)$ per insertion.

With those building blocks now clearly defined, we are ready to describe the construction of the sparse suffix tree.

LEMMA 5.2. *The sparse suffix tree containing any b suffixes $T[i_1..y], \dots, T[i_b..y]$ of $T[x..y]$ with $m = |T[x..y]|$ can be constructed using $\mathcal{O}(b\sigma \log b \log m)$ time plus $\mathcal{O}(n \log^* n)$ time shared by all invocations of the lemma.*

²Lemma 14 in [39] does not explicitly mention that it works for general unordered alphabet. However, the proof of the lemma relies solely on equality tests.

³Lemma 14 in [39] does not explicitly mention that it requires $\mathcal{O}(n + q)$ comparisons. However, they use a union-find approach where there can be at most $\mathcal{O}(n)$ comparisons with outcome "equal", and each LCE query performs only one comparison with outcome "not-equal", similarly to what we describe in the proof of Lemma 4.1.

Proof. As in the proof of Lemma 4.1, we consider the insertion of a suffix $T[i_j..y]$ into the sparse suffix tree with suffixes $T[i_1..y], T[i_2..y] \cdots T[i_{j-1}..y]$. At all times, we maintain the heavy path decomposition. Additionally, we maintain for each heavy path a predecessor data structure, where given some length ℓ , we can quickly identify the deepest explicit node on the heavy path that spells a string of length at least ℓ . The data structure can, e.g., be a balanced binary search tree with insertion and search operations in $\mathcal{O}(\log b)$ time (the final sparse suffix tree and thus each heavy path contains $\mathcal{O}(b)$ nodes). When rebuilding a subtree of the heavy path decomposition, we also have to rebuild the predecessor data structure for each of its heavy paths. Thus, rebuilding a size- q subtree takes $\mathcal{O}(q \log b)$ time (each node is on exactly one heavy path and has to be inserted into one predecessor data structure), and the amortized insertion time increases from $\mathcal{O}(\log m)$ to $\mathcal{O}(\log m \cdot \log b)$. Whenever we insert a suffix, we make at most one node explicit, and thus have to perform at most one insertion into a predecessor data structure. The time for this is $\mathcal{O}(\log b)$, which is dominated by the previous term.

When inserting $T[i_j..y]$, we look for the node u corresponding to the longest common prefix between $T[i_j..y]$ and the inserted suffixes, make u explicit if necessary and add a new leaf corresponding to $T[i_j..y]$ attached to u . Let v be the current node (initialized by the root, and always an explicit node) and v_1, \dots, v_d be its (explicit) children. If there is a heavy edge (v, v_a) for $1 \leq a \leq d$, let p be the corresponding heavy path. For each heavy path p , we store the label of one leaf (i.e., the starting position of one suffix) that is contained in the subtree of $e(p)$. Thus, we can use Corollary 5.1 to compute the longest common extension between the string spelled by $e(p)$ and $T[i_j..y]$. Now we use the predecessor data structure on the heavy path to find the deepest (either explicit or implicit) node v' on the path that spells a prefix of $T[i_j..y]$. If v' is implicit, we make it explicit and add the leaf. If v' is explicit and $v' \neq v$, we use v' as the new current node and continue. Otherwise, it holds $v' = v$, i.e., the suffix does not belong to the subtree rooted in v_a . In this case, we issue d LCE queries between $T[i_j..y]$ and each of the strings spelled by the nodes v_1, \dots, v_d . This either reveals that we can continue using one of the v_a as the new current node, or that we can create a new explicit node on some (v, v_a) edge and attach the leaf to it, or that we can simply attach a new leaf to v .

Now we analyse the time spent while inserting one suffix. We spent $\mathcal{O}(b \cdot \log m \cdot \log b)$ total time time for inserting $\mathcal{O}(b)$ nodes into the dynamic heavy path decomposition and the predecessor data structures. In each step of the insertion process, we either (i) move as far as possible along some heavy path or (ii) move along some non-heavy edge. For (i), we issue one LCE query and one predecessor query. For (ii) we issue $\mathcal{O}(\sigma)$ LCE queries. Due to Observation 5.1, both (i) and (ii) happen at most $\mathcal{O}(\log b)$ times per suffix. Thus, for all suffixes, we perform $\mathcal{O}(b \log b)$ predecessor queries and $\mathcal{O}(b \sigma \log b)$ LCE queries. The total time is $\mathcal{O}(b \log^2 b)$ for predecessor queries, and $\mathcal{O}(b \sigma \log b \log m)$ for LCE queries (apart from the $n \log^* n$ time shared by all invocations of Corollary 5.1). \square

LEMMA 5.3. *For any parameter $\Delta \in [1, m]$, a Δ -approximate LZ factorisation of any fragment $T[x..y]$ of length m can be computed in $\mathcal{O}(m \sigma \log^2 m / \sqrt{\Delta})$ time plus $\mathcal{O}(n \log^* n)$ time shared by all invocations of the lemma.*

Proof. Let $T' = T[x..y]$, and let $\{i_1, i_2, \dots, i_b\}$ be a Δ -cover of $\{1, \dots, m\}$, which implies $b = \Theta(m / \sqrt{\Delta})$. We obtain a sparse suffix tree of the suffixes $T'[i_1..m], \dots, T'[i_b..m]$, which takes $\mathcal{O}(b \sigma \log b \log m) \subseteq \mathcal{O}(m \sigma \log^2 m / \sqrt{\Delta})$ time according to Lemma 5.2, plus $\mathcal{O}(n \log^* n)$ time shared by all invocations of the lemma. Now we compute a Δ -approximate LZ factorisation of T' from the sparse suffix tree in $\mathcal{O}(b)$ time.

In the following proof, we use i_1, i_2, \dots, i_b interchangeably to denote both the difference cover positions, as well as their corresponding leaves in the sparse suffix tree. Assume that the order of difference cover positions is $i_1 < i_2 < \dots < i_b$. First, we determine for each $i_k > i_1$, the position $\text{src}(i_k) = i_h$ and the length $\text{len}(i_k) = \text{LCE}(i_h, i_k)$, where $i_h \in \{i_1, \dots, i_{k-1}\}$ is a position that maximizes $\text{LCE}(i_h, i_k)$. This is similar to what was done in [34] for the LZ77 factorisation. We start by assigning labels from $\{1, \dots, b\}$ to the nodes of the sparse suffix tree. A node has label k if and only if i_k is its smallest descendant leaf. We assign the labels as follows. Initially, all nodes are unlabelled. We assign label 1 to each node on the path from i_1 to the root. Then, we process the remaining leaves i_2, \dots, i_b in increasing order. For each i_k , we follow the path from i_k to the root. We assign label k to each unlabelled node that we encounter. As soon as we reach a node that has already been labelled, say, with label h and string-depth ℓ , we are done processing leaf i_k . It should be easy to see that i_h is also exactly the desired index that maximizes $\text{LCE}(i_h, i_k)$, and it holds $\text{LCE}(i_h, i_k) = \ell$. Thus, we have found $\text{src}(i_k) = i_h$ and $\text{len}(i_k) = \ell$. The total time needed is linear in the number of sparse suffix tree nodes, which is $\mathcal{O}(b)$.

Finally, we obtain a Δ -approximate LZ factorisation using src and len . The previously computed values can

be interpreted as follows: i_k could become the starting position of a length- $\text{len}(i_k)$ tail (with previous occurrence at position $\text{src}(i_k)$). For the Δ -approximate LZ factorisation, we will create the factors greedily in a left-to-right manner. Assume that we already factorised $T'[1..s-1]$, then the next phrase starts at position s , and thus the next tail starts within $T'[s..s+\Delta)$ (as a reminder, the head is by definition shorter than Δ). Let $S = \{i_1, i_2, \dots, i_b\} \cap \{s, \dots, s+\Delta-1\}$. If there is no $i_k \in S$ with $i_k + \text{len}(i_k) > s + \Delta - 1$, then the next phrase is simply $T'[s..\min(|T|, s+\Delta-1))$ with empty tail. Otherwise, the next phrase has (possibly empty) head $T'[s..i_k)$ and tail $T'[i_k..i_k + \text{len}(i_k))$ (with previous occurrence $\text{src}(i_k)$), where i_k is chosen from S such that it maximizes $i_k + \text{len}(i_k)$. Creating the phrase in this way clearly takes $\mathcal{O}(|S|)$ time. Since the next phrase starts at least at position $s + \Delta - 1$, none of the positions from $S \setminus \{s + \Delta - 1\}$ will ever be considered as starting positions of other tails. Thus, every i_k is considered during the creation of at most two phrases, and the total time needed to create all phrases is $\mathcal{O}(b)$.

It remains to be shown that the computed factorisation is indeed a Δ -approximate LZ factorisation, i.e., if we output a phrase $T'[s..e]$, then the unique (non-approximate) LZ phrase $T'[s..e']$ starting at position s satisfies $e' - 1 \leq e$. First, note that for the created approximate phrases (except possibly the last phrase of T) it holds $s + \Delta - 2 \leq e$. Assume $e' < s + \Delta$, then clearly $e' - 1 \leq e$. Thus, we only have to consider $e' > s + \Delta - 1$. Since $T'[s..e']$ is an LZ phrase, there is some $s' < s$ such that $\text{LCE}(s', s) = e' - s$. Let h be the constant-time computable function that defines the Δ -cover, and let $i_{k'} = s' + h(s', s)$ and $i_k = s + h(s', s)$. Note that $i_{k'} \in \{i_1, i_2, \dots, i_{k-1}\}$ and $i_k \in \{i_1, i_2, \dots, i_b\} \cap \{s, \dots, s + \Delta - 1\}$. Therefore, it holds $\text{len}(i_k) \geq \text{LCE}(i_{k'}, i_k) = \text{LCE}(s', s) - h(s', s) = (e' - s) - (i_k - s) = e' - i_k$. While computing the Δ -approximate phrase $T'[s..e]$, we considered i_k as the starting positions of the tail, which implies $e \geq i_k + \text{len}(i_k) - 1 \geq e' - 1$. \square

LEMMA 5.4. *There is an algorithm that, given any parameter $\Delta \in [1, m]$, estimate $\tilde{\sigma}$ and fragment $T[x..y]$ of length m , takes $\mathcal{O}(m\tilde{\sigma} \log^2 m / \sqrt{\Delta})$ time plus $\mathcal{O}(n \log^* n)$ time shared by all invocations of the lemma, and either computes a Δ -approximate LZ factorisation of $T[x..y]$ or determines that $\sigma > \tilde{\sigma}$.*

Proof. We simply use Lemma 5.3 to compute the factorisation. In the first step, we have to construct the sparse suffix tree using the algorithm from Lemma 5.2. While this algorithm takes $\mathcal{O}(m\sigma \log^2 m / \sqrt{\Delta})$ time, it should be easy to see that a more accurate time bound is $\mathcal{O}(md \log^2 m / \sqrt{\Delta})$, where d is the maximum degree of any node in the suffix tree. If during construction the maximum degree of a node becomes $\tilde{\sigma} + 1$, we immediately stop and return that $\sigma > \tilde{\sigma}$. Otherwise, we finish the construction in the desired time. \square

Now we can describe the algorithm that detects squares in $\mathcal{O}(n \lg \sigma + n \log^* n)$ time and $\mathcal{O}(n \lg \sigma)$ comparisons. We simply use the algorithm from Section 4, but use Lemma 5.4 instead of Lemma 4.6. Next, we analyse the time needed apart from the $\mathcal{O}(n \log^* n)$ time shared by all invocations of Lemma 5.4. Throughout the t^{th} phase, we use $\mathcal{O}(n \cdot \tilde{\sigma} \cdot \log^2(\sigma_t) / \sqrt{\Delta}) = \mathcal{O}(n \cdot (\sigma_t)^{1/4} / \log(\sigma_t) \cdot \log^2(\sigma_t) / \sqrt{\sigma_t}) = \mathcal{O}(n \log(\sigma_t) / (\sigma_t)^{1/4})$ comparisons to construct all the Δ -approximate factorisations. As before, if at any time we discover that $\tilde{\sigma} > (\sigma_t)^{1/4} / \log(\sigma_t)$, then we use Theorem 2.1 to finish the computation in $\mathcal{O}(n \lg \sigma_t) = \mathcal{O}(n \lg \sigma)$ time. Until then (or until we finished all $\lceil \log \log n \rceil$ phases), we use $\mathcal{O}(\sum_{t=0}^{t'} n \log(\sigma_t) / (\sigma_t)^{1/4})$ time, and by Corollary 4.1 this is $\mathcal{O}(n)$. For detecting squares, we still use Lemma 4.5, which as explained in Section 4 takes $\mathcal{O}(n)$ time and comparisons in total, plus additional $\mathcal{O}(Z)$ time, where Z is the number of approximate LZ factors considered during all invocations of the lemma. Since we apply the lemma to each approximate LZ factorisation exactly once, and by construction each factor has size at least Δ thus $Z = \mathcal{O}(n/\Delta)$.

The last thing that remains to be shown is how to implement the bookkeeping of blocks, i.e., in each phase we have to efficiently deactivate block pairs as described at the end of Section 4. We maintain the block pairs in $\lceil \log \log n \rceil$ bitvectors of total length $\mathcal{O}(n)$, where a set bit means that a block pair has been deactivated (recall that there are $\mathcal{O}(n)$ pairs in total). Bitvector t contains at position j the bit corresponding to block pair $B_j B_{j+1} = T[i..i + 2(\sigma_t)^2]$ with $i = (\sigma_t)^2 \cdot (j - 1)$. Note that translating between i and j takes constant time. For each sufficiently long tail in phase t , we simply iterate over the relevant block pairs in phase $t + 2$ and deactivate them, i.e., we set the corresponding bit. This takes time linear in the number of deactivated blocks. Since there are $\mathcal{O}(n)$ block pairs, and each block pair gets deactivated at most a constant number of times, the total cost for this bookkeeping is $\mathcal{O}(n)$.

The number of comparisons is dominated by the $\mathcal{O}(n \lg \sigma)$ comparisons used when finishing the computation with Theorem 2.1. The only other comparisons are performed by Lemma 4.5, which we already bounded by $\mathcal{O}(n)$,

and by LCE queries via Corollary 5.1. Since we ask $\mathcal{O}(n)$ such queries in total, the number of comparisons is also $\mathcal{O}(n)$. We have shown:

LEMMA 5.5. *The square detection algorithm from Section 4 can be implemented in $\mathcal{O}(n \lg \sigma + n \log^* n)$ time and $\mathcal{O}(n \log \sigma)$ comparisons.*

5.2 Final Improvement For our final improvement we need the notion of a string synchronising set introduced by Kempa and Kociumaka [50].

DEFINITION 5.1. *Consider a string $T[1..n]$ and a parameter $\tau \leq \frac{n}{2}$. A set $S \subseteq [1, n - 2\tau + 1]$ is a τ -synchronising set of $T[1..n]$ when it satisfies the following conditions:*

1. *if $T[i..i + 2\tau] = T[j..j + 2\tau]$, then $i \in S$ holds if and only if $j \in S$ (for $i, j \in [1, n - 2\tau + 1]$),*
2. *$S \cap [i, i + \tau] = \emptyset$ if and only if the period of $T[i..i + 3\tau - 2]$ is at most $\frac{\tau}{3}$ (for $i \in [1, n - 3\tau + 2]$).*

The first condition can be interpreted as saying that whether $i \in S$ depends only on the string $T[i..i + 2\tau]$. That is, a τ -synchronising set is obtained by selecting a set \mathcal{S} of length- 2τ strings, and then including in S the starting positions of all strings $s \in \mathcal{S}$ in $T[1..n]$. The second condition will not be relevant in our application, as we will always ensure that there are no periodic fragments.

THEOREM 5.1. (PROPOSITION 8.10 IN [50]) *Given a string $T[1..n]$ over linearly-sortable alphabet and a parameter $\tau \leq \frac{n}{2}$, we can construct a τ -synchronising set of size $\mathcal{O}(n/\tau)$ in $\mathcal{O}(n)$ time.*

The goal of this section is to show how to, given a parameter Δ and estimate $\tilde{\sigma}$ of the alphabet size, find a $(\Delta + 3\tau)$ -approximate LZ factorisation of any fragment $T[x..y]$ in $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$ time, where $\tau = \lfloor \sqrt{\Delta} \rfloor$ and $m = |T[x..y]|$ (with $m = \Theta(\Delta)$, as otherwise we are not required to detect anything). As in the previous section, the algorithm might detect that the size of the alphabet is larger than $\tilde{\sigma}$, and in such case we revert to the divide-and-conquer algorithm. Because τ is sufficiently small, by adjusting the constants a $(\Delta + 3\tau)$ -approximate LZ factorisation can be used instead of a Δ -approximate LZ factorisation in the approach described in the previous section, and this results in the promised $\mathcal{O}(n \log \sigma)$ overall time complexity.

Initially, we only consider some fragments of $T[x..y]$. We say that $T[i..i + 3\lfloor \sqrt{\Delta} \rfloor]$ is a dense fragment if $i \equiv 0 \pmod{\Delta}$. We start by remapping the characters in all dense fragments that intersect $T[x..y]$ to a linearly-sortable alphabet. This can be done in $\mathcal{O}(\tilde{\sigma})$ time for each position by maintaining a list of the already seen distinct characters. For each position in a dense fragment, we iterate over the characters in the list, and possibly append a new character to the list if it is not already present. As soon as the size of the list exceeds $\tilde{\sigma}$, we terminate the procedure. Otherwise, we replace each character by its position in the list. Overall, there are $\mathcal{O}(m/\sqrt{\Delta})$ positions in the dense fragments of $T[x..y]$, and thus the remapping takes $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$ time.

Next, we construct generalised suffix trees [43] of all dense fragments and their reversals. Again, because we now work with a linearly-sortable alphabet this takes only $\mathcal{O}(m/\sqrt{\Delta})$ time [31]. We consider fragments of the form $T[i \cdot 2\tau..(i + 1) \cdot 2\tau]$ having non-empty intersection with $T[x..y]$. We call such fragments chunks. We note that there are $\mathcal{O}(m/\sqrt{\Delta})$ chunks, and their total length is $\mathcal{O}(m)$. For each chunk, we find its longest prefix $T[i \cdot 2\tau..i \cdot 2\tau + \ell_i]$ and longest suffix $T[(i + 1) \cdot 2\tau - r_i..(i + 1) \cdot 2\tau]$ that occur in one of the dense fragments. This can be done efficiently by following the heavy path decomposition of the generalised suffix tree of all dense fragments and their reversals, respectively. On each current heavy path, we just naively match the characters as long as possible, and in case of a mismatch spend $\mathcal{O}(\tilde{\sigma})$ time to descend to the appropriate subtree. The total time per chunk is thus $\mathcal{O}(\tilde{\sigma} \log m)$ plus the lengths of the longest prefix and suffix. The former sums up to $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$, and we will later show that the latter can be amortised by deactivating blocks on the lower levels.

The situation so far is that we have remapped the alphabet of all dense fragments to linearly-sortable, and for every chunk we know its longest prefix and suffix that occur in one of the dense fragments. We concatenate (i) all dense fragments, and (ii) all fragments of the form $T[i \cdot 2\tau - r_{i-1}..i \cdot 2\tau + \ell_i]$ (intersected with $T[x..y]$) while adding distinct separators in between to form a new string T' . We stress that, because we have remapped the alphabet of all dense fragments, and the found longest prefix and suffix of each chunk also occur in some dense fragment, T' is over linearly-sortable alphabet.

The next step is to test square-freeness of T' . As we have remapped its alphabet to linearly-sortable, this takes only $\mathcal{O}(m/\sqrt{\Delta} + \sum_i(\ell_i + r_i))$ time by Theorem 2.2. We will later show that $\mathcal{O}(\sum_i(\ell_i + r_i))$ can be amortised by deactivated blocks on the lower levels. If there is a square in T' , due to the separators being distinct it must fully belong to a fragment of $T[x..y]$, thus we can terminate the whole algorithm. Thus, from now on, we assume that T' is square-free. In particular, this implies that there are no fragments $T'[i..i + 3\tau - 2]$ with period at most $\frac{\tau}{3}$ there. We use Theorem 5.1 to build a τ -synchronizing set S of T' , for $\tau = \sqrt{\Delta} = \sqrt{\frac{\sigma_i}{8}}$. We observe that, for every $i \in S$ such that $T'[i..i + 2\tau)$ does not contain any separator symbol, we can find the corresponding position i' in $T[x..y]$ such that $T'[i..i + 2\tau) = T[i'..i' + 2\tau)$. We create the *truncated* τ -synchronising set S' of $T[x..y]$ by including all such i' for which $T[i'..i' + 2\tau)$ has an occurrence in a dense fragment (we note that the same i' might be included in S' due to more than one i , as the fragments concatenated to obtain T' are not disjoint in $T[x..y]$). To implement this efficiently, we first precompute for every position in T' its corresponding position in $T[x..y]$ and the position of the nearest separator. This can be done in $\mathcal{O}(|T'|)$ time when constructing T' , and allows us to filter out positions i such that $T'[i..i + 2\tau)$ does not contain any separator symbol. Next, we build the suffix tree ST of T' in $\mathcal{O}(|T'|)$ time. For each node of ST corresponding to a fragment of length at least 2τ , we precompute its unique ancestor (implicit or explicit) corresponding to a fragment of length 2τ . Additionally, for each explicit node of ST we precompute whether its corresponding string has an occurrence in a dense fragment of T . This can be done in a bottom-up traversal in $\mathcal{O}(|T'|)$ time, and allows us to complete the filtering.

Although we built S' by considering T' instead of $T[x..y]$, it still has some properties of a string synchronising set of $T[x..y]$.

LEMMA 5.6. *The set S' satisfies the following conditions:*

1. For any $i, j \in [x, y - 2\tau + 1]$, if $T[i..i + 2\tau) = T[j..j + 2\tau)$ then $i \in S' \iff j \in S'$.
2. For any $i \in [x, y - \Delta - \tau + 1]$, it holds $S' \cap [i, i + \Delta + \tau) \neq \emptyset$.
3. It holds $|S'| = \mathcal{O}(m/\tau)$.

Proof. First, by the construction of the truncated τ -synchronising set, it holds $i \notin S'$ for all the $T[i..i + 2\tau)$ that do not have any occurrence in a dense fragment. Thus, we only have to consider the case where $T[i..i + 2\tau)$ has an occurrence in a dense fragment. Note that the chunk boundary $i' \cdot 2\tau$ with $i' = \lceil \frac{i}{2\tau} \rceil$ is contained in $T[i..i + 2\tau)$. We included $T[i' \cdot 2\tau - r_{i-1}..i' \cdot 2\tau + \ell_i)$ in T' , and since $T[i..i + 2\tau)$ has an occurrence in a dense fragment it holds $i' \cdot 2\tau - r_{i-1} \leq i$ and $i + 2\tau \leq i' \cdot 2\tau + \ell_i$. Thus, $T[i..i + 2\tau)$ has a corresponding fragment in T' . If there is another fragment $T[j..j + 2\tau) = T[i..i + 2\tau)$, then $T[j..j + 2\tau)$ has a corresponding fragment in T' by the same reasoning. This yields the claimed condition.

Second, let $i' = \lceil \frac{i}{\Delta} \rceil \cdot \Delta$ and consider the dense fragment $T[i'..i' + 3\tau)$. It is included in T' , say as $T'[i''..i'' + 3\tau)$, and by the assumption on T' being square-free and property 2 of a τ -synchronising set there will be a position $j \in S \cap [i'', i'' + \tau)$. Then, j has its corresponding position j' in $S' \cap [i', i' + \tau)$ that is not filtered out when forming S' (because $T[j'..j' + 2\tau)$ occurs inside the dense fragment $T[i'..i' + 3\tau)$). Finally, note that $[i', i' + \tau) \subset [i, i + \Delta + \tau)$ because of $i' \in [i, i + \Delta)$, and thus $j' \in S' \cap [i, i + \Delta + \tau)$.

Lastly, the length of T' is $\mathcal{O}(m)$. Since we used Theorem 5.1 to construct S , it holds $|S| = \mathcal{O}(\frac{m}{\tau})$. Each position in S contributes at most one position to S' , and thus $|S'| \leq |S| = \mathcal{O}(\frac{m}{\tau})$. \square

We now define a partial parsing of $T[x..y]$ with a $\$$ character appended at the end based on S' (in fact, the following would work for any set that satisfies the conditions from Lemma 5.6). Let $i_1 < i_2 < \dots < i_k$ be all the positions in S' , that is, $(i_j, i_{j+1}) \cap S' = \emptyset$ for every $j = 1, 2, \dots, k-1$. For every $j = 1, 2, \dots, k-1$, we create the phrase $T[i_j..i_{j+1} + 2\tau)$. We add a last phrase $T[i_k..y]\$$. We stress that consecutive phrases overlap by 2τ characters, and each phrase begins with a length- 2τ fragment starting at a position in S' . This, together with property 1 of Lemma 5.6, implies the following useful property.

OBSERVATION 5.2. *The set of distinct phrases is prefix-free.*

We would like to construct the compacted trie $\mathcal{T}_{\text{phrase}}$ of all such phrases, so that (in particular) we identify identical phrases. We first notice that each phrase begins with a fragment $T[i_j..i_{j+2\tau})$ that has its corresponding occurrence in T' , and using the suffix tree of T' we can partition all phrases according to their prefixes of length

2τ . This gives us the top part of $\mathcal{T}_{\text{phrase}}$ containing all such prefixes in $\mathcal{O}(m/\sqrt{\Delta})$ time, and for each phrase we can assume that we know the node of $\mathcal{T}_{\text{phrase}}$ corresponding to its length- 2τ prefix. Next, we iterate over the phrases, and insert them into $\mathcal{T}_{\text{phrase}}$ using a dynamic heavy path decomposition as in Section 5.1. For each phrase $T[i_j..i_{j+1} + 2\tau]$, we begin the insertion at the already known node corresponding to $T[i_j..i_j + 2\tau]$, and continue the insertion by following the heavy paths. On each heavy path, we naively match the characters as long as possible. The total time spent on maintaining the dynamic trie and its heavy path decomposition is $\mathcal{O}(m \log m/\sqrt{\Delta})$, while the time to insert a single phrase $T[i_j..i_{j+1} + 2\tau]$ is $\mathcal{O}(\log m)$ plus the length of the longest prefix of $T[i_j + 2\tau..i_{j+1} + 2\tau]$ equal to a prefix of $T[i_{j'} + 2\tau..i_{j'+1} + 2\tau]$, for some $j' < j$. The former sums up to another $\mathcal{O}(m \log m/\sqrt{\Delta})$, and we will later show that the latter can be amortised by deactivating blocks on the lower levels.

$\mathcal{T}_{\text{phrase}}$ allows us to form metacharacters corresponding to the phrases, and transform $T[x..y]$ into a string T_{parse} of length $\mathcal{O}(m/\sqrt{\Delta})$ consisting of these metacharacters. We build a suffix tree $\mathcal{S}_{\text{parse}}$ over this string over linearly-sortable metacharacters in $\mathcal{O}(m/\sqrt{\Delta})$ time. Next, we convert it into the sparse suffix tree $\mathcal{S}'_{\text{parse}}$ of all suffixes $T[i..j]$ as follows. Consider an explicit node $u \in \mathcal{S}_{\text{parse}}$ with children v_1, v_2, \dots, v_d , $d \geq 2$. We replace u with the subtree \mathcal{T}_u of T_{parse} induced by the leaves corresponding to the first metacharacters on the edges (u, v_i) , for $i = 1, 2, \dots, d$, and connect every v_i to the appropriate leaf of \mathcal{T}_u . This can be implemented in $\mathcal{O}(d)$ time, assuming constant-time lower common ancestor queries on $\mathcal{T}_{\text{phrase}}$ [10] and processing the leaves from left to right with a stack, similarly as in the Cartesian tree construction algorithm [76]. Overall, this sums up to $\mathcal{O}(m/\sqrt{\Delta})$. By following an argument from the proof of Lemma 5.3, $\mathcal{S}'_{\text{parse}}$ allows us to determine, for every suffix $T[i_j..y]$, its longest prefix equal to a prefix of some $T[i'..y]$ with $i' < i_j$, as long as its length is at least 2τ . Indeed, in such case we must have $i' \in S'$ by property 1 of Lemma 5.6, so in fact $i' = i_{j'}$ and it is enough to maximise the length of the common prefix with all earlier positions in S' , which can be done using $\mathcal{S}'_{\text{parse}}$. Thus, we either know that the length of this longest prefix is less than 2τ , or know its exact value (and the corresponding position $i' \in S'$).

LEMMA 5.7. *For any parameter $\Delta \in [1, m]$ and estimate $\tilde{\sigma}$ of the alphabet size, a $(\Delta + 3\tau)$ -approximate LZ factorisation of any fragment $T[x..y]$ can be computed in $\mathcal{O}(m/\sqrt{\Delta})$ time with $m = |T[x..y]|$ (assuming the preprocessing described earlier in this section).*

Proof. Let $e \in [x, y]$ and suppose we have already constructed the factorisation of $T[x..e - 1]$ and are now trying to construct the next phrase. Let e' be the next multiple of Δ , we have that $e' - e < \Delta$ and $T[e'..e' + 3\lfloor\sqrt{\Delta}\rfloor]$ is a dense substring. Thus, by property 2 of Lemma 5.6 we have $S' \cap [e', e' + \tau] \neq \emptyset$. Let $e'' \in S' \cap [e', e' + \tau]$. We first check if the longest common prefix between $T[e''..y]$ and any suffix starting at an earlier position is of length at least 2τ . If not, we simply set the head of the new phrase to be $T[e..e'' + 2\tau]$ and the tail to be empty. Otherwise, we know the length ℓ of this longest prefix. We set the head of the new phrase to be $T[e..e'']$ and the tail to be $T[e''..e'' + \ell]$. This takes constant time per phrase, and each phrase is of length at least τ , giving the claimed overall time complexity. It remains to argue correctness of every step.

Let $T[e..s]$ be the longest LZ phrase starting at position e , to show that we obtain a valid $(\Delta + 3\tau)$ -approximate phrase it suffices to show that $s \leq e'' + \max(2\tau, \ell)$. Let the previous occurrence of $T[e..s]$ be at position $p < e$. If $s - e'' < 2\tau$ then there is nothing to prove. Otherwise, $T[e''..s]$ is a string of length at least 2τ that also occurs starting earlier at position $p + e'' - e < e''$. Thus, we will correctly determine that $\ell \geq 2\tau$, and find a previous occurrence of the string maximising the value of ℓ . In particular, we will have $\ell \geq s - e''$ as required. \square

To achieve the bound of Theorem 1.3, we now proceed as in Section 4.4, except that instead of Lemma 5.4 we use Lemma 5.7. This takes $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$ for every $T[x..y]$ with $m = |T[x..y]|$, however we need to additionally account for the additional comparisons used when computing the longest common prefixes and suffixes of every chunk, and inserting $T[i_j + 2\tau..i_{j+1} + 2\tau]$ into $\mathcal{T}_{\text{phrase}}$. We observe that in both cases we can deactivate any block fully contained in the corresponding fragment of the input string. For $T[i \cdot 2\tau..i \cdot 2\tau + \ell_i]$ and $T[(i + 1) \cdot 2\tau - r_i..(i + 1) \cdot 2\tau]$ this follows because we test square-freeness of these fragments anyway in the current phase. For the longest prefix of $T[i_j + 2\tau..i_{j+1} + 2\tau]$ equal to $T[i_{j'} + 2\tau..i_{j'+1} + 2\tau]$, for some $j' < j$, this follows because such fragment cannot contain the leftmost occurrence of a square. There are $\mathcal{O}(m/\sqrt{\Delta})$ chunks and phrases. We can afford to spend $\mathcal{O}(\sqrt{\Delta})$ comparisons for each phrase because then the total number of comparisons sums to $\mathcal{O}(n)$ over all phrases. If a chunk or phrase contributes $x = \Omega(\sqrt[4]{\Delta})$ comparisons, then we explicitly deactivate the block pairs in phase $t + 3$ that are entirely contained in the corresponding fragment. Block

pairs in phase $t + 3$ are of length $\mathcal{O}(\sqrt[4]{\Delta})$, and thus we deactivate $\mathcal{O}(x)$ positions. Therefore, the comparisons performed when inserting chunks and phrases into their respective tries in all phases sums to $\mathcal{O}(n)$.

References

- [1] J. Allouche and J. O. Shallit. The ubiquitous Prouhet-Thue-Morse Sequence. In C. Ding, T. Helleseht, and H. Niederreiter, editors, *Sequences and their Applications - Proceedings of SETA 1998, Singapore, December 14-17, 1998*, Discrete Mathematics and Theoretical Computer Science, pages 1–16. Springer, 1998.
- [2] A. Amir, E. Eisenberg, and A. Levy. Approximate periodicity. In O. Cheong, K. Chwa, and K. Park, editors, *Algorithms and Computation - 21st International Symposium, ISAAC 2010, Jeju Island, Korea, December 15-17, 2010, Proceedings, Part I*, volume 6506 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2010.
- [3] A. Amir, E. Eisenberg, and A. Levy. Approximate periodicity. *Inf. Comput.*, 241:215–226, 2015.
- [4] A. Amir and A. Levy. Approximate period detection and correction. In L. Calderón-Benavides, C. N. González-Caro, E. Chávez, and N. Ziviani, editors, *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, volume 7608 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2012.
- [5] A. Apostolico and D. Breslauer. An optimal $\mathcal{O}(\log \log n)$ -time parallel algorithm for detecting all squares in a string. *SIAM J. Comput.*, 25(6):1318–1331, 1996.
- [6] A. Apostolico and R. Giancarlo. Periodicity and repetitions in parameterized strings. *Discret. Appl. Math.*, 156(9):1389–1398, 2008.
- [7] A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theor. Comput. Sci.*, 22:297–315, 1983.
- [8] H. Bannai, T. I. S. Inenaga, Y. Nakashima, M. Takeda, and K. Tsuruta. The "runs" theorem. *SIAM J. Comput.*, 46(5):1501–1514, 2017.
- [9] D. R. Bean, A. Ehrenfeucht, and G. F. McNulty. Avoidable patterns in strings of symbols. *Pacific Journal of Mathematics*, 85(2):261 – 294, 1979.
- [10] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.
- [11] J. Berstel and L. Boasson. Partial words and a theorem of Fine and Wilf. *Theor. Comput. Sci.*, 218(1):135–141, 1999.
- [12] F. Blanchet-Sadri, D. Bal, and G. Sisodia. Graph connectivity, partial words, and a theorem of Fine and Wilf. *Inf. Comput.*, 206(5):676–693, 2008.
- [13] F. Blanchet-Sadri and R. A. Hegstrom. Partial words and a theorem of Fine and Wilf revisited. *Theor. Comput. Sci.*, 270(1-2):401–419, 2002.
- [14] F. Blanchet-Sadri, S. Simmons, A. Tebbe, and A. Veprauskas. Abelian periods, partial words, and an extension of a theorem of Fine and Wilf. *RAIRO Theor. Informatics Appl.*, 47(3):215–234, 2013.
- [15] D. Breslauer. *Efficient String Algorithmics*. PhD thesis, Columbia University, 1992.
- [16] S. Brlek and S. Li. On the number of squares in a finite word. *CoRR*, abs/2204.10204, 2022.
- [17] M. G. Castelli, F. Mignosi, and A. Restivo. Fine and Wilf's theorem for three periods and a generalization of Sturmian words. *Theor. Comput. Sci.*, 218(1):83–94, 1999.
- [18] R. Cole and R. Hariharan. Tighter upper bounds on the exact complexity of string matching. *SIAM J. Comput.*, 26(3):803–856, 1997.
- [19] R. Cole, R. Hariharan, M. Paterson, and U. Zwick. Tighter lower bounds on the exact complexity of string matching. *SIAM J. Comput.*, 24(1):30–45, 1995.
- [20] S. Constantinescu and L. Ilie. Fine and Wilf's theorem for abelian periods. *Bulletin of the EATCS*, 89:167–170, 01 2006.
- [21] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley, 2nd edition, 2006.
- [22] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.
- [23] M. Crochemore. Transducers and repetitions. *Theor. Comput. Sci.*, 45(1):63–86, 1986.
- [24] M. Crochemore and L. Ilie. Maximal repetitions in strings. *J. Comput. Syst. Sci.*, 74(5):796–807, 2008.
- [25] M. Crochemore, L. Ilie, and L. Tinta. The "runs" conjecture. *Theor. Comput. Sci.*, 412(27):2931–2941, 2011.
- [26] M. Crochemore, C. S. Iliopoulos, T. Kociumaka, R. Kundu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Walen. Near-optimal computation of runs over general alphabet via non-crossing LCE queries. In *SPIRE*, volume 9954 of *Lecture Notes in Computer Science*, pages 22–34, 2016.
- [27] J. D. Currie. Pattern avoidance: themes and variations. *Theor. Comput. Sci.*, 339(1):7–18, 2005.
- [28] A. Deza, F. Franek, and A. Thierry. How many double squares can a string contain? *Discret. Appl. Math.*, 180:52–69, 2015.

- [29] J. Duval, T. Lecroq, and A. Lefebvre. Linear computation of unbordered conjugate on unordered alphabet. *Theor. Comput. Sci.*, 522:77–84, 2014.
- [30] J. Ellert and J. Fischer. Linear Time Runs Over General Ordered Alphabets. In N. Bansal, E. Merelli, and J. Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, volume 198 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 63:1–63:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [31] M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143. IEEE Computer Society, 1997.
- [32] N. J. Fine and H. S. Wilf. Uniqueness theorems for periodic functions. In *Proceedings of the American Mathematical Society*, volume 16, pages 109–114, 1965.
- [33] J. Fischer, S. Holub, T. I., and M. Lewenstein. Beyond the runs theorem. In C. S. Iliopoulos, S. J. Puglisi, and E. Yilmaz, editors, *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, volume 9309 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2015.
- [34] J. Fischer, T. I., D. Köppl, and K. Sadakane. Lempel-Ziv factorization powered by space efficient suffix trees. *Algorithmica*, 80(7):2048–2081, 2018.
- [35] A. S. Fraenkel and J. Simpson. How many squares can a string contain? *J. Comb. Theory, Ser. A*, 82(1):112–120, 1998.
- [36] F. Franek and Q. Yang. An asymptotic lower bound for the maximal number of runs in a string. *Int. J. Found. Comput. Sci.*, 19(1):195–203, 2008.
- [37] H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In D. S. Johnson, editor, *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, pages 434–443. SIAM, 1990.
- [38] Z. Galil and J. I. Seiferas. Time-space-optimal string matching. *J. Comput. Syst. Sci.*, 26(3):280–294, 1983.
- [39] P. Gawrychowski, T. Kociumaka, W. Rytter, and T. Walen. Faster longest common extension queries in strings over general alphabets. In R. Grossi and M. Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, volume 54 of *LIPIcs*, pages 5:1–5:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.
- [40] M. Giraud. Not so many runs in strings. In C. Martín-Vide, F. Otto, and H. Fernau, editors, *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers*, volume 5196 of *Lecture Notes in Computer Science*, pages 232–239. Springer, 2008.
- [41] M. Giraud. Asymptotic behavior of the numbers of runs and microruns. *Inf. Comput.*, 207(11):1221–1228, 2009.
- [42] G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A. M. Shur, and T. Walen. String periods in the order-preserving model. *Inf. Comput.*, 270, 2020.
- [43] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [44] Y. Han and M. Thorup. Integer sorting in $\mathcal{O}(n\sqrt{\log \log n})$ expected time and linear space. In *FOCS*, pages 135–144. IEEE Computer Society, 2002.
- [45] S. Holub. Prefix frequency of lost positions. *Theor. Comput. Sci.*, 684:43–52, 2017.
- [46] J. Hong and G. Chen. Efficient on-line repetition detection. *Theor. Comput. Sci.*, 407(1-3):554–563, 2008.
- [47] L. A. Idiatulina and A. M. Shur. Periodic partial words and random bipartite graphs. *Fundam. Informaticae*, 132(1):15–31, 2014.
- [48] L. Ilie. A note on the number of squares in a word. *Theor. Comput. Sci.*, 380(3):373–376, 2007.
- [49] J. Justin. On a paper by Castelli, Mignosi, Restivo. *RAIRO Theor. Informatics Appl.*, 34(5):373–377, 2000.
- [50] D. Kempa and T. Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In M. Charikar and E. Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 756–767. ACM, 2019.
- [51] T. Kociumaka, J. Radoszewski, W. Rytter, and T. Walen. A periodicity lemma for partial words. *Inf. Comput.*, 283:104677, 2022.
- [52] R. M. Kolpakov, G. Bana, and G. Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic Acids Res.*, 31(13):3672–3678, 2003.
- [53] R. M. Kolpakov and G. Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 596–604. IEEE Computer Society, 1999.
- [54] S. R. Kosaraju. Computation of squares in a string (preliminary version). In M. Crochemore and D. Gusfield, editors, *Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994, Proceedings*, volume 807 of *Lecture Notes in Computer Science*, pages 146–150. Springer, 1994.
- [55] D. Kosolobov. Online square detection. *CoRR*, abs/1411.2022, 2014.

- [56] D. Kosolobov. Lempel-Ziv Factorization May Be Harder Than Computing All Runs. In E. W. Mayr and N. Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 582–593, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [57] D. Kosolobov. Online detection of repetitions with backtracking. In F. Cicalese, E. Porat, and U. Vaccaro, editors, *Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, volume 9133 of *Lecture Notes in Computer Science*, pages 295–306. Springer, 2015.
- [58] D. Kosolobov. Computing runs on a general alphabet. *Inf. Process. Lett.*, 116(3):241–244, 2016.
- [59] D. Kosolobov. Finding the leftmost critical factorization on unordered alphabet. *Theor. Comput. Sci.*, 636:56–65, 2016.
- [60] S. Li, J. Pachocki, and J. Radoszewski. A note on the maximum number of k -powers in a finite word. *CoRR*, abs/2205.10156, 2022.
- [61] M. Maekawa. A \sqrt{N} algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.
- [62] M. G. Main and R. J. Lorentz. An $\mathcal{O}(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, 5(3):422–432, 1984.
- [63] W. Matsubara, K. Kusano, H. Bannai, and A. Shinohara. A series of run-rich strings. In A. Dediu, A. Ionescu, and C. Martín-Vide, editors, *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, volume 5457 of *Lecture Notes in Computer Science*, pages 578–587. Springer, 2009.
- [64] W. Matsubara, K. Kusano, A. Ishino, H. Bannai, and A. Shinohara. New lower bounds for the maximum number of runs in a string. In J. Holub and J. Zdárek, editors, *Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008*, pages 140–145. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2008.
- [65] Y. Matsuoka, T. Aoki, S. Inenaga, H. Bannai, and M. Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theor. Comput. Sci.*, 656:225–233, 2016.
- [66] O. Merkurev and A. M. Shur. Searching runs in streams. In N. R. Brisaboa and S. J. Puglisi, editors, *String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019, Segovia, Spain, October 7-9, 2019, Proceedings*, volume 11811 of *Lecture Notes in Computer Science*, pages 203–220. Springer, 2019.
- [67] O. Merkurev and A. M. Shur. Computing the maximum exponent in a stream. *Algorithmica*, 84(3):742–756, 2022.
- [68] S. J. Puglisi, J. Simpson, and W. F. Smyth. How many runs can a string contain? *Theor. Comput. Sci.*, 401(1-3):165–171, 2008.
- [69] W. Rytter. The number of runs in a string: Improved analysis of the linear upper bound. In B. Durand and W. Thomas, editors, *STACS 2006, 23rd Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, volume 3884 of *Lecture Notes in Computer Science*, pages 184–195. Springer, 2006.
- [70] A. M. Shur and Y. V. Gamzova. Partial words and the interaction property of periods. *Izvestiya: Mathematics*, 68(2):405–428, apr 2004.
- [71] A. M. Shur and Y. V. Konovalova. On the periods of partial words. In J. Sgall, A. Pultr, and P. Kolman, editors, *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001 Mariánské Lázně, Czech Republic, August 27-31, 2001, Proceedings*, volume 2136 of *Lecture Notes in Computer Science*, pages 657–665. Springer, 2001.
- [72] J. Simpson. Modified Padovan words and the maximum number of runs in a word. *Australas. J Comb.*, 46:129–146, 2010.
- [73] A. Thierry. A proof that a word of length n has less than $1.5n$ distinct squares. *CoRR*, abs/2001.02996, 2020.
- [74] A. Thue. Über unendliche Zeichenreihen. *Norske Vid. Selsk. Skr., I Mat.-Nat. Kl., Christiania*, 7:1–22, 1906.
- [75] R. Tijdeman and L. Zamboni. Fine and Wilf words for any periods. *Indagationes Mathematicae*, 14(1):135–147, 2003.
- [76] J. Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.