Order-Preserving Squares in Strings

Paweł Gawrychowski
Institute of Computer Science, University of Wrocław, Poland

Samah Ghazawi
Department of Computer Science, University of Haifa, Israel

Gad M. Landau
Department of Computer Science, University of Haifa, Israel
Department of Computer Science and Engineering, NYU Tandon School of Engineering, New York University, Brooklyn, NY, USA

Abstract

An order-preserving square in a string is a fragment of the form uv where $u \neq v$ and u is order-isomorphic to v. We show that a string w of length n over an alphabet of size σ contains $\mathcal{O}(\sigma n)$ order-preserving squares that are distinct as words. This improves the upper bound of $\mathcal{O}(\sigma^2 n)$ by Kociumaka, Radoszewski, Rytter, and Waleń [TCS 2016]. Further, for every σ and n we exhibit a string with $\Omega(\sigma n)$ order-preserving squares that are distinct as words, thus establishing that our upper bound is asymptotically tight. Finally, we design an $\mathcal{O}(\sigma n)$ time algorithm that outputs all order-preserving squares that occur in a given string and are distinct as words. By our lower bound, this is optimal in the worst case.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases repetitions, distinct squares, order-isomorphism

Digital Object Identifier 10.4230/LIPIcs.CPM.2023.13

Funding *Samah Ghazawi*: partially supported by the Israel Science Foundation grant 1475/18, and Grant No. 2018141 from the United States-Israel Binational Science Foundation (BSF).
Gad M. Landau: partially supported by the Israel Science Foundation grant 1475/18, and Grant No. 2018141 from the United States-Israel Binational Science Foundation (BSF).

1 Introduction

A natural definition of repetitions in strings is that of squares, which are fragments of the form uu, where u is a string. The study of repetitions in strings goes back at least to the work of Thue from 1906 [28], who constructed an infinite square-free word over the ternary alphabet. Since then, multiple definitions of repetitions have been proposed and studied, with the basic question being focused on analyzing how many such repetitions a string of length n can contain. Of course, any even-length fragment of the string $\mathtt{a}^n$ is a square, therefore we would like to count distinct squares. Using a combinatorial result of Crochemore and Rytter [5], Fraenkel and Simpson [10] proved that a string of length n contains at most $2n$ distinct squares (also see a simpler proof by Ilie [17]). They also provided an infinite family of strings of length n with $n - o(n)$ distinct squares. For many years, it was conjectured that the right upper bound is actually n. Interestingly, a proof of the conjecture for the binary alphabet would imply it for any alphabet [24]. Very recently, after a series of improvements on the upper bound [7, 18, 23, 27], the conjecture has been finally resolved by Brlek and Li [1], who showed an upper bound of $n - \sigma + 1$, where σ is the size of the alphabet.

For many of the applications, it seems more appropriate to work with different definitions of equality, giving us different notions of squares. Three interesting examples are (1) Abelian squares [6, 8, 9, 16, 19–21, 26] (also called Jumbled squares) are of interest in natural language

processing applications and in other domains where the classifications strongly depend on feature sets distribution, as opposed to feature sequences distributions. (2) Parameterized squares [20] are considered in applications for finding identical sections of code. (3) Order-preserving squares [4, 13, 20] could be used in applications of stock price analysis and musical melody matching.

The combinatorial properties of the three types of squares were studied by Kociumaka et al. [20]. Given a string of length n over an alphabet of size σ, first the authors bounded the number of abelian squares that are distinct as words by $\Theta(n^2)$. Second, bounded the number of parameterized squares that are distinct as words by $\mathcal{O}((\sigma!)^2 n)$ and bounded the number of nonequivalent parameterized squares (see definition within) by $\mathcal{O}(\sigma! n)$. Third, the authors provided $\mathcal{O}(\sigma^2 n)$ bound for the number of order-preserving squares that are distinct as words.

From an algorithmic perspective, various algorithms were proposed for computing abelian squares and order-preserving squares in a string of length n. Cummings and Smyth [6] proposed an $\Theta(n^2)$ time algorithm for computing all substrings that consist of a concatenation of two or more abelian-equivalent substrings. Kociumaka et al. [21] proposed an algorithm for computing the longest, the shortest, and the number of all abelian squares in $\mathcal{O}(n^2/\log^2 n)$ time using linear space. Gourdel et al. [13] proved that all nonshiftable order-preserving squares (see definition within) can be computed in $\mathcal{O}(n \log n)$ time. Additionally, Crochemore et al. [4] proposed the *incomplete* order-preserving suffix tree (see details within), denoted by T, that enables order-preserving pattern matching queries in time proportional to the pattern length. The suffix tree T can be constructed in $\mathcal{O}(n \log\log n)$ expected time and $\mathcal{O}(n \log^2 \log n / \log\log\log n)$ worst-case time. Moreover, the authors proved that using T, all occurrences of order-preserving squares can be computed in $\mathcal{O}(n \log n + occ)$ time, where occ is the total number of occurrences of order-preserving squares. Note that, the number of all occurrences of order-preserving squares might be unreasonably high. In particular, every regular square is considered to be an order-preserving square, hence $\mathtt{a}^n$ contains $\Theta(n^2)$ occurrences of order-preserving squares. Henceforth, a more natural approach is to generate only order-preserving squares that are distinct as words.

Our results. In this paper, we focus on order-preserving squares. Same-length strings u and v over an ordered alphabet are order-isomorphic, denoted $u \approx v$, when the order between the characters at the corresponding positions is the same in u and v. For example, the strings $u = \mathtt{acb}$ and $v = \mathtt{azd}$ are order-isomorphic, assuming $\mathtt{a} < \mathtt{b} < \mathtt{c} < \mathtt{d} < \mathtt{z}$. In this paper, order-preserving squares are strings of the form uv, where $u \approx v$ and additionally $u \neq v$.

The main result of our paper is that the number of order-preserving squares in a string of length n over an alphabet of size σ is $\mathcal{O}(\sigma n)$. This improves the bound of $\mathcal{O}(\sigma^2 n)$ by Kociumaka et al. [20]. We stress that in our definition of an order-preserving square, we require that $u \neq v$, while Kociumaka et al. [20] counted fragments of the form uv, where $u \approx v$, that are distinct as words. We believe that our definition is more natural in the context of this paper. At the same time, by the result of Brlek and Li [1] a string of length n contains less than n fragments of the form uu that are distinct as words, thus our result implies that the number of fragments uv such that $u \approx v$ that are distinct as words is also $\mathcal{O}(\sigma n)$. We complement our upper bound by designing, for each σ, an infinite family of strings of length n over an alphabet of size σ containing $\Theta(\sigma n)$ such fragments. We begin with describing the lower bound in Section 3, and then present the upper bound in Section 4.

▶ **Theorem 1.** *The number of order-preserving squares in a string of length n over an alphabet of size σ is $\mathcal{O}(\sigma n)$, and this bound is asymptotically tight even if we only consider order-preserving squares that are distinct as words.*

Next, we design an algorithm for reporting all order-preserving squares in a given string of length n over an alphabet of size σ in $\mathcal{O}(\sigma n)$ time, which (by our lower bound) is asymptotically optimal in the worst case. We again stress that in our definition of an order-preserving square, we require that $u \neq v$. However, all fragments of the form uu that are distinct as words can be reported in $\mathcal{O}(\sigma n)$ time using the algorithm of Gusfield and Stoye [14][1]. Thus, for $\sigma = o(\log n)$, this resolves one of the open questions by Crochemore et al. [4], who asked if there is an $o(n \log n)$ time algorithm for finding the longest order-preserving square. This is described in Section 5.

▶ **Theorem 2.** *All order-preserving squares in a string of length n over an alphabet of size σ can be found in $\mathcal{O}(\sigma n)$ time.*

High-level description of our techniques. For the lower bound, first, we consider the increasing string $w = 123 \dots n$ where $\sigma = n$. Clearly, any even-length fragment is an order-preserving square thus producing the maximum number, i.e. $\Omega(n^2) = \Omega(\sigma n)$, of order-preserving squares in a string of length n. To decrease the size of the alphabet σ, we replace w with a non-decreasing string $w = 11 \dots 122 \dots 2 \dots \sigma\sigma \dots \sigma$, where each character is repeated the same number of times. We exhibit $\Omega(\sigma n)$ order-preserving squares in w that are distinct as words. See Section 3 for more details.

For the upper bound, we build on the insight by Kociumaka et al. [20], where the high-level strategy is to consider each suffix of w separately. For each suffix and an alphabet character, they considered the leftmost occurrence of this character within the suffix. Thus, there are at most σ leftmost occurrences in each suffix. For a fixed suffix, they considered all of its prefixes as possible order-preserving squares uv. Next, they showed that, because $u \neq v$, the order-preserving square uv is defined by a pair (or pairs) of leftmost occurrences such that one occurrence belongs to u, and the other one belongs to v at the same relative position, where the length of uv is twice the difference between the leftmost occurrences. For example, let `acbadxyz` be the suffix, then the pair of positions 2 and 5 are leftmost occurrences defining the order-preserving square `acbadx` of length 6 that is a prefix of the given suffix. Note that, also 3 and 6 are leftmost occurrences defining the same order-preserving square `acbadx`. Thus, as a result, they upper bounded the number of order-preserving squares being a prefix of the considered suffix by $\binom{\sigma}{2}$, so $\binom{\sigma}{2}n$ in total.

In this paper, we adopt a similar approach, by separately upper bound the number of order-preserving squares that are prefixes of a suffix of the input string w. However, our goal is to show that there are only $\mathcal{O}(\sigma)$ such prefixes, so $\mathcal{O}(\sigma n)$ in total. To this end, we first partition the order-preserving squares into groups. Let O_k the set of all order-preserving squares uv such that $2^k \leq |uv| < 2^{k+1}$. Similarly, we partition the leftmost occurrences into groups. Let L_k the set of all leftmost occurrences i such that $2^k \leq i < 2^{k+1}$. Now, our strategy is to show that if $|O_k|$ is larger than some fixed constant then $|O_k| = \mathcal{O}(|L_{k-2}|)$. The structure of the argument is as follows. We first observe that two order-preserving squares uv and $u'v'$ imply that $|u| - \Delta$, where $\Delta = |u'| - |u|$, is a so-called order-preserving border of u. We write $u = b_1 b_2 \dots b_f b_{f+1}$, where $|b_1| = |b_2| = \dots |b_f| = \Delta$ and $|b_{f+1}| < \Delta$, and by carefully choosing uv and $u'v'$ from O_k conclude that b_2 contains a leftmost occurrence and f is proportional to $|O_k|$. Then, we argue that b_2 containing a leftmost occurrence implies

[1] They only claim $\mathcal{O}(n)$ time for fixed alphabets, however a closer look at the algorithm reveals that there are 3 phases: the first phase takes $\mathcal{O}(n)$ time (Theorem 6 and Lemma 7), the second phase also takes $\mathcal{O}(n)$ time (Section 4), and the third phase takes $\mathcal{O}(\sigma n)$ time (Lemma 11). Additionally, the algorithm assumes that the suffix tree is constructed in $\mathcal{O}(n)$ time, for larger alphabets this increases to $\mathcal{O}(\sigma n)$.

that, in fact, every b_j contains a leftmost occurrence, and thus $|O_k| = \mathcal{O}(|L_{k-2}|)$. Summing this over all k, and separately considering all k such that $|O_k|$ is less than the fixed constant, we are able to conclude that $\sum_k |O_k| = \sum_k \mathcal{O}(|L_k|) < \mathcal{O}(\sigma)$. See Section 4 for more details.

To obtain an efficient algorithm for reporting all order-preserving squares, we apply the order-preserving suffix tree as defined by Crochemore et al. [4]. This structure allows us to check if $w[i..i+2\ell-1]$ is an order-preserving square by checking if the LCA of two leaves is at string depth at least ℓ. First, we need to show how to construct the order-preserving tree in $\mathcal{O}(\sigma n)$ time. Second, we extend the above reasoning to efficiently generate only $\mathcal{O}(\sigma n)$ fragments that are then tested for being an order-preserving square in constant time each. While the underlying argument is essentially the same as when bounding the number of order-preserving squares, it needs to be executed differently for the purpose of an efficient implementation. See Section 5 for more details.

2 Preliminaries

Let $\Sigma = \{1, \ldots, \sigma\}$ be a fixed finite alphabet of size σ. Let $|s|$ denote the length of a string s. For a string s, the character at position i of s is denoted by $s[i]$, and $s[i..j]$ is the fragment of s starting at position i and ending at position j. We call two strings u and v order-isomorphic, denoted by $u \approx v$, when $|u| = |v|$ and, for each i, j, we have $u[i] \leq u[j]$ if and only if $v[i] \leq v[j]$. The concatenation of two strings u and v is denoted by uv. A string of the form uv is called an order-preserving square, or op-square, when $u \neq v$ and $u \approx v$. We call u its left arm and v its right arm. We stress that a regular square, that is, a string of the form xx, is not an op-square. Two op-squares uv and $u'v'$ are *distinct as words* if and only if $uv \neq u'v'$.

A trie is a rooted tree, with every edge labeled with a single character and edges outgoing from the same node having distinct labels. A node u of a trie represents the string obtained by reading the labels on the path from the root to u. A compacted trie is obtained from a trie by replacing maximal paths consisting of nodes with exactly one child with single edges labeled by the concatenation of the labels of the edges on the path. A suffix tree T of a string w is a compacted trie whose leaves correspond to the suffixes of $w\$$. The string depth of a node u of T is the length of the string that it corresponds to. An explicit node of T is simply a node of T. An implicit node of T is a node of the non-compacted trie corresponding to T, or in other words a location on an edge of T.

Next, we need some definitions specific to order-isomorphism. Following Kubica et al. [22], we call b an op-border of a string $s[1..n]$ when $s[1..b] \approx s[n-b+1..n]$. Following Gourdel et al. [13] (and Matsuoka et al. [25]), we call p an (initial) op-period of $s[1..n]$ when $s = b_1 b_2 \ldots b_f b_{f+1}$ with $|b_1| = |b_2| = \ldots = |b_f| = p$ and $|b_{f+1}| < p$ (so $f = \lfloor n/p \rfloor$), $b_1 \approx b_2 \approx \ldots \approx b_f$ and $b_1[1..|b_{f+1}|] \approx b_2[1..|b_{f+1}|] \approx \ldots \approx b_f[1..|b_{f+1}|] \approx b_{f+1}$. $b_1, b_2, \ldots, b_f$ are called the *blocks* defined by p in s, while b_{f+1} (possibly empty) is called the *incomplete block*. While in the classical setting p is a period of $s[1..n]$ if and only if $n-p$ is a border of $s[1..n]$, in the order-preserving setting, we only have an implication in one direction. For example, the string `aficdgbeh` has an op-period 3 while 6 is not its op-border.

▶ **Proposition 3.** *If b is an op-border of $s[1..n]$ then $n-b$ is an initial op-period of $s[1..n]$.*

Proof. Let $p = n - b$ and $f = \lfloor n/p \rfloor$. We represent $s[1..n]$ as $s = b_1 b_2 \ldots b_f b_{f+1}$ with $|b_1| = |b_2| = \ldots = |b_f| = p$ and $|b_{f+1}| < p$. By b being an op-border of $s[1..n]$, we have $s[1..b] \approx s[n-b+1..n]$, so $s[1..n-p] \approx s[p+1..n]$. We observe that $s[1..n-p] = b_1 b_2 \ldots b_{f-1} b_f[1..|b_{f+1}|]$ and $s[p+1..n] = b_2 b_3 \ldots b_f b_{f+1}$. Then, $b_1 b_2 \ldots b_{f-1} b_f[1..|b_{f+1}|] \approx$

$b_2b_3 \dots b_f b_{f+1}$ implies $b_i \approx b_{i+1}$, for every $i = 1, 2, \dots, f-1$, and $b_i[1..|b_{f+1}|] \approx b_{i+1}[1..|b_{f+1}|]$, for every $i = 1, 2, \dots, f$. Hence, we obtain $b_1 \approx b_2 \approx b_3 \approx \dots \approx b_{f-1} \approx b_f$ and $b_1[1..|b_{f+1}|] \approx b_2[1..|b_{f+1}|] \approx \dots \approx b_f[1..|b_{f+1}|] \approx b_{f+1}$, so $p = n - b$ is indeed an initial op-period of $s[1..n]$. ◀

Due to Proposition 3, if b is an op-border of $s[1..n]$ then $s[1..n] = b_1b_2 \dots b_f b_{f+1}$, where $b_1b_2 \dots b_f[1..|b_{f+1}|] \approx b_2b_3 \dots b_f b_{f+1}$, $|b_1| = |b_2| = \dots = |b_f| = n - b$ and $|b_{f+1}| < p$ (so $f = \lfloor n/(n-b) \rfloor$), $b_1 \approx b_2 \approx \dots \approx b_f$ and $b_1[1..|b_{f+1}|] \approx b_2[1..|b_{f+1}|] \approx \dots \approx b_f[1..|b_{f+1}|] \approx b_{f+1}$. We will say that these blocks are defined by b.

3 Lower Bound

Recall that $\Sigma = \{1, \dots, \sigma\}$. We define a string $w = 11 \dots 122 \dots 2 \dots \sigma\sigma \dots \sigma$, that is, a concatenation of σ blocks, each consisting of k repetitions of the same character. We note that $|w| = \sigma k$. For $i = 1, 2, \dots, \lfloor \sigma/2 \rfloor$, we consider all fragments of w of length $2ik$ starting at positions $j = 1, 2, \dots, |w| - 2ik + 1$. For $j = 1 \bmod k$, the fragment is a concatenation of $2i$ blocks, each block consisting of k repetitions of the same character. For $j \neq 1 \bmod k$, the fragment starts with $r \in [1, k-1]$ repetitions of the same character, then $2i - 1$ blocks, each block consisting of k repetitions of the same character, and finally $\ell = k - r$ repetitions of the same character. See Figure 1.

1 11112 22223 33 33444445555666667777 7

Figure 1 The red box corresponds to an op-square of length 10 containing 3 different characters. The blue box corresponds to an op-square of length 20 containing 5 different characters.

Each such fragment is an op-square. For $j = 1 \bmod k$, both the left and the right arm consist of i blocks consisting of k repetitions of character $c, c+1, \dots, c+i-1$. For $j \neq 1 \bmod k$, both the left and the right arm consist of first r repetitions of character c, then $i - 1$ blocks consisting of k repetitions of characters $c+1, c+2, \dots, c+i-2$, and then finally ℓ repetitions of character $c + i - 1$. Thus, the left and the right arm are always order-isomorphic. Further, for every choice of i and the starting position we obtain a different word, as two such fragments of the same length either start with different characters or differ in the length of the first block of the same character.

Now, we analyze the number of such op-squares in w. By considering every $1 \leq i \leq \lfloor \sigma/2 \rfloor$ and starting position $1, 2, \dots, |w| - 2ik + 1$, we obtain that the number of op-squares in w is at least:

$$\sum_{i=1}^{\lfloor \sigma/2 \rfloor} (|w| - 2ik + 1) = \sum_{i=1}^{\lfloor \sigma/2 \rfloor} (\sigma k - 2ik + 1) = \lfloor \sigma/2 \rfloor \cdot (\sigma k - k(\lfloor \sigma/2 \rfloor + 1) + 1)$$
$$\geq \lfloor \sigma/2 \rfloor \cdot (k(\lceil \sigma/2 \rceil - 1) + 1).$$

For $\sigma \geq 3$, this is at least $\sigma^2 k/12 = \sigma n/12$ for any k. For $\sigma = 1, 2$, we additionally assume $k \geq 2$ and count op-squares of the form 1^{2i}, there are $\lfloor k/2 \rfloor \geq n/6 \geq \sigma n/12$ of them. Thus, in either case for every $k \geq 2$ we obtain a string of length $n = k\sigma$ over Σ containing $\sigma n/12$ op-squares that are distinct as words.

▶ **Theorem 4.** *For any alphabet $\Sigma = \{1, 2, \dots, \sigma\}$, there exists an infinite family of strings of length $n = k\sigma$ over Σ containing $\Omega(\sigma n)$ op-squares distinct as words.*

4 Upper bound

Our goal in this section is to upper bound the number of op-squares in a given string w of length n over the alphabet $\Sigma = \{1, \ldots, \sigma\}$. Recall that uv is an op-square when $u \neq v$ and $u \approx v$. We will show that this number is $\mathcal{O}(\sigma n)$. As explained in the introduction, by the result of Brlek and Li [1], the number of regular squares, that is, fragments of the form uu that are distinct as words, is less than n. Thus, our result in fact allows us to upper bound the number of fragments of the form uv, where $u \approx v$, that are distinct as words by $\mathcal{O}(\sigma n)$.

We consider each suffix of w separately. For each suffix $w[i..n]$, we will upper bound the number of prefixes of $w[i..n]$ that are op-squares by $\mathcal{O}(\sigma)$. Therefore, to avoid cumbersome notation in the remaining part of this section we will assume that we have a string s of length m over the alphabet $\Sigma = \{1, \ldots, \sigma\}$, and we want to upper bound the number of op-squares uv that are prefixes of s by $\mathcal{O}(\sigma)$. See Figure 2.

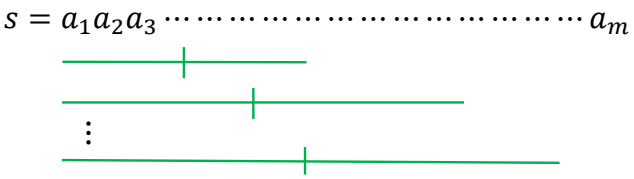

Figure 2 Green prefixes of s are op-squares.

Kociumaka et al. [20] observed that every op-square uv that is a prefix of s can be obtained as follows (recall that in our definition $u \neq v$). We call position i a leftmost occurrence and $s[i]$ a leftmost character when $s[j] \neq s[i]$ for every $j < i$. Then, there exists i and j such that both i and j are leftmost occurrences, where i belongs to u and j belongs to v, and further $|u| = j - i$. More formally:

▶ **Proposition 5** ([20, Lemma 4.2 and Corollary 4.3])**.** *We can construct an injective function g mapping op-squares that are prefixes of s to 2-element subsets of the alphabet as follows. We choose the smallest i belonging to v such that $s[i] = a$ does not occur in u, and let $s[i-|u|] = b$ be its counterpart in u, then set $g(uv) = \{a, b\}$. Both i and $i - |u|$ are leftmost occurrences.*

We split all op-squares that are prefixes of s into groups. Let O_k denote the group of op-squares that are prefixes of s having length at least 2^k and at most $2^{k+1} - 1$:

▶ **Definition 6.** $O_k = \{uv \mid u \neq v \text{ and } u \approx v \text{ and } 2^k \leq |uv| < 2^{k+1}\}$ *for* $0 \leq k \leq \log m$.

In other words, we split s into consecutive ranges of exponentially increasing lengths, such that the k-th range is of length 2^{k-1}, starts at position 2^k and ends at position $2^{k+1} - 1$ in s (where $0 \leq k \leq \log m$ and the final range may not be complete when $m < 2^{k+1} - 1$). Then, the set O_k consists of op-squares that end in the k-th range. See Figure 3.

The number of op-squares uv that are prefixes of s is $\sum_{k=0}^{\log m} |O_k|$. In order to upper bound the sum, we will separately upper bound the size of each group. We first need some propositions.

▶ **Proposition 7.** *For any $\{uv, u'v'\} \in O_k$ such that $|u| < |u'|$ and $\Delta = |u'| - |u|$, $|u| - \Delta$ is an op-border of both u and v.*

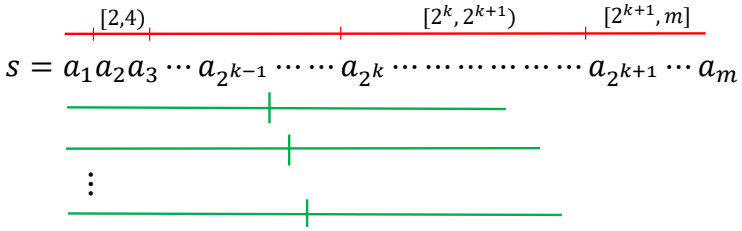

Figure 3 Green prefixes of s are op-squares ending in the k-th range. The red line illustrates the ranges.

Proof. Because $u \approx v$ it is enough to show that $|u| - \Delta$ is an op-border of u. By the assumption that both uv and $u'v'$ are op-squares we have:

$$u[1..|u| - \Delta] = u'[1..|u| - \Delta] \approx v'[1..|u| - \Delta] = v[\Delta + 1..|u|] \approx u[\Delta + 1..|u|].$$

See Figure 4. ◀

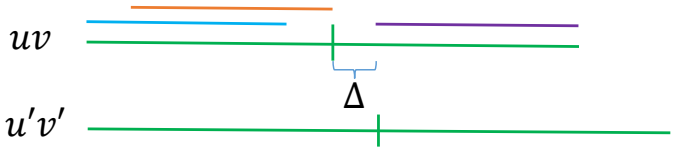

Figure 4 The green lines correspond to uv and $u'v'$. The blue line corresponds to $u[1..|u| - \Delta]$. The orange line corresponds to $u[\Delta + 1..|u|]$. The purple line corresponds to $v[\Delta + 1..|u|]$.

In the remaining part of this section, we will often consider $\{uv, u'v'\} \in O_k$ such that $|u| < |u'|$ and $\Delta = |u'| - |u|$. Then, by Proposition 7, we know that $|u| - \Delta$ is an op-border of u, and thus by Proposition 3 Δ is an initial op-period of u. Hence, u can be represented as a concatenation of $f = \lfloor |u|/\Delta \rfloor$ blocks $b_1, b_2, \ldots, b_f$ and one incomplete block b_{f+1}, where $|b_1| = |b_2| = \ldots = |b_f| = \Delta$ and $|b_{f+1}| < \Delta$, such that $b_1 b_2 \ldots b_f[1..|b_{f+1}|] \approx b_2 b_3 \ldots b_f b_{f+1}$, $b_1 \approx b_2 \approx \ldots \approx b_f$ and $b_1[1..|b_{f+1}|] \approx b_2[1..|b_{f+1}|] \approx \ldots \approx b_f[1..|b_{f+1}|] \approx b_{f+1}$. See Figure 5. For brevity, in the remaining part of the paper we will describe this situation by saying that $\{uv, u'v'\} \in O_k$ define blocks $b_1, b_2, \ldots, b_f, b_{f+1}$.

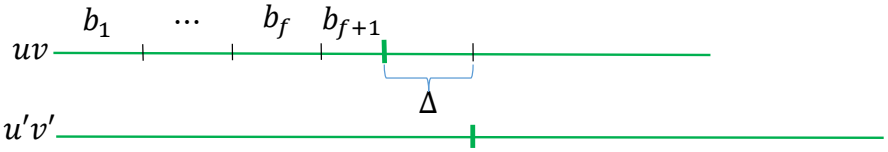

Figure 5 Blocks defined by $\{uv, u'v'\}$ in u.

▶ **Proposition 8.** *If $|O_k| \geq 3$ then there exist $\{uv, u'v', u''v''\} \in O_k$ such that $0 < |u'| - |u|, |u''| - |u'| < 2^k/(|O_k| - 2)$.*

Proof. The length of every op-square in O_k belongs to $[2^k, 2^{k+1})$, thus the length of its left arm falls within $[2^{k-1}, 2^k)$. Let $O_k = \{u_1 v_1, u_2 v_2, \ldots, u_\ell v_\ell\}$ with $|u_1| < |u_2| < \ldots < |u_\ell|$. Then, for some $i \in \{1, 2, \ldots, \lfloor (\ell - 1)/2 \rfloor\}$ we must have $|u_{2i+1}| < |u_{2i-1}| + 2^{k-1}/\lfloor (\ell - 1)/2 \rfloor$ (as otherwise we would have $u_\ell \geq u_1 + 2^{k-1}$). The sought op-squares are $u_{2i-1}v_{2i-1}, u_{2i}v_{2i}, u_{2i+1}v_{2i+1}$ because:

$$|u_{2i}| - |u_{2i-1}|, |u_{2i+1}| - |u_{2i}| < |u_{2i+1}| - |u_{2i-1}| < 2^{k-1}/\lfloor(\ell-1)/2\rfloor$$
$$\leq 2^{k-1}/(\ell/2-1) = 2^k/(|O_k|-2).$$ ◀

With all the propositions in hand, we are now ready for the technical lemmas. Our goal is to upper bound $\sum_k |O_k|$ by the number of leftmost occurrences. To this end, we need to show that, if some O_k is large then there are many leftmost occurrences in some range. This will be done by applying the following reasoning to the three op-squares chosen by applying Proposition 8. In the following, whenever we refer to a leftmost occurrence in block b we mean a leftmost occurrence falling within the positions in block b.

▶ **Lemma 9.** *If $|O_k| \geq 3$ then for any $\{uv, u'v', u''v''\} \in O_k$ where $|u| < |u'| < |u''|$ such that $\{uv, u'v'\}$ defines $b_1, \ldots, b_f, b_{f+1}$ and $\{u'v', u''v''\}$ defines $b'_1, \ldots, b'_f, b'_{f+1}$ there is a leftmost occurrence in block b_j such that $j \neq 1$ or there is a leftmost occurrence in block $b'_{j'}$ such that $j' \neq 1$.*

Proof. Let $\Delta = |u'| - |u|$ be the length of every block b_j and $\Delta' = |u''| - |u'|$ be the length of every block b'_j. By Proposition 5, we know that there must be a leftmost occurrence i that falls within u' and its corresponding leftmost occurrence $i + |u'|$ that falls within v'. If $s[i]$ belongs to a block b'_j with $j \neq 1$ then we are done. Thus, we assume that i belongs to b'_1. We claim that the leftmost occurrence $i + |u'|$ falls within v. To verify this, we calculate:

$$i + |u'| \leq \Delta' + |u'| = |u''| - |u'| + |u'| = |u''| < 2^k \leq |uv|.$$

We have established that $i + |u'|$ is a leftmost occurrence and falls within v. Thus, $s[i'] \neq s[i + |u'|]$ for every $i' \in [1, i + |u'|)$. Because $u \approx v$, this then implies that $s[i' - |u|] \neq s[i + |u'| - |u|]$ for every $i' \in [|u| + 1, i + |u'|)$. Thus, $i + |u'| - |u|$ is also a leftmost occurrence. We claim that $s[i + |u'| - |u|]$ cannot belong to b_1. To verify this, we calculate:

$$i + |u'| - |u| \geq 1 + |u'| - |u| = 1 + \Delta.$$

Thus, we have found a leftmost occurrence $i + |u'| - |u|$ that falls within u and belongs to a block b_j with $j \neq 1$. See Figure 6. ◀

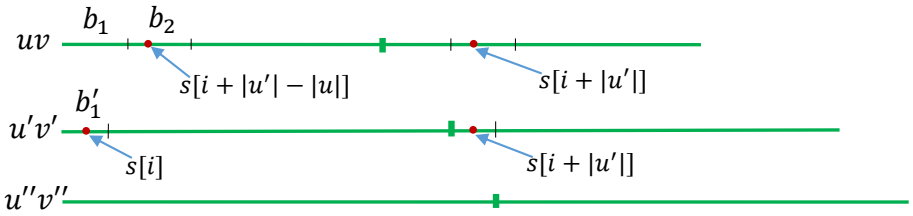

Figure 6 The red points correspond to the leftmost occurrences considered in the proof of Lemma 9.

Next, we show that if $\{uv, u'v'\} \in O_k$ define blocks $b_1, b_2, \ldots, b_f, b_{f+1}$ such that there is a leftmost occurrence in block b_j for some $j \neq 1$ then, in fact, there is a leftmost occurrence in every block b_j. This reasoning is done in two steps.

▶ **Lemma 10.** *Let b be an op-border of $u = s[1..|u|]$ that defines blocks $b_1, b_2, \ldots, b_f, b_{f+1}$, and assume that there is a leftmost occurrence in block b_j, for some $j \in [1, f+1]$. Then there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_j$.*

Proof. Let $\Delta = |b_1| = |b_2| = \ldots = |b_f|$ and $|b_{f+1}| < \Delta$. By induction, it is enough to show that if there is a leftmost occurrence in block b_j for some $j \geq 2$ then there is a leftmost occurrence in block b_{j-1}. Let i be a leftmost occurrence that belongs to b_j. Then $u[i'] \neq u[i]$ for every $i' \in [1, i)$. Because $b_1 b_2 \ldots b_{f-1} b_f[1..|b_{f+1}|] \approx b_2 b_3 \ldots b_f b_{f+1}$, this implies $u[i' - \Delta] \neq u[i - \Delta]$ for every $i' \in [\Delta + 1, i)$. But then $i - \Delta$ is also a leftmost occurrence, and it belongs to b_{j-1} as required. See Figure 7. ◀

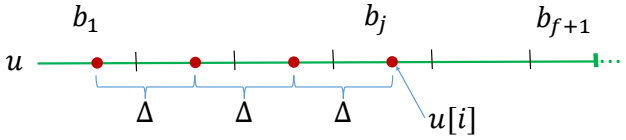

Figure 7 Each red point corresponds to a leftmost character at the same relative position in every block $b_1, b_2, \ldots, b_j$.

▶ **Lemma 11.** *Let b be an op-border of $u = s[1..|u|]$ that defines blocks $b_1, b_2, \ldots, b_f, b_{f+1}$, and assume that there is a leftmost character in block b_2. Then there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_f$.*

Proof. Let $\Delta = |b_1| = |b_2| = \ldots |b_f|$ and $|b_{f+1}| < \Delta$. By assumption, there is a leftmost character $s[i]$ in block b_2, where $i \in [\Delta + 1, 2\Delta]$. Our goal is to show that there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_f$.

Because $b_1 b_2 \ldots b_f[1..|b_{f+1}|] \approx b_2 b_3 \ldots b_f b_{f+1}$, each position $x \in [1..\Delta]$ satisfies exactly one of the following possibilities:

1. $s[x + p \cdot \Delta]$ is the same, for all integers $p \in [0, f)$,
2. $s[x + p \cdot \Delta] < s[x + (p+1) \cdot \Delta]$ for all integers $p \in [0, f-1)$,
3. $s[x + p \cdot \Delta] > s[x + (p+1) \cdot \Delta]$ for all integers $p \in [0, f-1)$.

Note that $i_0 = i - \Delta$ satisfies (2) or (3), because $s[i]$ is different than $s[1], s[2], \ldots, s[i-1]$, so in particular $s[i - \Delta] \neq s[i]$. By reversing the order of the alphabet, it is enough to establish the lemma assuming that i_0 satisfies (2). To this end, we choose some positions $i_1, i_2, \ldots, i_\ell$ in b_1 as follows. Let C be the set of characters that appear in b_1. The position $i_1 \in [1, \Delta]$ is chosen so that $s[i_1]$ is the strict successor of $s[i_0]$ in C, then $i_2 \in [1, \Delta]$ is chosen so that $s[i_2]$ is the strict successor of $s[i_1]$ in C, and so on. If there are multiple choices for the next $i_j \in [1, \Delta]$ then we take the smallest. We stop when one of the following two possibilities holds:

(a) $s[i_{\ell+1}]$ is not defined, i.e. $s[i_\ell]$ is the largest character in b_1.
(b) $i_{\ell+1}$ satisfies (1) or (3).

Notice that, by definition, the positions $i_0, i_1, \ldots, i_\ell$ all satisfy (2). Further, $s[i_0]$ is a leftmost character because $s[i]$ is a leftmost character, so $s[i'] \neq s[i]$ for every $i' \in [1, i)$, and $b_1 \approx b_2$ so $s[i' - \Delta] \neq s[i - \Delta]$ for every $i' \in [\Delta + 1, i)$. Next, we note that $s[i_1], \ldots, s[i_\ell]$ are all leftmost characters because we are always choosing the smallest i_j such that $s[i_j]$ is equal to a specific character, for $j = 1, 2, \ldots, \ell$.

We summarize the situation so far. For every integer $p \in [0, f)$, the fragment $s[i_\ell + p \cdot \Delta]$ belongs to block b_{p+1}, and we want to show that it is a leftmost character. We know that $s[i_\ell]$ is a leftmost character, thus by $b_1 \approx b_{p+1}$ we obtain that $s[i_\ell + p \cdot \Delta]$ does not occur earlier in b_{p+1}. We need to establish that it also does not occur earlier in $b_1, b_2, \ldots, b_p$. We separately consider the two possible cases (a) and (b).

(a) $s[i_{\ell+1}]$ is not defined, i.e. $s[i_\ell]$ is the largest character in b_1. We know that i_ℓ satisfies (2), so $s[i_\ell] < s[i_\ell + \Delta] < \ldots < s[i_\ell + (p-1)\cdot\Delta] < s[i_\ell + p\cdot\Delta]$. For all integers $q \in [0, p)$, by $b_1 \approx b_{q+1}$ we obtain that $s[i_\ell + q\cdot\Delta]$ is the largest character in b_{q+1}. So in fact $s[i_\ell + p\cdot\Delta]$ is larger than all characters in the whole block b_{q+1}, for every integer $q \in [0, p)$, making $i_\ell + p\cdot\Delta$ a leftmost occurrence.

(b) $i_{\ell+1}$ is defined and satisfies (1) or (3), so $s[i_{\ell+1}] \geq s[i_{\ell+1}+\Delta] \geq \ldots \geq s[i_{\ell+1}+(p-1)\cdot\Delta] \geq s[i_{\ell+1} + p\cdot\Delta]$. See Figure 8. We know that i_ℓ satisfies (2), so $s[i_\ell] < s[i_\ell + \Delta] < \ldots < s[i_\ell + (p-1)\cdot\Delta] < s[i_\ell + p\cdot\Delta]$. Recall that $s[i_{\ell+1}]$ is a strict successor of $s[i_\ell]$ in b_1. Thus, for every $i' \in [1, \Delta]$ we have that $s[i']$ does not belong to the interval $(s[i_\ell], s[i_{\ell+1}])$. Because we have $b_1 \approx b_{q+1}$, for every integer $q \in [0, p)$, this implies $s[i' + q\cdot\Delta]$ does not belong to the interval $(s[i_\ell + q\cdot\Delta], s[i_{\ell+1} + q\cdot\Delta])$. As observed earlier, $s[i_\ell + q\cdot\Delta] < s[i_\ell + p\cdot\Delta]$ and $s[i_{\ell+1} + q\cdot\Delta] \geq s[i_{\ell+1} + p\cdot\Delta]$. We conclude that, for every $i' \in [1, \Delta]$, we have that $s[i' + q\cdot\Delta]$ does not belong to the interval $[s[i_\ell + p\cdot\Delta], s[i_{\ell+1} + p\cdot\Delta])$ (the interval is non-empty, as both positions belong to the same block b_q, and by $b_{q+1} \approx b_1$ we have that $s[i_{\ell+1} + q\cdot\Delta]$ is a strict successor of $s[i_\ell + q\cdot\Delta]$ in b_q). In particular, $s[i' + q\cdot\Delta] \neq s[i_\ell + p\cdot\Delta]$, so $s[i_\ell + p\cdot\Delta]$ does not occur in b_{q+1}, making it a leftmost character.

Hence, for every integer $p \in [0, f)$, position $i_\ell + p\cdot\Delta$ is a leftmost occurrence. ◀

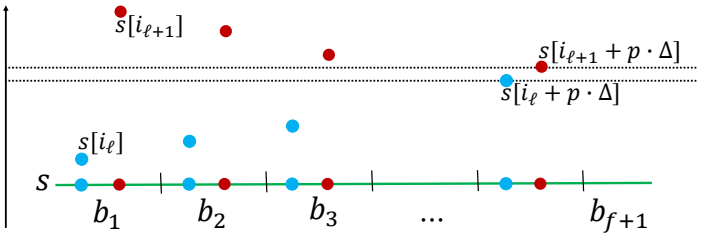

Figure 8 The red points correspond to $s[i_{\ell+1}], ..., s[i_{\ell+1} + p\cdot\Delta]$. The blue points correspond to $s[i_\ell], ..., s[i_\ell + p\cdot\Delta]$. The black arrow illustrates the character's axis.

By combining the above lemmas we obtain the following conclusion.

▶ **Lemma 12.** *If $|O_k| \geq 3$ then for any $\{uv, u'v', u''v''\} \in O_k$ where $|u| < |u'| < |u''|$ such that $\{uv, u'v'\}$ defines $b_1, \ldots, b_f, b_{f+1}$ and $\{u'v', u''v''\}$ defines $b'_1, \ldots, b'_f, b'_{f+1}$ there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_f$ or there is a leftmost occurrence in every block $b'_1, b'_2, \ldots, b'_{f'}$.*

Proof. Recall that by Proposition 7, $|u'| - |u|$ is an op-border of $u = s[1..|u|]$ while $|u''| - |u'|$ is an op-border of $u' = s[1..|u'|]$. By Lemma 10 there is a leftmost occurrence in block b_2 or in block b'_2. Then, by Lemma 11 applied either to the blocks $b_1, b_2, \ldots, b_f, b_{f+1}$ defined by the op-border $|u'| - |u|$ or the blocks $b'_1, b'_2, \ldots, b'_{f'}, b'_{f'+1}$ defined by the op-border $|u''| - |u'|$, there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_f$ or in every block $b'_1, b'_2, \ldots, b'_{f'}$. ◀

We are now ready to upper bound $\sum_k |O_k|$ by the number of leftmost characters. We will show that, if some O_k is large then there are many leftmost characters in some range. To this end, we define groups of leftmost occurrences. Let L_k be the set of the leftmost occurrences i such that $i \in [2^k, 2^{k+1})$:

▶ **Definition 13.** $L_k = \{i \mid i \in [2^k, 2^{k+1}) \wedge \forall_{j\in[1,i)} s[i] \neq s[j]\}$ *for* $0 \leq k \leq \log m$.

Note that the groups are disjoint, i.e. $L_k \cap L_{k'} = \emptyset$ for any k and k'. Thus $\sum_{k=0}^{\log m} |L_k| \leq \sigma$. With this definition in hand, we are ready to show the main technical lemma.

▶ **Lemma 14.** *The number of op-squares that are prefixes of s is $\mathcal{O}(\sigma)$.*

Proof. To establish the lemma we want to connect $|O_k|$ with $|L_k|$, and then sum over all possible values of k. $k = 0, 1$ will be considered separately, and for larger k we apply different arguments for $|O_k| \geq 11$ and $|O_k| \leq 10$.

We first consider $k \geq 2$ such that $|O_k| \geq 11$. In particular, when $|O_k| \geq 3$, so by Proposition 8, there exist $\{uv, u'v', u''v''\} \in O_k$ such that $0 < |u'| - |u|, |u''| - |u'| < 2^k/(|O_k| - 2)$. By Lemma 12, for any $\{uv, u'v', u''v''\} \in O_k$ where $|u| < |u'| < |u''|$ such that $\{uv, u'v'\}$ defines $b_1, \ldots, b_f, b_{f+1}$ in u and $\{u'v', u''v''\}$ defines $b'_1, \ldots, b'_{f'}, b'_{f'+1}$ in u' either there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_f$ or there is a leftmost occurrence in every block $b'_1, b'_2, \ldots, b'_{f'}$ In either case, we have found $\{uv, u'v'\} \in O_k$ with $0 < \Delta < 2^k/(|O_k| - 2)$, where $\Delta = |u'| - |u|$, such that $\{uv, u'v'\}$ defines $b_1, \ldots, b_f, b_{f+1}$ with $f = \lfloor |u|/\Delta \rfloor$ and there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_f$. We want to establish a lower bound on the number of leftmost occurrences in L_{k-2}. To this end, it is enough to show a lower bound on the number of blocks b_i that are fully contained in the range $[2^{k-2}, 2^{k-1})$. Recall that $|u| \in [2^{k-1}, 2^k)$, and $u = b_1 b_2 \ldots b_f b_{f+1}$. Thus, the fragment $u[2^{k-2}..2^{k-1} - 1]$ consists of a suffix (possibly empty) of some b_j, then $b_{j+1}, b_{j+2}, \ldots, b_{j+\ell}$, and then a prefix of $b_{j+\ell+1}$ (where $b_{j+\ell+1}$ might be the incomplete block b_{f+1} that should not be counted in the lower bound). Thus, the number of blocks b_i that are fully contained in the range $[2^{k-2}, 2^{k-1})$ is at least $\lfloor 2^{k-2}/\Delta \rfloor - 1$. See Figure 9. Combining this with the upper bound on Δ, we obtain the following inequality:

$$|L_{k-2}| \geq \left\lfloor \frac{2^{k-2}}{\Delta} \right\rfloor - 1 \geq \frac{2^{k-2}}{\Delta} - 2 > \frac{|O_k| - 2}{4} - 2 = \frac{|O_k| - 10}{4}.$$

Using the assumption $|O_k| \geq 11$, we conclude that $|L_{k-2}| > |O_k|/44$. Hence:

$$\sum_{k \geq 2 : |O_k| \geq 11} |O_k| < \sum_{k \geq 2} 44 \cdot |L_{k-2}| \leq 44 \sum_k |L_k| \leq 44 \cdot \sigma.$$

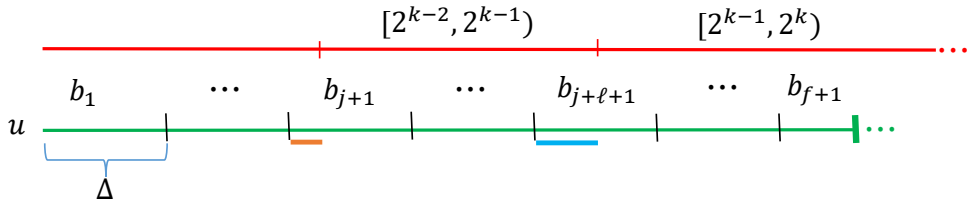

Figure 9 The orange line corresponds to the suffix of b_j. The blue line corresponds to the prefix of $b_{j+\ell+1}$. The red line illustrates the ranges.

Next, we consider $k \geq 2$ such that $|O_k| \leq 10$. Of course, we have the trivial upper bound $\sum_{k:|O_k| \leq 10} |O_k| \leq \sum_{k=0}^{\log m} 10 = \mathcal{O}(\log m)$. As in the previous case, we want to use the leftmost occurrences to improve the bound. Recall that, by Proposition 5, every op-square $uv \in O_k$ is defined by a pair of leftmost occurrences i and j, where i belongs to u and j belongs to v. Because $|uv| \in [2^k, 2^{k+1})$, we conclude that j falls within the range $[2^{k-1} + 1, 2^{k+1})$, so must belong to $L_{k-1} \cup L_k$. Hence, the set O_k can be non-empty only when L_{k-1} or L_k is non-empty. Hence:

$$\sum_{k\geq 2:|O_k|\leq 10} |O_k| \leq \sum_{k\geq 2:|O_k|>0} 10 \leq \sum_{k\geq 2:|L_{k-1}|>0} 10 + \sum_{k\geq 2:|L_k|>0} 10$$
$$\leq 10\sum_{k\geq 2} |L_{k-1}| \quad + 10\sum_{k\geq 2} |L_k| \quad \leq 20\sigma.$$

To upper bound $\sum_k |O_k|$, we split the sum into three parts. For $k = 0, 1$, we have $|O_0| \leq 1$ and $|O_1| \leq 2$. Then, for $k \geq 2$ we separately consider all k with $|O_k| \geq 11$ and $|O_k| \leq 10$ and plug in the above upper bounds. Overall, we obtain:

$$\sum_k |O_k| \leq 1 + 2 + 44 \cdot \sigma + 20 \cdot \sigma = \mathcal{O}(\sigma).$$

Thus, the number of op-squares that are prefixes of s is $\mathcal{O}(\sigma)$. ◀

We conclude the section with the main theorem.

▶ **Theorem 15.** *The number of op-squares in a string w of length n over an alphabet of size σ is $\mathcal{O}(\sigma n)$.*

Proof. We consider each suffix of w separately. For each suffix $w[i..n]$, we apply Lemma 14 to conclude that the number of op-squares that are prefixes of $w[i..n]$ is upper bounded by $\mathcal{O}(\sigma)$. Thus, summing over all i we obtain that the number of op-squares in w is $\mathcal{O}(\sigma n)$. ◀

5 Algorithm

In this section, we describe the algorithm that reports all occurrences of op-squares in a string $w[1..n]$ over an alphabet of size σ in $\mathcal{O}(\sigma n)$ time.

The high-level idea of the algorithm is to generate $\mathcal{O}(\sigma n)$ candidates for op-squares and then test each of them in constant time, see the pseudocode in Algorithm 1. To this end, we first describe a mechanism for checking if $w[i..i+\ell-1] \approx w[i+\ell..i+2\ell-1]$ in constant time. This can be implemented with an LCA query on the order-preserving suffix tree of w, as explained in [3]. However, we need to explain how to construct this structure in $\mathcal{O}(\sigma n)$ time.

Order-preserving suffix tree. Following [3], for a string $w[1..n]$ we define $\mathsf{code}(w)$ as $(\phi(w,1), \phi(w,2), \ldots, \phi(w,n))$, where $\phi(w,i) = (\mathsf{prev}_<(w,i), \mathsf{prev}_=(w,i))$ and $\mathsf{prev}_<(w,i) = |\{k < i : w[k] < w[i]\}|$, $\mathsf{prev}_=(w,i) = |\{k < i : w[k] = w[i]\}|$. We observe that $\mathsf{code}(w) = \mathsf{code}(w')$ if and only if $w \approx w'$. Then, the order-preserving suffix tree of $w[1..n]$ is the compacted trie of all strings of the form $\mathsf{code}(w[i..n])\$$, for $i = 1, 2, \ldots, n$. It is easy to see that $w[i..i+\ell-1] \approx w[i+\ell..i+2\ell-1]$ if and only if the lowest common ancestor of the leaves corresponding to $\mathsf{code}(w[i..n])\$$ and $\mathsf{code}(w[i+\ell..n])\$$ is at string depth at least ℓ. Therefore, assuming that we have already built the order-preserving suffix tree of $w[1..n]$, such a test can be implemented in constant time after $\mathcal{O}(n)$ preprocessing for LCA queries [15]. It remains to explain how to construct the order-preserving suffix tree. We stress that while [3] does provide an efficient $\mathcal{O}(n \log n / \log\log n)$ time construction algorithm (in fact, the full version [4] further improves the time complexity to $\mathcal{O}(n\sqrt{\log n})$), such complexity is incompatible with our goal. Due to the lack of space, the proof is moved to the appendix.

▶ **Lemma 16.** *Given a string $w[1..n]$ over an alphabet of size σ, we can construct its order-preserving suffix tree in $\mathcal{O}(\sigma n)$ time and space.*

The main part of the algorithm is efficiently generating $\mathcal{O}(\sigma n)$ candidates for op-squares. Then, each of them is tested in constant time as explained above, assuming the preprocessing from Lemma 16.

Algorithm 1 Report all occurrences of op-squares in a string $w[1..n]$ over an alphabet of size σ in $\mathcal{O}(\sigma n)$ time.

```
Preprocess w[1..n] for retrieving the characters of any code(w[i..n])$
Construct the order-preserving suffix tree T
Preprocess T for LCA queries
i ← n
while i > 0 do
    s ← w[i..n]
    Retrieve the leftmost occurrences x_1, x_2, ...x_t in s
    foreach x_j that is the smallest or the largest of its group L_k do
        s' ← s[1..2^{k-1}]
        foreach fragment s[y..y + 2^{k-1} − 1] ≈ s' such that x_j ∈ [y, y + 2^{k-1} − 1] do
            Store s[1..2(y − 1)] as a candidate in R[x_j][k][i]
        end
    end
    foreach candidate s[1..2(y − 1)] in R[x_j][k][i] do
        v_1 ← the leaf corresponding to code(w[i..n])$ in T
        v_2 ← the leaf corresponding to code(w[i + (y − 1)..n])$ in T
        if the string depth of LCA_T(v_1, v_2) is at least y − 1 then
            Report s[1..2(y − 1) − 1] as an op-square
        end
    end
    i ← i − 1
end
```

Leftmost occurrences. As in the proof of the $\mathcal{O}(\sigma n)$ upper bound on the number of op-squares, we will consider the suffixes of the input string $w[1..n]$ one-by-one. For $i = n, n-1, \ldots, 1$ in this order, let $s = w[i..n]$ be the currently considered suffix, and $x_1 < x_2 < \ldots < x_t$ be the leftmost occurrences in s. By spending $\mathcal{O}(\sigma)$ time per each suffix, we can assume that the positions $x_1, x_2, \ldots, x_t$ are known, as after moving from $w[i..n]$ to $w[i-1..n]$ we only have to insert the new leftmost occurrence $i-1$ and possibly remove the previous leftmost occurrence i' such that $w[i-1] = w[i']$ (unless $w[i-1]$ has not been seen before), which can be done in $\mathcal{O}(t) = \mathcal{O}(\sigma)$ time. By Proposition 5, every prefix of s that is an op-square can be obtained by choosing two leftmost characters at positions x_q and x_j, where $q < j$, and setting the length of the possible square to be $2(x_j - x_q)$. This gives us $\mathcal{O}(\sigma^2)$ candidates for prefixes that could be op-squares. However, our goal is to generate only $\mathcal{O}(\sigma)$ such candidates. To achieve this goal, we first provide some combinatorial properties in Lemma 17, Lemma 18, and Proposition 19.

Recall that all leftmost occurrences are partitioned into groups $L_0, L_1, \ldots$. Next, we show that it is enough to consider x_j that is the smallest or the largest element in its group.

▶ **Lemma 17.** *Consider an op-square $s[1..2\ell]$. Then there exists $q < j$ such that the leftmost occurrences x_q and x_j satisfy $x_j - x_q = \ell$, $x_j \in [\ell+1, 2\ell]$ and x_j is either the smallest or the largest element of its group.*

The proof is described in the appendix. Figure 10 illustrates the scenario of the lemma.

To generate the candidates, we iterate over all j such that x_j is the smallest or largest element of its group L_k. Consider $q < j$ such that $x_j - x_q = \ell$ and $x_j \in [\ell+1, 2\ell]$ for an op-square $s[1..2\ell]$. Then, because $x_j \in [2^k, 2^{k+1})$, $\ell \geq 2^{k-1}$. To avoid clutter, let $s' = s[1..2^{k-1}]$.

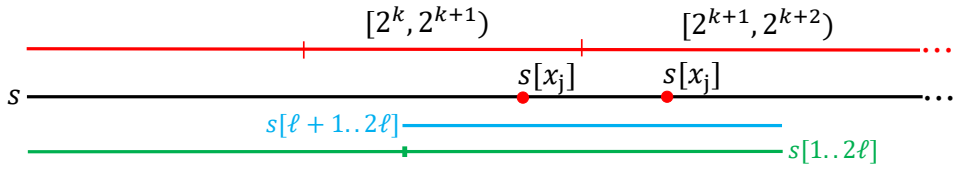

Figure 10 The op-square $s[1..2\ell]$ is colored in green. The red points are the two possibilities for $s[x_j]$. $s[\ell + 1..2\ell]$ is colored in blue. The red line illustrates the ranges.

Because $s[1..2\ell]$ is assumed to be an op-square, we have $s[\ell..\ell + 2^{k-1} - 1] \approx s'$. This suggests the following natural strategy to generate the candidates: we iterate over all fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$, and output $s[1..2(y - 1)]$ as a possible op-square (as explained earlier, each such candidate is then tested in constant time). See Figure 11. We first bound the number of such fragments by $\mathcal{O}(1 + |L_{k-2}|)$, and then explain how to generate them in the same time complexity.

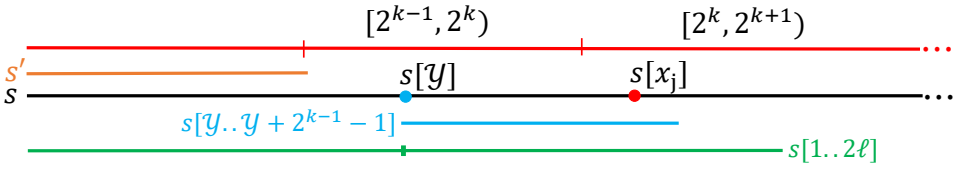

Figure 11 The op-square $s[1..2\ell]$, the fragment $s[y..y + 2^{k-1} - 1]$, and s are colored in green, blue, and black, respectively. s' is colored in orange. The red line illustrates the ranges.

▶ **Lemma 18.** *The number of fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$ is upper bounded by $\mathcal{O}(1 + |L_{k-2}|)$.*

The proof of the lemma relies on showing that ℓ' is an op-border of s' and thus we can define blocks of length $\Delta = 2^{k-1} - \ell'$ in s' and then apply Lemma 10 and Lemma 11 to achieve the desired bound. The full proof is described in the appendix. Hence, for every k such that L_k is non-empty, we generate $\mathcal{O}(1+|L_{k-2}|)$ candidates. The overall number of candidates generated by following the above strategy is $\sum_{k:L_k \neq \emptyset} \mathcal{O}(1 + |L_{k-2}|) = \mathcal{O}(\sigma + \sum_k |L_{k-2}|) = \mathcal{O}(\sigma)$ as promised. It remains to show how to access all fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$ in time proportional to their number.

Accessing candidates. We will solve a more general problem, and show how to ensure that, when considering $s = w[i..n]$, for every leftmost occurrence x_j in s we have access to a list of all fragments $s[y..y+2^{k-1}-1]$ such that $x_j \in [y, y+2^{k-1}-1]$ and $s[1..2^{k-1}-1] \approx s[y..y+2^{k-1}-1]$, where $2^k \leq x_j < 2^{k+1}$. We call this list the *result* for i and x_j.

Recall that $s = w[i..n]$, and we consider $i = n, n-1, \ldots, 1$ in this order. When we consider $s = w[i..n]$, position i becomes a leftmost occurrence and remains to be so until we reach $s = w[\mathsf{prev}_i..n]$ such that $w[i] = w[\mathsf{prev}_i]$ (possibly, it is a leftmost occurrence till the very end of the scan). We can calculate prev_i for every i in $\mathcal{O}(\sigma n)$ time by maintaining a list of leftmost occurrences as described earlier. We say that a position i is k-active at position i' when $i' \in [\mathsf{prev}_i, i]$ and $2^k \leq i - i' + 1 < 2^{k+1}$. We observe that, as we consider longer and longer suffixes of w, position i is first 0-active, then 1-active, and so on until it becomes

k_i-active, and then it is never active again. Further, indices i' such that i is k-active at i' form a contiguous range $[\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]$ (the length of each such range is 2^k, except possibly for $k = k_i$ when it is shorter). The total length of these ranges is small as shown below.

▶ **Proposition 19.** $\sum_{i,k:k\leq k_i} 2^k = \mathcal{O}(\sigma n)$

Proof. For $k \geq 1$ we can upper bound 2^k by $2 \cdot |[\mathsf{begin}_{i,k-1}, \mathsf{end}_{i,k-1}]|$. Then the sum becomes:

$$\sum_{i,k:k\leq k_i} 2^k = n + 2 \cdot \sum_{i,k:1\leq k\leq k_i} |[\mathsf{begin}_{i,k-1}, \mathsf{end}_{i,k-1}]| \leq n + 2 \cdot \sum_{i,k:k\leq k_i} |[\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]|.$$

We observe that every position $i' \in [\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]$ in the suffix $w[i..n]$ corresponds to the relative position $i - i' + 1$ being a leftmost occurrence in the suffix $w[i'..n]$. Because there are at most σ leftmost characters in any suffix $w[i'..n]$, this allows us to upper bound the sum by $\mathcal{O}(\sigma n)$. ◀

Storing candidates. This allows us to physically store the results as follows. For every leftmost occurrence x, we have an array indexed by $k \leq k_i$, denoted $R[x]$. Each entry of this array is an array indexed by $i' \in [\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]$, denoted $R[x][k]$. Finally, each entry of that array, denoted $R[x][k][i']$, is a pointer to a list of ys such that $x \in [y, y + 2^{k-1} - 1]$ and $w[i'..i' + 2^{k-1} - 1] \approx w[y..y + 2^{k-1} - 1]$ (note that it is a pointer to a list and not its physical copy). The arrays allow us to access the result for every x_j, $k \leq k_{x_j}$ and i' in constant time, by retrieving the pointer $R[x_j][k][i']$ (where we first verify that $i' \in [\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]$). The total length of all arrays $R[x]$ is only $\mathcal{O}(\sigma n)$ by Proposition 19. Further, the total length of all lists of occurrences that we need to prepare (assuming that we store every $R[x][k][i]$ as a pointer to such a list and not their physical copies) is also $\mathcal{O}(\sigma n)$ by the following argument. Consider i and $k \leq k_i$. Then, we need a list of positions y such that $i \in [y, y + 2^{k-1} - 1]$ and $w[y..y + 2^{k-1} - 1]$ is order-isomorphic to a specific string s'. Thus, we can partition all positions y such that $i \in [y, y + 2^{k-1} - 1]$ into groups corresponding to order-isomorphic fragments $w[y..y + 2^{k-1} - 1]$, and then store a pointer to the appropriate list (possibly null, if there is no y). The total number of positions y, over all i and $k \leq k_i$, is $\mathcal{O}(\sigma n)$ by Proposition 19, which bounds the total length of all the lists.

Generating candidates. It remains to describe how to efficiently calculate the results. This requires partitioning all fragments $w[y..y + 2^{k-1} - 1]$ such that $i \in [y, y + 2^{k-1} - 1]$ and $k \leq k_i$ into order-isomorphic groups, and finding for every $i, k \leq k_i, i' \in [\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]$ a pointer to the list of fragments $w[y..y + 2^{k-1} - 1]$ with $i \in [y, y + 2^{k-1} - 1]$ that are order-isomorphic to $w[i'..i' + 2^{k-1} - 1]$. Both steps can be implemented with the order-preserving suffix tree that is preprocessed in $\mathcal{O}(n)$ time and space for computing a (deterministic) fingerprint of any $\mathsf{code}(w[x..x + 2^\ell - 1])$ in constant time. Here, a fingerprint is meant as an integer consisting of $\mathcal{O}(\log n)$ bits, denoted $\mathsf{fingerprint}_\ell(x)$, such that $\mathsf{fingerprint}_\ell(x) = \mathsf{fingerprint}_\ell(x')$ iff $\mathsf{code}(w[x..x + 2^\ell - 1]) = \mathsf{code}(w[x'..x' + 2^\ell - 1])$ (or equivalently $w[x..x + 2^\ell - 1] \approx w[x'..x' + 2^\ell - 1]$). We first describe such a mechanism and then provide a more detailed description of how to apply it. The proof of the following lemma directly follows from prior work [11, 12] and is described in the appendix.

▶ **Lemma 20.** *A compacted trie on n leaves can be preprocessed in $\mathcal{O}(n)$ time, so that for any leaf u and integer k we can query in constant time for a $\mathcal{O}(\log n)$-bit fingerprint of the ancestor of u at string depth 2^k.*

We apply Lemma 20 on the order-preserving suffix tree. This allows us to calculate any $\mathsf{fingerprint}_\ell(x)$ with the required properties in constant time. Now consider any i and $k \leq k_i$. We first compute $\mathsf{fingerprint}_{k-1}(y)$ for every y such that $i \in [y, y + 2^{k-1} - 1]$. This takes $\mathcal{O}(2^k)$ time. Next, we compute $\mathsf{fingerprint}_{k-1}(i')$ for every $i' \in [\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]$, also in $\mathcal{O}(2^k)$ time because $|[\mathsf{begin}_{i,k}, \mathsf{end}_{i,k}]| \leq 2^{k-1}$. We sort all fingerprints and partition them into groups corresponding to order-isomorphic fragments. We need to implement this step in $\mathcal{O}(2^k)$ time as well. To this end, we observe that we need to sort $\mathcal{O}(2^k)$ integers consisting of $\mathcal{O}(\log n)$ bits, which can be done with radix sort in $\mathcal{O}(2^k + n)$ time. To avoid paying $\mathcal{O}(n)$ for each i and $k \leq k_i$, we observe that this is an offline problem, and all sets corresponding to different i and $k \leq k_i$ can be sorted together. In more detail, we sort tuples of the form $(i, k_i, \mathsf{fingerprint}_\ell(y), y)$ and $(i, k_i, \mathsf{fingerprint}_\ell(i'), i')$. The total number of all tuples is $\mathcal{O}(\sigma n)$ by Lemma 19 and, as each of them can be treated as an integer consisting of $\mathcal{O}(\log n)$ bits, they can be sorted in $\mathcal{O}(\sigma n + n) = \mathcal{O}(\sigma n)$ time. Then, we extract the results for each i and $k \leq k_i$ from the output. For each i and $k \leq k_i$, we consider every group of equal fingerprints. From each group, we first create a list containing all positions y corresponding to $\mathsf{fingerprint}_{k-1}(y)$ belonging to the group. Then, for every $\mathsf{fingerprint}_{k-1}(i')$ belonging to the group we store a pointer to this list. Overall, this takes $\mathcal{O}(\sigma n)$ time and allows us to compute all the results in the same time complexity.

6 Open Problems

An interesting follow-up to our results is first bounding the number of order-preserving squares that are not order-isomorphic, and then designing an algorithm that reports all such squares. In addition, investigating the bounds for parameterized squares is of interest. Moreover, we are not aware of an algorithm reporting parameterized squares in a string, hence, designing such an algorithm is desired.

References

1 S. Brlek and S. Li. On the number of squares in a finite word. *arXiv*, 2022. `arXiv:2204.10204`.
2 R. Cole and R. Hariharan. Faster suffix tree construction with missing suffix links. *STOC*, pages 407–415, 2000.
3 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń. Order-preserving incomplete suffix trees and order-preserving indexes. *SPIRE*, 8214:84–95, 2013.
4 M. Crochemore, C. S. Iliopoulos, T. Kociumaka, M. Kubica, A. Langiu, S. P. Pissis, J. Radoszewski, W. Rytter, and T. Waleń. Order-preserving indexing. *Theoretical Computer Science*, 638:122–135, 2016.
5 M. Crochemore and W. Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13:405–425, 1995.
6 L. J. Cummings and W. F. Smyth. Weak repetitions in strings. *J. Combinatorial Math. Combinatorial Comput*, 24:33–48, 1997.
7 A. Deza, F. Franek, and A. Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015.
8 P. Erdős. Some unsolved problems. *Magy. Tud. Akad. Mat. Kut. Intéz. Közl.*, 6:221–254, 1961.
9 A. A. Evdokimov. Strongly asymmetric sequences generated by a finite number of symbols. *Dokl. Akad. Nauk SSSR*, 179(6):1268–1271, 1968.
10 A. S. Fraenkel and J. Simpson. How many squares can a string contain? *Combinatorial Theory, Series A*, 82(1):112–120, 1998.
11 P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. *ESA*, 6942:421–432, 2011.

12 P. Gawrychowski. Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic. *arXive*, 2011. arXiv:1104.4203.

13 G. Gourdel, T. Kociumaka, J. Radoszewski, W. Rytter, A. Shur, and T. Waleń. String periods in the order-preserving model. *Information and Computation*, 270:104463, 2020.

14 D. Gusfield and J. Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Computer and System Sciences*, 69(4):525–546, 2004.

15 D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

16 M. Huova, J. Karhumäki, and A. Saarela. Problems in between words and abelian words: k-abelian avoidability. *Theoretical Computer Science*, 454:172–177, 2012.

17 L. Ilie. A simple proof that a word of length n has at most $2n$ distinct squares. *Journal of Combinatorial Theory, Series A*, 112(1):163–164, 2005.

18 L. Ilie. A note on the number of squares in a word. *Theoretical Computer Science*, 380(3):373–376, 2007.

19 V. Keränen. Abelian squares are avoidable on 4 letters. *Automata, Languages and Programming*, pages 41–52, 1992.

20 T. Kociumaka, J. Radoszewski, W. Rytter, and T. Waleń. Maximum number of distinct and nonequivalent nonstandard squares in a word. *Theoretical Computer Science*, 648(C):84–95, 2016.

21 T. Kociumaka, J. Radoszewski, and B. Wiśniewski. Subquadratic-time algorithms for abelian stringology problems. *Mathematical Aspects of Computer and Information Sciences*, pages 320–334, 2016.

22 M. Kubica, T. Kulczyński, J. Radoszewski, W. Rytter, and T. Waleń. A linear time algorithm for consecutive permutation pattern matching. *Information Processing Letters*, 113(12):430–433, 2013.

23 N. H. Lam. On the number of squares in a string. *AdvOL-Report 2*, 2013.

24 F. Manea and S. Seki. Square-density increasing mappings. *In 10th WORDS*, 9304:160–169, 2015.

25 Y. Matsuoka, T. Aoki, S. Inenaga, H. Bannai, and M. Takeda. Generalized pattern matching and periodicity under substring consistent equivalence relations. *Theoretical Computer Science*, 656:225–233, 2016.

26 P. A. B. Pleasants. Non-repetitive sequences. *Mathematical Proceedings of the Cambridge Philosophical Society*, 68(2):267–274, 1970.

27 A. Thierry. A proof that a word of length n has less than $1.5n$ distinct squares. *arXiv*, 2020. arXiv:2001.02996.

28 A. Thue. Über unendliche Zeichenreihen. *Norske Vid Selsk. Skr. I Mat-Nat Kl.(Christiana)*, 7:1–22, 1906.

A Missing Proofs

▶ **Lemma 16.** *Given a string $w[1..n]$ over an alphabet of size σ, we can construct its order-preserving suffix tree in $\mathcal{O}(\sigma n)$ time and space.*

Proof. As explained in [3], the order-preserving suffix tree of $w[1..n]$ can be constructed using the general framework of Cole and Hariharan [2] for constructing a suffix tree for a quasi-suffix collection of strings $w_1, w_2, \ldots, w_n$. The running time of their algorithm is $\mathcal{O}(n)$ with almost inverse exponential failure probability, assuming that one can access the j-th character of any w_i in constant time. The mechanism for accessing the j-th character of w_i is called the *character oracle*. In this particular application, the string $w_i = \phi(w[i..n])\$$. We will first describe how to implement a constant-time character oracle for such strings, and then explain why randomization is not needed in our setting.

We need to implement a new *character oracle* that returns $\phi(w[i..n], j)$, for any i, j, in constant time after $\mathcal{O}(\sigma n)$ time and space preprocessing. This requires being able to calculate $\mathsf{prev}_<(w[i..n], j)$ and $\mathsf{prev}_=(w[i..n], j)$ in constant time. To this end, we define a two-dimensional array $\mathsf{cnt}[i, x] = |\{k : k < i, w[k] < x\}|$, for $i = 0, 1, \ldots, n$ and $x = 0, 1, \ldots, \sigma$. All entries in this array can be computed in $\mathcal{O}(\sigma n)$ total time and space. Then, we can calculate any $\mathsf{prev}_<(w[i..n], j)$ and $\mathsf{prev}_=(w[i..n], j)$ as follows:

$$\begin{aligned}\mathsf{prev}_<(w[i..n], j) =& \mathsf{cnt}[i + j - 2, w[i + j - 1]] - \mathsf{cnt}[i - 1, w[i + j - 1]]\\ \mathsf{prev}_=(w[i..n], j) =& (\mathsf{cnt}[i + j - 2, w[i + j - 1]] - \mathsf{cnt}[i + j - 2, w[i + j - 1] - 1])\\ &- (\mathsf{cnt}[i - 1, w[i + j - 1]] - \mathsf{cnt}[i - 1, w[i + j - 1] - 1]).\end{aligned}$$

To remove randomization, we observe that its only source in the algorithm of Cole and Hariharan is the need to maintain, for each explicit node of the current tree, a dictionary indexed by the next character on an outgoing edge. If we could show that there are at most $\mathcal{O}(\sigma)$ such edges, then the dictionary could be implemented as a simple list, increasing the construction time to $\mathcal{O}(\sigma n)$, which is within our claimed bound.

Consider a non-leaf node v of the current tree. It corresponds to a proper prefix of some $\mathsf{code}(w[i..n])\$$, which by the definition of $\mathsf{code}(.)$ is equal to $\mathsf{code}(w[i..j])$, for some j. Let $c_1 < c_2 < \ldots < c_k$ be the distinct characters of $w[i..j]$, and denote by occ_x the number of occurrences of c_x in $w[i..j]$. Now consider an edge outgoing from v, and let $\mathsf{code}(w[i'..j' + 1])$ correspond to the first node (implicit or explicit) after v there. We know that $\mathsf{code}(w[i'..j']) = \mathsf{code}(w[i..j])$, so the distinct characters of $w[i'..j']$ are $c'_1 < c'_2 < \ldots < c'_k$ with occ_x being the number of occurrences of c'_x in $w[i'..j']$. Then, we analyze the possible values of $(\mathsf{prev}_<(w[i'..j' + 1], j' - i' + 2), \mathsf{prev}_=(w[i'..j' + 1], j' - i' + 2))$, that is, the first character on the considered edge. The first number is always equal to $\sum_{y=1}^{x-1} c_y$, for some $x \in [1, k + 1]$. Then, the second number is either 0 or occ_x. Thus, overall we have only $2k \leq 2\sigma$ possible first characters, which bounds the degree of any v by $\mathcal{O}(\sigma)$. ◀

▶ **Lemma 17.** *Consider an op-square $s[1..2\ell]$. Then there exists $q < j$ such that the leftmost occurrences x_q and x_j satisfy $x_j - x_q = \ell$, $x_j \in [\ell + 1, 2\ell]$ and x_j is either the smallest or the largest element of its group.*

Proof. By Proposition 5, we know that there is a leftmost character in $s[\ell + 1..2\ell]$. Choose the largest k such that $2^k \leq \ell$ (so $2^{k+1} > \ell$). Consider two ranges $[2^k, 2^{k+1})$ and $[2^{k+1}, 2^{k+2})$ corresponding to groups L_k and L_{k+1}, respectively. Because $2^k \leq \ell$ and $2^{k+1} > \ell$, we have $2^k < \ell + 1$, $2^{k+1} \in [\ell + 1, 2\ell]$ and $2\ell < 2^{k+2}$. Consequently, the fragment $s[\ell + 1..2\ell]$ can be represented as the concatenation of a suffix of $s[2^k..2^{k+1})$ and a prefix of $s[2^{k+1}..2^{k+2})$. The leftmost occurrence that falls within $s[\ell + 1..2\ell]$ belongs to the suffix or the prefix. See Figure 10. If it falls within the suffix, the largest element of L_k belongs to $[\ell + 1, 2\ell]$. If it falls within the prefix, the smallest element of L_{k+1} belongs to $[\ell + 1, 2\ell]$. Let $x_j \in [\ell + 1, 2\ell]$ be the corresponding leftmost occurrence. To complete the proof we need to establish that there exists $q < j$ such that $x_j - x_q = \ell$. The character $s[x_j]$ is distinct from all $s[1], s[2], \ldots, s[x_j - 1]$, and by $s[1..\ell] \approx s[\ell + 1..2\ell]$ we obtain that $s[x_j - \ell]$ is distinct from all $s[1], s[2], \ldots, s[x_j - \ell - 1]$. Thus, the position $x_j - \ell$ is a leftmost occurrence, hence $x_j - \ell = x_q$ for some $q < j$ as required. ◀

▶ **Lemma 18.** *The number of fragments $s[y..y + 2^{k-1} - 1]$ such that $x_j \in [y, y + 2^{k-1} - 1]$ and $s' \approx s[y..y + 2^{k-1} - 1]$ is upper bounded by $\mathcal{O}(1 + |L_{k-2}|)$.*

Proof. Consider all such fragments $s[y_1..y_1 + 2^{k-1} - 1], s[y_2..y_2 + 2^{k-1} - 1], \ldots, s[y_t..y_t + 2^{k-1} - 1]$. Because $x_j \in [y_z, y_z + 2^{k-1} - 1]$ for every $z = 1, 2, \ldots, t$, either $t = 1$ or by the pigeonhole principle there exists z such that $y_{z+1} - y_z < 2^{k-1}/(t-1)$. If $t = 1$ then we are done. Otherwise, let $\ell' = |s[y_{z+1}..y_z + 2^{k-1} - 1]|$. By assumption, $s' \approx s[y_z..y_z + 2^{k-1} - 1]$ and $s' \approx s[y_{z+1}..y_{z+1} + 2^{k-1} - 1]$, so by the transitivity of $\approx$ also $s[y_z..y_z + 2^{k-1} - 1] \approx s[y_{z+1}..y_{z+1} + 2^{k-1} - 1]$. We conclude that $s'[1..\ell'] \approx s'[y_{z+1} - y_z + 1..2^{k-1}]$, or in other words ℓ' is an op-border of s'. Let $b_1, b_2, \ldots, b_f, b_{f+1}$ be the blocks defined by ℓ' in $s' = s[1..|s'|] = w[i..i + |s'| - 1]$, where each block is of length $\Delta = 2^{k-1} - \ell'$. See Figure 12. Recall that x_j is a leftmost occurrence in $s = w[i..n]$, and by the definition of y_z and y_{z+1} we have $x_j \in [y_{z+1}, y_z + 2^{k-1} - 1]$. Then, by $s'[1..\ell'] \approx s'[y_{z+1} - y_z + 1..2^{k-1}]$ we obtain that $x_j - y_z + 1 \in [y_{z+1} - y_z + 1, 2^{k-1}]$ is also a leftmost occurrence in $s = w[i..n]$. Hence, we have a leftmost occurrence in block b_j, for some $j \geq 2$. This allows us to apply Lemma 10 and then Lemma 11 to conclude that there is a leftmost occurrence in every block $b_1, b_2, \ldots, b_f$. We calculate a lower bound on how many of these leftmost occurrences fall within the range $[2^{k-2}, 2^{k-1})$:

$$\begin{aligned}\left\lfloor \frac{2^{k-2}}{\Delta} \right\rfloor - 1 &> \frac{2^{k-2}}{2^{k-1} - \ell'} - 2 \\ &= \frac{2^{k-2}}{2^{k-1} - (y_z + 2^{k-1} - y_{z+1})} - 2 = \frac{2^{k-2}}{y_{z+1} - y_z} - 2 \\ &> \frac{2^{k-2}}{2^{k-1}/(t-1)} - 2 = (t-5)/2.\end{aligned}$$

For $t < 6$, we are done as the number of fragments is $\mathcal{O}(1)$. Otherwise, we obtain that $|L_{k-2}| \geq (t-5)/2 \geq t/12$, thus $t = \mathcal{O}(1 + |L_{k-2}|)$ always holds as claimed. ◀

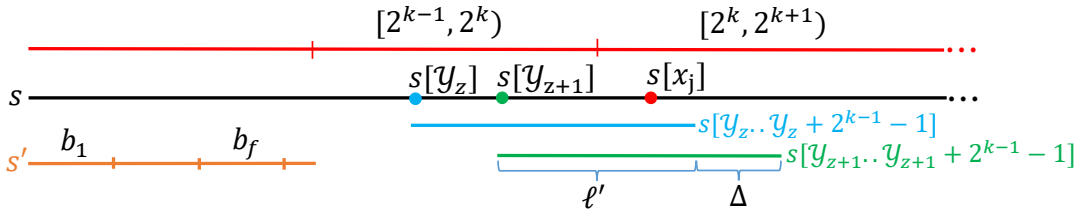

Figure 12 s, s', $s[y_z..y_z + 2^{k-1} - 1]$, and $s[y_{z+1}..y_{z+1} + 2^{k-1} - 1]$ are colored in black, orange, blue, and green, respectively. The red line illustrates the ranges.

▶ **Lemma 20.** *A compacted trie on n leaves can be preprocessed in $\mathcal{O}(n)$ time, so that for any leaf u and integer k we can query in constant time for a $\mathcal{O}(\log n)$-bit fingerprint of the ancestor of u at string depth 2^k.*

Proof. This follows by applying the method used to solve the substring fingerprint problem mentioned in [11, Lemma 14]. Following the description in the full version [12, Lemma 12], a compacted trie on n leaves can be preprocessed in $\mathcal{O}(n)$ time so that we can locate the (implicit or explicit) node corresponding to the ancestor at string depth 2^k of a given leaf in constant time. If the sought node is implicit (and does not explicitly exist in the compacted trie) we retrieve the edge that contains it. Next, if the node is explicit then we return its identifier. If the node is implicit then we return the identifier of the edge that contains it. Thus, the required range of identifiers is $[2n]$. ◀