University of Warsaw

Faculty of Mathematics, Informatics and Mechanics

Wiktor Zuba

Efficient enumerations in words

PhD dissertation

Supervisor:
prof. dr hab. Wojciech Rytter
Institute of Informatics
University of Warsaw

June 2021

Author's declaration:
I hereby declare that this dissertation is my own work.

June 8, 2021

Wiktor Zuba
....
Wiktor Zuba

Supervisor's declaration:
The dissertation is ready to be reviewed

June 8, 2021

W Rytter
....
Wojciech Rytter

Abstract

The dissertation presented here consists of a set of results in text processing algorithms from four different scientific publications. The papers focus on algorithms and data structures that allow efficient enumeration and counting of certain classes of factors. Most of the results are based on different variations of the same key property - regularity of the set of occurrences of factors of specific types.

In "Efficient Representation and Counting of Antipower Factors in Words" we focused on a class of k-antipowers (words consisting of k pairwise different segments of the same length). The results of the paper consist of effective algorithms for enumerating all occurrences, counting all occurrences and counting different k-antipower factors in a string, working in $O(nk \log k + \text{output})$, $O(nk \log k)$ and $O(nk^4 \log k \log n)$ time, respectively.

The paper "Counting Distinct Patterns in Internal Dictionary Matching" deals with a problem in which we are given an input word T and its internal dictionary D. We want to answer questions of the form: "How many distinct words from D are the factors of $T[i\mathinner{.\,.}j]$?". We have shown few algorithms and data structures, that allows us to do that effectively. The algorithms differ in exploited properties and complexities (each one can be more effective from the other depending on the ratio between T and D).

In "The Number of Repetitions in 2D-Strings" we focused on the extensions of the notions of squares and runs to the case of two-dimensional strings. We gave new bounds on the maximal number of their occurrences in strings and effective algorithms for enumerating them.

In the fourth paper titled "Efficient Enumeration of Distinct Factors Using Package Representations", we proposed a new compact representation of string factors. It allows us to obtain new effective algorithms for enumeration and counting of distinct factors it contains. In particular it allowed us to obtain a new simple algorithm for finding distinct squares and a faster algorithm for finding all distinct antipowers in a string (improving one of the results from the first of enclosed papers).

Keywords: algorithm, data structure, enumeration, counting, factor, subword, antipower, runs, two-dimensional string, internal dictionary

Efektywne zliczanie w słowach.

Streszczenie

Przedstawiona rozprawa stanowi zbiór wyników prac na temat algorytmów przetwarzających dane tekstowe. Prace skupiają się na zaprezentowaniu algorytmów i struktur danych pozwalających na efektywne znajdowanie i zliczanie podsłów należących do konkretnych klas podsłów. Innym elementem wspólnym poniższych prac jest to, że wykorzystują one regularności w słowach związane z wystąpieniami podsłów szczególnego typu.

W pracy "Efficient Representation and Counting of Antipower Factors in Words" zajęliśmy się klasą k-antypotęg (słów składających się z k parami różnych członów o tej samej długości). Rezultatem pracy są efektywne algorytmy wypisywania wszystkich wystąpień, zliczania wszystkich wystąpień oraz zliczania różnych k-antypotęg w słowie, działające w czasach odpowiednio $O(nk \log k +$ wynik$)$, $O(nk \log k)$ i $O(nk^4 \log k \log n)$.

Praca "Counting Distinct Patterns in Internal Dictionary Matching" przedstawia problem w którym mając dane słowo wejściowe T oraz słownik jego podsłów D chcielibyśmy móc efektywnie odpowiadać na pytania "Ile różnych słów ze słownika D jest podsłowami słowa $T[i \mathinner{.\,.} j]$?". Przedstawiliśmy kilka algorytmów/struktur danych, które pozwalają na rozwiązanie tego problemu. Algorytmy różnią się wykorzystywanymi założeniami oraz złożonościami (każdy może być lepszy od pozostałych w zależności od zastosowania, oraz stosunku wielkości T i D).

W pracy "The Number of Repetitions in 2D-Strings" zajęliśmy się uogólnieniami pojęć kwadratów i maksymalnych powtórzeń na teksty dwuwymiarowe. Podaliśmy nowe ograniczenia na maksymalną ich ilość w tekstach, oraz efektywne algorytmy ich wyznaczania.

Na końcu przedstawiam pracę "Efficient Enumeration of Distinct Factors Using Package Representations", w której wprowadziliśmy nowy, skompresowany sposób reprezentacji wystąpień wielu podsłów. Pozwala ona na otrzymanie nowych, efektywnych algorytmów wyznaczania i zliczania różnych słów przez nią reprezentowanych. W szczególności pozwoliło nam to na otrzymanie nowego prostego algorytmu wyznaczania kwadratów i szybszego algorytmu wyznaczania antypotęg (poprawiając jeden z wyników pierwszej z przedstawionych prac).

Słowa kluczowe: algorytm, struktura danych, wyliczanie, zliczanie, podsłowo, antypotęga, maksymalne powtórzenie, słowo dwuwymiarowe, słownik wewnętrzny

Contents

Chapter 1

Extended abstract

1.1 Introduction

1.1.1 Motivation

The study of regular words and especially of algorithms finding and counting such words in texts is one of the most popular areas of text algorithms. There are many algorithms, which work differently on periodic and aperiodic parts of strings, as such parts may show distinct properties. For example, high periodicity of a word results in a very good compression rate - we can describe it using only its period and length. On the other hand it may make searching (for it or in it) much more difficult, as an aperiodic word of length m may occur in a word of length n at most $2\frac{n}{m}$ times. That is not true if the word is periodic - for instance a word *aaaaaa* appears in *aaaaaaaaaaaaaaaa* $n - m + 1 = 11$ times. The number of such (distinct) structures in a word can by itself serve as a good measure of the words regularity or complexity.

The presented dissertation is composed of publications in which (together with my coauthors) we have concerned a few classes of such regular (or irregular in a specific sense) subwords. In each of the papers we have shown new, effective algorithms for enumerating and counting (all occurrences or all distinct) factors from a specific class.

1.1.2 Contents

The dissertation is composed of four articles written together with my coauthors during the course of my doctoral studies at the Faculty of Mathematics, Informatics and Mechanics of the University of Warsaw:

1. Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, Wiktor Zuba: *Efficient Representation and Counting of Antipower Factors in Words.* LATA 2019: 421-433

The paper is included in the version accepted for publication in the Special Issue of Information & Computation derived from LATA 2019 conference.

2. Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, Wiktor Zuba: *Counting Distinct Patterns in Internal Dictionary Matching.* CPM 2020: 8:1-8:15

3. Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Wiktor Zuba: *The Number of Repetitions in 2D-Strings.* ESA 2020: 32:1-32:18

4. Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Wiktor Zuba: *Efficient Enumeration of Distinct Factors Using Package Representations.* SPIRE 2020: 247-261

All those papers contain original results in text algorithms. They were written by the authors representing the University of Warsaw together with scientists from King's College London (Panagiotis Charalampopoulos had a long term stay in Warsaw) and Tomasz Kociumaka, who changed his affiliation from the University of Warsaw to Bar-Ilan University in the meantime.

In section 1.2 of this abstract I introduce the basic notions - regular classes of words and factors, which are used throughout the presented papers.

In section 1.3 I show a detailed overview of our results together with sketches of methods used to obtain them.

In section 1.4 I list the publications I coauthored during the course of my doctoral studies, which however are not included as a part of this dissertation.

Chapter 2 contains a polish version of this extended abstract, while chapters 3, 4, 5 and 6 contain the papers included in the dissertation. The bibliography at the end contains references to papers cited in this abstract.

1.2 Preliminaries

To present the results we need to define the string regularities they use or search for first. Many of them are used by more then just one of the papers included in this dissertation.

Definition 1.1. Let us assume that $x = y_0 y_1 \cdots y_{k-1}$ where $k \geqslant 2$ and y_i are words of length d. We say that:

- x is a k-**power** if all y_i's are the same (x is a **square** if in addition $k = 2$);
- x is a k-**antipower** if all y_i's are pairwise distinct;
- x is a **weak** k-**power** if it is not a k-antipower, that is, if $y_i = y_j$ for some $i \neq j$;
- x is a **gapped square** if $y_0 = y_{k-1}$.

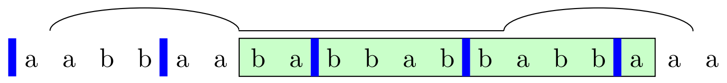


Figure 1.1: Regularities in standard strings. A run (with the period equal to 3) is depicted by a green background. Blue lines mark an occurrence of a 4-antipower. Notice that the words of length 16 starting one or two positions later are not 4-antipowers as they are gapped squares generated by a maximal 3-gapped repeat.

Definition 1.2. We say that:

- p is a **period** of a word w if $w[i] = w[i+p]$ for all $i \in [0\,..\,|w|-p)$ (by the period of a word we usually mean the shortest one);
- a fragment $w[i\,..\,j]$ is a **run** (a maximal repetition) if its length is equal to at least a double of its shortest period and it cannot be extended by a one position to the left or to the right without breaking the period;
- a fragment $w[i\,..\,j]$ is a **maximal α-gapped repeat** (for $\alpha \geqslant 1$) if it is of a form uvu for $p = |uv| \leqslant \alpha|u|$ and it cannot be extended by a one position to the left or to the right without breaking the period.

Definition 1.3. For two-dimensional texts we define repetitions analogously.

- A **tandem** is a two-dimensional text of even width, in which its left and right halves are equal as texts (extension of square to the case where instead of letters we have columns of letters).
- A **quartic** is a two-dimensional text Q of even height and width, such that Q and $Q^{\perp}$ are tandems (it is composed of four identical texts in a 2×2 grid).
- A **2D-run** is a $i \times j$ fragment of a text such that its vertical period is smaller or equal to $\frac{i}{2}$ and its horizontal period is smaller or equal to $\frac{j}{2}$. Moreover it cannot be extended by a row or a column from any side (if such exist in the text) without breaking any of those periods.

By a square generated by a run I mean a square fully contained in it with length equal to exactly double of its period. All such squares can be easily obtained from a run, and every square is generated by a run. The generation of gapped-squares by runs and maximal α-gapped repeats and the generation of quartics by 2D-runs work analogously.

A square is primitively rooted if its root (half) is not a k-power for any $k > 1$. Analogously a quartic is primitively rooted if its root (quarter) is not a k-power for any $k > 1$ neither horizontally nor diagonally.

Definition 1.4. By an **internal dictionary** we mean a set of factors of a given word represented by the starting and ending positions of a single occurrence of each factor.

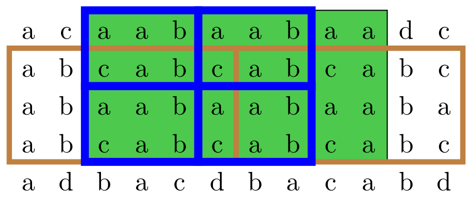


Figure 1.2: Different types of repetitions in 2D-strings. A tandem is marked by brown rims, a quartic by the blue ones, while the green background depicts a 2D-run.

1.3 Presentation of the main results

1.3.1 Efficient Representation and Counting of Antipower Factors in Words

The paper presented here was presented on LATA 2019 conference and published in conference proceedings. Extended version of that paper was later accepted for publication (it is not published yet) in the Special Issue of Information & Computation journal derived from that conference. It is included in this dissertation in the mentioned extended version (in chapter 3).

Introduction

Badkobeh et al. in [5] presented a lower and upper bound $\Theta(n^2/k)$ for the maximal number of occurrences of k-antipowers in a string. Furthermore they presented an $\mathcal{O}(n^2/k)$ time algorithm that finds all those occurrences. Due to the lower bound on the maximal size of the output the algorithm is optimal in the pessimistic case. It does not mean however, that nothing more in this field can be done.

In our publication we have focused on the problems that are not directly influenced by those bounds - counting of antipowers and enumerating them with complexity parameterized by the size of the output (more effective if a word contains a small number of antipowers).

More formally - we presented algorithms, which for a word of length n count all the occurrences of k-antipower factors in $\mathcal{O}(nk \log k)$ time, find all those occurrences in $\mathcal{O}(nk \log k + \text{output})$ time, and count all distinct k-antipowers in a word in $\mathcal{O}(nk^4 \log k \log n)$ time. We also presented a data structure that allows us to check efficiently if a given subword is a k-antipower.

In (the most common) case when k is small in comparison to the length of the word (smaller than $\sqrt{n}$) our algorithm has a better asymptotic performance time than the one previously known.

Weak powers and a compact representation

Since antipowers itself are very irregular it is not that easy to effectively find them. Due to that, we focused on computing a representation of all the other words, that is of all weak k-powers. Every weak k-power on the other hand is a consequence of the existence of a gapped square (possibly for a smaller k, but certainly for the same d).

Observation 1.1.

(a) Each (gapped) square is generated by a run with (not necessarily shortest) period $(q+1) \cdot d$ or a maximal $(q+1)$-gapped repeat with a period equal to $(q+1) \cdot d$ for some $q \leqslant k-2$.

(b) Each run and each maximal α-gapped repeat generate a single interval of positions in which (gapped) squares of a given length start, moreover this interval can be computed in constant time.

(c) In a word of length n there are at most $\mathcal{O}(n)$ general runs and their representation can be computed in $\mathcal{O}(n)$ time ([6]).

(d) There are only $\mathcal{O}(n\alpha)$ maximal α-gapped repeats in a word and their representation can be computed in $\mathcal{O}(n\alpha)$ time ([11, 12]).

With the use of those properties we can compute the representation of weak k-powers for all d's as $\mathcal{O}(nk)$ interval chains (set of intervals of the same lengths and beginnings forming an arithmetic progression) in total. Then, using geometric methods we can determine their set-theoretic sum in $\mathcal{O}(nk \log k)$ total time. This immediately gives us the number of all weak k-powers, hence also of all k-antipower factors. Instead of counting the size of the complement of this set we can enumerate through all of the positions it does not contain and thus obtain an algorithm that finds all k-antipowers in $\mathcal{O}(nk \log k + \text{output})$ time.

Antipower queries

Paper [5] included descriptions of two data structures, that after being constructed answer queries "Is a subword $w[i \mathinner{.\,.} j]$ a k-antipower?". The first structure requires $\mathcal{O}(n)$ space and answers those questions in $\mathcal{O}(k)$ time, while the other answers them in a constant time, but requires storing information of $\mathcal{O}(n^2)$ size.

In our paper we have shown a new structure parameterized by a value $r \in [1, n]$. After being built in $\mathcal{O}(n^2/r)$ time our structure stores information of size $\mathcal{O}(n^2/r)$ and answers such questions in $\mathcal{O}(r)$ time.

If $r < k$ then we know that the base of the power d equals at most $n/r > n/k$. To answer the questions we store for each $d \leqslant n/r$ separately a data structure based on the range minimum queries structure, that is of $\mathcal{O}(n)$ size and answers questions in constant time. Otherwise ($r \geqslant k$) to obtain the right complexities it is enough to just use the mentioned data structure from [5].

Counting distinct k-antipowers

In the journal version of the paper we have added a new result - an algorithm for counting distinct k-antipowers in a word.

Due to a potentially large number of all occurrences of antipowers, the methods enumerating through those cannot obtain a satisfactory computation time.

To obtain such a result we made use of the connection between antipowers and weak powers once again.

The number of distinct k-antipower factors can be obtained as a subtraction of two other values. The first one is equal to the number of (all) distinct factors of the word of length divisible by k. We can compute it in $\mathcal{O}(n)$ time with the use of a suffix tree.

The second value needed is the number of distinct weak k-powers. To obtain that we have strengthened the definition of weak powers, so that each such subword can be generated only in one way. Then with the use of a reduction to a graph problem we have acquired an algorithm computing that number in $\mathcal{O}(nk^4 \log k \log n)$ total time.

1.3.2 Counting Distinct Patterns in Internal Dictionary Matching

In [8] my coauthors introduced a problem of pattern matching with internal dictionary - for a given word T and its internal dictionary D they presented a data structure that allows effective answering of questions for subwords $T[i\mathinner{.\,.}j]$:

- does $T[i\mathinner{.\,.}j]$ contain a word from the dictionary?
- return all subwords of $T[i\mathinner{.\,.}j]$ that belong to the dictionary
- return all distinct factors from D contained in $T[i\mathinner{.\,.}j]$
- count all subwords of $T[i\mathinner{.\,.}j]$ that belong to the dictionary
- count the number of distinct factors from D contained in $T[i\mathinner{.\,.}j]$

The structure requires $\tilde{\mathcal{O}}(n+d)$ space and construction time (where $n = |T|$ denotes the length of the word, $d = |D|$ the number of factors contained in the dictionary, and $\tilde{\mathcal{O}}$ is the asymptotic notation ignoring polilogarithmic factors). It answers the first four questions in $\tilde{\mathcal{O}}(1 + \text{output})$ time.

Unfortunately for the last question only a data structure answering those question $\mathcal{O}(\log n)$ approximately was shown.

Space	**Preprocessing time**	**Query time**	**Variant**
$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(1)$	2-approximation
$\tilde{\mathcal{O}}(n^2/m^2+d)$	$\tilde{\mathcal{O}}(n^2/m+d)$	$\tilde{\mathcal{O}}(m)$	exact
$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(m)$	exact
$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(\log n)$	$D =$ squares, exact

Table 1.1: Our results for Count Distinct queries ($m \in [1, n]$ is an arbitrarily chosen parameter).

2-approximation

To improve the result from the previous paper we have designed a data structure which stores all the exact results for the base words - that is for all subwords of length $\lfloor (1+\delta)^p \rfloor$ for all natural p's and a fixed value $\delta = \frac{1}{9}$. To obtain the result for any given subword $T[i\,..\,j]$ we make use of two maximal length base words $T[i\,..\,i']$ and $T[j'\,..\,j]$ contained in $T[i\,..\,j]$ to obtain its division into three parts $F_1F_2F_3$ ($F_1F_2 = T[i\,..\,i']$, $F_2F_3 = T[j'\,..\,j]$).

The result is obtained by counting all the factors from the dictionary, which have an occurrence starting in F_1 and ending in F_3, which at the same time occurs in neither of the two base words. We can do that effectively thanks to a large ratio of the length of F_2 to the lengths of F_1 and F_3. To the obtained value we add the results stored for the two base words, which however results in only a 2-approximate result.

There are $\mathcal{O}(n \log n)$ base words in a word of length n. The exact values for them can be counted in $\mathcal{O}(n \log^{1+\epsilon} n + d)$ total time (for any constant $\epsilon > 0$) with the use of the property that we can efficiently update the result after a single position shift. The structure used to count the words described earlier costs us additional $\mathcal{O}(n + d \log n)$ space and $\mathcal{O}(n \log n / \log \log n + d \log^{3/2} n)$ construction time. It answers the questions in $\mathcal{O}(\log^2 n / \log \log n)$ time.

Exact counting

Instead of the base words we can focus on storing the results for all the words of the form $T[c_1m+1\,..\,c_2m]$ for a chosen m and all the values $c_1 < c_2$ to obtain a data structure of $\mathcal{O}(n^2/m^2+n+d)$ size construable in $\mathcal{O}((n^2 \log^\epsilon n)/m + n\sqrt{\log n}+d)$ time. We can update the result obtained after extending a subword by a one position in $\mathcal{O}(\log^\epsilon n)$ time, and to obtain the exact result for any chosen factor we only require up to $\mathcal{O}(m)$ such operations (we extend one of the words for which the result is stored).

We can also approach the problem differently, namely, we can inspect the structure of the dictionary.

If the dictionary does not contain a set of k distinct prefixes of the same word, then at any given position only up to $k-1$ dictionary words can start. In this case we can store for each of the dictionary words the set of positions in T, where such a word occurs. We can construct such a data structure of $\mathcal{O}(nk \log n)$ size in $\mathcal{O}(nk \log n)$ time. In return, it allows us to respond to our queries in $\mathcal{O}(\log n)$ time.

To obtain a dictionary of such a form we can simply remove from it all such large groups of prefixes (building new dictionaries). There are at most d/k such groups, and for each of those we can answer the queries in $\mathcal{O}(\log^\epsilon n)$ time (for any $\epsilon > 0$) using a bounded LCP structure ([14]) of $\mathcal{O}(n\sqrt{\log n})$ size.

For $k = \lceil \frac{d}{m} \rceil$ this gives us a query time equal to $\mathcal{O}(m \log^\epsilon n + \log n)$.

Counting distinct squares factors

For the popular case, where we want to count the number of distinct square factors in a given subword we have described a yet different algorithm.

In a word of length n there can be up to $\mathcal{O}(n^2)$ squares (only $\mathcal{O}(n)$ distinct ones), however thanks to the runs (only $\mathcal{O}(n)$ such can appear in the word, and all such can be computed in $\mathcal{O}(n)$ time) we can distinguish $m = \mathcal{O}(n \log n)$ occurrences which really affect our results. With the use of a geometrical data structure of $\mathcal{O}(m \log m)$ size and construction time from [13] we can answer the queries for the number of distinct squares in a subword in $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ time.

1.3.3 The Number of Repetitions in 2D-Strings

In the paper from chapter 5 we have focused on the repetitions in two-dimensional texts. The table 1.2 contains a summary of the current knowledge about the maximal possible number of 2D-runs, tandems and quartics (distinct or all occurrences of primitively rooted ones) in a two-dimensional string. It also shows the time complexity of the fastest known algorithms which find those structures in a text. Notice, that even though the maximal number of 2D-runs in a string was previously studied the gap between the known lower and upper bounds was linear. In our paper we have reduced this gap to a polilogarithmic one.

	Bounds on the number in an $n \times n$ string	**Computation time**
2D-runs	$\Omega(n^2)$,$\mathcal{O}(n^3)$[2] $\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$	$\mathcal{O}(n^2 \log n+\text{output})$[2] $\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$
Occurrences of primitively rooted quartics	$\Theta(n^2 \log^2 n)$ [3]	$\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$
Distinct quartics	$\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$	$\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$
Occurrences of primitively rooted tandems	$\Theta(n^3 \log n)$[3]	$\mathcal{O}(n^3 \log n)$[4]
Distinct tandems	$\Omega(n^2), \mathcal{O}(n^4)$(trivial) $\boldsymbol{\Theta(n^3)}$	$\boldsymbol{\mathcal{O}(n^3)}$

Table 1.2: Overview of the previous knowledge and of our results (written in bold).

2D-runs

To obtain our upper bound on the number of 2D-runs we made use of a lemma from [1], to assign to every such 2D-run a maximal horizontal repetition (no condition on vertical periodicity or maximality) of height equal to a power of two (largest possible not exceeding the height of the 2D-run). Such repetition meets three important conditions as well - its upper left or lower left corner equals the respective corner of the 2D-run, its horizontal period is equal to the horizontal period of the 2D-run, and its width equals at least the width of the 2D-run (see figure 1.3)

With the use of those properties, of the periodicity lemma, and of the three squares lemma we proved that the horizontal repetitions occupying the rows from i to $i + 2^k - 1$ can be assigned to at most $\mathcal{O}(n \log n)$ distinct 2D-runs in total. By multiplying this value by the number of feasible choices of i and k we

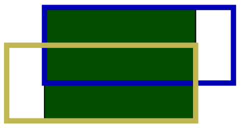


Figure 1.3: Assignment of a maximal horizontal repetition of height 2^k to a 2D-run.

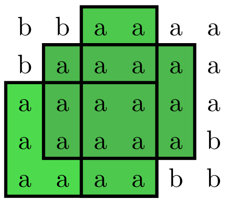


Figure 1.4: Many overlapping 2D-runs

obtain an upper bound on the number of 2D-runs in an $n \times n$ string equal to $\mathcal{O}(n^2 \log^2 n)$.

Our bound immediately shows that the algorithm finding all the 2D-runs in a 2D-string from [2] works in $\mathcal{O}(n^2 \log^2 n)$ time (their algorithm has time complexity parameterized by the size of the output).

Primitively rooted quartics

In standard strings we can easily extract all the occurrences of primitively rooted squares from the runs - it is enough to take all the subwords of the runs of length equal to the double of their period.

In 2D situation it is somewhat similar - each primitively rooted quartic is a rectangle of width equal to the double of the horizontal period and height equal to the double of the vertical period of a 2D-run, fully contained in it. However, while in one dimension two runs with the same period cannot have a big enough overlap to generate the same square, in 2D many such overlaps can easily appear (see figure 1.4), hence simple adaptation of such a one-dimensional algorithm would result in reporting some of the quartics multiple times.

This problem can be solved with the use of a sweeping line technique ([7]), which allows us to count set-theoretic sums of many families of rectangles on an $n \times n$ grid in $\mathcal{O}(n + r + \text{output})$ total time (r denotes the number of all the rectangles).

By dividing all the 2D-runs according to their periods, for each such group of runs we count the set-theoretic sum of the rectangles containing the starting positions of quartics generated by each 2D-run. This way we obtain the desired algorithm.

Due to our bound on r from the previous paragraph (the number of 2D-runs), and the $\mathcal{O}(n^2 \log^2 n)$ bound on the number of the occurrences of primitively rooted quartics from [3] our algorithm runs in $\mathcal{O}(n^2 \log^2 n)$ total time (optimal in the pessimistic case by the mentioned bound).

Distinct quartics

In an $n \times n$ unary string there are $\Theta(n^4)$ occurrences of general quartics. Due to that in the pessimistic case a trivial algorithm finding all such occurrences in an optimal time. A more interesting problems are the one of bounding the maximal number of distinct quartics (differing as texts and not only by their positioning) and the one of enumerating through all of them.

[3] already gave us a bound on the number of distinct primitively rooted quartics. All the other ones are composed of many adjacent occurrences of the primitively rooted ones forming a larger rectangle. In other words, while a primitively rooted quartic is constructed from occurrences of a primitive text W arranged in a 2×2 grid, in the case of the other ones it is a $2k \times 2l$ grid ($k > 1$ or $l > 1$). We divide such quartics into the thin ones ($k = 1$ or $l = 1$), and the thick ones (both $k, l > 1$).

Then again we perform a yet another division - we group quartics for which their primitive string W has size in $[2^a, 2^{a+1}) \times [2^b, 2^{b+1})$ for the same values $a, b \leqslant \log n$.

Due to the periodicity lemma and the three squares lemma any position can be the top left corner of the rightmost occurrence (same quartic does not occur anywhere further to the right) of at most four distinct thin quartics from a single group (it works analogously to a proof of a bound of the number of distinct squares in a standard string [10]).

In the case of the thick quartics the multitude of the grid points in which an occurrence of the W string begins in a large power of W allows us to uniquely assign such points to every smaller power of W. Moreover, due to the periodicity and the three squares lemmas such points cannot belong to a grid for a power of a different string W' which belongs to the same group.

Due to this assignments of the points in $[1, n] \times [1, n]$ to the quartics and due to the fact, that there are only $\log^2 n$ choices of a and b we obtain an upper bound on the number of distinct quartics of size $\mathcal{O}(n^2 \log^2 n)$.

To compute all the distinct quartics in a text we use our previous algorithm to find all the occurrences of primitively rooted quartics, and then we group them by their roots.

Since other quartics are constructed from the occurrences of the primitive ones (with the same root) we can distinguish the top left corners of all such occurrences, and then find the largest possible grids they form.

This problem can be solved with the sweep line approach to obtain an algorithm working in $\mathcal{O}(n^2 \log^2 n)$ time.

Distinct tandems

In the same paper we also considered the related problem of finding all the distinct tandems in a string (as with the quartics, there can be $\Theta(n^4)$ occurrences of all the tandems in a 2D-string).

In this problem we can easily identify the tandems occupying the rows from i to j with squares in a standard string by assigning single letters to all the columns of height $j-i+1$. In a standard word of length n there can be up to $\Theta(n)$ distinct squares, hence the same bound applies to the number of distinct tandems occupying the distinguished rows. Multiplying this value by a number of possible choices of i and j we obtain $\mathcal{O}(n^3)$ as the upper bound on the number of distinct tandems.

$\Omega(n^3)$ as a lower bound can be shown just as easy - it is obtained for a text in which every row is a different unary word.

The assignment of standard letters to the columns of height equal to $j-i+1$ can be performed constructively. Thanks to that, we can obtain an $\mathcal{O}(n^3)$ time algorithm for finding all distinct tandems in a 2D-string, with the use of an algorithm finding distinct squares in a standard string. Due to our lower bound on the maximal size of the output our algorithm is optimal in the pessimistic case.

1.3.4 Efficient Enumeration of Distinct Factors Using Package Representations

In the last of the described papers we introduced a new representation of sets of subwords, which in many cases gives a compact representation, and at the same time allows effective operations on the set.

By a package we mean a set of subwords of the same length, whose starting positions form an interval. Our representation is composed of many such packages represented as triples (starting position, length of words, length of interval). Our greatest interest in this representation lies in the property, that the set of the distinct factors it represents can be found efficiently.

More formally, we are interested in the set

$$\mathsf{Factors}(\mathcal{F}) = \{T[j\mathinner{..}j+l) : j \in [i, i+k] \text{ and } (i,l,k) \in \mathcal{F}\}.$$

There are many families of subwords, which can be effectively represented in this way - for example k-powers (when each package corresponds to a run) and k-antipowers (when we can use the representation from section 1.3.1). The obtained representations have size and computation time equal to $\mathcal{O}(n)$ and $\mathcal{O}(nk^2)$ respectively (each interval chain can be represented by at most k ordinary intervals).

To enumerate through $\mathsf{Factors}(\mathcal{F})$, and compute $|\mathsf{Factors}(\mathcal{F})|$ most effectively we consider two cases. The simpler one, called special, is the one in which if a factor belongs to $\mathsf{Factors}(\mathcal{F})$, then each of its occurrences must be contained in a package from $\mathcal{F}$ (it cannot be that some occurrences are represented, and some are not).

In this case for a word of length n and $\mathcal{F}$ composed of m packages we make use of a data structure for the longest previous factors ([9]) to obtain efficient enumeration and counting of the result in $\mathcal{O}(n+m+\text{output})$ and $\mathcal{O}(n+m)$ time respectively. In the case of aforementioned distinct k-powers we obtain a new, simple algorithm computing them in $\mathcal{O}(n)$ time, and in the case of distinct k-antipowers we obtain a new algorithm working in $\mathcal{O}(nk^2 + \text{output})$ time. This means, that the newly obtained algorithm has a better performance time than the algorithm described in section 1.3.1, which counts those antipower factors in $\mathcal{O}(nk^4 \log k \log n)$ time, and is much more complicated.

The second case, called general does not require this extra assumption, which however makes solving it much more difficult. To do that we adapt a method we used in the paper about k-antipowers (counting of distinct such factors) to obtain algorithms finding and counting $\mathsf{Factors}(\mathcal{F})$ in $\mathcal{O}(n \log^2 n + m \log n + \text{output})$ and $\mathcal{O}(n \log^2 n + m \log n)$ time respectively.

1.4 Other publications

During the course of my doctoral studies several other papers coauthored by me have been published. I decided not to include them in this dissertation, since they are not directly connected to its topic.

1. Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Faster Recovery of Approximate Periods over Edit Distance.* SPIRE 2018: 233-240

2. Wojciech Rytter, Wiktor Zuba: *Syntactic View of Sigma-Tau Generation of Permutations.* LATA 2019: 447-459
   Extended version of the paper was accepted for publication in Theoretical Computer Science journal.

3. Mai Alzamel, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Quasi-Linear-Time Algorithm for Longest Common Circular Factor.* CPM 2019: 25:1-25:14

4. Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Circular Pattern Matching with k Mismatches.* FCT 2019: 213-228

5. Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Weighted Shortest Common Supersequence Problem Revisited.* SPIRE 2019: 221-238

6. Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Shortest Covers of All Cyclic Shifts of a String.* WALCOM 2020: 69-80

7. Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Tomasz Walen, Wiktor Zuba: *Unary Words Have the Smallest Levenshtein k-Neighbourhoods.* CPM 2020: 10:1-10:12

8. Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Internal Quasiperiod Queries.* SPIRE 2020: 60-75

9. Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Shortest covers of all cyclic shifts of a string.* Theor. Comput. Sci. 866: 70-81 (2021)

10. Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Circular pattern matching with k mismatches.* J. Comput. Syst. Sci. 115: 73-85 (2021)

Chapter 2

Autoreferat

2.1 Wstęp

2.1.1 Motywacja

Badanie regularnych fragmentów słów, w szczególności ich znajdowanie i zliczanie jest jedną z najbardziej popularnych części algorytmiki tekstowej. Wiele algorytmów inaczej działa na fragmentach słów które są okresowe niż na takich w których ta własność nie występuje, jako że fragmenty te cechują często inne własności. Przykładowo duża okresowość słowa pozwala na łatwą jego kompresję - opisanie go przy pomocy samego okresu i długości, z drugiej strony może to utrudniać wyszukiwanie takiego słowa - jeśli słowo długości m jest nieokresowe, to w tekście długości n może się pojawić najwyżej $2\frac{n}{m}$ razy, nie jest to jednak prawdą w przypadku słów okresowych (na przykład słowo $aaaaaa$ pojawia się w słowie $aaaaaaaaaaaaaaaa$ $n-m+1 = 11$ razy). Sama ilość takich (różnych) struktur w słowie może służyć za miarę jego regularności lub skomplikowania.

W pracach wchodzących w skład doktoratu wraz z współautorami zajęliśmy się kilkoma klasami podsłów regularnych lub takich, które można przedstawić w zwarty sposób. W każdej z prac przedstawiliśmy nowe, efektywne algorytmy wypisywania i zliczania wszystkich wystąpień, lub wszystkich różnych podsłów pewnej klasy.

2.1.2 Skład pracy

Rozprawa składa się z czterech artykułów, napisanych podczas moich studiów doktorskich na Wydziale Matematyki, Informatyki i Mechaniki Uniwersytetu Warszawskiego we współpracy z innymi autorami:

1. Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, Wiktor Zuba: *Efficient Representation and Counting of Antipower Factors in Words.* LATA 2019: 421-433

W wersji przyjętej do druku w Special Issue konferencji LATA 2019, które ma być wydane w czasopiśmie Information & Computation.

2. Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, Wiktor Zuba: *Counting Distinct Patterns in Internal Dictionary Matching.* CPM 2020: 8:1-8:15

3. Panagiotis Charalampopoulos, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Wiktor Zuba: *The Number of Repetitions in 2D-Strings.* ESA 2020: 32:1-32:18

4. Panagiotis Charalampopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Tomasz Waleń, Wiktor Zuba: *Efficient Enumeration of Distinct Factors Using Package Representations.* SPIRE 2020: 247-261

Wszystkie prace zawierają oryginalne wyniki z algorytmiki na słowach. Prace zostały napisane przez autorów reprezentujących Uniwersytet Warszawski we współpracy z naukowcami z King's College London (Panagiotis Charalampopoulos przebywał przez dłuższy czas w Warszawie) oraz Tomaszem Kociumaką, który w międzyczasie zmienił afiliację z Uniwersytetu Warszawskiego na Uniwersytet Bar-Ilan.

W sekcji 2.2 niniejszego autoreferatu przedstawiam podstawowe pojęcia - definicje regularnych klas słów i podsłów, które są wykorzystywane w przedstawianych pracach.

W sekcji 2.3 pokazuję dokładny przegląd wyników załączonych prac oraz szkice metod użytych do ich uzyskania.

W sekcji 2.4 wymieniam publikacje których jestem współautorem i które zostały opublikowane podczas moich studiów doktoranckich, które jednak nie zostały włączone w skład tej rozprawy.

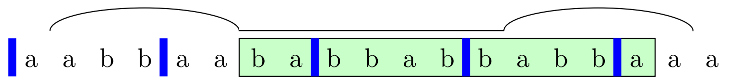
W rozdziale 1 zamieszczone jest to samo streszczenie w języku angielskim, zaś w rozdziałach 3, 4, 5 oraz 6 kolejne prace wchodzące w skład rozprawy. Bibliografia na końcu zawiera referencje użyte w niniejszym streszczeniu.

2.2 Preliminaria

Aby przedstawić wyniki załączonych prac należy zacząć od przedstawienia typów regularności w słowach, przez nie wykorzystywanych. Wiele z tych pojęć pojawia się w więcej niż jednej z tych prac.

Definicja 2.1. Niech $x = y_0 y_1 \cdots y_{k-1}$ dla $k \geqslant 2$ i słów y_i o tej samej długości d. Wtedy mówimy, że:

- x jest **k-potęgą** jeśli wszystkie y_i są takie same (x jest **kwadratem** jeśli dodatkowo $k = 2$);
- x jest **k-antypotęgą** jeśli wszystkie y_i są parami różne;



Rysunek 2.1: Regularności w słowach jednowymiarowych. Zielonym tłem zaznaczono maksymalne powtórzenie (o okresie 3). Niebieskie linie pokazują wystąpienie 4-antypotęgi. Zauważmy, że jedną i dwie pozycje dalej nie występują 4-antypotęgi o tej samej podstawie ponieważ te wyrazy długości 16 są przerywanymi kwadratami generowanymi przez zaznaczone nad słowem maksymalne 3-przerywane wystąpienie.

- x jest **słabą k-potęgą** jeśli nie jest k-antypotęgą, to jest jeśli $y_i = y_j$ dla pewnych $i \neq j$;
- x jest **przerywanym kwadratem** jeśli $y_0 = y_{k-1}$.

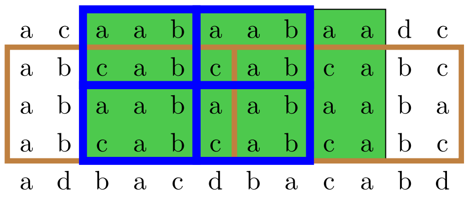
Definicja 2.2. Mówimy, że:

- p jest **okresem** słowa w jeśli $w[i] = w[i+p]$ dla wszystkich $i \in [0\,..\,|w|-p)$ (przez okres słowa najczęściej rozumiemy jego najkrótszy okres);
- fragment słowa $w[i\,..\,j]$ jest **maksymalnym powtórzeniem** jeśli jego długość jest równa co najmniej dwóm jego najkrótszym okresom i nie można go rozszerzyć o jedną pozycję w lewo ani w prawo bez zmiany tego okresu;
- fragment słowa $w[i\,..\,j]$ jest **maksymalnym α-przerywanym powtórzeniem** (dla $\alpha \geqslant 1$) jeśli jest postaci uvu dla $p = |uv| \leqslant \alpha|u|$ i nie można go rozszerzyć o jedną pozycję w lewo ani w prawo bez zmiany okresu.

Definicja 2.3. Dla tekstów dwuwymiarowych definiujemy analogiczne powtórzenia.

- **Tandemem** nazywamy dwuwymiarowy tekst o parzystej długości, w którym lewa i prawa połowa tekstu są takie same (uogólnienie kwadratu na przypadek gdzie pojedynczą literą jest cała kolumna).
- **Kwartyką** nazywamy dwuwymiarowy tekst K o parzystej długości i szerokości, taki że zarówno K jak i $K^{\perp}$ są tandemami (cztery identyczne teksty ułożone w kratę 2×2).
- **maksymalnym 2D-powtórzeniem** nazywamy fragment tekstu dwuwymiarowego o wymiarach $i \times j$, taki że jego okres poziomy jest wielkości co najwyżej $\frac{j}{2}$ a pionowy co najwyżej $\frac{i}{2}$. Co więcej powiększenie tego fragmentu o dodatkową kolumnę lub wiersz z dowolnej strony (o ile takie istnieją w całym tekście) skutkowałoby zmianą tych okresów.

Przez kwadrat generowany przez maksymalne powtórzenie rozumiem kwadrat zawarty w tym powtórzeniu o długości równej dokładnie dwóm okresom powtórzenia.



Rysunek 2.2: Różne rodzaje powtórzeń w tekstach dwuwymiarowych. Kolorem brązowym zaznaczono tandem, niebieskim kwartykę, zaś zielonym tłem maksymalne 2D-powtórzenie.

Wszystkie takie kwadraty można łatwo wyznaczyć z powtórzenia i każdy kwadrat jest generowany przez jakieś maksymalne powtórzenie. Analogicznie wygląda sytuacja generowania przerywanych kwadratów przez powtórzenia i α-przerywane powtórzenia oraz generowania kwartyk przez 2D-powtórzenia.

Kwadrat ma pierwiastek pierwotny gdy jego pierwiastek (połowa) nie jest k-potęgą dla żadnego $k > 1$. Analogicznie działa to w przypadku kwartyk, dla których jego pierwiastek (ćwiartka) nie jest k-potęgą dla $k > 1$ ani poziomo ani pionowo.

Definicja 2.4. **Słownikiem wewnętrznym** nazywamy zbiór pozycji początkowych i końcowych wybranych podsłów zadanego słowa.

2.3 Przegląd wyników

2.3.1 Efficient Representation and Counting of Antipower Factors in Words

Zaprezentowana tutaj praca została zaprezentowana na konferencji LATA 2019 i opublikowana w sprawozdaniu z konferencji. Rozszerzona wersja tej pracy została później przyjęta do druku w Special Issue tej konferencji, wydawanym w w czasopiśmie Information & Computation. Pracę właśnie w tej formie (przyjętej do druku, lecz jeszcze nie opublikowanej) załączyłem do niniejszej rozprawy (w rozdziale 3).

Wstęp

W pracy [5] przedstawione zostało górne i dolne ograniczenie $\Theta(n^2/k)$ na maksymalną liczbę wystąpień k-antypotęg w słowie. Praca ta podała również algorytm wyznaczający te wystąpienia w czasie $\mathcal{O}(n^2/k)$. Ze względu na ograniczenia na wielkość wyjścia algorytm ten jest optymalny w przypadku pesymistycznym. Nie oznacza to jednak, że nic w tej dziedzinie nie można poprawić.

W naszej publikacji skupiliśmy się na problemach które nie podlegają temu ograniczeniu - zliczaniu antypotęg oraz ich wyznaczaniu ze złożonością zależną od rozmiaru wyjścia (efektywniejszą gdy rozważane słowo zawiera mało antypotęg).

Pisząc dokładniej - zaprezentowaliśmy algorytmy, które dla słowa długości n zliczają wszystkie podsłowa będące k-antypotęgami w czasie $\mathcal{O}(nk \log k)$, wyznaczają wszystkie te podsłowa w czasie $\mathcal{O}(nk \log k+$rozmiar wyniku), oraz wyznaczają liczbę różnych k-antypotęg w słowie w czasie $\mathcal{O}(nk^4 \log k \log n)$. Zaprezentowaliśmy również nową strukturę danych pozwalającą na szybkie sprawdzenie, czy dane podsłowo jest k-antypotęgą.

Jak widać w (najczęściej wykorzystywanym) przypadku gdy rozważane k jest małe w stosunku do długości słowa (mniejsze niż $\sqrt{n}$) nasz algorytm uzyskuje lepszy asymptotyczny czas działania od poprzednio znanych.

Słabe potęgi i zwarta reprezentacja

Ze względu na nieregularną naturę antypotęg efektywne ich znajdowanie nie jest proste. Dlatego też, skupiliśmy się na wyznaczeniu reprezentacji tych podsłów, które nimi nie są, czyli słabych k-potęg. Jednocześnie każda słaba k-potęga jest konsekwencją wystąpienia przerywanego kwadratu (być może dla mniejszego k, lecz tego samego d).

Obserwacja 2.1.

(a) Każdy (przerywany) kwadrat jest generowany przez maksymalne powtórzenie o (nie koniecznie najkrótszym) okresie $(q + 1) \cdot d$ lub maksymalne $(q + 1)$-przerywane powtórzenie o okresie $(q + 1) \cdot d$ dla $q \leqslant k - 2$.

(b) Każde maksymalne powtórzenie i każde przerywane powtórzenie generuje pojedynczy przedział pozycji w którym pojawiają się (przerywane) kwadraty określonej długości, co więcej możemy ten przedział wyznaczyć w czasie stałym.

(c) Wszystkich ogólnych maksymalnych powtórzeń w słowie o długości n jest $\mathcal{O}(n)$ i ich reprezentację można wyliczyć w czasie $\mathcal{O}(n)$ ([6]).

(d) Wszystkich maksymalnych α-przerywanych powtórzeń w słowie jest $\mathcal{O}(n\alpha)$ i ich reprezentację można wyliczyć w czasie $\mathcal{O}(n\alpha)$ ([11, 12]).

Korzystając z tych własności potrafimy wyliczyć reprezentację słabych k-potęg dla wszystkich d w postaci łącznie $\mathcal{O}(nk)$ łańcuchów przedziałów (zbiór przedziałów o tej samej długości i początkach tworzących postęp arytmetyczny). Następnie korzystając z metod geometrycznych potrafimy wyznaczyć sumę teoriomnogościową tych łańcuchów w łącznym czasie $\mathcal{O}(nk \log k)$. Daje nam to bezpośrednio ilość słabych k-potęg dla każdego d, a więc i ilość k-antypotęg. Zamiast liczyć wielkość zbioru pozycji pokrytych przez reprezentację wystąpień słabych k-potęg możemy wyznaczyć wszystkie pozycje niepokryte i otrzymać algorytm wyznaczający k-antypotęgi w czasie $\mathcal{O}(nk \log k + \text{wynik})$.

Pytania o podsłowa

W pracy [5] zaprezentowano również dwie struktury danych które po zbudowaniu pozwalają odpowiadać na pytania "Czy dane podsłowo $w[i\,..\,j]$ jest k-antypotęgą?". Pierwsza struktura o rozmiarze $\mathcal{O}(n)$ odpowiada na pytania w czasie $\mathcal{O}(k)$, zaś druga robi to w czasie stałym, lecz wymaga zapamiętania informacji rozmiaru $\mathcal{O}(n^2)$.

W naszej pracy przedstawiliśmy nową strukturę, parametryzowaną przez wartość $r \in [1, n]$. Po zbudowaniu w czasie $\mathcal{O}(n^2/r)$ struktura ma rozmiar $\mathcal{O}(n^2/r)$ i odpowiada na pytania w czasie $\mathcal{O}(r)$.

Jeśli $r < k$, to wiemy, że podstawa potęgi d wynosi co najwyżej $n/r > n/k$. Aby odpowiadać na pytania dla każdego $d \leqslant n/r$ osobno budujemy strukturę danych opartą na range minimum queries, o rozmiarze $\mathcal{O}(n)$ i czasie zapytania $\mathcal{O}(1)$. W komplementarnym przypadku aby uzyskać pożądany rezultat wystarczy skorzystać ze wspomnianej struktury z pracy [5].

Zliczanie różnych k-antypotęg

W wersji do czasopisma dodaliśmy kolejny wynik - zliczanie różnych k-antypotęg w słowie.

Ze względu na potencjalną dużą ilość wszystkich wystąpień antypotęg metody wykorzystujące ich wyznaczenie nie pozwalają na osiągnięcie satysfakcjonującej szybkości. Aby uzyskać taki wynik ponownie skorzystaliśmy z powiązania antypotęg z bardziej regularnymi słabymi potęgami.

Liczbę różnych k-antypotęg możemy uzyskać odejmując od siebie dwie inne wartości. Pierwszą wartością jest ilość (wszystkich) różnych podsłów o długości podzielnej przez k. Możemy ją łatwo uzyskać w czasie $\mathcal{O}(n)$ przy użyciu drzewa sufiksowego.

Drugą potrzebną wartością jest liczba różnych słabych k-potęg. Aby ją uzyskać najpierw wzmocniliśmy definicję słabych potęg, tak aby każda była generowana na dokładnie jeden sposób. Następnie przez redukcję do problemu grafowego udało nam się wykonać liczenie różnych takich konstrukcji w łącznym czasie $\mathcal{O}(nk^4 \log k \log n)$.

2.3.2 Counting Distinct Patterns in Internal Dictionary Matching

W pracy [8] moi współautorzy wprowadzili problem wyszukiwania przy użyciu słownika wewnętrznego - dla zadanego słowa T i jego słownika wewnętrznego D przedstawili konstrukcję struktury danych pozwalającą efektywnie odpowiadać na pytania dla danego podsłowa $T[i\,..\,j]$:

- czy $T[i\,..\,j]$ zawiera jakieś słowo ze słownika?
- zwróć wszystkie podsłowa słowa $T[i\,..\,j]$ które należą do słownika
- zwróć wszystkie słowa ze słownika D które są zawarte w $T[i\,..\,j]$
- zwróć liczbę podsłów $T[i\,..\,j]$ które należą do słownika
- zwróć liczbę różnych słów ze słownika D które są zawarte w $T[i\,..\,j]$

Przedstawiona struktura danych miała rozmiar i czas konstrukcji $\tilde{\mathcal{O}}(n+d)$ (gdzie $n = |T|$ oznacza długość słowa, $d = |D|$ liczbę słów w słowniku, zaś $\tilde{\mathcal{O}}$ notację asymptotyczną pomijającą czynniki polilogarytmiczne) i odpowiadającą na pierwsze cztery pytania w czasie $\tilde{\mathcal{O}}(1 + \text{wynik})$.

W przypadku ostatniego pytania tylko $\mathcal{O}(\log n)$ aproksymacja wyniku została zaprezentowana.

Rozmiar	**Czas konstrukcji**	**Czas zapytania**	**Wariant**
$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(1)$	2-aproksymacja
$\tilde{\mathcal{O}}(n^2/m^2+d)$	$\tilde{\mathcal{O}}(n^2/m+d)$	$\tilde{\mathcal{O}}(m)$	dokładny
$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(m)$	dokładny
$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(\log n)$	$D=$kwadraty, dokładny

Tabela 2.1: Nasze rezultaty dla zapytań policz różne ($m \in [1,n]$ jest dowolnie wybranym parametrem).

2-aproksymacja

Aby poprawić wynik z poprzedniej pracy stworzyliśmy strukturę danych, która pamięta wyniki dla podsłów bazowych, czyli wszystkich podsłów długości $\lfloor (1+\delta)^p \rfloor$ dla wszystkich liczb naturalnych p i ustalonego $\delta = \frac{1}{9}$. Aby otrzymać wynik dla dowolnego podsłowa $T[i\,..\,j]$ korzystamy z maksymalnych słów bazowych postaci $T[i\,..\,i']$ oraz $T[j'\,..\,j]$ zawartych w $T[i\,..\,j]$ i otrzymujemy jego podział na trzy części $F_1F_2F_3$ ($F_1F_2 = T[i\,..\,i']$, $F_2F_3 = T[j'\,..\,j]$).

Wynik uzyskujemy poprzez zliczenie słów ze słownika, które mają wystąpienie zaczynające się w F_1 i kończące w F_3, jednak nie występujące w żadnym z wymienionych słów bazowych, co udaje nam się zrobić efektywnie dzięki dużemu stosunkowi długości F_2 do F_1 i F_3. Do tego dodajemy ilości słów występujących w obu słowach bazowych, co jednak skutkuje uzyskaniem jedynie 2-aproksymacji ze względu na możliwe powtórzenia.

Wszystkich słów bazowych jest $\mathcal{O}(n\log n)$, zaś wartości dla nich potrafimy policzyć w łącznym czasie $\mathcal{O}(n\log^{1+\epsilon} n+d)$ (dla dowolnego stałego $\epsilon > 0$) wykorzystując to, że wynik po przesunięciu o jedną pozycję można zaktualizować niewielkim kosztem. Struktura danych używana do zliczania słów opisanych w poprzednim akapicie kosztuje nas dodatkowe $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$ czasu preprocessingu oraz zajmuje $\mathcal{O}(n+d\log n)$ pamięci. Pozwala ona odpowiadać na pytania w czasie $\mathcal{O}(\log^2 n/\log\log n)$.

Obliczanie dokładne

Jeśli zamiast dla słów bazowych zapamiętamy wyniki dla wszystkich podsłów postaci $T[c_1m+1\,..\,c_2m]$ dla wybranego m oraz wszystkich $c_1 < c_2$ otrzymamy strukturę danych o rozmiarze $\mathcal{O}(n^2/m^2+n+d)$ obliczaną w czasie $\mathcal{O}((n^2\log^\epsilon n)/m + n\sqrt{\log n} + d)$. Rozszerzając takie słowo po jednej pozycji możemy uzyskać wynik dla dowolnego słowa przy użyciu $\mathcal{O}(m)$ operacji o koszcie $\mathcal{O}(\log^\epsilon n)$.

Innym podejściem do problemu jest zagłębienie się w strukturę słownika.

Jeśli słownik nie zawiera zbioru k różnych prefiksów tego samego słowa, to na dowolnej pozycji może się zaczynać co najwyżej $k-1$ różnych słów z tego słownika. W takim przypadku możemy dla każdego słowa ze słownika zapamiętać wszystkie pozycje na których ono występuje. Daje nam to strukturę danych wielkości $\mathcal{O}(nk \log n)$, obliczalną w czasie $\mathcal{O}(nk \log n)$ i pozwalającą odpowiadać na pytania w czasie $\mathcal{O}(\log n)$.

Aby uzyskać słownik w takiej postaci wydzielamy z niego (jako nowe słowniki) słowa niespełniające tej własności, to jest duże grupy słów będących prefiksami tego samego słowa. Takich grup (o wielkości co najmniej k) jest co najwyżej d/k, zaś dla każdej takiej grupy potrafimy efektywnie odpowiadać na pytania przy pomocy struktury do pytań o ograniczone LCP ([14]) o rozmiarze $\mathcal{O}(n\sqrt{\log n})$ w czasie $\mathcal{O}(\log^\epsilon n)$ dla dowolnego $\epsilon > 0$.

Dla $k = \lceil \frac{d}{m} \rceil$ daje nam to w sumie czas zapytania wielkości $\mathcal{O}(m \log^\epsilon n + \log n)$.

Słownik kwadratów

Dla popularnego przypadku obliczania liczby kwadratów w podsłowie opisaliśmy odrębny algorytm. Dzięki szczególnym własnościom takich podsłów możemy otrzymać lepsze wyniki.

Wszystkich wystąpień kwadratów w słowie może być $\mathcal{O}(n^2)$ (różnych kwadratów tylko $\mathcal{O}(n)$), jednak dzięki skorzystaniu z maksymalnych powtórzeń, których w całym słowie jest $\mathcal{O}(n)$ (i które można obliczyć w czasie $\mathcal{O}(n)$) potrafimy wyznaczyć $m = \mathcal{O}(n \log n)$ istotnych wystąpień kwadratów, które są wystarczająco reprezentatywne dla naszych potrzeb. Dzięki skorzystaniu z geometrycznej struktury danych o rozmiarze i czasie konstrukcji $\mathcal{O}(m \log m)$ z pracy [13] potrafimy odpowiedzieć na pytanie o liczbę różnych takich słów w podsłowie w czasie $\mathcal{O}(\log m) = \mathcal{O}(\log n)$.

2.3.3 The Number of Repetitions in 2D-Strings

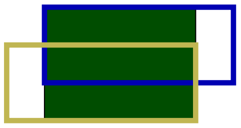
W pracy zamieszczonej w rozdziale 5 zajęliśmy się problemem powtórzeń w tekstach dwuwymiarowych. W tabeli 2.2 przedstawiłem podsumowanie naszych wyników oraz dotychczasowej wiedzy na temat możliwej ilości maksymalnych 2D-powtórzeń jak i maksymalnej liczby różnych tandemów i kwartyk oraz wszystkich wystąpień tych o pierwotnym pierwiastku. Przedstawiłem również czasy algorytmów pozwalających wyznaczyć te struktury w tekście. Warto zauważyć, że ilość 2D-powtórzeń była już wcześniej badana, jednak granica pomiędzy dolnym i górnym ograniczeniem pozostawała liniowa. W przedstawianej pracy udało nam się zredukować tą wielkość do polilogarytmicznej.

maksymalne 2D-powtórzenia

Aby uzyskać ograniczenie na liczbę maksymalnych 2D-powtórzeń skorzystaliśmy z lematu z pracy [1] aby przypisać każde takie powtórzenie do maksymalnego powtórzenia poziomego (brak wymagania pionowej okresowości i maksymalności) o wysokości będącej potęgą dwójki (największą możliwą, nie większą niż wysokość 2D-powtórzenia). Takie powtórzenie poziome spełnia również trzy istotne warunki - jego lewy górny lub lewy dolny narożnik jest równy odpowiadającemu narożnikowi 2D-powtórzenia, jego okres jest równy okresowi poziomemu

	Ograniczenia na ilość w tekście $n \times n$	**Czas wyznaczania**
maksymalne 2D-powtórzenia	$\Omega(n^2)$,$\mathcal{O}(n^3)$[2] $\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$	$\mathcal{O}(n^2 \log n$+wynik)[2] $\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$
Wystąpienia kwartyk o pierwotnych pierwiastkach	$\Theta(n^2 \log^2 n)$ [3]	$\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$
Różne kwartyki	$\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$	$\boldsymbol{\mathcal{O}(n^2 \log^2 n)}$
Wystąpienia tandemów o pierwotnych pierwiastkach	$\Theta(n^3 \log n)$[3]	$\mathcal{O}(n^3 \log n)$[4]
Różne tandemy	$\Omega(n^2)$, $\mathcal{O}(n^4)$(trywialne) $\boldsymbol{\Theta(n^3)}$	$\boldsymbol{\mathcal{O}(n^3)}$

Tabela 2.2: Przegląd wcześniejszych wyników i naszych rezultatów (pogrubione).



Rysunek 2.3: Przypisanie maksymalnemu 2D-powtórzeniu maksymalnego powtórzenia poziomego o wysokości 2^k.

powiązanego 2D-powtórzenia oraz jego długość jest co najmniej taka jak długość 2D-powtórzenia (rysunek 2.3).

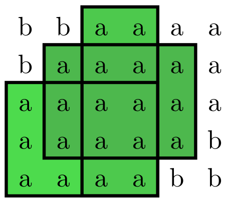
Korzystając z tych własności, lematu o okresowości oraz lematu o trzech kwadratach dowiedliśmy, że wszystkim poziomym powtórzeniom okupującym wiersze od i do $i + 2^k - 1$ może zostać przypisane łącznie $\mathcal{O}(n \log n)$ różnych maksymalnych 2D-powtórzeń. Mnożąc tą wartość, przez ilość możliwych wyborów i oraz k otrzymujemy ograniczenie $\mathcal{O}(n^2 \log^2 n)$ na liczbę maksymalnych 2D-powtórzeń w tekście rozmiaru $n \times n$.

Nasze ograniczenie na ilość maksymalnych 2D-powtórzeń automatycznie daje nowe ograniczenie na czas działania algorytmu z pracy [2] (algorytm ten działa w czasie zależnym od rozmiaru wyniku).

Kwartyki o pierwotnych pierwiastkach

W przypadku jednowymiarowym wszystkie kwadraty o pierwotnych pierwiastkach można bardzo łatwo wyznaczyć znając maksymalne powtórzenia w słowie - wystarczy wziąć wszystkie słowa o długości dwóch okresów maksymalnego powtórzenia, które są w nim zawarte.

W przypadku dwuwymiarowym jest podobnie - każda kwartyka o pierwotnym pierwiastku jest prostokątem o długości dwóch okresów poziomych i wysokości dwóch okresów pionowych pewnego maksymalnego 2D-powtórzenia, całkowicie w nim zawartym. Pojawia się jednak pewien problem - o ile w przypadku jednowymiarowym dwa maksymalne powtórzenia o tym samym okresie nie mogą



Rysunek 2.4: Wiele przecinających się maksymalnych 2D-powtórzeń o tych samych okresach

mieć dużego przecięcia i w szczególności nie mogą generować tego samego kwadratu, o tyle w 2D coś takiego może bardzo łatwo zachodzić (rysunek 2.4), przez to zwykłe użycie takiego algorytmu mogłoby skutkować wielokrotnym zwracaniem tych samych kwartyk.

Rozwiązaniem tego problemu jest metoda zamiatania - algorytm z pracy [7], który pozwala na liczenie sum teoriomnogościowych kilku różnych rodzin prostokątów na kracie $n \times n$ w łącznym czasie $\mathcal{O}(n + r + \text{wynik})$, gdzie r to liczba wszystkich prostokątów.

Dzieląc wszystkie maksymalne 2D-powtórzenia na grupy o takich samych okresach i dla każdej takiej biorąc prostokąty wyznaczające lewe górne rogi generowanych kwartyk otrzymujemy pożądany algorytm.

Dzięki naszemu ograniczeniu na liczbę r z poprzedniej podsekcji oraz ograniczeniu $\mathcal{O}(n^2 \log^2 n)$ na liczbę kwartyk o pierwotnym pierwiastku z pracy [3] algorytm działa w łącznym czasie $\mathcal{O}(n^2 \log^2 n)$ (optymalnym ze względu na dolne ograniczenie z tej samej pracy).

Różne kwartyki

W tekście rozmiaru $n \times n$ złożonym z samych liter a zawarte jest $\Theta(n^4)$ wystąpień dowolnych kwartyk, przez co trywialny algorytm znajduje wszystkie takie wystąpienia w optymalnym pesymistycznym czasie. Dlatego też ciekawszym problemem jest wyznaczenie różnych kwartyk (fragmenty różniące się nie tylko położeniem, ale też jako teksty).

Wynik z pracy [3] daje ograniczenie na liczbę kwartyk o pierwiastku pierwotnym. Inne kwartyki złożone są z wielu przylegających do siebie wystąpień takich samych pierwotnych kwartyk tworzących większy prostokąt. Innymi słowy o ile w przypadku tych poprzednich kwartyk wystąpienia pierwotnego tekstu W tworzyły kratę 2×2, to w przypadku tych pozostałych tworzą kraty $2k \times 2l$, gdzie $k > 1$ lub $l > 1$. Kwartyki takie dzielimy na wąskie, czyli takie dla których $k = 1$ lub $l = 1$ i szerokie dla których $k, l > 1$.

Następnie dokonujemy kolejnego podziału - kwartyki dla których słowo pierwotne W ma rozmiary w przedziałach $[2^a, 2^{a+1}) \times [2^b, 2^{b+1})$ dla pewnych $a, b \leqslant \log n$ rozważamy razem.

Ze względu na lematy o okresowości oraz o trzech kwadratach każdy punkt może być lewym górnym rogiem ostatniego wystąpienia (w tym samym wierszu, bardziej na prawo w tekście nie występuje taka sama kwartyka) co najwyżej czterech różnych kwartyk wąskich z jednej grupy (analogicznie działa dowód na ograniczenie liczby różnych kwadratów w standardowym tekście [10]).

W przypadku kwartyk szerokich dzięki mnogości punktów kratowych w których zaczyna się wystąpienie słowa W każdej takiej kwartyce będącej potęgą W możemy przypisać unikalny punkt na tej kracie. Dodatkowo dzięki lematowi o okresowości taki punkt nie może należeć do kraty dla kwartyki będącej potęgą innego W' o rozmiarze należącym do $[2^a, 2^{a+1}) \times [2^b, 2^{b+1})$.

Dzięki przypisywaniu punktów z $[1, n] \times [1, n]$ do kwartyk oraz $\log^2 n$ wyborom a, b otrzymujemy ograniczenia $\mathcal{O}(n^2 \log^2 n)$ na ilość różnych kwartyk każdego z typów.

W celu wyznaczenia wszystkich unikalnych kwartyk w tekście korzystamy z naszego wcześniejszego algorytmu by wyznaczyć wszystkie wystąpienia kwartyk o pierwiastkach pierwotnych i grupujemy je według pierwiastków.

Kwartyki będące potęgami słowa pierwotnego W powstają z połączenia wielu kwartyk pierwotnych o tym samym pierwiastku. Możemy więc wyznaczyć lewe górne rogi wszystkich wystąpień takich kwartyk i wyszukać maksymalne kraty utworzone przez takie punkty.

Problem ten rozwiązujemy używając ponownie metody zamiatania by uzyskać algorytm działający w czasie $\mathcal{O}(n^2 \log^2 n)$.

Różne tandemy

Jako powiązanym problemem zajęliśmy się też wyznaczaniem różnych tandemów (podobnie jak w przypadku kwartyk wszystkich wystąpień tandemów może być $\Theta(n^4)$).

W wypadku tego problemu można łatwo utożsamić tandemy zaczynające się w wierszu i oraz kończące w wierszu j z kwadratami w standardowym słowie (jednowymiarowym), poprzez przypisanie kolumnom wysokości $j - i + 1$ standardowych liter. W standardowym słowie długości n może być $\Theta(n)$ różnych kwadratów, stąd też takie samo ograniczenie obowiązuje tandemy okupujące wyznaczone wiersze. Mnożąc tę wartość przez liczbę wyborów wartości i oraz j otrzymujemy ograniczenie $\mathcal{O}(n^3)$ na liczbę różnych tandemów.

Dolne ograniczenie $\Omega(n^3)$ na maksymalną liczbę takich tandemów otrzymujemy równie prosto - otrzymujemy je na przykład dla tekstu w którym każdy wiersz jest słowem nad alfabetem jednoliterowym, dla każdego wiersza innym.

Przypisanie kolumnom wysokości $j-i+1$ standardowych liter można zrobić konstrukcyjnie. Dzięki temu i dzięki skorzystaniu z algorytmów wyszukujących różne kwadraty w standardowym słowie otrzymujemy algorytm wyznaczający różne tandemy w słowie dwuwymiarowym w czasie $\mathcal{O}(n^3)$. Ze względu na dolne ograniczenie maksymalnego rozmiaru wyniku żaden bardziej skomplikowany algorytm nie pozwala na uzyskanie lepszego rezultatu w pesymistycznym przypadku.

2.3.4 Efficient Enumeration of Distinct Factors Using Package Representations

W ostatniej z opisywanych prac zajęliśmy się nową reprezentacją podzbiorów podsłów która w wielu przypadkach zapewnia zwięzłą reprezentację, która jednocześnie pozwala na efektywne operowanie opisywanym podzbiorem.

Przez pakiet rozumiemy zbiór podsłów tej samej długości występujących na kolejnych pozycjach w słowie. Nasza reprezentacja składa się ze zbioru takich pakietów reprezentowanych jako trójki (pierwsza pozycja początkowa, długość słów, długość przedziału). Rzeczą, która interesuje nas szczególnie jest zbiór wszystkich podsłów które należą do przynajmniej jednego z tych pakietów.

Bardziej formalnie

$$\mathsf{Factors}(\mathcal{F}) = \{T[j \mathinner{.\,.} j+l) : j \in [i, i+k] \text{ and } (i,l,k) \in \mathcal{F}\}.$$

Przykładami rodzin podsłów, dla których łatwo można uzyskać taką reprezentację są potęgi (kiedy to każdy pakiet reprezentuje bezpośrednio jedno maksymalne powtórzenie) oraz k-antypotęgi (gdy możemy wykorzystać bezpośrednio ich reprezentację z pracy opisanej w sekcji 2.3.1). Reprezentacje te mają rozmiar (i czas obliczania) równy odpowiednio $\mathcal{O}(n)$ i $\mathcal{O}(nk^2)$ (każdy łańcuch przedziałów reprezentuje co najwyżej k zwykłych przedziałów).

W celu znalezienia zbioru $\mathsf{Factors}(\mathcal{F})$, czyli różnych podsłów należących do reprezentacji oraz $|\mathsf{Factors}(\mathcal{F})|$, czyli ilości takich słów rozpatrujemy dwa przypadki. Pierwszy prostszy przypadek, nazywany w pracy specjalnym, to taki w którym jeśli podsłowo należy do $\mathsf{Factors}(\mathcal{F})$, to każde jego wystąpienie w rozważanym słowie należy do któregoś z pakietów należących do $\mathcal{F}$.

W tym przypadku dla słowa długości n i $\mathcal{F}$ składającej się z m pakietów korzystając ze struktury danych najdłuższych poprzednich podsłów (longest previous factors [9]) otrzymujemy wyznaczenie i zliczanie wyniku w czasie odpowiednio $\mathcal{O}(n + m + \text{wynik})$ i $\mathcal{O}(n + m)$. W przypadku przedstawionych przykładów daje nam to nowy, prosty algorytm wyznaczania różnych potęg w słowie, działający w czasie $\mathcal{O}(n)$, oraz nowy algorytm wyznaczania różnych k-antypotęg działający w czasie $\mathcal{O}(nk^2 + \text{wynik})$, a więc lepszy od naszego algorytmu z pracy opisanej w sekcji 2.3.1, który zwraca ilość różnych k-antypotęg, działa w czasie $\mathcal{O}(nk^4 \log k \log n)$ i jest dużo bardziej skomplikowany.

Drugi rozważany przypadek, nazywany w pracy ogólnym nie wymaga tego dodatkowego założenia, co jednak czyni go istotnie trudniejszym do rozwiązania. Aby go rozwiązać adaptujemy metodę użytą poprzednio w pracy o k-antypotęgach (zliczanie różnych k-antypotęg) i uzyskujemy algorytmy wyznaczania i zliczania wyniku $\mathsf{Factors}(\mathcal{F})$, działające w czasie odpowiednio $\mathcal{O}(n \log^2 n + m \log n + \text{wynik})$ i $\mathcal{O}(n \log^2 n + m \log n)$.

2.4 Pozostałe publikacje

Podczas moich studiów doktorskich zostały opublikowane również inne prace których jestem współautorem. Zdecydowałem się nie dołączać ich do niniejszej rozprawy ze względu na niedostateczne dopasowanie do jej tematu.

1. Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Faster Recovery of Approximate Periods over Edit Distance.* SPIRE 2018: 233-240

2. Wojciech Rytter, Wiktor Zuba: *Syntactic View of Sigma-Tau Generation of Permutations.* LATA 2019: 447-459
   Rozszerzona wersja pracy została przyjęta do druku w czasopiśmie Theoretical Computer Science.

3. Mai Alzamel, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Quasi-Linear-Time Algorithm for Longest Common Circular Factor.* CPM 2019: 25:1-25:14

4. Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Circular Pattern Matching with k Mismatches.* FCT 2019: 213-228

5. Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Weighted Shortest Common Supersequence Problem Revisited.* SPIRE 2019: 221-238

6. Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Shortest Covers of All Cyclic Shifts of a String.* WALCOM 2020: 69-80

7. Panagiotis Charalampopoulos, Solon P. Pissis, Jakub Radoszewski, Tomasz Walen, Wiktor Zuba: *Unary Words Have the Smallest Levenshtein k-Neighbourhoods.* CPM 2020: 10:1-10:12

8. Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Internal Quasiperiod Queries.* SPIRE 2020: 60-75

9. Maxime Crochemore, Costas S. Iliopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Shortest covers of all cyclic shifts of a string.* Theor. Comput. Sci. 866: 70-81 (2021)

10. Panagiotis Charalampopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszynski, Tomasz Walen, Wiktor Zuba: *Circular pattern matching with k mismatches.* J. Comput. Syst. Sci. 115: 73-85 (2021)

Chapter 3

Efficient Representation and Counting of Antipower Factors in Words

Efficient Representation and Counting of Antipower Factors in Words

Tomasz Kociumaka[a,b,1], Jakub Radoszewski[b,2,*], Wojciech Rytter[b,1], Juliusz Straszyński[b,2], Tomasz Waleń[b], Wiktor Zuba[b]

[a] *Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel*
[b] *Institute of Informatics, University of Warsaw, Warsaw, Poland*

Abstract

A k-antipower (for $k \geq 2$) is a concatenation of k pairwise distinct words of the same length. The study of fragments of a word being antipowers was initiated by Fici et al. (ICALP 2016) and first algorithms for computing such fragments were presented by Badkobeh et al. (Inf. Process. Lett., 2018). We address two open problems posed by Badkobeh et al. We propose efficient algorithms for counting and reporting fragments of a word which are k-antipowers. They work in $\mathcal{O}(nk \log k)$ time and $\mathcal{O}(nk \log k + C)$ time, respectively, where C is the number of reported fragments. For $k = o(\sqrt{n/\log n})$, this improves upon the time complexity $\mathcal{O}(n^2/k)$ of the solution by Badkobeh et al. We also show that the number of different k-antipower factors of a word of length n can be computed in $\mathcal{O}(nk^4 \log k \log n)$ time. Our main algorithmic tools are runs and gapped repeats. Finally, we present an improved data structure that checks, for a given fragment of a word and an integer k, if the fragment is a k-antipower.

This is a full and extended version of a paper presented at LATA 2019. In particular, no algorithm counting different antipower factors has been announced in the conference proceedings.

Keywords: antipower, gapped repeat, run (maximal repetition)

---

1. Introduction

Typical types of regular words are powers. If equality is replaced by inequality, other versions of powers are obtained. Antipowers are a new type of

*Corresponding author

Email addresses: `kociumaka@mimuw.edu.pl` (Tomasz Kociumaka), `jrad@mimuw.edu.pl` (Jakub Radoszewski), `rytter@mimuw.edu.pl` (Wojciech Rytter), `jks@mimuw.edu.pl` (Juliusz Straszyński), `walen@mimuw.edu.pl` (Tomasz Waleń), `w.zuba@mimuw.edu.pl` (Wiktor Zuba)

[1]Supported by the Polish National Science Center, grant no 214/13/B/ST6/00770.

[2]Supported by the "Algorithms for text processing with errors and uncertainties" project carried out within the HOMING program of the Foundation for Polish Science co-financed by the European Union under the European Regional Development Fund.

regularity of words, based on diversity rather than on equality, that was recently introduced by Fici et al. [1, 2]. Algorithmic study of antipowers was initiated by Badkobeh et al. [3]. Very recently, a related concept of antiperiods was considered by Alamro et al. [4].

Let us assume that $x = y_0 \cdots y_{k-1}$, where $k \geq 2$ and y_i are words of the same length d. We then say that:

- x is a *k-power* if all y_i's are the same;
- x is a *k-antipower* (or a (k,d)-antipower) if all y_i's are pairwise distinct;
- x is a *weak k-power* (or a weak (k,d)-power) if it is not a k-antipower, that is, if $y_i = y_j$ for some $i \neq j$;
- x is a *gapped (q,d)-square* if $y_0 = y_{k-1}$ and $q = k-2$.

In the first three cases, the length d is called the *base* of the power or antipower x.

If w is a word, then by $w[i\mathinner{.\,.}j]$ we denote a *fragment* of w composed of letters $w[i], \dots, w[j]$. The corresponding word $w[i] \cdots w[j]$ is called a *factor* of w. The fragment $w[i\mathinner{.\,.}j]$ (which is an occurrence of the factor $w[i] \cdots w[j]$) can be concisely represented by the indices i and j. Badkobeh et al. [3] considered fragments of a word that are antipowers and obtained the following result.

Fact 1.1 ([3]). *The maximum number of k-antipower fragments in a word of length n is $\Theta(n^2/k)$, and they can all be reported in $\mathcal{O}(n^2/k)$ time. In particular, all k-antipower fragments of a specified base d can be reported in $\mathcal{O}(n)$ time.*

Badkobeh et al. [3] asked for an output-sensitive algorithm that reports all k-antipower fragments in a given word. We present such an algorithm. En route to enumerating k-antipowers, we (complementarily) find weak k-powers. Also gapped (q,d)-squares play an important role in our algorithm.

2-antipowers can be called *antisquares*. An antisquare is simply an even-length word that is not a square. The number of fragments of a word of length n being squares can obviously be $\Theta(n^2)$, e.g., for the word a^n. However, the number of different square factors in a word of length n is $\mathcal{O}(n)$; see [5, 6]. In comparison, the number of different antisquare factors of a word of length n can already be $\Theta(n^2)$. For example, this is true for a de Bruijn word. Still, we show that the number of different antisquare factors of a word can be computed in $\mathcal{O}(n)$ time and that the number of different k-antipower factors for relatively small values of k can also be computed efficiently.

For a given word w, an *antipower query* (i,j,k) asks to check if a fragment $w[i\mathinner{.\,.}j]$ is a k-antipower. Badkobeh et al. [3] proposed the following data structures for answering such queries.

Fact 1.2 ([3]). *Antipower queries can be answered (a) in $\mathcal{O}(k)$ time with a data structure of size $\mathcal{O}(n)$; (b) in $\mathcal{O}(1)$ time with a data structure of size $\mathcal{O}(n^2)$.*

In either case, answering n antipower queries using Fact 1.2 requires $\Omega(n^2)$ time in the worst case (including construction of the data structure). We show a trade-off between the data structure space (and construction time) and query time that allows answering any n antipower queries more efficiently.

Our results. We assume an integer alphabet $\{1,\ldots,n^{\mathcal{O}(1)}\}$. Our first result is an algorithm that computes the number C of k-antipower fragments of a word of length n in $\mathcal{O}(nk\log k)$ time and reports them in $\mathcal{O}(nk\log k + C)$ time.

Our second result is an algorithm that computes the number of different factors of a word of length n that are k-antipowers in $\mathcal{O}(nk^4\log k\log n)$ time.

Our third result is a construction in $\mathcal{O}(n^2/r)$ time of a data structure of size $\mathcal{O}(n^2/r)$, for any $r\in\{1,\ldots,n\}$, which answers antipower queries in $\mathcal{O}(r)$ time. Thus, any n antipower queries can be answered in $\mathcal{O}(n\sqrt{n})$ time and space.

This is a full and extended version of [7].

Structure of the paper. Our algorithms are based on a relation between weak powers and two notions of periodicity of words: gapped repeats and runs. In Section 2, we recall important properties of these notions. Section 4 shows a simple algorithm that counts k-antipower fragments in a word of length n in $\mathcal{O}(nk^3)$ time. In Section 5, it is improved in three steps to an $\mathcal{O}(nk\log k)$-time algorithm. One of the steps applies static range trees that are recalled in Section 3. Algorithms for reporting k-antipower fragments and answering antipower queries are presented in Section 6. The reporting algorithm makes a more sophisticated application of the static range tree that is also described in Section 3. Finally, an algorithm that counts the number of different k-antipower factors in a word of length n in $\mathcal{O}(nk^4\log k\log n)$ time is shown in Section 7.

2. Preliminaries

The length of a word w is denoted by $|w|$ and the letters of w are numbered 0 through $|w|-1$, with $w[i]$ representing the ith letter. Let $[i\mathinner{.\,.}j]$ denote the integer interval $\{i,i+1,\ldots,j\}$ and $[i\mathinner{.\,.}j)$ denote $[i\mathinner{.\,.}j-1]$. By $w[i\mathinner{.\,.}j]$ we denote the *fragment* of w between the ith and the jth letter, inclusively. If $i>j$, the fragment is empty. Let us further denote $w[i\mathinner{.\,.}j)=w[i\mathinner{.\,.}j-1]$. The word $w[i]\cdots w[j]$ that corresponds to the fragment $w[i\mathinner{.\,.}j]$ is called a *factor* of w. Thus the two main counting algorithms that we develop count different k-antipower fragments and different k-antipower factors of the input word, respectively.

By w^R we denote the reversed word w. We say that p is a *period* of the word w if $w[i]=w[i+p]$ holds for all $i\in[0\mathinner{.\,.}|w|-p)$.

An α-gapped repeat γ (for $\alpha\ge 1$) in a word w is a triple (i,j,p) such that $w[i\mathinner{.\,.}j]$ is of the form uvu for $p=|uv|\le\alpha|u|$. The two occurrences of u are called *arms* of the α-gapped repeat and p, denoted $\mathsf{per}(\gamma)$, is called *the period* of the α-gapped repeat. Note that an α-gapped repeat is also an α'-gapped repeat for every $\alpha'>\alpha$. An α-gapped repeat is called *maximal* if its arms can be extended simultaneously with the same character neither to the right nor to the left. In short, we call maximal α-gapped repeats *α-MGRs* and the set of α-MGRs in a word w is further denoted by $\mathit{MGReps}_\alpha(w)$.

Kolpakov et al. [8] showed the first upper bound $\mathcal{O}(n\alpha^2)$ on the number of α-MGRs and proposed an $\mathcal{O}(n\alpha^2)$-time algorithm computing α-MGRs. This result was later improved by Tanimura et al. [9] and Crochemore et al. [10]

resulting in an upper bound of $\mathcal{O}(n\alpha)$ and an $\mathcal{O}(n\alpha)$-time algorithm working for constant-sized alphabets. Gawrychowski et al. [11] provided a constant of 18 in the upper bound and obtained an algorithm that works for integer alphabets. The constant was later improved by I and Köppl [12] to $3(\pi^2/6+5/2) \approx 12.435$. We use the following fact that summarizes the state of the art.

Fact 2.1 ([11, 12]). *Given a word w of length n and a parameter α, the set $MGReps_\alpha(w)$ can be computed in $\mathcal{O}(n\alpha)$ time and satisfies $|MGReps_\alpha(w)| \le 3(\pi^2/6+5/2)\alpha n$.*

Other results related to gapped repeats were shown in [13, 14].

A *run* (a maximal repetition) in a word w is a triple (i,j,p) such that $w[i\mathinner{.\,.} j]$ is a fragment with the smallest period p, $2p \le j-i+1$, that can be extended neither to the left nor to the right preserving the period p. Its *exponent* e is defined as $e=(j-i+1)/p$. Kolpakov and Kucherov [15] showed that a word of length n has $\mathcal{O}(n)$ runs, that the sum of their exponents is $\mathcal{O}(n)$, and that they can be computed in $\mathcal{O}(n)$ time. Bannai et al. [16] recently refined these combinatorial results and developed a simpler the linear-time algorithm.

Fact 2.2 ([16]). *A word of length n has at most n runs, and the sum of their exponents does not exceed $3n$. All these runs can be computed in $\mathcal{O}(n)$ time.*

Further work [17, 18] led to improved constants in the upper bound, although only for the case of binary strings.

A *generalized run* in a word w is a triple $\gamma=(i,j,p)$ such that $w[i\mathinner{.\,.} j]$ is a fragment with a period p, not necessarily the shortest one, $2p \le j-i+1$, that can be extended neither to the left nor to the right preserving the period p. By $\mathsf{per}(\gamma)$ we denote p, called *the period* of the generalized run γ. The set of generalized runs in a word w is denoted by $GRuns(w)$.

A run (i,j,p) with exponent e corresponds to $\left\lfloor \frac{e}{2} \right\rfloor$ generalized runs: (i,j,p), $(i,j,2p)$, $(i,j,3p)$, …, $(i,j,\left\lfloor \frac{e}{2} \right\rfloor p)$. By Fact 2.2, we obtain the following.

Corollary 2.3. *For a word w of length n, $|GRuns(w)| \le 1.5n$ and this set can be computed in $\mathcal{O}(n)$ time.*

Our algorithm uses a relation between weak powers, α-MGRs, and generalized runs; see Fig. 1 for an example presenting the interplay of these notions.

The *interval representation* of a finite set X of integers is

$$X = [i_1 \mathinner{.\,.} j_1] \cup [i_2 \mathinner{.\,.} j_2] \cup \cdots \cup [i_t \mathinner{.\,.} j_t],$$

where $i_1 \le j_1$, $j_1+1 < i_2$, $i_2 \le j_2$, …, $j_{t-1}+1 < i_t$, and $i_t \le j_t$. We denote this representation by $\mathcal{R}(X)$. The value t, denoted $|\mathcal{R}(X)|$, is called the *size* of the representation. The following simple lemma allows implementing basic operations on interval representations.

Lemma 2.4. *Assume that $\mathcal{X}_1,\dots,\mathcal{X}_r$ are non-empty families of subintervals of $[0\mathinner{.\,.} n)$. The interval representations of $\bigcup\mathcal{X}_1, \bigcup\mathcal{X}_2, \dots, \bigcup\mathcal{X}_r$ can be computed in $\mathcal{O}(n+m)$ time, where m is the total size of the families $\mathcal{X}_i$. Similarly, the interval representation of $\bigcap_{i=1}^{r}(\bigcup\mathcal{X}_i)$ can be computed in $\mathcal{O}(n+m)$ time.*

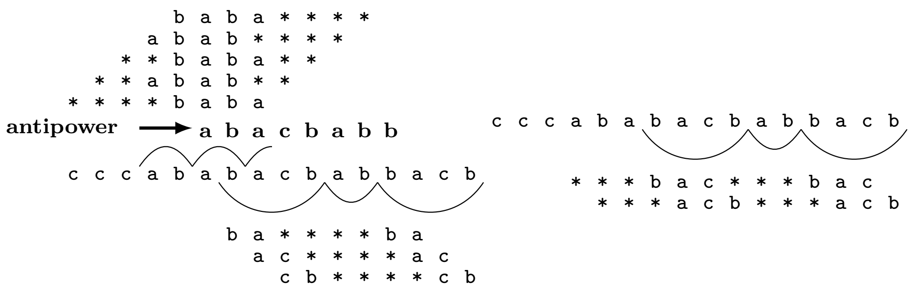


Figure 1: To the left: all weak $(4,2)$-powers and the only $(4,2)$-antipower in a word of length 16. An asterisk denotes any character. The first five weak $(4,2)$-powers are generated by the run `ababa` with period 2, and the last three are generated by the 1.5-MGR `bacb ab bacb`, whose period 6 is divisible by 2. To the right: all weak $(4,3)$-powers in the same word are generated by the same MGR because its period is a multiple of 3.

Proof. We start by sorting the endpoints of the intervals and grouping them by the index i of the family $\mathcal{X}_i$. This can be done in $\mathcal{O}(n+m)$ time using bucket sort [19]. Next, for each i, to compute the interval representation of $\bigcup \mathcal{X}_i$, we scan the endpoints left to right maintaining the number of intervals containing the current point. We start an interval when this number becomes positive and end one when the number drops to 0. This processing takes $\mathcal{O}(m)$ time.

In order to compute the representation of the intersection, we use the same type of a counter when simultaneously processing the interval representations of $\bigcup \mathcal{X}_1, \bigcup \mathcal{X}_2, \dots, \bigcup \mathcal{X}_r$, but start an interval only when the counter becomes equal to r and end one when the counter drops below r. $\square$

Let $\mathcal{J}$ be a family of subintervals of $[0\mathinner{.\,.} m)$, initially empty. Let us consider the following operations on $\mathcal{J}$, where I is an interval: `insert`(I): $\mathcal{J} := \mathcal{J} \cup \{I\}$; `delete`$(I)$: $\mathcal{J} := \mathcal{J} \setminus \{I\}$ for $I \in \mathcal{J}$; and `count`, which returns $|\bigcup \mathcal{J}|$. It is folklore knowledge that all these operations can be performed efficiently using a static range tree (sometimes called a segment tree; see, e.g., [20, Section 2.1]). In Section 3, we prove the following lemma for completeness.

Lemma 2.5. *There exists a data structure of size $\mathcal{O}(m)$ that, after $\mathcal{O}(m)$-time initialization, supports* `insert` *and* `delete` *in $\mathcal{O}(\log m)$ time and* `count` *in $\mathcal{O}(1)$ time.*

Let us introduce another operation `report` that returns all elements of the set $A = [0\mathinner{.\,.} m) \setminus \bigcup \mathcal{J}$. We also show in Section 3 that a static range tree can be augmented to support this operation efficiently.

Lemma 2.6. *There exists a data structure of size $\mathcal{O}(m)$ that, after $\mathcal{O}(m)$-time initialization, supports* `insert` *and* `delete` *in $\mathcal{O}(\log m)$ time and* `report` *in $\mathcal{O}(|A|)$ time.*

3. Applications of static range tree

Let m be a power of two. We define a *basic interval* as an interval of the form $[2^i a \mathinner{.\,.} 2^i(a+1)) \subseteq [0 \mathinner{.\,.} m-1)$ such that i and a are non-negative integers. For example, the basic intervals for $m = 8$ are $[0\mathinner{.\,.}1), \ldots, [7\mathinner{.\,.}8)$, $[0\mathinner{.\,.}2), [2\mathinner{.\,.}4), [4\mathinner{.\,.}6), [6\mathinner{.\,.}8), [0\mathinner{.\,.}4), [4\mathinner{.\,.}8), [0\mathinner{.\,.}8)$. In a *static range tree* (sometimes called a segment tree; see [20, Section 2.1]), each node is identified with a basic interval. The children of a node $J = [2^i a \mathinner{.\,.} 2^i(a+1))$, for $i > 0$, are $\mathit{lchild}(J) = [2^{i-1}{\cdot}2a \mathinner{.\,.} 2^{i-1}(2a{+}1))$ and $\mathit{rchild}(J) = [2^{i-1}(2a{+}1) \mathinner{.\,.} 2^{i-1}(2a{+}2))$. Thus, a static range tree is a full binary tree of size $\mathcal{O}(m)$ rooted at $\mathit{root} = [0\mathinner{.\,.}m)$.

Every interval $I \subseteq [0\mathinner{.\,.}m)$ can be decomposed into a disjoint union of at most $2\log m$ basic intervals. The decomposition can be computed in $\mathcal{O}(\log m)$ time recursively starting from the root. Let J be a node considered in the algorithm. If $J \subseteq I$, the algorithm adds J to the decomposition. Otherwise, for each child J' of the node J, if $J' \cap I \neq \emptyset$, the algorithm makes a recursive call to the child. At each level of the tree, the algorithm makes at most two recursive calls. The resulting set of basic intervals is denoted by $\mathit{Decomp}(I)$; see Fig. 2.

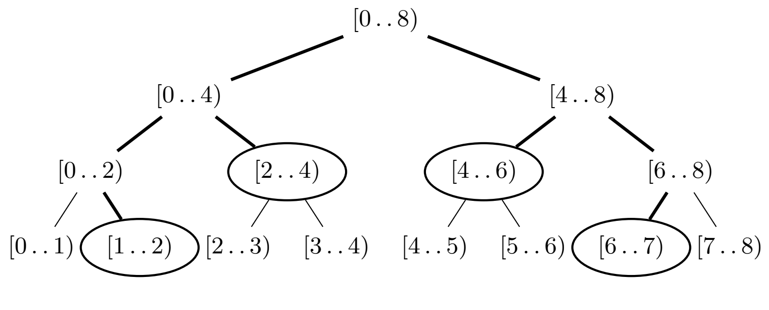


Figure 2: A static range tree for $m = 8$ with the set of nodes that comprises $\mathit{Decomp}(\,[1\mathinner{.\,.}7)\,)$. The paths visited in the recursive decomposition algorithm are shown in bold.

Proofs of the Lemmas 2.5 and 2.6 follow. Instead of Lemma 2.5, we show an equivalent lemma with an operation `count′` which returns $|[0\mathinner{.\,.}m) \setminus \bigcup \mathcal{J}|$.

Lemma 3.1. *There exists a data structure of size $\mathcal{O}(m)$ that, after $\mathcal{O}(m)$-time initialization, supports* `insert` *and* `delete` *in $\mathcal{O}(\log m)$ time and* `count′` *in $\mathcal{O}(1)$ time.*

Proof. Let m' be the smallest power of two satisfying $m' \geq m$. Observe that the data structure for m can be simulated by an instance constructed for m': it suffices to insert an interval $[m\mathinner{.\,.}m')$ in the initialization phase to make sure that integers $i \geq m$ will not be counted when `count′` is invoked. Henceforth, we may assume without loss of generality that m is a power of two.

We maintain a static range tree; every node J stores two values (see Fig. 3):

- $\mathit{bi}(J) = |\,\{I \in \mathcal{J} \,:\, J \in \mathit{Decomp}(I)\}\,|$

- $val(J) = |\, J \setminus \bigcup\{J' : J' \subseteq J,\, J' \in Decomp(I),\, I \in \mathcal{J}\}\,|$.

The value $val(J)$ can also be defined recursively:

- If $bi(J) > 0$, then $val(J) = 0$.
- If $bi(J) = 0$ and J is a leaf, then $val(J) = 1$.
- Otherwise, $val(J) = val(lchild(J)) + val(rchild(J))$.

This allows computing $val(J)$ from $bi(J)$ and the values in the children of J.

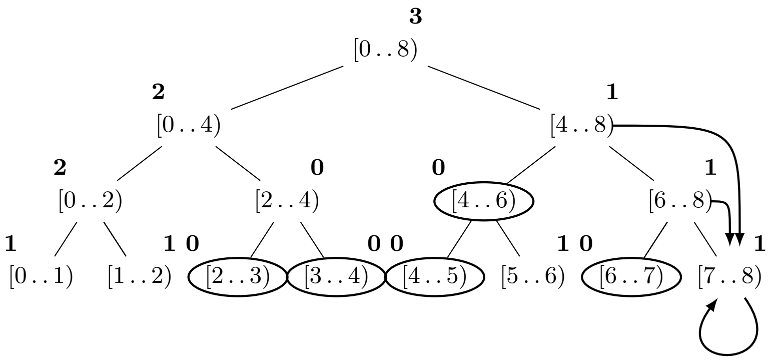


Figure 3: A static range tree for $m = 8$ that stores the family $\mathcal{J} = \{[2\,..\,3), [3\,..\,5), [4\,..\,7), [6\,..\,7)\}$. The values $val(J)$ are shown in bold. The arrows present selected *jump* pointers (described in the proof of Lemma 2.6).

The data structure can be initialized bottom-up in $\mathcal{O}(m)$ time. The respective operations on the data structure are now implemented as follows:

- `insert`(I): Compute $Decomp(I)$ recursively. For each $J \in Decomp(I)$, increment $bi(J)$. For each node J encountered in the recursive computation, recompute $val(J)$.
- `delete`(I): Similar to `insert`, but we decrement $bi(J)$ for each node $J \in Decomp(I)$.
- `count`$'$: Return $val(root)$.

The complexities of the respective operations follow. $\square$

Lemma 2.6. *There exists a data structure of size $\mathcal{O}(m)$ that, after $\mathcal{O}(m)$-time initialization, supports* `insert` *and* `delete` *in $\mathcal{O}(\log m)$ time and* `report` *in $\mathcal{O}(|A|)$ time.*

Proof. As in the proof of Lemma 3.1, we assume without loss of generality that m is a power of two. Again, the data structure applies a static range tree. We also reuse the values $bi(J)$ for nodes; we generalize the $val(J)$ values, though.

If J and J' are basic intervals and $J' \subseteq J$, then we define $val_J(J')$ as 0 if there exists a basic interval J'' on the path from J to J' (i.e., such that $J' \subseteq J'' \subseteq J$) for which $bi(J'') > 0$, and as $val(J')$ otherwise. These values satisfy the following properties.

Observation 3.2. *For every node J, (a) $val_J(J) = val(J)$ and (b) $val_{root}(J) = |J \setminus \bigcup \mathcal{J}|$.*

By point (b) of the observation, our goal in a `report` query is to report all leaves J such that $val_{root}(J) = 1$. The first idea how to do it would be to recursively visit all the nodes J' of the tree such that $val_{root}(J') > 0$. However, this approach would cost $\Omega(|A| \log m)$ time since, for every leaf, all the nodes on the path to the root would need to be visited.

In order to efficiently answer `report` queries, we introduce *jump* pointers, stored in each node J, such that $jump(J)$ is the lowest such node J' in the subtree of J such that $val_J(J') = val_J(J)$; see Fig. 3.

The pointer $jump(J)$ can be computed in $\mathcal{O}(1)$ time from the values in the children of J:

$$jump(J) = \begin{cases} J & \text{if } J \text{ is a leaf or } 0 < val(lchild(J)) < val(J), \\ jump(lchild(J)) & \text{if } val(rchild(J)) = 0, \\ jump(rchild(J)) & \text{otherwise.} \end{cases}$$

This formula allows recomputing the *jump* pointers on the paths visited during a call to `insert` or `delete` without altering the time complexity.

Let us consider a subtree that is composed of all the nodes J with positive values $val_{root}(J)$. Using *jump* pointers, we make a recursive traversal of the subtree that avoids visiting long paths of non-branching nodes of the subtree. It visits all the leaves and branching nodes of the subtree and, in addition, both children of each branching node. With this traversal, a `report` query is therefore answered in $\mathcal{O}(|A|)$ time. □

4. Computing a compact representation of weak k-powers

Let us denote by $Squares(q, d)$ the set of starting positions of gapped (q, d)-square fragments in the input word w.

We say that an occurrence at position i of a gapped (q, d)-square is *generated* by a gapped repeat γ if $\mathsf{per}(\gamma) = (q+1)d$ and $w[i \mathinner{.\,.} i+(q+2)d)$ is fully contained in this gapped repeat. In other words, if γ is of the form uvu, then $u = u_1u_2u_3$, $|u_2| = d$, $|u_3vu_1| = qd$, and γ starts at position $i - |u_1|$ of the input word; cf. Fig. 4.

Similarly, an occurrence at position i of a gapped (q, d)-square is *generated* by a generalized run γ if $\mathsf{per}(\gamma) = (q+1)d$ and $w[i \mathinner{.\,.} i+(q+2)d)$ is fully contained in this generalized run. See Fig. 5 for a concrete example.

Lemma 4.1. *Let $q \geq 0$ and $1 \leq d \leq n/(q + 2)$ be integers.*

(a) *Every gapped (q, d)-square fragment is generated by a $(q + 1)$-MGR with period $(q + 1)d$ or by a generalized run with period $(q + 1)d$.*

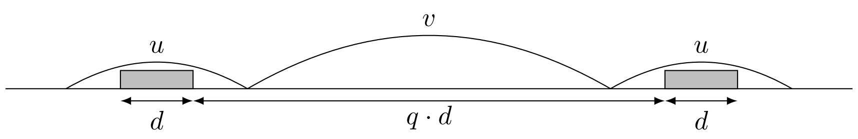
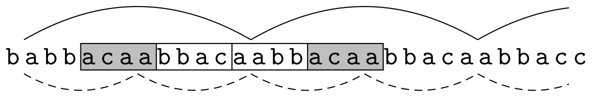


Figure 4: An occurrence of a gapped (q, d)-square generated by a gapped repeat with period $(q+1)d$. Gray rectangles represent equal words.



b a b b a c a a b b a c a a b b a c a a b b a c a a b b a c c

Figure 5: An occurrence of a gapped $(2, 4)$-square `acaa bbac aabb acaa` generated by a generalized run with period 12. Note that the generalized run has its origin in a (generalized) run with period 6 (depicted below) that *does not* generate this gapped square.

(b) *Each gapped repeat with period $(q+1)d$ and each generalized run γ with period $(q+1)d$ generates a single interval of positions where gapped (q, d)-squares occur; see Fig. 6. Moreover, this interval can be computed in constant time.*

Proof. **(a)** Let i be the starting position of an occurrence of a gapped (q, d)-square x of length $\ell := |x| = (q+2)d$. Observe that x has period $p := (q+1)d$. We denote by $w[i' \mathinner{..} j']$ the longest factor with period p that contains x (i.e., such that $i' \le i$ and $i + \ell - 1 \le j'$), and denote $\gamma := (i', j', p)$.

If $|w[i' \mathinner{..} j']| < 2p$, then γ is a gapped repeat with period p, and it is maximal by definition. Moreover, it is a $(q+1)$-MGR since its arms have length at least d.

If $|w[i' \mathinner{..} j']| \ge 2p$, then γ is a generalized run that generates the gapped square x. In particular, this happens for $q = 0$.

(b) Let $\gamma = (i, j, p)$ be a gapped repeat or a generalized run with period $p = (q+1)d$. Then γ generates gapped (q, d)-squares that start at positions in the interval $[i \mathinner{..} j + 1 - (p + d)]$. □

The interval defined in Lemma 4.1**(b)** will be further denoted by $\mathit{Squares}(q, d, \gamma)$.

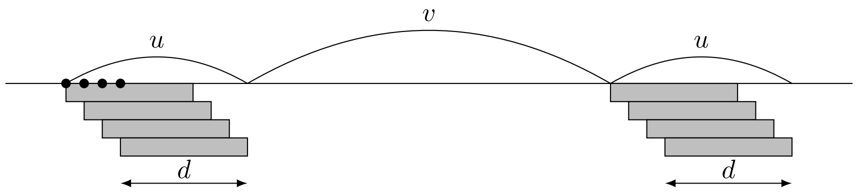


Figure 6: An interval, represented as a sequence of four consecutive positions (black dots), of starting positions of gapped (q, d)-square fragments generated by a gapped repeat with period $(q+1)d$.

Corollary 4.2. *For all integers $q \geq 0$ and $d \geq 1$, we have $\mathcal{R}(Squares(q,d)) = \{Squares(q,d,\gamma) \,:\, \gamma \in MGReps_{q+1}(w) \cup GRuns(w)$ and $\mathsf{per}(\gamma) = (q+1)d\}$. Moreover, for every $q \geq 0$,*

$$\sum_{d=1}^{\lfloor n/(q+2) \rfloor} |\mathcal{R}(Squares(q,d))| = \mathcal{O}(n(q+1))$$

and these representations $\mathcal{R}(Squares(q,d))$ can be computed in $\mathcal{O}(n(q+1))$ time.

Proof. Lemma 4.1 shows that $Squares(q,d)$ can be expressed as the union of intervals $Squares(q,d,\gamma)$, where γ is a generalized run with period $(q+1)d$ or a $(q+1)$-MGR with period $(q+1)d$. Moreover, by maximality of generalized runs and $(q+1)$-MGRs, no two such intervals $Squares(q,d,\gamma)$ overlap (contain a common position) or touch (contain adjacent positions).

As for the second claim, the total number of interval chains in these representations is $\mathcal{O}(n(q+1))$ because the number of $(q+1)$-MGRs and generalized runs γ is bounded by $\mathcal{O}(n(q+1))$ due to Facts 2.1 and 2.2, respectively. Since each interval $Squares(q,d,\gamma)$ can be computed in constant time (by Lemma 4.1**(b)**), this also yields an $\mathcal{O}(n(q+1))$ bound on the construction time. $\square$

Let us denote

$$Chain_k(q,d,i) \;=\; \{\, i,\, i-d,\, i-2d,\, \ldots,\, i-(k-q-2)d \,\}.$$

This definition can be extended to intervals I. To this end, let us denote

$$I \ominus r \;=\; \{\, i-r \,:\, i \in I \,\}$$

and define

$$Chain_k(q,d,I) \;=\; I \,\cup\, (I \ominus d) \,\cup\, (I \ominus 2d) \,\cup \cdots \cup\, (I \ominus (k-q-2)d).$$

This set, further referred to as an *interval chain*, can be stored in $\mathcal{O}(1)$ space. A *chain representation* of a set of integers is its representation as a union of interval chains, limited to some base interval. The size of the chain representation is the number of chains.

We denote by $WeakP_k(d)$ the set of starting positions in w of weak (k,d)-power fragments. The following lemma shows how to compute small chain representations of the sets $WeakP_k(d)$ (with base intervals $[0\mathinner{.\,.} n-kd]$).

Lemma 4.3.

(a) $WeakP_k(d) = \bigcup_{q=0}^{k-2} \bigcup_{I \in \mathcal{R}(Squares(q,d))} Chain_k(q,d,I) \cap [0\mathinner{.\,.} n-kd]$.

(b) *For $d = 1, \ldots, \lfloor n/k \rfloor$, the sets $WeakP_k(d)$ have chain representations of total size $\mathcal{O}(nk^2)$; these representations can be computed in $\mathcal{O}(nk^2)$ time.*

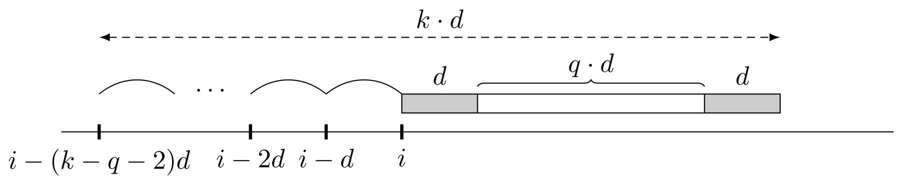


Figure 7: The fact that $i \in \mathit{Squares}(q,d)$ witnesses inclusion $(\mathit{Chain}_k(q,d,i) \cap [0\mathinner{.\,.} n-kd]) \subseteq \mathit{WeakP}_k(d)$.

Proof. As for point **(a)**, $x = y_0 \cdots y_{k-1}$ for $|y_0| = \cdots = |y_{k-1}| = d$ is a weak (k,d)-power if and only if $y_i \cdots y_j$ is a gapped $(j-i-1,d)$-square for some $0 \le i < j < k$. Conversely, a gapped (q,d)-square occurring at position i implies occurrences of weak (k,d)-powers at positions in the set $\mathit{Chain}_k(q,d,i)$, limited to the interval $[0\mathinner{.\,.} n-kd]$ due to the length constraint; see Fig. 7.

We obtain point **(b)** by applying the formula from point **(a)** to compute the chain representations of sets $\mathit{WeakP}_k(d)$ for all $d = 1,\dots,\lfloor n/k \rfloor$. The interval representations $\mathcal{R}(\mathit{Squares}(q,d))$ are computed using Corollary 4.2, which costs $\mathcal{O}(n(q+1))$ time for each $q \in [0\mathinner{.\,.} k-2]$ and $\mathcal{O}(nk^2)$ time in total. The total size of the interval representations and the chain representations is also $\mathcal{O}(nk^2)$. $\square$

Lemma 4.3 lets us count k-antipowers by computing the sizes of the complementary sets $\mathit{WeakP}_k(d)$. Thus, we obtain the following preliminary result.

Proposition 4.4. *The number of k-antipower fragments in a word of length n can be computed in $\mathcal{O}(nk^3)$ time.*

Proof. See Algorithm 1. We use Corollary 4.2 and Lemma 4.3 to express the sets $\mathit{WeakP}_k(d)$ for all $d = 1,\dots,\lfloor n/k \rfloor$ as unions of $\mathcal{O}(nk^2)$ interval chains.

Algorithm 1: SimpleCount(w,n,k)

```
$(\mathcal{C}_d)_{d=1}^{\lfloor n/k \rfloor} := (\emptyset,\dots,\emptyset)$
for $q := 0$ to $k-2$ do
    foreach $(q+1)$-MGR or generalized run $\gamma$ in $w$ do
        $p := \mathsf{per}(\gamma)$
        $d := \frac{p}{q+1}$
        if $d \in \mathbb{Z}$ then
            $I := \mathit{Squares}(q,d,\gamma)$
            $\mathcal{C}_d := \mathcal{C}_d \cup \{\, \mathit{Chain}_k(q,d,I) \,\}$
$\mathit{antipowers} := 0$
for $d := 1$ to $\lfloor n/k \rfloor$ do
    $\mathit{WeakP}_k(d) := (\bigcup \mathcal{C}_d) \cap [0\mathinner{.\,.} n-kd]$
    $\mathit{antipowers} := \mathit{antipowers} + (n-kd+1) - |\,\mathit{WeakP}_k(d)|$
return $\mathit{antipowers}$
```

That is, the total size of the sets $\mathcal{C}_d$ is $\mathcal{O}(nk^2)$. Each of the interval chains consists of at most k intervals. Hence, Lemma 2.4 can be applied to compute interval representations of the sets $\mathit{WeakP}_k(d)$ in $\mathcal{O}(nk^3)$ total time. Finally, the size of the complement of the set $\mathit{WeakP}_k(d)$ (in $[0\mathinner{.\,.} n-kd]$) is the number of (k,d)-antipowers. $\square$

Next, we improve the time complexity of this algorithm to $\mathcal{O}(nk \log k)$.

5. Counting k-antipower fragments in $\mathcal{O}(nk \log k)$ time

We improve the algorithm SimpleCount threefold. First, we show that the chain representation of weak k-powers actually consists of only $\mathcal{O}(nk)$ chains. Then, instead of converting the chain representations to the interval representations, we introduce a geometric interpretation that reduces the problem to computing the area of the union of $\mathcal{O}(nk)$ axis-aligned rectangles. This area could be computed directly in $\mathcal{O}(nk \log n)$ time, but we improve this complexity to $\mathcal{O}(nk \log k)$ by exploiting properties of the dimensions of the rectangles.

5.1. First improvement of SimpleCount

First, we improve the $\mathcal{O}(nk^2)$ bounds of Lemma 4.3**(b)**. By inspecting the structure of MGRs, we show that the formula from Lemma 4.3**(a)** generates only $\mathcal{O}(nk)$ interval chains in total for $d \ge 2k-2$. Trivially, the sets $\mathit{WeakP}_k(d)$ for $d < 2k-2$ can be represented using $\mathcal{O}(n)$ interval chains each. A careful implementation lets us compute all such chain representations in $\mathcal{O}(nk)$ time.

We say that an α-MGR for integer α with period p is *nice* if $\alpha \mid p$ and $p \ge 2\alpha^2$. Let $\mathit{NMGReps}_\alpha(w)$ denote the set of nice α-MGRs in the word w. The following lemma provides a combinatorial foundation of the improvement.

Lemma 5.1. *For a word w of length n and an integer $\alpha > 1$, $|\mathit{NMGReps}_\alpha(w)| \le 9(\pi^2/6 + 5/2)n$.*

Proof. Let us consider a partition of the word w into blocks of α letters (the final $n \bmod \alpha$ letters are not assigned to any block). Let uvu be a nice α-MGR in w. We know that $2\alpha^2 \le |uv| \le \alpha|u|$, so $|u| \ge 2\alpha$. Now, let us fit the considered α-MGR into the structure of blocks. Since $\alpha \mid |uv|$, the indices in w of the occurrences of the left and the right arm are equal modulo α. We shrink both arms to u' such that u' is the maximal inclusion-wise interval of blocks which is encompassed by each arm u. Then, let us expand v to v' so that it fills the space between the two occurrences of u'.

We have $u = xu'y$ and $v' = yvx$ for some words x, y, so $|u'v'| = |u'| + |x| + |y| + |v| = |uv|$. Moreover, $|u'| \ge \frac{1}{3}|u|$ since u encompasses at least one full block of w. Consequently, $|u'v'| \le 3\alpha|u'|$.

Let t be a word whose letters correspond to whole blocks in w and u'', v'' be factors of t that correspond to u' and v', respectively. We have $|u''| = |u'|/\alpha$ and $|v''| = |v'|/\alpha$, so $u''v''u''$ is a 3α-gapped repeat in t. It is also a 3α-MGR because it can be expanded by one block neither to the left nor to the right, as it would contradict the maximality of the original nice α-MGR. This concludes that every nice α-MGR in w has a corresponding 3α-MGR in t. Also, every 3α-MGR in t corresponds to at most one nice α-MGR in w, as it can be translated into blocks of w and expanded in a single way to a 3α-MGR (that can happen to be a nice α-MGR).

We conclude that the number of nice α-MGRs in w is at most the number of 3α-MGRs in t. As $|t| \le n/\alpha$, this is at most $9(\pi^2/6+5/2)n$ due to Fact 2.1. $\square$

Lemma 5.2. *For $d = 1, \dots, \lfloor n/k \rfloor$, the sets $\mathit{WeakP}_k(d)$ have chain representations of total size $\mathcal{O}(nk)$ which can be computed in $\mathcal{O}(nk)$ time. Moreover, $\sum_{d=2k-2}^{\lfloor n/k \rfloor} \sum_{q=0}^{k-2} |\mathcal{R}(\mathit{Squares}(q,d))| = \mathcal{O}(nk)$.*

Proof. The chain representations of sets $\mathit{WeakP}_k(d)$ are computed for $d < 2k-2$ and for $d \ge 2k-2$ separately.

From Fact 1.1, we know that all (k,d)-antipowers for given k and d can be found in $\mathcal{O}(n)$ time. This lets us compute the set $\mathit{WeakP}_k(d)$ (and its trivial chain representation) in $\mathcal{O}(n)$ time. Across $d \in [1 \,..\, 2k-2)$, this gives $\mathcal{O}(nk)$ chains and $\mathcal{O}(nk)$ time.

Henceforth we consider the case that $d \ge 2k-2$. Let us note that if a gapped (q,d)-square with $d \ge 2(q+1)$ is generated by a $(q+1)$-MGR, then this $(q+1)$-MGR is nice. Indeed, by Lemma 4.1**(a)** this $(q+1)$-MGR has period $p = (q+1)d \ge 2(q+1)(k-1) \ge 2(q+1)^2$. This observation lets us express the formula of Corollary 4.2 for $\mathcal{R}(\mathit{Squares}(q,d))$ using $\mathit{NMGReps}_{q+1}(w)$ instead of $\mathit{MGReps}_{q+1}(w)$ provided that $d \ge 2(q+1)$.

By Fact 2.2 and Lemma 5.1, for every q we have only

$$|\mathit{NMGReps}_{q+1}(w) \cup \mathit{GRuns}(w)| = \mathcal{O}(n)$$

MGRs and generalized runs to consider. Hence, the total size of chain representations of sets $\mathit{WeakP}_k(d)$ for $d \ge 2k-2$ is $\mathcal{O}(nk)$ as well. The same applies to the total size of interval representations of sets $\mathit{Squares}(q,d)$ for $d \ge 2k-2$.

The last piece of the puzzle is the following claim.

Claim 5.3. *The sets $\mathit{NMGReps}_\alpha(w)$ for $\alpha \in [1 \,..\, k-1]$ can be built in $\mathcal{O}(nk)$ time.*

Proof. The union of those sets is a subset of $\mathit{MGReps}_{k-1}(w)$. Therefore, we can consider each $(k-1)$-MGR uvu with period $p = |uv|$ and report all $\alpha \in [\alpha_L \,..\, \alpha_R]$ such that $\alpha \mid p$, where

$$\alpha_L = \left\lceil \frac{p}{|u|} \right\rceil, \quad \alpha_R = \min\left(k-1, \left\lfloor \sqrt{\frac{p}{2}} \right\rfloor\right).$$

We will use an auxiliary table `next` such that

$$\texttt{next}_p[\alpha] = \min\{\alpha' \in [\alpha+1 \,..\, k) \,:\, \alpha' \mid p\}.\ ^{3}$$

[3] We assume that $\min \emptyset = \infty$.

This table has size $\mathcal{O}(nk)$. For every $p \in [1\,..\,n]$, all values $\mathtt{next}_p[\alpha]$ for $\alpha \in [1\,..\,k)$ can be computed, right to left, in $\mathcal{O}(k)$ time. Then, all values α for which uvu is a nice α-MGR can be computed by iterating $\alpha := \mathtt{next}_p[\alpha]$ until a value greater than α_R is reached, starting from $\alpha = \alpha_L - 1$. Thus, the total time of constructing the sets $\mathit{NMGReps}_\alpha(w)$ is $\mathcal{O}(|\mathit{MGReps}_{k-1}(w)| + \sum_{\alpha=1}^{k-1} |\mathit{NMGReps}_\alpha(w)|) = \mathcal{O}(nk)$. □

This concludes the proof. □

5.2. Second improvement of SimpleCount

We reduce the problem to computing unions of sets of orthogonal rectangles with bounded integer coordinates.

For a given value of d, let us fit the integers from $[0\,..\,n-kd]$ into the cells of a grid of width d so that the first row consists of numbers 0 through $d-1$, the second of numbers d to $2d-1$, etc. Let us call this grid $\mathcal{G}_d$. The main idea behind the lemma presented below is shown in Fig. 8.

Lemma 5.4. *The set $\mathit{Chain}_k(q,d,I)$ is a union of $\mathcal{O}(1)$ orthogonal rectangles in $\mathcal{G}_d$, each of height at most k or width exactly d. The coordinates of the rectangles can be computed in $\mathcal{O}(1)$ time.*

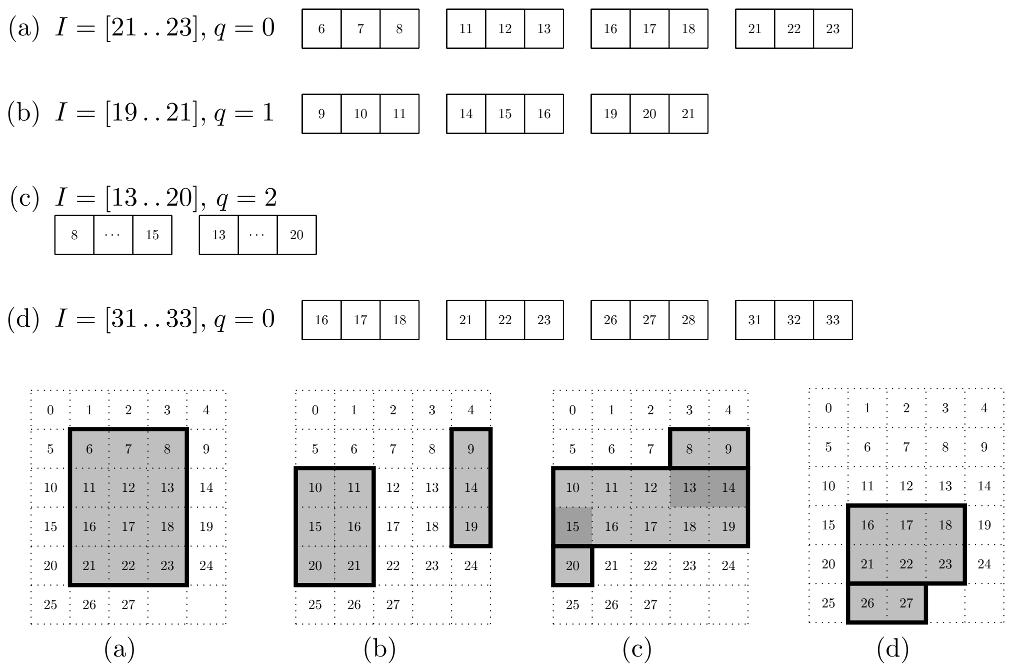

(a) $I = [21\,..\,23]$, $q = 0$
(b) $I = [19\,..\,21]$, $q = 1$
(c) $I = [13\,..\,20]$, $q = 2$
(d) $I = [31\,..\,33]$, $q = 0$
(a)
(b)
(c)
(d)

Figure 8: Examples of decompositions of various interval chains $\mathit{Chain}_k(q,d,I)$ into orthogonal rectangles in the grid $\mathcal{G}_d$ for $d = 5$, $k = 5$, $n = 52$.

Proof. Translating the set $\mathit{Chain}_k(q, d, I)$ onto our grid representation, it becomes a union of horizontal strips, each corresponding to an interval $I \ominus ad$, for $a \in [0 \mathinner{..} k-q-2]$, that possibly wrap around into the subsequent rows. Those strips have their beginnings in the same column, occupying consecutive positions. Depending on the column index of the beginning of a strip and its length, we have three cases:

- The strip does not wrap around at all (Fig. 8(a)). Then, the union of all strips is simply a single rectangle. Its height is exactly $k - q - 1$.
- The strip's length is smaller than the length of the row, but it wraps around at some point (Fig. 8(b)). Then, there exists a column which does not intersect with any strip. The strips' parts that have wrapped around (that is, to the left of the column) form a rectangle and similarly the strips' parts that have not wrapped around form a rectangle as well. Both of these rectangles have height equal to $k - q - 1$.
- The strip's length is greater than or equal to the length of the row. In this case, excluding the first and the last row, the union of the strips is actually a rectangle fully encompassing all columns (Fig. 8(c)). Therefore the union of all strips can be represented as a union of three rectangles: the first row, the last row and what is in between. Both the first and the last row have height equal to 1 and the rectangle in between has width equal to d.

In some cases, such decomposition into orthogonal rectangles may include cells that are not on the grid, that is, negative numbers or numbers greater than $n - kd$; see Fig. 8(d). Then, the rectangles need to be trimmed to fit inside the grid. Moreover, in the case of numbers greater than $n - kd$, we may need to consider the part of the union that is located in the last row of the grid as a separate rectangle, since it may have a smaller width than the rectangle it originates from (as shown on Fig. 8(d)). $\square$

Thus, by Lemma 5.2, our problem reduces to computing the area of unions of rectangles in subsequent grids $\mathcal{G}_d$. In total, the number of rectangles is $\mathcal{O}(nk)$.

5.3. Third improvement of SimpleCount

Assume that r axis-aligned rectangles in the plane are given. The area of their union can be computed in $\mathcal{O}(r \log r)$ time using a classic sweep line algorithm of Bentley [21] (obtaining a more efficient solution to this problem, even for integer coordinates bounded by $\mathcal{O}(n)$, seems hard [22]). This approach would yield an $\mathcal{O}(nk \log n)$-time algorithm for counting k-antipowers. We refine this approach in the case that the rectangles have bounded height or maximum width and their coordinates are bounded.

Lemma 5.5. *Assume that r axis-aligned rectangles in $[0 \mathinner{..} m_h] \times [0 \mathinner{..} m_w]$ with integer coordinates are given and that each rectangle has height at most k or width exactly m_w. The area of their union can be computed in $\mathcal{O}(r \log k + m_h + m_w)$ time and $\mathcal{O}(r + m_h + m_w)$ space.*

Proof. We assume first that all rectangles have height at most k.

Let us partition the plane into horizontal strips of height k. Thus, each of the rectangles is divided into at most two pieces. The algorithm performs a line sweep in each of the strips.

Let the sweep line move from left to right. The events in the sweep correspond to the left and right sides of rectangles. The events can be sorted left-to-right, across all strips simultaneously, in $\mathcal{O}(r + m_h)$ time using bucket sort [19].

For each strip, the sweep line stores a data structure that allows insertion and deletion of intervals with integer coordinates in $[0\mathinner{.\,.} k]$ and querying for the total length of the union of the intervals that are currently stored. This corresponds to the operations of the data structure from Lemma 2.5 for $m = k$ (with elements corresponding to unit intervals), which supports insertions and deletions in $\mathcal{O}(\log k)$ time and queries in $\mathcal{O}(1)$ time after $\mathcal{O}(k)$-time preprocessing per strip. The total preprocessing time is $\mathcal{O}(m_h)$ and, since the total number of events in all strips is at most $2r$, the sweep works in $\mathcal{O}(r \log k)$ time.

Finally, let us consider the width-m_w rectangles. Each of them induces an interval on the second component. First, in $\mathcal{O}(r + m_h)$ time, the union S of these intervals, represented as a union of pairwise disjoint maximal intervals, is computed by bucket sorting the endpoints of the intervals. Then, each maximal interval in S is partitioned by the strips and the resulting subintervals are inserted into the data structures of the respective strips before the sweep. In total, at most $2r + m_h/k$ additional intervals are inserted, so the time complexity is still $\mathcal{O}((r + m_h/k) \log k + m_h + m_w) = \mathcal{O}(r \log k + m_h + m_w)$. □

We arrive at the main result of this section.

Theorem 5.6. *The number of k-antipower fragments in a word of length n can be computed in $\mathcal{O}(nk \log k)$ time and $\mathcal{O}(nk)$ space.*

Proof. We use Lemma 5.2 to express the sets $\mathit{WeakP}_k(d)$ for $d = 1, \dots, \lfloor n/k \rfloor$ as unions of $\mathcal{O}(nk)$ interval chains. This takes $\mathcal{O}(nk)$ time. Using Lemma 5.4, each chain is represented on the corresponding grid $\mathcal{G}_d$ as the union of a constant number of rectangles, with each rectangle of height at most k or width exactly d. Denoting by r_d the number of rectangles in $\mathcal{G}_d$, we can bound the total number of rectangles by $\sum_d r_d = \mathcal{O}(nk)$.

As the next step, in each grid we assign consecutive numbers to the (sorted) first coordinates of rectangle vertices and repeat the same procedure for the second coordinates. This can be done in $\mathcal{O}(nk)$ time, for all the grids simultaneously, using bucket sort [19]. This gives new coordinates to rectangle vertices; the new coordinates also store the original values. After this transformation, rectangles with height at most k retain this property and rectangles with width d retain maximal width. Moreover, the coordinates of the rectangles are now within $[0\mathinner{.\,.} 2r_d)$, so the application of Lemma 5.5 costs $\mathcal{O}(r_d \log k)$ time and $\mathcal{O}(r_d)$ space. One can readily verify that the underlying procedure can be adapted to compute the total area in the original coordinate values. In total, computing $|\mathit{WeakP}_k(d)|$ for all d costs $\mathcal{O}(nk \log k)$ time and $\mathcal{O}(nk)$ space.

In the end, the number of (k,d)-antipower fragments is calculated as $n - kd + 1 - |\mathit{WeakP}_k(d)|$. $\square$

6. Reporting antipowers and answering antipower queries

The same technique can be used to report all k-antipower fragments. In the grid representation, they correspond to grid cells of $\mathcal{G}_d$ that are not covered by any rectangle. Hence, in Lemma 5.5, instead of computing the area of the rectangles with the aid of Lemma 2.5, we need to report all grid cells excluded from rectangles using Lemma 2.6. The computation takes $\mathcal{O}(r \log k + d + C_d)$ time where C_d is the number of reported cells. By plugging this routine into the algorithm of Theorem 5.6, we obtain the following result.

Theorem 6.1. *All fragments of a word of length n being k-antipowers can be reported in $\mathcal{O}(nk \log k + C)$ time and $\mathcal{O}(nk)$ space, where C is the size of the output.*

Finally, we present our data structure for answering antipower queries that introduces a smooth trade-off between the two data structures of Badkobeh et al. [3] (see Fact 1.2). Let us recall that an antipower query (i,j,k) asks to check if a fragment $w[i\mathinner{.\,.}j]$ of the word w is a k-antipower.

Theorem 6.2. *Assume that a word of length n is given. For every $r \in [1\mathinner{.\,.}n]$, there is a data structure of size $\mathcal{O}(n^2/r)$ that can be constructed in $\mathcal{O}(n^2/r)$ time and answers antipower queries in $\mathcal{O}(r)$ time.*

Proof. Let w be a word of length n and let $r \in [1\mathinner{.\,.}n]$. If an antipower query (i,j,k) satisfies $k \le r$, we answer it in $\mathcal{O}(k)$ time using Fact 1.2(a). This is always $\mathcal{O}(r)$ time, and the data structure requires $\mathcal{O}(n)$ space.

Otherwise, if $w[i\mathinner{.\,.}j]$ is a k-antipower, then its base is at most n/r. Our data structure will let us answer antipower queries for every such base in $\mathcal{O}(1)$ time.

Let us consider a positive integer $b \le n/r$. We group the length-b fragments of w by the remainder modulo b of their starting position. For a remainder $g \in [0\mathinner{.\,.}b-1]$ and index $i \in [0\mathinner{.\,.}\lfloor\frac{n-g}{b}\rfloor)$, we store, as $A^b_g[i]$, the smallest index $j > i$ such that $w[jb+g\mathinner{.\,.}(j+1)b+g) = w[ib+g\mathinner{.\,.}(i+1)b+g)$ ($j = \infty$ if it does not exist). We also store a data structure for range minimum queries over A^b_g for each group; it uses linear space, takes linear time to construct, and answers queries in constant time (see [23]). The tables take $\mathcal{O}(n)$ space for a fixed b, which gives $\mathcal{O}(n^2/r)$ in total. They can also be constructed in $\mathcal{O}(n^2/r)$ total time, as shown in the following claim.

Claim 6.3. *The tables A^b_g for all $b \in [1\mathinner{.\,.}m]$ and $g \in [0\mathinner{.\,.}b-1]$ can be constructed in $\mathcal{O}(nm)$ time.*

Proof. Let us assign to each fragment of w of length at most m an identifier in $[0\mathinner{.\,.}n)$ such that factors corresponding to two equal-length fragments are equal if and only if their identifiers are equal. For length-1 fragments, this requires sorting the alphabet symbols, which can be done in $\mathcal{O}(n)$ time for an integer alphabet. For factors of length $\ell > 1$, we construct pairs that consist of the identifiers of the length-$(\ell-1)$ prefix and length-1 suffix and bucket sort the pairs. This gives $\mathcal{O}(nm)$ time in total.

To construct the tables A_g^b for a given b, we use an auxiliary array D that is indexed by identifiers in $[0\,..\,n)$. Initially, all its elements are set to ∞. For a given g, the indices i are considered in descending order. For each i, we take as x the identifier of the factor $w[ib+g\,..\,(i+1)b+g)$, set $A_g^b[i]$ to $D[x]$ and then $D[x]$ to i. Afterwards, in the same loop, all such values $D[x]$ are reset to ∞. For given b and g, both loops take $\mathcal{O}(n/b)$ time. $\square$

Given an antipower query (i,j,k) such that $(j-i+1)/k = b$, we set

$$g = i \bmod b, \quad i' = \left\lfloor \tfrac{i}{b} \right\rfloor, \quad j' = \left\lfloor \tfrac{j+1}{b} \right\rfloor - 2,$$

and ask a range minimum query on $A_g^b[i'], \dots, A_g^b[j']$. Then, $w[i\,..\,j]$ is a k-antipower if and only if the query returns a value that is at least $j'+2$. $\square$

7. Counting different k-antipower factors

7.1. Warmup: Counting different antisquare factors

Let us first show how to count different antisquare factors, that is, different 2-antipowers in a word w of length n.

Recall that the *suffix tree* of a word w is a compact trie representing all the suffixes of the word $w\$$, where \$ is a special end-marker. The root, the branching nodes, and the leaves are explicit in the suffix tree, whereas the remaining nodes are stored implicitly. Explicit and implicit nodes of the suffix tree are simply called its nodes. Each implicit node is represented as its position within a compacted edge. The string-depth of a node v is the length of the path from v to the root in the uncompacted version of the trie. The *locus* of a factor of w is the node it corresponds to. The suffix tree of a word of length n can be constructed in $\mathcal{O}(n)$ time [24].

Proposition 7.1. *The number of different antisquare factors in a word of length n can be computed in $\mathcal{O}(n)$ time.*

Proof. The algorithm counts different factors of even length and subtracts the number of different square factors. The latter can be computed in $\mathcal{O}(n)$ time [25, 26]. The former can be computed by counting (explicit and implicit) nodes of the suffix tree of w at even string-depths. For every edge of the suffix tree, this number can be easily retrieved in constant time. $\square$

We will use the same idea, i.e., subtract the number of weak k-powers from the number of all factors of length divisible by k, to count the number of different k-antipower factors. The algorithm requires at some point an auxiliary data structure that answers the following queries related to the suffix tree.

A *weighted ancestor query* in the suffix tree, given a leaf v and a non-negative integer d, returns the ancestor of v located at depth d (being an explicit or implicit node). A weighted ancestor query can be used to compute, for a factor u of w given by its occurrence, the locus of u in the suffix tree.

Fact 7.2 ([27, Section 7])**.** *A batch of m weighted ancestor queries (for any rooted tree of n nodes with positive polynomially-bounded integer weights of edges) can be answered in $\mathcal{O}(n+m)$ time.*

7.2. Representing the set of weak powers

We say that $x = y_0 \cdots y_{k-1}$, where $|y_0| = \cdots = |y_{k-1}| = d$, is a *weak (k,i,j,d)-power* if $i < j$, $y_i = y_j$, and this is the "leftmost" pair of equal factors among $y_0, \ldots, y_{k-1}$, i.e., for any $i' < j'$ such that $y_{i'} = y_{j'}$, either $i' > i$, or $i' = i$ and $j' > j$. This definition satisfies the following uniqueness property.

Observation 7.3. *A weak (k,d)-power is a weak (k,i,j,d)-power for exactly one pair of indices $0 \le i < j < k$.*

We denote by $\mathit{WeakP}_{k,i,j}(d)$ the set of starting positions of weak (k,i,j,d)-powers in w; see Fig. 9. The following lemma shows that this set can be computed efficiently. Let us recall that $\mathit{Squares}(q,d)$ denotes the set of starting positions of gapped (q,d)-square fragments.

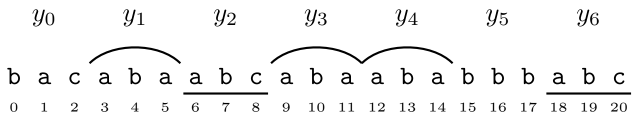


Figure 9: This weak $(7,3)$-power is actually a weak $(7,1,3,3)$-power. We have $0 \in \mathit{WeakP}_{7,1,3}(3)$, since $3 \in \mathit{Squares}(1,3)$, $3 \notin \mathit{Squares}(0,3)$, and $0 \notin \mathit{Squares}(q,3)$ for $q \in [0\mathinner{..}5]$.

Lemma 7.4. *For a given k, the sets $\mathit{WeakP}_{k,i,j}(d)$ for all $d = 1, \ldots, \lfloor n/k \rfloor$ and $0 \le i < j < k$ have interval representations of total size $\mathcal{O}(nk^4 \log k)$ which can be computed in $\mathcal{O}(nk^4 \log k)$ time.*

Proof. Let us note that $a \in \mathit{WeakP}_{k,i,j}(d)$ if and only if all the following conditions are satisfied:

1. $a + i \cdot d \in \mathit{Squares}(j-i-1, d)$
2. $a + i \cdot d \notin \mathit{Squares}(q, d)$ for $q < j-i-1$
3. for every $c \in [0\mathinner{..}i)$ and $q \le k-c-2$, we have $a + c \cdot d \notin \mathit{Squares}(q,d)$.

Intuitively, if $y_0 \cdots y_{k-1}$, with all factors of length d, is a weak (k,i,j,d)-power, then the first condition corresponds to $y_i = y_j$, the second condition to $y_i \ne y_{j'}$ for $i < j' < j$, and the third condition to $y_{i'} \ne y_{j'}$ for $i' < i$ and $i' < j' < k$.

Hence, $\mathit{WeakP}_{k,i,j}(d) = (A_{i,j}(d) \setminus (B_{i,j}(d) \cup C_{i,j}(d))) \cap [0\mathinner{.\,.} n-kd]$, where

$$A_{i,j}(d) = \mathit{Squares}(j-i-1,d) \ominus (i \cdot d),$$
$$B_{i,j}(d) = \bigcup_{q=0}^{j-i-2} \left(\mathit{Squares}(q,d) \ominus (i \cdot d)\right),$$
$$C_{i,j}(d) = \bigcup_{c=0}^{i-1} \bigcup_{q=0}^{k-c-2} \left(\mathit{Squares}(q,d) \ominus (c \cdot d)\right).$$

By Corollary 4.2, the interval representations of all sets $\mathit{Squares}(q,d)$ for $0 \le q \le k-2$ and $1 \le d \le n/k$ can be computed in $\mathcal{O}(nk^2)$ time. By Lemma 5.2, the total size of interval representations of sets $\mathit{Squares}(q,d)$ over all $d \ge 2k-2$ is $\mathcal{O}(nk)$. We further have:

Claim 7.5. $\sum_{q=0}^{k-2} \sum_{d=1}^{2k-3} |\mathcal{R}(\mathit{Squares}(q,d))| = \mathcal{O}(nk \log k)$.

Proof. The interval representation of the set $\mathit{Squares}(q,d)$ has size $\mathcal{O}(n/d)$. Indeed, if $a < b < a+d$ and $a,b \in \mathit{Squares}(q,d)$, then $c \in \mathit{Squares}(q,d)$ for any $a < c < b$, so the endpoints of any two consecutive intervals in the representation are at least d positions apart. Hence, the total size of interval representations of the sets in question is $\mathcal{O}(k \sum_{d=1}^{2k-3} n/d) = \mathcal{O}(nk \log k)$. □

In conclusion, $\sum_{q=0}^{k-2} \sum_{d=1}^{\lfloor n/k \rfloor} |\mathcal{R}(\mathit{Squares}(q,d))| = \mathcal{O}(nk \log k)$.

If $X_1, \dots, X_p \subseteq \mathbb{Z}$, then $|\mathcal{R}(\bigcup_{i=1}^p X_i)| \le \sum_{i=1}^p |\mathcal{R}(X_i)|$. Consequently, for any i, j, and d,

$$|\mathcal{R}(A_{i,j}(d))| + |\mathcal{R}(B_{i,j}(d))| + |\mathcal{R}(C_{i,j}(d))| \le$$
$$|\mathcal{R}(\mathit{Squares}(j-i-1,d))| + \sum_{q=0}^{k-2} |\mathcal{R}(\mathit{Squares}(q,d))| + (k-2) \sum_{q=0}^{k-2} |\mathcal{R}(\mathit{Squares}(q,d))| \le$$
$$\le k \sum_{q=0}^{k-2} |\mathcal{R}(\mathit{Squares}(q,d))|.$$

Hence, over all i,j,d, the size of these interval representations does not exceed

$$k^2(k+2) \sum_{d=1}^{\lfloor n/k \rfloor} \sum_{q=0}^{k-2} |\mathcal{R}(\mathit{Squares}(q,d))| = \mathcal{O}(nk^4 \log k).$$

Finally, Lemma 2.4 can be used to compute the sets $A_{i,j}(d) \setminus (B_{i,j}(d) \cup C_{i,j}(d))$ in $\mathcal{O}(nk^4 \log k)$ total time (note that set subtraction can be computed as intersection with set complement). □

We say that a weak (k,i,j,d)-power $y_0 \cdots y_{k-1}$ is *generated* by an MGR or a generalized run γ if the $(j-i-1,d)$-square $y_i \cdots y_j$ is generated by γ.

We denote by $\mathit{WeakP}_{k,i,j}(d,\gamma)$ the set of starting positions of weak (k,i,j,d)-powers generated by γ. By Lemma 4.1**(b)**, this set is an interval. It can be readily verified that the intervals generated in the above lemma can be labeled by the MGR or generalized run γ that generated them. This labelling is unique due to the following simple observation.

Observation 7.6. *For any different MGRs or generalized runs γ_1, γ_2, the intervals $\mathit{WeakP}_{k,i,j}(d,\gamma_1)$ and $\mathit{WeakP}_{k,i,j}(d,\gamma_2)$ neither overlap nor touch (contain adjacent positions).*

Proof. It suffices to note that for any $q \le k-2$ and d, the sets $\mathit{Squares}(q,d,\gamma_1)$ and $\mathit{Squares}(q,d,\gamma_2)$ neither overlap not touch (as already noted in the proof of Corollary 4.2). □

Let us first show how to count different weak (k,i,j,d)-powers for $i>0$. The case of $i=0$ will be handled in Section 7.5.

Definition 7.7. Let $i>0$. We say that a function g that assigns to every weak (k,i,j,d)-power factor x of w a position $g(x) \in [0\mathinner{.\,.} kd)$ is a *synchronizer* if for a given MGR or generalized run γ, for every $a \in \mathit{WeakP}_{k,i,j}(d,\gamma)$ the value $a+g(w[a]\cdots w[a+kd-1])$ is the same.

Note that a synchronizer function is defined on factors of w, not on fragments; i.e., it has the same value for every occurrence of the same weak (k,i,j,d)-power factor.

We will now show how to efficiently construct a synchronizer in the case of $i>0$. For a fragment $\alpha = w[a\mathinner{.\,.} b]$ of w, let us denote $\mathsf{start}(\alpha)=a$ and $\mathsf{end}(\alpha)=b$.

Lemma 7.8. *A function* synch *that assigns to every weak (k,i,j,d)-power x, for $i>0$, such that $x=w[a]\cdots w[a+kd-1]$ and $a \in \mathit{WeakP}_{k,i,j}(d,\gamma)$, the position $\mathsf{start}(\gamma)-a$, is a synchronizer; see Fig. 10.*

Proof. Clearly, for any positions $a_1,a_2 \in \mathit{WeakP}_{k,i,j}(d,\gamma)$ we have

$$a_1+\mathsf{synch}(w[a_1]\cdots w[a_1+kd-1]) = a_2+\mathsf{synch}(w[a_2]\cdots w[a_2+kd-1]) = \mathsf{start}(\gamma).$$

Now let us show that synch is indeed a function on the set of weak k-power factors, i.e., that its value does not depend on the particular occurrence of a weak k-power and that is satisfies $\mathsf{synch}(x) \in [0\mathinner{.\,.}|x|)$. Let $w[a]\cdots w[a+kd-1] = y_0\cdots y_{k-1} = x$ be an occurrence of a weak (k,i,j,d)-power for length-d words $y_0,\dots,y_{k-1}$ and let γ be the MGR or generalized run that generates it. We have $y_i=y_j$ and γ has period $p=(j-i)d$. Let $r=\max\{b<i\cdot d : x[b]\ne x[b+p]\}$. We have $r>(i-1)d$, since otherwise we would have $y_{i-1}=y_{j-1}$ and x would not be a weak (k,i,j,d)-power. Then position $r+1$ corresponds to the starting position of γ, i.e., $\mathsf{synch}(x)=\mathsf{start}(\gamma)-a=r+1$. Hence, indeed this value does not depend on the position a and $\mathsf{synch}(x)\in[0\mathinner{.\,.}|x|)$. □

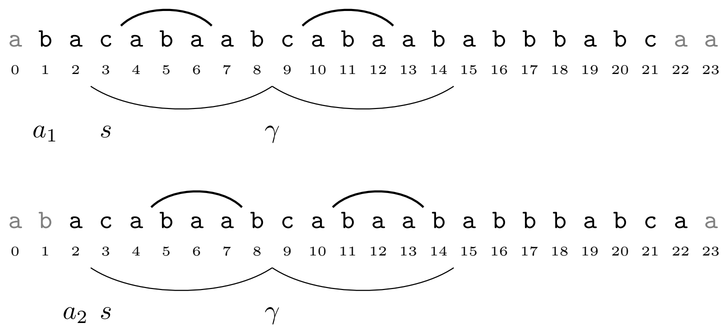


Figure 10: Here, γ is a run starting at position $s = \mathsf{start}(\gamma) = 2$ and we have $a_1 = 1, a_2 = 2 \in \mathit{WeakP}_{7,1,3}(3, \gamma)$. Moreover, $\mathsf{synch}(w[a_1] \cdots w[a_1 + 20]) = s - a_1 = 2$ and $\mathsf{synch}(w[a_2] \cdots w[a_2 + 20]) = s - a_2 = 1$.

7.3. Reduction to Path Pairs Problem

We say that T is a *compact tree* if it is a rooted tree with positive integer weights on edges. If an edge weight is $e > 1$, this edge contains $e - 1$ implicit nodes. We make an assumption that the depth of a compact tree with N explicit nodes does not exceed N. A *path* in a compact tree is an upwards or downwards path that connects two explicit nodes. Let us introduce the following convenient auxiliary problem.

Problem 7.9. Path Pairs Problem

Input: Two compact trees T and T' containing up to N explicit nodes each and a set P of M pairs (π, π') of equal-length paths where π is a path going downwards in T and π' is a path going upwards in T'.

Output: $|\bigcup_{(\pi,\pi')\in P} \mathsf{Induced}(\pi, \pi')|$, where by $\mathsf{Induced}(\pi, \pi')$ we denote the set of pairs of (explicit or implicit) nodes (u, u') such that u is the ith node on π and u' is the ith node on π', for some i.

Example 7.10. Let us consider the instance of Path Pairs Problem from Fig. 11. We have $P = \{(\pi_1, \pi'_1), (\pi_2, \pi'_2), (\pi_3, \pi'_3)\}$, where

- $\pi_1 = 1 \to 6$, $\pi'_1 = 8 \to 3$ (solid lines),
- $\pi_2 = 2 \to 10$, $\pi'_2 = 10 \to 1$ (dotted lines),
- $\pi_3 = 1 \to 14$, $\pi'_3 = 13 \to 1$ (dashed lines).

Then

$$\begin{aligned}
\mathsf{Induced}(\pi_1, \pi'_1) &= \{(1,8),(2,7),(3,6),(4,5),(5,4),(6,3)\},\\
\mathsf{Induced}(\pi_2, \pi'_2) &= \{(2,10),(3,6),(4,5),(7,4),(8,3),(9,2),(10,1)\},\\
\mathsf{Induced}(\pi_3, \pi'_3) &= \{(1,13),(2,12),(12,11),(13,2),(14,1)\}.
\end{aligned}$$

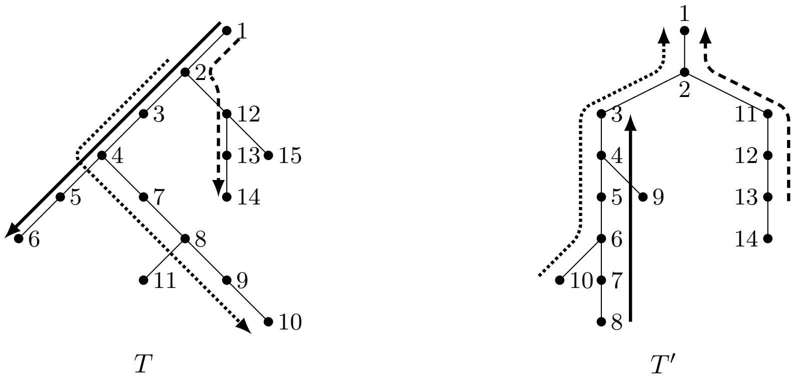


Figure 11: Illustration of Path Pairs Problem and Example 7.10. For simplicity, the trees in this example do not contain implicit nodes.

In total $|\bigcup_{i=1}^{3} \mathsf{Induced}(\pi_i, \pi_i')| = 16$ and $\mathsf{Induced}(\pi_1, \pi_1') \cap \mathsf{Induced}(\pi_2, \pi_2') = \{(3,6),(4,5)\}$.

Synchronizers let us reduce the problem in scope to the auxiliary problem.

Lemma 7.11. *Computing the number of different weak (k,i,j,d)-powers for given k and all $0 < i < j < k$, $d \le \frac{n}{k}$ in a word of length n reduces in $\mathcal{O}(nk^4 \log k)$ time to an instance of the* Path Pairs Problem *with $M, N = \mathcal{O}(nk^4 \log k)$.*

Proof. Let us consider the suffix tree T of w and the suffix tree T' of w^R.

For an interval $[a\,..\,b]$ in the interval representation of $\mathit{WeakP}_{k,i,j}(d)$, let us denote $q = a + \mathsf{synch}(w[a] \cdots w[a+kd-1])$. By Observation 7.6, $q = c + \mathsf{synch}(w[c] \cdots w[c+kd-1])$ for all $c \in [a\,..\,b]$. We create a downwards path π in T that connects the loci of $w[q\,..\,a+kd)$ and $w[q\,..\,b+kd)$ and an upwards path π' in T' that connects the loci of $(w[a\,..\,q))^R$ and $(w[b\,..\,q))^R$. We use weighted ancestor queries (Fact 7.2) to find the endpoints of the paths in the suffix trees, which can be explicit or implicit nodes.

Finally, the endpoints of the paths are made explicit in both trees. This can be achieved by grouping the endpoints by the compact edges they belong to and sorting them, within each edge, in the order of non-decreasing string-depth, which can be done in linear time via radix sort.

The resulting instance of the Path Pairs Problem is equivalent to counting the number of different weak powers thanks to the fact that synchronizers are defined for factors.

By Lemma 7.4, the number of intervals in the interval representation of $\mathit{WeakP}_{k,i,j}(d)$ over all $0 < i < j$ and d is $\mathcal{O}(nk^4 \log k)$. Each of them produces one pair of paths. In the end, we obtain $\mathcal{O}(nk^4 \log k)$ paths in two compact trees containing $\mathcal{O}(nk^4 \log k)$ explicit nodes each. The conclusion follows. □

7.4. Solution to Path Pairs Problem

Let us recall the notion of a *heavy-path decomposition* of a rooted tree T that was introduced in [28]. Here, we only consider explicit nodes of T. A path in T is a sequence of nodes. For each non-leaf node u of T, the heavy edge (u, v) is a downwards edge for which the subtree rooted at v has the maximal number of leaves (in case of several such subtrees, we fix one of them). The remaining edges are called light. A heavy path is a maximal path containing only heavy edges. A known property of the heavy-path decomposition is that the path from any leaf u in T towards the root visits at most $\log N$ heavy paths, where N is the number of nodes of T.

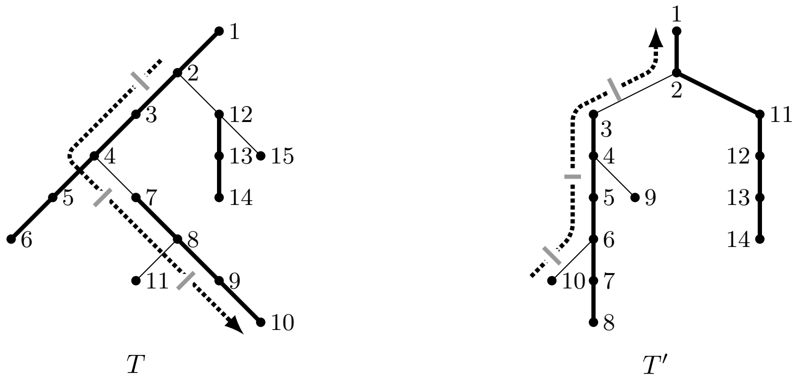


Figure 12: Partitioning of the second pair of paths from Example 7.10 into $2 \to 2, 3 \to 4, 7 \to 8, 9 \to 10$ and $10 \to 10, 6 \to 5, 4 \to 3, 2 \to 1$ along the heavy paths (drawn as thick edges).

In the solution to the Path Pairs Problem, we compute the heavy path decompositions of both trees T and T'. For each pair of paths (π, π') in P, we decompose each path π, π' into maximal fragments belonging to different heavy paths. Note that the decomposition of the upwards path π' can be computed in $\mathcal{O}(\log n)$ time assuming that each tree node stores the topmost node in its heavy path and the decomposition of the downwards path π can be computed in $\mathcal{O}(\log n)$ time by traversing π in the reverse direction. Then we further decompose the paths π, π' into maximal subpaths $\pi = \pi_1, \ldots, \pi_\ell$, $\pi' = \pi'_1, \ldots, \pi'_\ell$ so that the lengths of π_i and π'_i are the same and each π_i and each π'_i is a fragment of one heavy path in T and in T', respectively; we have $\ell \le 2 \log N$. For an illustration, see Fig. 12. Finally, we create new pairs of paths (π_i, π'_i), label each of them by the pair of heavy paths they belong to, and group them by their labels. This can be done using radix sort in $\mathcal{O}(N + M \log N)$ time, since the number of new path pairs is $\mathcal{O}(M \log N)$ and the number of heavy paths in each tree is $\mathcal{O}(N)$.

In the end, we obtain $\mathcal{O}(M \log N)$ very simple instances of the Path Pairs Problem, in each of which the compact trees T and T' are single paths corresponding to pairs of heavy paths from the original compact trees. We call such an instance *special*. The total number of path pairs across the special instances is $\mathcal{O}(M \log N)$. In the lemma below, we use the assumption that the depth of the initial compact tree is at most N in order to efficiently sort the depths of path endpoints.

Lemma 7.12. *The answers to up to K special instances of the* Path Pairs Problem *containing compact trees of depth at most N and at most K paths in total can be computed in $\mathcal{O}(N+K)$ time.*

Proof. For convenience let us reverse the order of edges in the tree T' of each instance so that both paths in each path pair lead downwards. Let us number the (explicit and implicit) nodes of trees T and T' top-down as $0, 1, \ldots, \mathcal{O}(N)$ in every instance. Then a path pair (π, π') such that π connects nodes with numbers i and j and π' connects nodes with numbers i' and j', with $j-i = j'-i'$, can be viewed as a diagonal segment that connects points (i, i') and (j, j') in a 2D grid. Thus, each instance reduces to counting the number of grid points that are covered by the segments. Again for convenience we can rotate each grid by 45 degrees to make the segments horizontal.

This problem can easily be solved by a top-down, and then left-to-right sweep. We only need the segment endpoints to be ordered first by the vertical, and then by the horizontal coordinate. This ordering can be achieved using radix sort in $\mathcal{O}(N+K)$ time across all instances. $\square$

This concludes the proof of the following lemma.

Lemma 7.13. Path Pairs Problem *can be solved in $\mathcal{O}(N + M \log N)$ time.*

7.5. Counting different weak powers with $i = 0$

We say that word v is a *cyclic shift* of word u if there exist words x and y such that $u = xy$ and $v = yx$. For a word s, by $\mathsf{minrot}(s)$ we denote a position $i \in [0\,..\,|s|)$ such that $s[i\,..\,|s|)s[0\,..\,i)$ is the lexicographically minimum cyclic shift of s. If there is more than one such position (i.e., that s is a power of a shorter word), we select as $\mathsf{minrot}(s)$ the *leftmost* such position.

If $i = 0$, we partition every interval $A = \mathit{WeakP}_{k,0,j}(d, \gamma)$ into four (possibly empty) intervals. Let

$$J_1 = [\mathsf{start}(\gamma)\,..\,\mathsf{end}(\gamma) - kd + 1], \quad J_2 = J_1 \cap [0\,..\,\mathsf{start}(\gamma) + \mathsf{per}(\gamma)).$$

We denote $\mathsf{pos}(\gamma) = \mathsf{minrot}(\gamma[0\,..\,\mathsf{per}(\gamma)))$ and

$$I_1 = J_2 \cap [0\,..\,\mathsf{start}(\gamma) + \mathsf{pos}(\gamma)], \quad I_2 = J_2 \setminus I_1, \quad I_3 = J_1 \setminus J_2,$$
$$I_4 = [\mathsf{start}(\gamma)\,..\,\mathsf{end}(\gamma)] \setminus J_1.$$

Example 7.14. If $\mathsf{per}(\gamma) + kd \le |\gamma|$, then

$$I_1 = [\mathsf{start}(\gamma)\,..\,\mathsf{pos}(\gamma)], \qquad I_2 = (\mathsf{pos}(\gamma)\,..\,\mathsf{start}(\gamma) + \mathsf{per}(\gamma)),$$
$$I_3 = [\mathsf{start}(\gamma) + \mathsf{per}(\gamma)\,..\,\mathsf{end}(\gamma) - kd + 1], \qquad I_4 = [\mathsf{end}(\gamma) - kd + 2\,..\,\mathsf{end}(\gamma)].$$

We define $WeakP^q_{k,j}(d,\gamma)$ as $WeakP_{k,0,j}(d,\gamma) \cap I_q$ for $q = 1,2,3,4$; see Fig. 13 for an example. By the following observation, these intervals will be of interest only for $q = 1,2,4$, since each weak power generated for $q = 3$ is also generated for some $q \in \{1,2,4\}$.

Observation 7.15. *If $a \in WeakP^3_{k,j}(d,\gamma)$, then $a' = a - \mathsf{per}(\gamma)$ satisfies $a' \in WeakP_{k,0,j}(d,\gamma)$ and $w[a \mathinner{.\,.} a+kd) = w[a' \mathinner{.\,.} a'+kd)$. In this case, γ is a generalized run.*

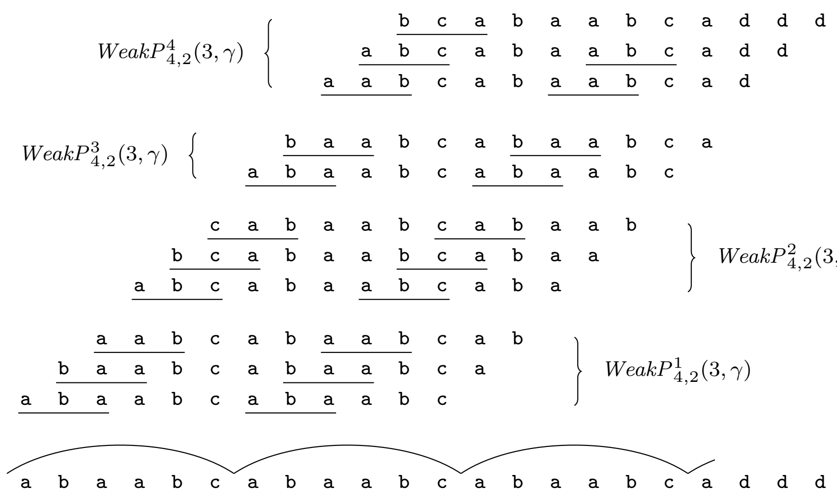


Figure 13: The sets $WeakP^q_{4,2}(3,\gamma)$ for a run γ. Note that the weak powers from the third set occur also in the first set. For $a \in WeakP^1_{4,2}(3,\gamma)$, $a + \mathsf{synch}(a) = \mathsf{start}(\gamma) + 2$. For $a \in WeakP^2_{4,2}(3,\gamma)$, $a+\mathsf{synch}(a) = \mathsf{start}(\gamma)+8$. For $a \in WeakP^4_{4,2}(3,\gamma)$, $a+\mathsf{synch}(a) = \mathsf{end}(\gamma)$.

We can then extend Definition 7.7 by saying that a function synch on weak $(k,0,j,d)$-power factors that assigns to each of them a number in $[0 \mathinner{.\,.} kd)$ is a *0-synchronizer* if $a + \mathsf{synch}(w[a] \cdots w[a+kd-1])$ is the same for each element $a \in WeakP^q_{k,j}(d,\gamma)$, for a given MGR or generalized run γ and $q \in \{1,2,4\}$. This lets us extend Lemma 7.8 as follows.

Lemma 7.16. *A function synch that assigns to every weak $(k,0,j,d)$-power x, such that $x = w[a] \cdots w[a+kd-1]$ and $a \in WeakP_{k,0,j}(k,d,\gamma)$, a number:*

- $\mathsf{start}(\gamma) + \mathsf{pos}(\gamma) - a$ *if* $a \in WeakP^1_{k,j}(d,\gamma)$
- $\mathsf{start}(\gamma) + \mathsf{pos}(\gamma) + \mathsf{per}(\gamma) - a$ *if* $a \in WeakP^2_{k,j}(d,\gamma)$
- $\mathsf{end}(\gamma) - a$ *if* $a \in WeakP^4_{k,j}(d,\gamma)$

is a 0-synchronizer. (See also Fig. 13.)

Proof. The proof in the case that $a \in WeakP^4_{k,j}(d,\gamma)$ is analogous to the proof of Lemma 7.8. In the first two cases, for a weak k-power $y_0 \ldots y_{k-1}$ we have

$\mathsf{synch}(y_0 \dots y_{k-1}) = \mathsf{minrot}(y_0 \dots y_{j-1})$. Therefore, for any positions $a_1, a_2 \in \mathit{WeakP}^q_{k,j}(d,\gamma)$, we have

$$a_1 + \mathsf{synch}(w[a_1] \cdots w[a_1 + kd - 1]) = a_2 + \mathsf{synch}(w[a_2] \cdots w[a_2 + kd - 1]).$$

This shows that synch is indeed a 0-synchronizer. $\square$

We use the following internal queries in texts by Kociumaka [29] to efficiently partition the intervals comprising $\mathit{WeakP}_{k,i,j}(d,\gamma)$ into maximal intervals that belong to $\mathit{WeakP}^q_{k,i,j}(d,\gamma)$.

Fact 7.17 ([29])**.** *One can preprocess a word w of length n in $\mathcal{O}(n)$ time so that $\mathsf{minrot}(s)$ can be computed in $\mathcal{O}(1)$ time for any factor s of w.*

Then, the problem of counting different weak $(k,0,j,d)$-powers reduces to the Path Pairs Problem, as in the previous section.

Lemma 7.18. *Computing the number of different weak $(k,0,j,d)$-powers for given k and all $0 < j < k$, $d \le \frac{n}{k}$ in a word of length n reduces in $\mathcal{O}(nk^3 \log k)$ time to an instance of the* Path Pairs Problem *with $M, N = \mathcal{O}(nk^3 \log k)$.*

We finally arrive at the main result of this section.

Theorem 7.19. *The number of different k-antipower factors in a word of length n can be computed in $\mathcal{O}(nk^4 \log k \log n)$ time.*

Proof. Let w be a word of length n. We reduce counting different k-antipower factors of w to counting the numbers of different factors of w of length that is divisible by k and of different weak k-power factors of w. As in the proof of Proposition 7.1, the former can be computed in $\mathcal{O}(n)$ time using the suffix tree of w. By Observation 7.3, every weak (k,d)-power is a weak (k,i,j,d)-power for exactly one pair of indices $0 \le i < j < k$. We reduce counting the number of different weak (k,i,j,d)-power factors of w to instances of the Path Pairs Problem with $N, M = \mathcal{O}(nk^4 \log k)$ using Lemmas 7.11 and 7.18 for $i > 0$ and $i = 0$, respectively, and solve these instances in $\mathcal{O}(nk^4 \log k \log n)$ time using Lemma 7.13. $\square$

References

[1] G. Fici, A. Restivo, M. Silva, L. Q. Zamboni, Anti-powers in infinite words, in: I. Chatzigiannakis, M. Mitzenmacher, Y. Rabani, D. Sangiorgi (Eds.), Automata, Languages and Programming, ICALP 2016, Vol. 55 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 124:1–124:9. doi:10.4230/LIPIcs.ICALP.2016.124.

[2] G. Fici, A. Restivo, M. Silva, L. Q. Zamboni, Anti-powers in infinite words, Journal of Combinatorial Theory, Series A 157 (2018) 109–119. doi:10.1016/j.jcta.2018.02.009.

[3] G. Badkobeh, G. Fici, S. J. Puglisi, Algorithms for anti-powers in strings, Information Processing Letters 137 (2018) 57–60. doi:10.1016/j.ipl.2018.05.003.

[4] H. Alamro, G. Badkobeh, D. Belazzougui, C. S. Iliopoulos, S. J. Puglisi, Computing the antiperiod(s) of a string, in: N. Pisanti, S. P. Pissis (Eds.), 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019, Vol. 128 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019, pp. 32:1–32:11. doi:10.4230/LIPIcs.CPM.2019.32.

[5] A. S. Fraenkel, J. Simpson, How many squares can a string contain?, Journal of Combinatorial Theory. Series A 82 (1) (1998) 112–120. doi:10.1006/jcta.1997.2843.

[6] A. Deza, F. Franek, A. Thierry, How many double squares can a string contain?, Discrete Applied Mathematics 180 (2015) 52–69. doi:10.1016/j.dam.2014.08.016.

[7] T. Kociumaka, J. Radoszewski, W. Rytter, J. Straszyński, T. Waleń, W. Zuba, Efficient representation and counting of antipower factors in words, in: C. Martín-Vide, A. Okhotin, D. Shapira (Eds.), Language and Automata Theory and Applications - 13th International Conference, LATA 2019, Vol. 11417 of Lecture Notes in Computer Science, Springer, 2019, pp. 421–433. doi:10.1007/978-3-030-13435-8_31.

[8] R. Kolpakov, M. Podolskiy, M. Posypkin, N. Khrapov, Searching of gapped repeats and subrepetitions in a word, Journal of Discrete Algorithms 46-47 (2017) 1–15. doi:10.1016/j.jda.2017.10.004.

[9] Y. Tanimura, Y. Fujishige, T. I, S. Inenaga, H. Bannai, M. Takeda, A faster algorithm for computing maximal α-gapped repeats in a string, in: C. S. Iliopoulos, S. J. Puglisi, E. Yilmaz (Eds.), String Processing and Information Retrieval, SPIRE 2015, Vol. 9309 of Lecture Notes in Computer Science, Springer, 2015, pp. 124–136. doi:10.1007/978-3-319-23826-5_13.

[10] M. Crochemore, R. Kolpakov, G. Kucherov, Optimal bounds for computing α-gapped repeats, in: A. Dediu, J. Janousek, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications, LATA 2016, Vol. 9618 of Lecture Notes in Computer Science, Springer, 2016, pp. 245–255. doi:10.1007/978-3-319-30000-9_19.

[11] P. Gawrychowski, T. I, S. Inenaga, D. Köppl, F. Manea, Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes - finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets, Theory of Computing Systems 62 (1) (2018) 162–191. doi:10.1007/s00224-017-9794-5.

[12] T. I, D. Köppl, Improved upper bounds on all maximal α-gapped repeats and palindromes, Theoretical Computer Science 753 (2019) 1–15. doi:10.1016/j.tcs.2018.06.033.

[13] M. Dumitran, P. Gawrychowski, F. Manea, Longest gapped repeats and palindromes, Discrete Mathematics and Theoretical Computer Science 19 (4) (2017) 4. doi:10.23638/DMTCS-19-4-4.

[14] D. Kosolobov, Online detection of repetitions with backtracking, in: F. Cicalese, E. Porat, U. Vaccaro (Eds.), Combinatorial Pattern Matching - 26th Annual Symposium, CPM 2015, Vol. 9133 of Lecture Notes in Computer Science, Springer, 2015, pp. 295–306. doi:10.1007/978-3-319-19929-0_25.

[15] R. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: 40th Annual Symposium on Foundations of Computer Science, FOCS 1999, IEEE Computer Society, 1999, pp. 596–604. doi:10.1109/SFFCS.1999.814634.

[16] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, The "runs" theorem, SIAM Journal on Computing 46 (5) (2017) 1501–1514. doi:10.1137/15M1011032.

[17] J. Fischer, S. Holub, T. I, M. Lewenstein, Beyond the runs theorem, in: C. S. Iliopoulos, S. J. Puglisi, E. Yilmaz (Eds.), String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, Vol. 9309 of Lecture Notes in Computer Science, Springer, 2015, pp. 277–286. doi:10.1007/978-3-319-23826-5_27.

[18] S. Holub, Prefix frequency of lost positions, Theoretical Computer Science 684 (2017) 43–52. doi:10.1016/j.tcs.2017.01.026.

[19] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.
URL http://mitpress.mit.edu/books/introduction-algorithms

[20] M. Rubinchik, A. M. Shur, Counting palindromes in substrings, in: G. Fici, M. Sciortino, R. Venturini (Eds.), String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Proceedings, Vol. 10508 of Lecture Notes in Computer Science, Springer, 2017, pp. 290–303. doi:10.1007/978-3-319-67428-5_25.

[21] J. L. Bentley, Algorithms for Klee's rectangle problems, Unpublished notes, Computer Science Department, Carnegie Mellon University (1977).

[22] H. Yu, Cell-probe lower bounds for dynamic problems via a new communication model, in: D. Wichs, Y. Mansour (Eds.), Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, ACM, 2016, pp. 362–374. doi:10.1145/2897518.2897556.

[23] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, P. Sumazin, Lowest common ancestors in trees and directed acyclic graphs, Journal of Algorithms 57 (2) (2005) 75–94. doi:10.1016/j.jalgor.2005.08.001.

[24] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, Journal of the ACM 47 (6) (2000) 987–1011. doi:10.1145/355541.355547.

[25] D. Gusfield, J. Stoye, Linear time algorithms for finding and representing all the tandem repeats in a string, Journal of Computer and System Sciences 69 (4) (2004) 525–546. doi:10.1016/j.jcss.2004.03.004.

[26] M. Crochemore, C. S. Iliopoulos, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, Extracting powers and periods in a word from its runs structure, Theoretical Computer Science 521 (2014) 29–41. doi:10.1016/j.tcs.2013.11.018.

[27] T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, A linear-time algorithm for seeds computation, ACM Transactions on Algorithms 16 (2) (2020) 27:1–27:23. doi:10.1145/3386369.

[28] D. D. Sleator, R. E. Tarjan, A data structure for dynamic trees, Journal of Computer and System Sciences 26 (3) (1983) 362–391. doi:10.1016/0022-0000(83)90006-5.

[29] T. Kociumaka, Minimal suffix and rotation of a substring in optimal time, in: R. Grossi, M. Lewenstein (Eds.), 27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016, Vol. 54 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, pp. 28:1–28:12. doi:10.4230/LIPIcs.CPM.2016.28.

Chapter 4

Counting Distinct Patterns in Internal Dictionary Matching

Counting Distinct Patterns in Internal Dictionary Matching

Panagiotis Charalampopoulos
King's College London, UK
University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Tomasz Kociumaka
Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Manal Mohamed
London, UK
manalabd@gmail.com

Jakub Radoszewski
University of Warsaw, Poland
Samsung R&D, Warsaw, Poland
jrad@mimuw.edu.pl

Wojciech Rytter
University of Warsaw, Poland
rytter@mimuw.edu.pl

Juliusz Straszyński
University of Warsaw, Poland
jks@mimuw.edu.pl

Tomasz Waleń
University of Warsaw, Poland
walen@mimuw.edu.pl

Wiktor Zuba
University of Warsaw, Poland
w.zuba@mimuw.edu.pl

Abstract

We consider the problem of preprocessing a text T of length n and a dictionary $\mathcal{D}$ in order to be able to efficiently answer queries $\textsc{CountDistinct}(i, j)$, that is, given i and j return the number of patterns from $\mathcal{D}$ that occur in the *fragment* $T[i \mathinner{.\,.} j]$. The dictionary is *internal* in the sense that each pattern in $\mathcal{D}$ is given as a fragment of T. This way, the dictionary takes space proportional to the number of patterns $d = |\mathcal{D}|$ rather than their total length, which could be $\Theta(n \cdot d)$. An $\tilde{\mathcal{O}}(n+d)$-size [1] data structure that answers $\textsc{CountDistinct}(i, j)$ queries $\mathcal{O}(\log n)$-approximately in $\tilde{\mathcal{O}}(1)$ time was recently proposed in a work that introduced internal dictionary matching [ISAAC 2019]. Here we present an $\tilde{\mathcal{O}}(n+d)$-size data structure that answers $\textsc{CountDistinct}(i, j)$ queries 2-approximately in $\tilde{\mathcal{O}}(1)$ time. Using range queries, for any m, we give an $\tilde{\mathcal{O}}(\min(nd/m, n^2/m^2) + d)$-size data structure that answers $\textsc{CountDistinct}(i, j)$ queries exactly in $\tilde{\mathcal{O}}(m)$ time. We also consider the special case when the dictionary consists of all square factors of the string. We design an $\mathcal{O}(n \log^2 n)$-size data structure that allows us to count distinct squares in a text fragment $T[i \mathinner{.\,.} j]$ in $\mathcal{O}(\log n)$ time.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases dictionary matching, internal pattern matching, squares

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.8

Related Version Full version at `https://arxiv.org/abs/2005.05681`.

Funding *Panagiotis Charalampopoulos*: Partially supported by ERC grant TOTAL under the EU's Horizon 2020 Research and Innovation Programme (agreement no. 677651).
Tomasz Kociumaka: Supported by ISF grants no. 1278/16 and 1926/19, a BSF grant no. 2018364, and an ERC grant MPM (no. 683064) under the EU's Horizon 2020 Research and Innovation Programme.
Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Juliusz Straszyński: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Tomasz Waleń: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Wiktor Zuba: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

[1] The $\tilde{\mathcal{O}}(\cdot)$ notation suppresses $\log^{\mathcal{O}(1)} n$ factors for inputs of size n.

1 Introduction

Internal Dictionary Matching was recently introduced in [5] as a generalization of Internal Pattern Matching. In the classical Dictionary Matching problem, we are given a dictionary $\mathcal{D}$ consisting of d patterns, and the goal is to preprocess $\mathcal{D}$ so that, presented with a text T, we can efficiently compute the occurrences of the patterns from $\mathcal{D}$ in T. In Internal Dictionary Matching, the text T is given in advance, the dictionary $\mathcal{D}$ is a set of fragments of T, and the Dictionary Matching queries can be asked for any fragment of T.

The Internal Pattern Matching problem consists in preprocessing a text T of length n so that we can efficiently compute the occurrences of a fragment of T in another fragment of T. A data structure of nearly linear size that allows for sublogarithmic-time Internal Pattern Matching queries was presented in [15], while a linear-size data structure allowing for constant-time Internal Pattern Matching queries in the case that the ratio between the lengths of the two factors is constant was presented in [18]. Other types of internal queries have been also studied; we refer the interested reader to [17].

In [5], several types of Internal Dictionary Matching queries about fragments $T[i\mathinner{..}j]$ in a string T were considered: $\textsc{Exists}(i,j)$, $\textsc{Report}(i,j)$, $\textsc{ReportDistinct}(i,j)$, $\textsc{Count}(i,j)$, $\textsc{CountDistinct}(i,j)$. Data structures of size $\tilde{\mathcal{O}}(n+d)$ and query time $\tilde{\mathcal{O}}(1+\mathsf{output})$ were shown for answering each of the first four queries, with $\textsc{Count}$ queries requiring most advanced techniques. For $\textsc{CountDistinct}$ queries, only a data structure answering these queries $\mathcal{O}(\log n)$-approximately was shown. In this work, we focus on more efficient data structures for such queries. $\textsc{CountDistinct}$ queries are formally defined as follows.

$\textsc{CountDistinct}$
Input: A text T of length n and a dictionary $\mathcal{D}$ consisting of d patterns, each given as a fragment $T[a\mathinner{..}b]$ of T (represented only by integers a,b).
Query: $\textsc{CountDistinct}(i,j)$: Count all distinct patterns $P \in \mathcal{D}$ that occur in $T[i\mathinner{..}j]$.

Observe that the input size is $n+d$, while the total length of strings in $\mathcal{D}$ could be $\Theta(n \cdot d)$.

We also consider a special case of this problem when the dictionary $\mathcal{D}$ is the set of all squares (i.e., strings of the form UU) in T. The case that $\mathcal{D}$ is the set of palindromes in T was considered by Rubinchik and Shur in [20].

▶ **Example 1.** Let us consider the following text:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	a	d	a	a	a	a	b	a	a	b	b	a	a	c

For the dictionary $\mathcal{D} = \{\mathtt{aa}, \mathtt{aaaa}, \mathtt{abba}, \mathtt{c}\}$, we have:

$$\textsc{CountDistinct}(5,12) = 2,\ \textsc{CountDistinct}(2,6) = 2,\ \textsc{CountDistinct}(2,12) = 3.$$

In particular, $T[5\mathinner{..}12]$ contains two distinct patterns from $\mathcal{D}$: `aa` (two occurrences) and `abba`. When the dictionary $\mathcal{D}$ represents all squares in T, we have

$$\textsc{CountDistinct}(5,12) = 3,\ \textsc{CountDistinct}(2,6) = 2,\ \textsc{CountDistinct}(2,12) = 4.$$

In particular, $T[5\mathinner{..}12]$ contains three distinct squares: `aa` (two occurrences), `bb` and `aabaab`.

Let us note that one could answer $\textsc{CountDistinct}(i,j)$ queries in time $\mathcal{O}(j-i)$ by running $T[i\mathinner{..}j]$ over the Aho–Corasick automaton of $\mathcal{D}$ [1] or in time $\tilde{\mathcal{O}}(d)$ by performing Internal Pattern Matching [18] for each element of $\mathcal{D}$ individually. Neither of these approaches is satisfactory as they can require $\Omega(n)$ time in the worst case.

Our results and a roadmap. We start with preliminaries in Section 2 and an algorithmic toolbox in Section 3. Our results for the case of a static dictionary are summarized in Table 1. Our solutions exploit string periodicity using runs and use data structures for variants of the (colored) orthogonal range counting problem and for auxiliary internal queries on strings.

Table 1 Our results for CountDistinct queries. Here, m is an arbitrary parameter.

Space	Preprocessing time	Query time	Variant	Section
$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(1)$	2-approximation	4
$\tilde{\mathcal{O}}(n^2/m^2+d)$	$\tilde{\mathcal{O}}(n^2/m+d)$	$\tilde{\mathcal{O}}(m)$	exact	5.1
$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(m)$	exact	5.2
$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(n\log^2 n)$	$\mathcal{O}(\log n)$	$\mathcal{D}=$ squares, exact	6

For the case of a dynamic dictionary, where queries are interleaved with insertions and deletions of patterns in the dictionary, it was shown in [5] that the product of the time to process an update and the time to answer an Exists(i,j) query cannot be $\mathcal{O}(n^{1-\epsilon})$ for any constant $\epsilon > 0$, unless the Online Boolean Matrix-Vector Multiplication conjecture [13] is false. In the full version of this paper, we outline a general scheme that adapts our data structures for the case of a dynamic dictionary. In particular, we show how to answer CountDistinct(i,j) queries 2-approximately in $\tilde{\mathcal{O}}(m)$ time and process each update in $\tilde{\mathcal{O}}(n/m)$ time, for any m.

2 Preliminaries

We begin with basic definitions and notation. Let $T = T[1]T[2]\cdots T[n]$ be a *string* of length $|T| = n$ over a linearly sortable alphabet Σ. The elements of Σ are called *letters*. By ε we denote an *empty string*. For two positions i and j on T, we denote by $T[i\,..\,j] = T[i]\cdots T[j]$ the *fragment* of T that starts at position i and ends at position j (the fragment is empty if $j < i$). A fragment is called *proper* if $i > 1$ or $j < n$. A fragment of T is represented in $\mathcal{O}(1)$ space by specifying the indices i and j. A *prefix* of T is a fragment that starts at position 1 and a *suffix* is a fragment that ends at position n. By UV and U^k we denote the concatenation of strings U and V and k copies of the string U, respectively. A *cyclic rotation* of a string U is any string V such that $U = XY$ and $V = YX$ for some strings X and Y.

Let U be a string of length m with $0 < m \leq n$. We say that U is a *factor* of T if there exists a fragment $T[i\,..\,i+m-1]$, called an *occurrence* of U in T, that is matches U. We then say that U occurs at the *starting position* i in T.

A positive integer p is called a *period* of T if $T[i] = T[i+p]$ for all $i = 1,\ldots,n-p$. We refer to the smallest period as *the period* of the string, and denote it by $\mathsf{per}(T)$. A string is called *periodic* if its period is no more than half of its length and *aperiodic* otherwise. The weak version of the periodicity lemma [9] states that if p and q are periods of a string T and satisfy $p + q \leq |T|$, then $\gcd(p,q)$ is also a period of T. A string T is called *primitive* if it cannot be expressed as U^k for a string U and an integer $k > 1$.

The elements of the dictionary $\mathcal{D}$ are called *patterns*. Henceforth, we assume that $\varepsilon \notin \mathcal{D}$, i.e., that the length of each $P \in \mathcal{D}$ is at least 1. We also assume that each pattern of $\mathcal{D}$ is given by the starting and ending positions of its occurrence in T. Thus, the size of the dictionary $d = |\mathcal{D}|$ refers to the number of patterns in $\mathcal{D}$ and not their total length. A *compact trie* of $\mathcal{D}$ is the trie of $\mathcal{D}$ in which all non-terminal nodes with exactly one child become implicit. The path-label $\mathcal{L}(v)$ of a node v is defined as the path-ordered concatenation of the string-labels of the edges in the root-to-v path. We refer to $|\mathcal{L}(v)|$ as the *string-depth* of v.

3 Algorithmic Tools

3.1 Modified Suffix Trees

A $\mathcal{D}$-*modified suffix tree* [5], denoted as $\mathcal{T}_{T,\mathcal{D}}$, of a given text T of length n and a dictionary $\mathcal{D}$ is obtained from the trie of $\mathcal{D} \cup \{T[i\mathinner{.\,.} n] : 1 \le i \le n\}$ by contracting, for each non-terminal node u other than the root, the edge from u to the parent of u. As a result, all the nodes of $\mathcal{T}_{T,\mathcal{D}}$ (except for the root) correspond to patterns in $\mathcal{D}$ or to suffixes of T. For $1 \le i \le n$, the node representing $T[i\mathinner{.\,.} n]$ is labelled with i; see Figure 1. For a dictionary $\mathcal{D}$ whose patterns are given as fragments of a text T, we can construct $\mathcal{T}_{T,\mathcal{D}}$ in $\mathcal{O}(|\mathcal{D}| + |T|)$ time [5].

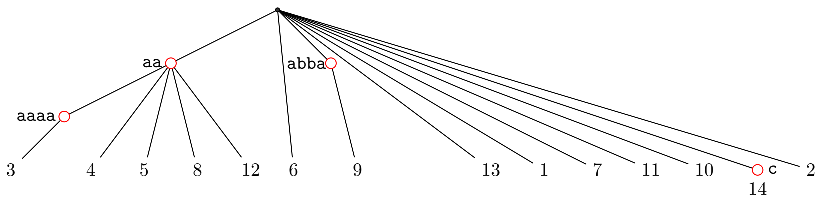


Figure 1 Example of a $\mathcal{D}$-modified suffix tree for text T = adaaaabaabbaac and dictionary $\mathcal{D} = \{$aa, aaaa, abba, c$\}$ (figure from [5]).

Let us denote by $\mathsf{Occ}(\mathcal{D})$ the set of all occurrences of dictionary patterns in T, that is, the set of all fragments of T that match a pattern in $\mathcal{D}$. Using $\mathcal{T}_{T,\mathcal{D}}$, the set $\mathsf{Occ}(\mathcal{D})$ can be computed in time $\mathcal{O}(n + d + |\mathsf{Occ}(\mathcal{D})|)$.

We say that a tree is a *weighted tree* if it is a rooted tree with an integer weight on each node v, denoted by $\omega(v)$, such that the weight of the root is zero and $\omega(u) < \omega(v)$ if u is the parent of v. We say that a node v is a *weighted ancestor at depth* ℓ of a node u if v is the top-most ancestor of u with weight of at least ℓ.

▶ **Theorem 2** ([2, Section 6.2.1]). *After $\mathcal{O}(n)$-time preprocessing, weighted ancestor queries for nodes of a weighted tree $\mathcal{T}$ of size n can be answered in $\mathcal{O}(\log\log n)$ time per query.*

The $\mathcal{D}$-modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$ is a weighted tree with the weight of each node defined as the length of the corresponding string. We define the *locus* of a fragment $T[i\mathinner{.\,.} j]$ in $\mathcal{T}_{T,\mathcal{D}}$ to be the weighted ancestor of the leaf i at string-depth $j - i + 1$.

3.2 Auxiliary Internal Queries

In a *Bounded LCP* query, one is given two fragments U and V of T and needs to return the longest prefix of U that occurs in V; we denote such a query by *BoundedLCP*(U, V). Kociumaka et al. [18] presented several tradeoffs for this problem, including the following.

▶ **Lemma 3** ([18],[17, Corollary 7.3.4]). *Given a text T of length n, one can construct in $\mathcal{O}(n\sqrt{\log n})$ time an $\mathcal{O}(n)$-size data structure that answers Bounded LCP queries in $\mathcal{O}(\log^\epsilon n)$ time, for any constant $\epsilon > 0$.*

Recall that $\textsc{Count}(i, j)$ returns the number of all occurrences of all the patterns of $\mathcal{D}$ in $T[i\mathinner{.\,.} j]$. The following result was proved in [5].

▶ **Lemma 4** ([5]). *The $\textsc{Count}(i, j)$ queries can be answered in $\mathcal{O}(\log^2 n/\log\log n)$ time with an $\mathcal{O}(n + d\log n)$-size data structure, constructed in $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$ time.*

3.3 Geometric Toolbox

For a set of n points in 2D, a range counting query returns the number of points in a given rectangle.

▶ **Theorem 5** (Chan and Pătrașcu [4]). *Range counting queries for n integer points in 2D can be answered in time $\mathcal{O}(\log n/\log\log n)$ with a data structure of size $\mathcal{O}(n)$ that can be constructed in time $\mathcal{O}(n\sqrt{\log n})$.*

A quarterplane is a range of the form $(-\infty, x_1] \times (-\infty, x_2]$. By reversing coordinates we can also consider quarterplanes with some dimensions of the form $[x_i, \infty)$. Let us state the following result on orthant color range counting due to Kaplan et al. [14] in the special case of two dimensions.

▶ **Theorem 6** ([14, Theorem 2.3]). *Given n colored integer points in 2D, we can construct in $\mathcal{O}(n \log n)$ time an $\mathcal{O}(n \log n)$-size data structure that, given any quarterplane Q, counts the number of distinct colors with at least one point in Q in $\mathcal{O}(\log n)$ time.*

We show how to apply geometric methods to a special variant of the CountDistinct problem, where we are interested in a small subset of occurrences of each pattern.

Let $\mathcal{D} = \{P_1, P_2, \dots, P_d\}$ and $\mathcal{S}$ be a family of sets $S_1, \dots, S_d$ such that $S_k \subseteq \mathsf{Occ}(P_k)$, where $\mathsf{Occ}(P_k)$ is the set of positions of T where P_k occurs. Let $\|\mathcal{S}\| = \sum_k |S_k|$. For each pattern P_k, we call the positions in the set S_k the *special positions of* P_k. Counting distinct patterns occurring at their special positions in $T[i\,..\,j]$ is called $\textsc{CountDistinct}_{\mathcal{S}}(i, j)$.

▶ **Lemma 7.** *The $\textsc{CountDistinct}_{\mathcal{S}}(i, j)$ queries can be answered in $\mathcal{O}(\log n)$ time with a data structure of size $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ that can be constructed in $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ time.*

Proof. We assign a different integer color c_k to every pattern $P_k \in \mathcal{D}$. Then, for each fragment $T[a\,..\,b] = P_k$ such that $a \in S_k$, we add point (a, b) with color c_k in an initially empty 2D grid $\mathcal{G}$. A $\textsc{CountDistinct}_{\mathcal{S}}(i, j)$ query reduces to counting different colors in the range $[i, \infty) \times (-\infty, j]$ of $\mathcal{G}$. The complexities follow from Theorem 6. ◀

3.4 Runs

A *run* (also known as a *maximal repetition*) is a periodic fragment $R = T[a\,..\,b]$ which can be extended neither to the left nor to the right without increasing the period $p = \mathsf{per}(R)$, i.e., $T[a-1] \neq T[a+p-1]$ and $T[b-p+1] \neq T[b+1]$ provided that the respective positions exist. If $\mathcal{R}$ is the set of all runs in a string T of length n, then $|\mathcal{R}| \leq n$ [3] and $\mathcal{R}$ can be computed in $\mathcal{O}(n)$ time [19]. The *exponent* $\mathsf{exp}(R)$ of a run R with period p is $|R|/p$. The sum of exponents of runs in a string of length n is $\mathcal{O}(n)$ [3, 19].

The *Lyndon root* of a periodic string U is the lexicographically smallest rotation of its $\mathsf{per}(U)$-length prefix. If L is the Lyndon root of a periodic string U, then U may be represented as (L, r, a, b); here $U = L[|L|-a+1\,..\,|L|]L^r L[1\,..\,b]$, and r is called the *rank* of U. Note that the minimal rotation of a fragment of a text can be computed in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$-time preprocessing [16].

For a periodic fragment U, let $\mathsf{run}(U)$ be the run with the same period that contains U.

▶ **Lemma 8** ([3, 7, 17]). *For a periodic fragment U, $\mathsf{run}(U)$ and its Lyndon root are uniquely determined and can be computed in constant time after linear-time preprocessing.*

We use runs in 2-approximate $\textsc{CountDistinct}(i, j)$ queries and in counting squares.

4 Answering CountDistinct 2-Approximately

4.1 CountDistinct for Extended or Contracted Fragments

For two positions ℓ and r, we define $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$ as the longest prefix of $T[\ell\,..\,r]$ that matches some pattern $P \in \mathcal{D}$; the length of such prefix is at most $r - \ell + 1$. Let us show how to compute the locus of $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$ in the $\mathcal{D}$-modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$. To this end, we preprocess $\mathcal{T}_{T,\mathcal{D}}$ for weighted ancestor queries and store at every node v of $\mathcal{T}_{T,\mathcal{D}}$ a pointer $p(v)$ to the nearest ancestor u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$. To return $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$, we find the locus u of $T[\ell\,..\,r]$ in the $\mathcal{D}$-modified suffix tree. We return $p(u)$ if $|\mathcal{L}(u)| = |T[\ell\,..\,r]|$ and $p(v)$, where v is the parent of u, otherwise.

Lemma 9 applies the $\mathcal{D}$-modified suffix tree to the problem of maintaining the count of distinct patterns occurring in a fragment subject to extending or shrinking the fragment.

▶ **Lemma 9.** *For any constant $\epsilon > 0$, given* CountDistinct(i, j), *one can compute* CountDistinct$(i \pm 1, j)$ *and* CountDistinct$(i, j \pm 1)$ *in $\mathcal{O}(\log^{\epsilon} n)$ time with an $\mathcal{O}(n + d)$-size data structure that can be constructed in $\mathcal{O}(n\sqrt{\log n} + d)$ time.*

Proof. We only present a data structure for CountDistinct$(i \pm 1, j)$ queries. Queries CountDistinct$(i, j \pm 1)$ can be handled analogously by building the same data structure for the reverses of all the strings in scope.

We show how to compute the number of patterns $P \in \mathcal{D}$ whose only occurrence in some fragment $T[\ell\,..\,r]$ starts at position ℓ. The computation of CountDistinct$(i \pm 1, j)$ follows directly by setting $j = r$ and ℓ equal to $i - 1$ or i.

Data structure. We preprocess T for Bounded LCP queries (Lemma 3) and construct the $\mathcal{D}$-modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$ of text T and dictionary $\mathcal{D}$. In addition, we preprocess $\mathcal{T}_{T,\mathcal{D}}$ for weighted ancestor queries and store at every node v of $\mathcal{T}_{T,\mathcal{D}}$ the number $\#(v)$ of the ancestors u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$.

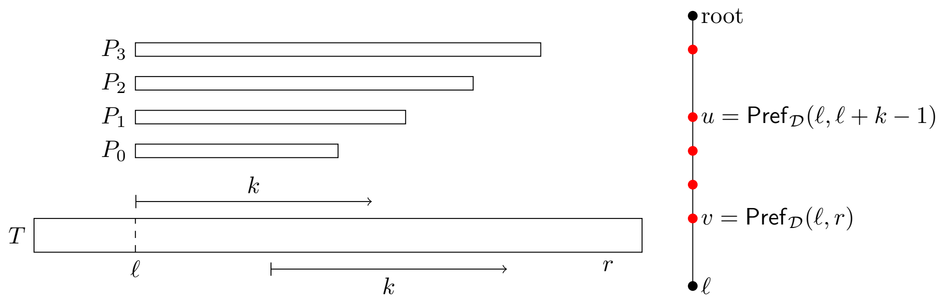

P_3
P_2
P_1
P_0
k
T
ℓ
k
r
root
$u = \mathsf{Pref}_{\mathcal{D}}(\ell, \ell + k - 1)$
$v = \mathsf{Pref}_{\mathcal{D}}(\ell, r)$
ℓ

Figure 2 The setting of Lemma 9. Left: text T. Right: the path from the root of $\mathcal{T}_{T,\mathcal{D}}$ to the leaf with path-label $T[\ell\,..\,n]$. The nodes of the path whose path-labels match some patterns from $\mathcal{D}$ are drawn in red. Here, P_0 is the longest pattern that occurs at ℓ and also has an occurrence in $T[\ell + 1\,..\,r]$; its locus in $\mathcal{T}_{T,\mathcal{D}}$ is $u = \mathsf{Pref}_{\mathcal{D}}(\ell, \ell + k - 1)$. The patterns that occur in $T[\ell\,..\,r]$ only at position ℓ are P_1, P_2 and P_3. The locus of P_3 is $v = \mathsf{Pref}_{\mathcal{D}}(\ell, r)$. Then, $\#(v) - \#(u) = 5 - 2 = 3$.

Query. We want to count patterns longer than $k = |BoundedLCP(T[\ell\,..\,r], T[\ell + 1\,..\,r])|$. Let $u = \mathsf{Pref}_{\mathcal{D}}(\ell, \ell + k - 1)$ and $v = \mathsf{Pref}_{\mathcal{D}}(\ell, r)$. The desired number of patterns is equal to $\#(v) - \#(u)$. See Figure 2 for a visualization. ◀

4.2 Auxiliary Operation

Two fragments $U = T[i_1 \mathinner{.\,.} j_1]$ and $V = T[i_2 \mathinner{.\,.} j_2]$ are called *consecutive* if $i_2 = j_1 + 1$. We denote the overlap $T[\max\{i_1, i_2\} \mathinner{.\,.} \min\{j_1, j_2\}]$ of U and V by $U \cap V$.

3-Fragments-Counting
Input: A text T of length n and a dictionary $\mathcal{D}$ consisting of d patterns
Query: Given three consecutive fragments F_1, F_2, F_3 in T such that $|F_1| = |F_3|$ and $|F_2| \geq 8 \cdot |F_1|$, count distinct patterns P from $\mathcal{D}$ that have an occurrence starting in F_1 and ending in F_3 and do not occur in either F_1F_2 or F_2F_3

Let us fix $|F_1| = |F_3| = x$ and $|F_2| = y \geq 8x$. Additionally, let us call an occurrence of $P \in \mathcal{D}$ that starts in fragment F_a and ends in fragment F_b an (F_a, F_b)-occurrence. We will call an (F_1, F_3)-occurrence an *essential occurrence.*

We say that a string S is *highly periodic* if $\mathsf{per}(S) \leq \frac{1}{4}|S|$. We first consider the case that all patterns in $\mathcal{D}$ are not highly periodic.

▶ **Lemma 10.** *If each $P \in \mathcal{D}$ is not highly periodic, then*

$$\text{3-Fragments-Counting}(F_1, F_2, F_3) = \\ \text{Count}(F_1F_2F_3) - \text{Count}(F_1F_2) - \text{Count}(F_2F_3) + \text{Count}(F_2).$$

Proof. Let us start with the following claim.

▷ Claim 11. Any $P \in \mathcal{D}$ that has an essential occurrence occurs exactly once in $F_1F_2F_3$.

Proof. We have $|F_1F_2F_3| = x + y + x = 2x + y$. String P has an essential occurrence, so $|P| \geq y$. Therefore, if there are two occurrences of P in $F_1F_2F_3$, then they overlap in

$$2|P| - (2x + y) \geq 2|P| - (\tfrac{1}{4}|P| + |P|) = \tfrac{3}{4}|P|$$

positions. This implies that P is highly periodic, which is a contradiction. ◁

Claim 11 shows that 3-Fragments-Counting(F_1, F_2, F_3) is equal to the number of essential occurrences. Let us prove that the stated formula does not count any (F_a, F_b)-occurrences other than (F_1, F_3)-occurrences.

- Each (F_1, F_2)-occurrence is registered when we add Count$(F_1F_2F_3)$ and unregistered when we subtract Count(F_1F_2). Similarly for (F_2, F_3)-occurrences.
- Each (F_2, F_2)-occurrence is registered when we add Count$(F_1F_2F_3)$, Count(F_2) and unregistered when we subtract Count(F_1F_2), Count(F_2F_3).
- Each (F_1, F_1)-occurrence is registered when we add Count$(F_1F_2F_3)$ and unregistered when we subtract Count(F_1F_2). Similarly for (F_3, F_3)-occurrences. ◀

We now proceed with answering 3-Fragments-Counting queries for the dictionary of highly periodic patterns.

▶ **Lemma 12.** *If F_2 is aperiodic, then there are no essential occurrences of highly periodic patterns. Otherwise, all essential occurrences of highly periodic patterns are generated by the same run, that is,* $\mathsf{run}(F_2)$.

Proof. The first claim follows from the fact that such an occurrence of a pattern $P \in \mathcal{D}$ has an overlap of length at least $2\mathsf{per}(P)$ with F_2 and hence $\mathsf{per}(P) \le \frac{1}{2}|F_2|$ is a period of F_2.

As for the second claim, it suffices to show that, for any pattern $P \in \mathcal{D}$ that has an essential occurrence, we have $\mathsf{per}(P) = \mathsf{per}(F_2)$. The inequalities $|F_2| \ge 2\mathsf{per}(F_2)$ and $|F_2| \ge 2\mathsf{per}(P)$ imply $|F_2| \ge \mathsf{per}(F_2) + \mathsf{per}(P)$. Hence, by the periodicity lemma, $q = \gcd(\mathsf{per}(P), \mathsf{per}(F_2))$ is a period of F_2. As $q \le \mathsf{per}(F_2)$, we conclude that $q = \mathsf{per}(F_2)$. Thus, $\mathsf{per}(F_2)$ divides $\mathsf{per}(P)$, and therefore $\mathsf{per}(P) = \mathsf{per}(F_2)$. This concludes the proof. ◀

For a periodic factor U of T, let $\textsc{Periodic}(U)$ denote the set of distinct patterns from $\mathcal{D}$ that occur in U and have the same shortest period. Let us make the following observation.

▶ **Observation 13.** *If all $P \in \mathcal{D}$ are highly periodic, F_2 is periodic, and $R = \mathsf{run}(F_2)$, then*

$$\textit{3-}\textsc{Fragments-Counting}(F_1, F_2, F_3) = \\ |\textsc{Periodic}(F_1F_2F_3 \cap R)| - |\textsc{Periodic}(F_1F_2 \cap R) \cup \textsc{Periodic}(F_2F_3 \cap R)|.$$

Next we now show how to efficiently evaluate the right-hand side of the formula in the observation above, using Theorem 5 for efficiently answering range counting queries in 2D.

We group all highly periodic patterns by Lyndon root and rank; for a Lyndon root L and a rank r, we denote by $\mathcal{D}^p_{L,r}$ the corresponding set of patterns. Then, we build the data structure of Theorem 5 for the set of points obtained by adding the point (a, b) for each $(L, r, a, b) \in \mathcal{D}^p_{L,r}$. We refer to the 2D grid underlying this data structure as $\mathcal{G}_{L,r}$. Note that the total number of points in the data structures over all Lyndon roots and ranks is $\mathcal{O}(d)$.

Each occurrence of a pattern (L, r, a, b) lies within some run in $\mathcal{R}$ with Lyndon root L. Let us state a simple fact.

▶ **Fact 14.** *A periodic string (L, r, a, b) occurs in a periodic string (L, r', a', b') if and only if at least one of the following conditions is met:*
(1) *$r = r'$, $a \le a'$, and $b \le b'$;*
(2) *$r = r' - 1$ and $a \le a'$;*
(3) *$r = r' - 1$ and $b \le b'$;*
(4) *$r \le r' - 2$.*

▶ **Lemma 15.** *One can compute $|\textsc{Periodic}(U)|$ for any periodic fragment U in time $\mathcal{O}(\log n / \log\log n)$ using a data structure of size $\mathcal{O}(n + d)$ that can be constructed in time $\mathcal{O}(n + d\sqrt{\log n})$.*

Proof. For $U = (L, r, a, b)$, we count points contained in at least one of the rectangles
(1) $(-\infty, a] \times (-\infty, b]$ in $\mathcal{G}_{L,r}$,
(2) $(-\infty, a] \times (-\infty, |L|]$ in $\mathcal{G}_{L,r-1}$,
(3) $(-\infty, |L|] \times (-\infty, b]$ in $\mathcal{G}_{L,r-1}$,
and we add to the count the number of patterns of the form (L, r', a, b) with $r' < r - 1$. For the latter term, it suffices to store an array $X_L[1 \mathinner{.\,.} t]$ such that $X_L[r] = \sum_{i=1}^{r} |\mathcal{D}^p_{L,i}|$, where t is the maximum rank of a pattern with Lyndon root L. The total size of these arrays is $\mathcal{O}(n)$ by the linearity of the sum of exponents of runs in a string [3, 19]. ◀

▶ Remark 16. In particular, in the proof of the above lemma, we count points that are contained within at least one out of a constant number of rectangles. Therefore, not only we can easily compute $|\textsc{Periodic}(U)|$, but similarly we are able to compute $|\textsc{Periodic}(U_1) \cup \textsc{Periodic}(U_2)|$ for some periodic factors U_1, U_2 of T.

We are now ready to prove the main result of this subsection.

▶ **Lemma 17.** *The 3-*Fragments-Counting(F_1, F_2, F_3) *queries can be answered in time* $\mathcal{O}(\log^2 n/\log\log n)$ *with a data structure of size* $\mathcal{O}(n + d\log n)$ *that can be constructed in* $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$ *time.*

Proof. By Lemma 10, in order to count the patterns that are not highly periodic, it suffices to perform three Count queries. To this end, we employ the data structure of Lemma 4 which answers Count queries in $\mathcal{O}(\log^2 n/\log\log n)$ time, occupies space $\mathcal{O}(n + d\log n)$, and can be constructed in time $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$.

We now proceed to counting highly periodic patterns. First, we check whether F_2 is periodic; this can be done in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$-time preprocessing of T [18, 17]. If F_2 is not periodic, then by Lemma 12 no highly periodic pattern has an essential occurrence, and we are thus done. If F_2 is periodic, three $|\text{Periodic}(U)|$ queries suffice to obtain the answer due to Observation 13. They can be efficiently answered due to Lemma 15 and Remark 16; the complexities are dominated by those for building the data structure for Count queries. ◀

4.3 Approximation Algorithm

Let us fix $\delta = \frac{1}{9}$. A fragment of length $\lfloor(1+\delta)^p\rfloor$ for any positive integer p will be called a *p-basic fragment*. Our data structure stores CountDistinct(i, j) for every basic fragment $T[i\,..\,j]$. Using Lemma 9, these values can be computed in $\mathcal{O}(n\log^{1+\epsilon} n + d)$ time with a sliding window approach. The space requirement is $\mathcal{O}(n\log n + d)$.

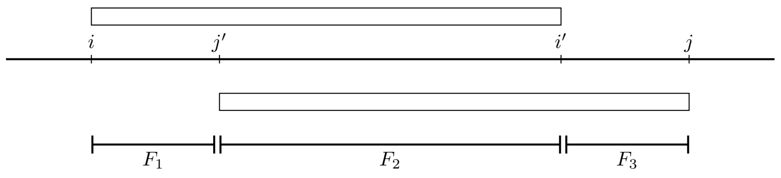


Figure 3 A 2-approximation of CountDistinct(i, j) is achieved using precomputed counts for basic factors $T[i\,..\,i']$ and $T[j'\,..\,j]$.

In order to answer an arbitrary CountDistinct(i, j) query, let $T[i\,..\,i']$ and $T[j'\,..\,j]$ be the longest prefix and suffix of $T[i\,..\,j]$ being a basic factor; see Figure 3. We sum up CountDistinct(i, i') and CountDistinct(j', j) and the result of a 3-Fragments-Counting query for $F_1 = T[i\,..\,j'-1]$, $F_2 = T[j'\,..\,i']$, $F_3 = T[i'+1\,..\,j]$. (Note that $(|F_1| + |F_2|)\cdot(1+\delta) > |F_1| + |F_2| + |F_3|$ implies $\delta(|F_1| + |F_2|) > |F_3|$, and since $|F_1| = |F_3|$, we have that $|F_1| = |F_3| \le \frac{1}{8}|F_2|$.) Now, a pattern $P \in \mathcal{D}$ is counted at least once if and only if it occurs in $T[i\,..\,j]$. Also, a pattern $P \in \mathcal{D}$ is counted at most twice (exactly twice if and only if it occurs in both F_1F_2 and F_2F_3). The above discussion and Lemma 17 yield the following result.

▶ **Theorem 18.** *The* CountDistinct(i, j) *queries can be answered 2-approximately in time* $\mathcal{O}(\log^2 n/\log\log n)$ *with a data structure of size* $\mathcal{O}((n + d)\log n)$ *that can be constructed in time* $\mathcal{O}(n\log^{1+\epsilon} n + d\log^{3/2} n)$ *for any constant* $\epsilon > 0$.

5 Time-Space Tradeoffs for Exact Counting

5.1 Tradeoff for Large Dictionaries

The following result is yet another application of Lemma 9.

▶ **Theorem 19.** *For any $m \in [1,n]$ and any constant $\epsilon > 0$, the* COUNTDISTINCT(i,j) *queries can be answered in $\mathcal{O}(m \log^\epsilon n)$ time using an $\mathcal{O}(n^2/m^2 + n + d)$-size data structure that can be constructed in $\mathcal{O}((n^2 \log^\epsilon n)/m + n\sqrt{\log n} + d)$ time.*

Proof. A fragment of the form $T[c_1m{+}1\,..\,c_2m]$ for integers c_1 and c_2 will be called a *canonical fragment*. Our data structure stores COUNTDISTINCT(i',j') for every canonical fragment $T[i'\,..\,j']$ and the data structure of Lemma 9. Hence the space complexity $\mathcal{O}(n^2/m^2 + n + d)$.

We can compute in $\mathcal{O}(n \log^\epsilon n)$ time COUNTDISTINCT(i',j) for a given i' and all j using Lemma 9. There are $\mathcal{O}(n/m)$ starting positions of canonical fragments and hence the counts for all canonical fragments can be computed in $\mathcal{O}((n^2 \log^\epsilon n)/m)$ time. Additional preprocessing time $\mathcal{O}(n\sqrt{\log n} + d)$ originates from Lemma 9.

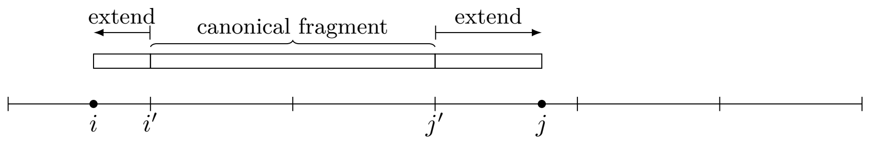


Figure 4 An illustration of the setting in the query algorithm underlying Theorem 19.

We can answer a COUNTDISTINCT(i,j) query in $\mathcal{O}(m \log^\epsilon n)$ time as follows. Let $T[i'\,..\,j']$ be the maximal canonical fragment contained in $T[i\,..\,j]$. We retrieve COUNTDISTINCT(i',j') for $T[i'\,..\,j']$. Then, we apply Lemma 9 $\mathcal{O}(m)$ times; each time we extend the fragment for which we count, until we obtain COUNTDISTINCT(i,j). See Figure 4. ◀

5.2 Tradeoff for Small Dictionaries

We call a set of strings $\mathcal{H}$ a *path-set* if all elements of $\mathcal{H}$ are prefixes of its longest element. We now show how to efficiently handle dictionaries that do not contain large path-sets.

▶ **Lemma 20.** *If $\mathcal{D}$ does not contain any path-set of size greater than k, then we can construct in $\mathcal{O}(kn \log n)$ time an $\mathcal{O}(kn \log n)$-size data structure that answers* COUNTDISTINCT(i,j) *queries in $\mathcal{O}(\log n)$ time.*

Proof. Let $\mathcal{D} = \{P_1, \dots, P_d\}$ and $\mathcal{S} = \{\mathsf{Occ}(P_1), \dots, \mathsf{Occ}(P_d)\}$. Every position of T contains at most k occurrences of patterns from $\mathcal{D}$. This implies that $\|\mathcal{S}\| \le kn$. We can obviously treat a COUNTDISTINCT(i,j) query as a COUNTDISTINCT$_{\mathcal{S}}(i,j)$ query. The complexities follow from Lemma 7. ◀

A proof of the following lemma is rather standard and is included in the full version of the paper.

▶ **Lemma 21.** *For any $k \in [1,n]$, we can compute a maximal family $\mathcal{F}$ of pairwise-disjoint path-sets in $\mathcal{D}$, each consisting of at least k elements, in $\mathcal{O}(n + d)$ time.*

We now combine Lemmas 3, 20 and 21 to get the main result of this section.

▶ **Theorem 22.** *For any $m \in [1,n]$ and any constant $\epsilon > 0$, the* COUNTDISTINCT(i,j) *queries can be answered in $\mathcal{O}(m \log^\epsilon n + \log n)$ time using an $\mathcal{O}((nd \log n)/m + d)$-size data structure that can be constructed in $\mathcal{O}((nd \log n)/m + d)$ time.*

Proof. We first apply Lemma 21 for $k = \lceil d/m \rceil$. We then have a decomposition of $\mathcal{D}$ to a family $\mathcal{F}$ of at most m path-sets and a set $\mathcal{D}'$ with no path-set of size greater than $\lfloor d/m \rfloor$. We directly apply Lemma 20 for $\mathcal{D}'$. In order to handle path-sets, we build the data structure of Lemma 3. Then, upon a CountDistinct(i, j) query, for each path-set $\mathcal{H} \in \mathcal{F}$, we compute the longest pattern in $\mathcal{H}$ that occurs in $T[i\,..\,j]$ using a Bounded LCP query followed by a predecessor query [24] in a structure that stores the lengths of the elements of $\mathcal{H}$, with the lexicographic rank in $\mathcal{H}$ stored as satellite information. The data structure of [24] is randomized, but it can be combined with deterministic dictionaries [21] using a simple two-level approach (see [23]), resulting in a deterministic *static* data structure. ◀

▶ Remark 23. Let us fix the query time to be $\mathcal{O}(m \log^{\epsilon} n)$ for $m = \Omega(\log n)$. Then, Theorem 22 outperforms Theorem 19 in terms of the required space for $d = o(n/(m \log n))$. For example, for $m = d = n^{1/4}$, the data structure of Theorem 22 requires space $\tilde{\mathcal{O}}(n)$ while the one of Theorem 19 requires space $\tilde{\mathcal{O}}(n\sqrt{n})$.

6 Internal Counting of Distinct Squares

The number of occurrences of squares could be quadratic, but we can construct a much smaller $\mathcal{O}(n \log n)$-size subset of these occurrences (called *boundary occurrences*) that, from the point of view of CountDistinct queries, gives almost the same answers. This is the main trick in this section. Distinct squares with a boundary occurrence in a given fragment can be counted in $\mathcal{O}(\log n)$ time due to Lemma 7. The remaining squares can be counted based on their structure: we show that they are all generated by the same run.

Now, the dictionary $\mathcal{D}$ is the set of all squares in T. By the following fact, $d = \mathcal{O}(n)$ and $\mathcal{D}$ can be computed in $\mathcal{O}(n)$ time.

▶ **Fact 24** ([7, 8, 10, 12]). *A string T of length n contains $\mathcal{O}(n)$ distinct square factors and they can all be computed in $\mathcal{O}(n)$ time.*

We say that an occurrence of a square U^2 is *induced* by a run R if it is contained in R and the shortest periods of U and R are the same. Every occurrence of a square is induced by exactly one run.

We need the following fact (note that it is false for the set of *all* runs; see [11]).

▶ **Fact 25.** *The sum of the lengths of all highly periodic runs is $\mathcal{O}(n \log n)$.*

Proof. We will prove that each position in T is contained in $\mathcal{O}(\log n)$ highly periodic runs. Let us consider all highly periodic runs R containing some position i, such that $m \le \mathsf{per}(R) < \frac{3}{2}m$ for some even integer m. Suppose for the sake of contradiction that there are at least 5 such runs. Note that each such run fully contains one of the fragments $T[i - 3m + 1 + t\,..\,i + t]$ for $t \in \{0, m, 2m, 3m\}$. By the pigeonhole principle, one of these four fragments is contained in at least two runs, say R_1 and R_2. In particular, the overlap of these runs is at least $3m \ge \mathsf{per}(R_1) + \mathsf{per}(R_2)$, which is a contradiction by the periodicity lemma. ◀

We define a family of occurrences $\mathcal{B} = B_1, \ldots, B_d$ such that, for each square U_i^2, the set B_i contains the leftmost and the rightmost occurrence of U_i^2 in every run. We call these *boundary occurrences*. Boundary occurrences of squares have the following property.

▶ **Lemma 26.** *$\|\mathcal{B}\| = \mathcal{O}(n \log n)$ and the set family $\mathcal{B}$ can be computed in $\mathcal{O}(n \log n)$ time.*

Proof. Let us define the *root* of a square U^2 to be U. A square is primitively rooted if its root is a primitive string. Let *p-squares* be primitively rooted squares, *np-squares* be the remaining ones. The number of occurrences of p-squares in a string of length n is $\mathcal{O}(n \log n)$ and they can all be computed in $\mathcal{O}(n \log n)$ time; see [6, 22].

We now proceed to np-squares. Note that for any highly periodic run R, the leftmost occurrence of each np-square induced by R starts in one of the first $\mathsf{per}(R)$ positions of R; a symmetric property holds for rightmost occurrences and last $\mathsf{per}(R)$ positions. In addition, it can be readily verified that such a position is the starting (resp. ending) position of at most $\mathsf{exp}(R)$ squares induced by R. It thus suffices to bound the sum of $\mathsf{exp}(R) \cdot \mathsf{per}(R)$ over all highly periodic runs R. The fact that $\mathsf{exp}(R) \cdot \mathsf{per}(R) = |R|$ concludes the proof of the combinatorial part by Fact 25.

For the algorithmic part, it suffices to iterate over the $\mathcal{O}(n)$ runs of T. ◀

▶ **Lemma 27.** *If $T[i\,..\,j]$ is non-periodic, $\textsc{CountDistinct}(i,j) = \textsc{CountDistinct}_{\mathcal{B}}(i,j)$.*

Proof. Let us consider an occurrence of a square U^2 inside $T[i\,..\,j]$. Let R be the run that induces this occurrence. By the assumption of the lemma, R does not contain $T[i\,..\,j]$. Then at least one of the boundary occurrences of U^2 in R is contained in $T[i\,..\,j]$. ◀

For a periodic fragment F of T, by $\mathit{RunSquares}(F)$ we denote the number of distinct squares that are induced by F (being a run if interpreted as a standalone string). The value $\mathit{RunSquares}(F)$ can be computed in $\mathcal{O}(1)$ time, as it was shown in e.g. [7].

Let F_1 be a prefix and F_2 be a suffix of a periodic fragment F, such that each of F_1 and F_2 is of length at most $\mathsf{per}(F)$ – and hence they are disjoint. By $\mathit{BSq}(F, F_1, F_2)$ ("bounded squares") we denote the number of distinct squares induced by F which have an occurrence starting in F_1 or ending in F_2.

▶ **Lemma 28.** *Given $\mathsf{per}(F)$, the $\mathit{BSq}(F, F_1, F_2)$ queries can be answered in $\mathcal{O}(1)$ time.*

Proof. We are to count distinct squares induced by F that start in F_1 or end in F_2.

We introduce an easier version of BSq queries. Let $\mathit{BSq}'(F, F_1) = \mathit{BSq}(F, F_1, \varepsilon)$ be the number of squares induced by F which start in its prefix F_1 of length at most $p := \mathsf{per}(F)$.

Reduction of $\boldsymbol{BSq}$ to $\boldsymbol{BSq'}$. First, observe that the set of squares induced by F starting at some position $q \in [1, p]$ and the set of squares induced by F ending at some position $q' \in [|F| - p + 1, |F|]$ are equal if $q \equiv q' + 1 \pmod{p}$ and disjoint otherwise. Also note that $F_2 = UV$ for some prefix V and some suffix U of $F[p]F[1\,..\,p-1]$; we consider this rotation of $F[1\,..\,p]$ to offset the $+1$ factor in the above modular equation. Let $|U| = a$ and $|V| = b$.

Then, by the aforementioned observation, we are to count distinct squares that start in some position in the set $[1, |F_1|] \cup [1, b] \cup [p - a + 1, p]$; see Figure 5.

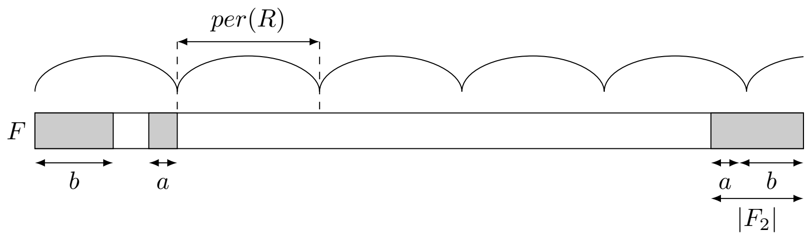


Figure 5 Reduction of BSq to BSq'; the case that $|F_1| \le b$.

Hence the computation of $\mathit{BSq}(F, F_1, F_2)$ is reduced to at most two instances of the special case when F_2 is the empty string.

Computation of $BSq'(F, F_1)$. The number of squares induced by F starting at $F[i]$ is $\lfloor(|F|-i+1)/(2p)\rfloor$. Consequently, $BSq'(F,F_1) = \sum_{i=1}^{|F_1|}\lfloor(|F|-i+1)/(2p)\rfloor = |F_1|\cdot t - \max\{0, |F_1|-k-1\}$, where $t = \lfloor|F|/(2p)\rfloor$ and $k = |F| \bmod (2p)$. ◀

▶ **Lemma 29.** *Assume that $F = T[i\,..\,j]$ is periodic and $R = T[a\,..\,b] = \mathsf{run}(T[i\,..\,j])$. Let $F_1 = T[i\,..\,a+p-1]$ and $F_2 = T[b-p+1\,..\,j]$, where $\mathsf{per}(R) = p$. Then:*

$$\textsc{CountDistinct}(i,j) = \textsc{CountDistinct}_{\mathcal{B}}(i,j) + RunSquares(F) - BSq(F,F_1,F_2). \quad (1)$$

Proof. In the sum $\textsc{CountDistinct}_{\mathcal{B}}(i,j) + RunSquares(F)$, all squares are counted once except for squares whose boundary occurrences are induced by R, which are counted twice. They are exactly counted in the term $BSq(F,F_1,F_2)$; see Figure 6. ◀

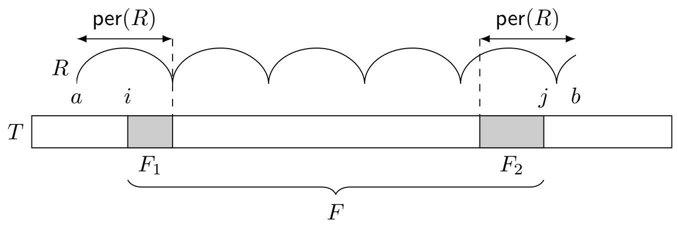


Figure 6 The setting in Lemma 29. Note that F_1 is empty if $i \geq a + \mathsf{per}(R)$; similarly for F_2.

▶ **Theorem 30.** *If $\mathcal{D}$ is the set of all square factors of T, then $\textsc{CountDistinct}(i,j)$ queries can be answered in $\mathcal{O}(\log n)$ time using a data structure of size $\mathcal{O}(n\log^2 n)$ that can be constructed in $\mathcal{O}(n\log^2 n)$ time.*

Proof. We precompute the set $\mathcal{B}$ in $\mathcal{O}(n\log n)$ time using Lemma 26 and perform $\mathcal{O}(n\log^2 n)$ time and space preprocessing for $\textsc{CountDistinct}_{\mathcal{B}}(i,j)$ queries.

In order to answer a $\textsc{CountDistinct}(i,j)$ query, first we ask a $\mathsf{run}(T[i\,..\,j])$ query of Lemma 8 to check if $T[i\,..\,j]$ is periodic.

We compute $\textsc{CountDistinct}_{\mathcal{B}}(i,j)$ which takes $\mathcal{O}(\log n)$ time due to Lemma 7. If $T[i\,..\,j]$ is non-periodic, then it is the final result due to Lemma 27.

Otherwise $T[i\,..\,j]$ is periodic. Let F, F_1, F_2 be as in Lemma 29. We answer $RunSquares(F)$ and $BSq(F,F_1,F_2)$ queries in $\mathcal{O}(1)$ time using the algorithm from [7] and Lemma 28, respectively. Finally, $\textsc{CountDistinct}(i,j)$ is computed using (1). ◀

7 Final Remarks

The general framework for dynamic dictionaries, presented in the full version of this paper, essentially consists in rebuilding a static data structure after every k updates. We return correct answers by performing individual queries for the patterns inserted or deleted from the dictionary since the data structure was built. In particular, we show that an application of this framework – with some tweaks – to the data structure of Section 4 yields the following.

▶ **Theorem 31.** *For any $k \in [1,n]$, we can construct a data structure in $\tilde{\mathcal{O}}(n+d)$ time, which processes each update to the dictionary in $\tilde{\mathcal{O}}(n/k)$ time and answers $\textsc{CountDistinct}(i,j)$ queries 2-approximately in $\tilde{\mathcal{O}}(k)$ time.*

We leave open the problem of whether an $\tilde{\mathcal{O}}(n+d)$-size data structure answering $\textsc{CountDistinct}(i,j)$ queries exactly in time $\tilde{\mathcal{O}}(1)$ exists.

References

1 Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

2 Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

3 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

4 Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 161–173. SIAM, 2010. doi:10.1137/1.9781611973075.15.

5 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, volume 149 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ISAAC.2019.22.

6 Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981. doi:10.1016/0020-0190(81)90024-7.

7 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

8 Antoine Deza, Frantisek Franek, and Adrien Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015. doi:10.1016/j.dam.2014.08.016.

9 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.

10 Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.

11 Amy Glen and Jamie Simpson. The total run length of a word. *Theoretical Computer Science*, 501:41–48, 2013. doi:10.1016/j.tcs.2013.06.004.

12 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004. doi:10.1016/j.jcss.2004.03.004.

13 Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th Annual ACM on Symposium on Theory of Computing, STOC 2015*, pages 21–30. ACM, 2015. doi:10.1145/2746539.2746609.

14 Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient colored orthogonal range counting. *SIAM Journal on Computing*, 38(3):982–1011, 2008. doi:10.1137/070684483.

15 Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.

16 Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPIcs*, pages 28:1–28:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CPM.2016.28.

17 Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018. URL: https://mimuw.edu.pl/~kociumaka/files/phd.pdf.

18 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.

19 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999*, pages 596–604. IEEE Computer Society, 1999. doi:10.1109/SFFCS.1999.814634.

20 Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, volume 10508 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5_25.

21 Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *Automata, Languages and Programming, ICALP 2008, Part I*, volume 5125 of *Lecture Notes in Computer Science*, pages 84–95. Springer, 2008. doi:10.1007/978-3-540-70575-8_8.

22 Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002. doi:10.1016/S0304-3975(01)00121-9.

23 Mikkel Thorup. Space efficient dynamic stabbing with fast queries. In *35th Annual ACM Symposium on Theory of Computing, STOC 2003*, pages 649–658. ACM, 2003. doi:10.1145/780542.780636.

24 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

Chapter 5

The Number of Repetitions in 2D-Strings

The Number of Repetitions in 2D-Strings

Panagiotis Charalampopoulos
Department of Informatics, King's College London, UK
Institute of Informatics, University of Warsaw, Poland
panagiotis.charalampopoulos@kcl.ac.uk

Jakub Radoszewski
Institute of Informatics, University of Warsaw, Poland
Samsung R&D Poland, Warsaw, Poland
jrad@mimuw.edu.pl

Wojciech Rytter
Institute of Informatics, University of Warsaw, Poland
rytter@mimuw.edu.pl

Tomasz Waleń
Institute of Informatics, University of Warsaw, Poland
walen@mimuw.edu.pl

Wiktor Zuba
Institute of Informatics, University of Warsaw, Poland
w.zuba@mimuw.edu.pl

Abstract

The notions of periodicity and repetitions in strings, and hence these of runs and squares, naturally extend to two-dimensional strings. We consider two types of repetitions in 2D-strings: *2D-runs* and *quartics* (quartics are a 2D-version of squares in standard strings). Amir et al. introduced 2D-runs, showed that there are $\mathcal{O}(n^3)$ of them in an $n \times n$ 2D-string and presented a simple construction giving a lower bound of $\Omega(n^2)$ for their number (*Theoretical Computer Science*, 2020). We make a significant step towards closing the gap between these bounds by showing that the number of 2D-runs in an $n \times n$ 2D-string is $\mathcal{O}(n^2 \log^2 n)$. In particular, our bound implies that the $\mathcal{O}(n^2 \log n + \mathsf{output})$ run-time of the algorithm of Amir et al. for computing 2D-runs is also $\mathcal{O}(n^2 \log^2 n)$. We expect this result to allow for exploiting 2D-runs algorithmically in the area of 2D pattern matching.

A quartic is a 2D-string composed of 2×2 identical blocks (2D-strings) that was introduced by Apostolico and Brimkov (*Theoretical Computer Science*, 2000), where by quartics they meant only *primitively rooted* quartics, i.e. built of a primitive block. Here our notion of quartics is more general and analogous to that of squares in 1D-strings. Apostolico and Brimkov showed that there are $\mathcal{O}(n^2 \log^2 n)$ occurrences of primitively rooted quartics in an $n \times n$ 2D-string and that this bound is attainable. Consequently the number of distinct primitively rooted quartics is $\mathcal{O}(n^2 \log^2 n)$. The straightforward bound for the maximal number of distinct general quartics is $\mathcal{O}(n^4)$. Here, we prove that the number of distinct general quartics is also $\mathcal{O}(n^2 \log^2 n)$. This extends the rich combinatorial study of the number of distinct squares in a 1D-string, that was initiated by Fraenkel and Simpson (*Journal of Combinatorial Theory, Series A*, 1998), to two dimensions.

Finally, we show some algorithmic applications of 2D-runs. Specifically, we present algorithms for computing all occurrences of primitively rooted quartics and counting all general distinct quartics in $\mathcal{O}(n^2 \log^2 n)$ time, which is quasi-linear with respect to the size of the input. The former algorithm is optimal due to the lower bound of Apostolico and Brimkov. The latter can be seen as a continuation of works on enumeration of distinct squares in 1D-strings using runs (Crochemore et al., *Theoretical Computer Science*, 2014). However, the methods used in 2D are different because of different properties of 2D-runs and quartics.

2012 ACM Subject Classification Theory of computation → Pattern matching

Keywords and phrases 2D-run, quartic, run, square

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.32

Funding *Panagiotis Charalampopoulos*: Partially supported by ERC grant TOTAL under the EU's Horizon 2020 Research and Innovation Programme (agreement no. 677651).
Jakub Radoszewski: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Tomasz Waleń: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.
Wiktor Zuba: Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

1 Introduction

Periodicity is one of the main and most elegant notions in stringology. It has been studied extensively both from the combinatorial and the algorithmic perspective; see e.g. the books [18, 25, 39]. A classic combinatorial result is the periodicity lemma due to Fine and Wilf [27]. From the algorithmic side, periodicity often poses challenges in pattern matching, due to the following fact: a pattern P can have many occurrences in a text T that are "close" to each other if and only if P has a "small" period. On the other hand, the periodic structure indeed allows us to overcome such challenges; see [18, 25].

Runs, also known as maximal repetitions, are a fundamental notion in stringology. A run is a periodic fragment of the text that cannot be extended without changing the period. Runs were introduced in [35]. Kolpakov and Kucherov presented an algorithm to compute all runs in a string in time linear with respect to the length of the string over a linearly-sortable alphabet [38]. Runs fully capture the periodicity of the underlying string and, since the publication of the algorithm for their linear-time computation, they have assumed a central role in algorithm design for strings. They have been exploited for text indexing [36], answering internal pattern matching queries in texts [16, 37], or reporting repetitions in a string [2, 15, 22], to name a few applications.

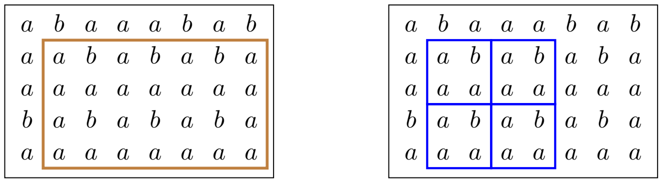
Kolpakov and Kucherov also posed the so-called runs conjecture which states that there are at most n runs in a string of length n. A long line of work on the upper [19, 20, 21, 31, 42, 43, 44] and lower bounds [30, 41, 45] was concluded by Bannai et al. who positively resolved the runs conjecture in [10] (see also an alternative proof in [23] and a tighter upper bound for binary strings from [28]).

A square is a concatenation of two copies of the same string. Fraenkel and Simpson [29] showed that a string of length n contains at most $2n$ distinct square factors. This bound was improved in [26, 34]. All distinct squares in a string of length n can be computed in $\mathcal{O}(n)$ time assuming an integer alphabet [11, 22, 33] (see [46] for an earlier $\mathcal{O}(n \log n)$ algorithm).

Pattern matching and combinatorics on 2D strings have been studied for more than 40 years, see e.g. [1, 4, 9, 14, 18, 25]. In this paper we consider 2-dimensional versions of runs, introduced by Amir et al. [5, 6], and of repetitions in 2D-strings, introduced by Apostolico and Brimkov [7]. As discussed in [6, 8], one could potentially exploit such repetitions in a 2D-string, which could for instance be an image, in order to compress it.

A *2D-run* in a 2D-string A is a subarray of A that is both horizontally periodic and vertically periodic and that cannot be extended by a row or column without changing the horizontal or vertical periodicity (a formal definition follows in Section 2); see Figure 1(a). Amir et al. [5, 6] have shown that the maximum number of 2D-runs in an $n \times n$ array is $\mathcal{O}(n^3)$ and presented an example with $\Theta(n^2)$ 2D-runs. In [6] they presented an $\mathcal{O}(n^2 \log n + \mathsf{output})$-time algorithm for computing 2D-runs.

A *quartic* is a configuration that is composed of 2×2 occurrences of an array W (see Figure 1(b)) and a *tandem* is a configuration consisting of two occurrences of an array W that share one side (Apostolico and Brimkov [7] also considered another type of tandems, which share one corner; see also [3]). An array W is called *primitive* if it cannot be partitioned into non-overlapping replicas of some array W'. Apostolico and Brimkov [7] considered only quartics and tandems with primitive W (we call them *primitively rooted*) and showed tight asymptotic bounds $\Theta(n^2 \log^2 n)$ and $\Theta(n^3 \log n)$ for the maximum number of occurrences of such quartics and tandems in an $n \times n$ array, respectively. In [8] they presented an optimal $\mathcal{O}(n^3 \log n)$-time algorithm for computing all *occurrences* of tandems with primitive W. This extends a result that a 1D-string of length n contains $\mathcal{O}(n \log n)$ occurrences of primitively rooted squares and they can all be computed in $\mathcal{O}(n \log n)$ time; see [17, 46]. In this paper we consider the numbers of *all distinct* quartics, which is a more complicated problem.



(a) a 2D-run **(b)** a quartic

Figure 1 Examples of a 2D-run and a quartic.

When computing 2D-runs we consider positioned runs: two 2D-runs with same content but starting in different points are considered distinct. However in case of quartics, similarly as in case of 1D-squares, we consider unpositioned quartics; if two quartics have the same content but start in different positions, we consider them equal.

Our Results.

- We show that the number of 2D-runs in an $n \times n$ array is $\mathcal{O}(n^2 \log^2 n)$. This improves upon the $\mathcal{O}(n^3)$ upper bound of Amir et al. [5, 6] and proves that their algorithm computes all 2D-runs in an $n \times n$ 2D-string in $\mathcal{O}(n^2 \log^2 n)$ time **(Section 3)**.
- We show that the number of distinct quartics in an $n \times n$ array is $\mathcal{O}(n^2 \log^2 n)$. This can be viewed as an extension of the bounds on the maximum number of distinct square factors in a 1D-string [26, 29] **(Section 4)**.
- We present algorithmic implications of the new upper bound for 2D-runs. We show that all occurrences of primitively rooted quartics can be computed in quasi-linear, $\mathcal{O}(n^2 \log^2 n)$ time, which is optimal by the bound of Apostolico and Brimkov [7]. Thus our algorithm complements the result of Apostolico and Brimkov [8] who gave an optimal algorithm for computing all occurrences of primitively rooted tandems. We also show that all distinct quartics can be computed in quasi-linear, $\mathcal{O}(n^2 \log^2 n)$ time, which extends efficient computation of distinct squares in 1D-strings [11, 22, 33] to 2D **(Section 5)**.
- As an easy side result, we show tight $\Theta(n^3)$ bounds for the maximum number of distinct tandems in an $n \times n$ array and how to report them in $\mathcal{O}(n^3)$ time **(Section 2)**.

2 Preliminaries

1D-Strings. We denote by $[a, b]$ the set $\{i \in \mathbb{Z} : a \le i \le b\}$. Let $S = S[1]S[2] \cdots S[|S|]$ be a *string* of length $|S|$ over an alphabet Σ. The elements of Σ are called *letters*. For two positions i and j on S, we denote by $S[i\,..\,j] = S[i] \cdots S[j]$ the *fragment* of S that starts at position i and ends at position j (it equals ε if $j < i$). A positive integer p is called a *period* of S if $S[i] = S[i + p]$ for all $i = 1, \ldots, |S| - p$. We refer to the smallest period as *the period* of the string, and denote it by $\mathsf{per}(S)$.

▶ **Lemma 1** (Periodicity Lemma (weak version), Fine and Wilf [27]). *If p and q are periods of a string S and satisfy $p+q \le |S|$, then $\gcd(p,q)$ is also a period of S.*

A string S is called *periodic* if $\mathsf{per}(S) \le |S|/2$. By ST and S^k we denote the concatenation of strings S and T and k copies of the string S, respectively. A string S is called *primitive* if it cannot be expressed as U^k for a string U and an integer $k>1$.

A string of the form U^2 for string U is called a *square*. A square U^2 is called *primitively rooted* if U is primitive. We will make use of the following important property of squares.

▶ **Lemma 2** (Three Squares Lemma, [24]). *Let U, V and W be three strings such that U^2 is a proper prefix of V^2, V^2 is a proper prefix of W^2 and U is primitive. Then $|U|+|V| \le |W|$.*

A *run* (also known as *maximal repetition*) in S is a periodic fragment $R = S[i\mathinner{.\,.} j]$ which cannot be extended either to the left or to the right without increasing the period $p = \mathsf{per}(R)$, i.e. if $i>1$ then $S[i-1] \ne S[i+p-1]$ and if $j<|S|$ then $S[j+1] \ne S[j-p+1]$. Let $\mathcal{R}(S)$ denote the set of all runs of string S. For periodic fragment $U = S[a\mathinner{.\,.} b]$, the run that extends U is the unique run $R = S[i\mathinner{.\,.} j]$ such that $i \le a \le b \le j$ and $\mathsf{per}(R) = \mathsf{per}(U)$. An occurrence of a square U^2 is said to be *induced* by a run R if R extends U^2. Every square is induced by exactly one run [22].

2D-Strings. Let A be an $m \times n$ array (2D-string). We denote the height and width of A by $\mathsf{height}(A) = m$ and $\mathsf{width}(A) = n$, respectively. By $A[i,j]$ we denote the cell in the ith row and jth column of A; see Figure 2(a). By $A[i_1\mathinner{.\,.} i_2, j_1\mathinner{.\,.} j_2]$ we denote the subarray formed of rows $i_1,\dots,i_2$ and columns $j_1,\dots,j_2$.

A positive integer p is a *horizontal period* of A if the i-th column of A equals the $(i+p)$-th column of A for all $i = 1,\dots,n-p$. We denote the smallest horizontal period of A by $\mathsf{hper}(A)$. Similarly, a positive integer q is a *vertical period* of A if the i-th row of A equals the $(i+q)$-th row of A for all $i = 1,\dots,m-q$; the smallest vertical period of A is denoted by $\mathsf{vper}(A)$.

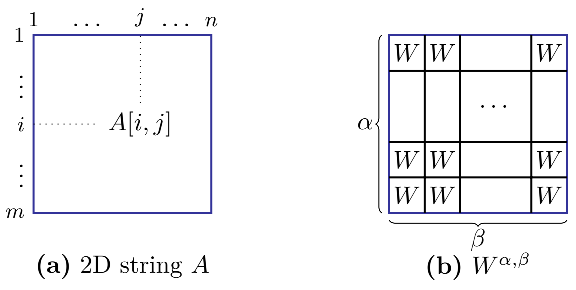


Figure 2 A 2D-string and the structure of $W^{\alpha,\beta}$.

An $r \times c$ subarray $B = A[i_1\mathinner{.\,.} i_2, j_1\mathinner{.\,.} j_2]$ of A is a *2D-run* if $\mathsf{hper}(B) \le c/2$, $\mathsf{vper}(B) \le r/2$ and extending B by a row or column, i.e. either of $A[i_1-1, j_1\mathinner{.\,.} j_2]$, $A[i_2+1, j_1\mathinner{.\,.} j_2]$, $A[i_1\mathinner{.\,.} i_2, j_1-1]$, or $A[i_1\mathinner{.\,.} i_2, j_2+1]$, would result in a change of the smallest vertical or the horizontal period.

If W is a 2D array, then by $W^{\alpha,\beta}$ we denote an array that is composed of $\alpha \times \beta$ copies of W; see Figure 2(b). A *tandem* of W is an array of the form $W^{1,2}$ and a *quartic* of W is the array $W^{2,2}$. A 2D array A is called *primitive* if $A = B^{\alpha,\beta}$ for positive integers α, β implies that $\alpha = \beta = 1$. The *primitive root* of an array A is the unique primitive array B for which $A = B^{\alpha,\beta}$ for $\alpha, \beta \ge 1$.

Apostolico and Brimkov [7] proved the following upper bound, and showed that it is tight by giving a corresponding lower bound.

▶ **Fact 3** (Lemma 5 in [7]). *A 2D array of size $n \times n$ has $\mathcal{O}(n^2 \log^2 n)$ occurrences of primitively rooted quartics.*

We say that a quartic $Q = W^{2,2}$ is *induced* by a 2D-run R if Q is a subarray of R and $\mathsf{hper}(R)$ and $\mathsf{vper}(R)$ divide the width and height of W, respectively.

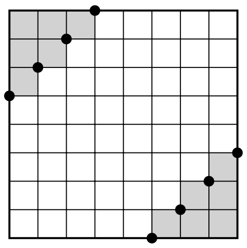


Figure 3 Shaded positions contain letters b, all the other the letters a. Each rectangle with top-left and bottom-right corners marked is a 2D-run; altogether there are 18 distinct 2D-runs, including two of the form $b^{2,2}$. There are also 10 distinct quartics $a^{\alpha,\beta}$, where $0 < \alpha, \beta \leq 8$ are even and $\alpha + \beta \leq 10$. There is also the quartic $b^{2,2}$ (altogether 11 distinct quartics). The centrally placed quartic $a^{2,2}$ is contained in 16 2D-runs. There are only two distinct primitively rooted quartics.

▶ **Observation 4.** *Every quartic is induced by a 2D-run. However; the same quartic can be induced even by $\Theta(n^2)$ 2D-runs; say the middle quartic $a^{2,2}$ in Figure 3.*

▶ Remark 5. The fact that a string of length n has $\mathcal{O}(n \log n)$ occurrences of primitively rooted squares immediately shows (by the fact that a square is induced by exactly one run) that it has $\mathcal{O}(n \log n)$ runs. However, an analogous argument applied for quartics and 2D-runs does not give a non-trivial upper bound for the number of the latter because of Observation 4.

In our algorithms, we use a variant of the Dictionary of Basic Factors in 2D (2D-DBF in short) that is similar to the one presented in [25]. Namely, to each subarray of A whose width and height is an integer power of 2 we assign an integer identifier from $[0, n^2]$ so that two arrays with the same dimensions are equal if and only if their identifiers are equal. The total number of such subarrays is $\mathcal{O}(n^2 \log^2 n)$ and the identifiers can be assigned in $\mathcal{O}(n^2 \log^2 n)$ time; see [25]. Using 2D-DBF, we can assign an identifier to a subarray of A of arbitrary dimensions $r \times c$ being a quadruple of 2D-DBF identifiers of its four $2^i \times 2^j$ subarrays that share one of its corners, where $2^i \leq r < 2^{i+1}$ and $2^j \leq c < 2^{j+1}$. Such quadruples preserve the property that two subarrays of the same dimensions are equal if and only if the 2D-DBF quadruples are the same.

As an illustration, we show a tight bound for the number of distinct tandems and an optimal algorithm for computing them.

▶ **Theorem 6.** *The maximum number of distinct tandems in an $n \times n$ array A is $\Theta(n^3)$. All distinct tandems in an $n \times n$ array can be reported in the optimal $\Theta(n^3)$ time.*

Proof. Let us fix two row numbers $i < i'$ in A. Then, the number of distinct tandems with top row i and bottom row i' is $\mathcal{O}(n)$ by the fact that a string of length n contains $\mathcal{O}(n)$ squares [26, 29]. Thus, in total there are $\mathcal{O}(n^3)$ distinct tandems. For the lower bound, let the ith row of A be filled with occurrences of the letter i. Every subarray of A of even width is a tandem. For each distinct triplet of top and bottom rows and even width, we obtain a distinct tandem.

Let us proceed to the algorithm. For a height $h \in [1, n]$, we assign integer identifiers from $[1, n^2]$ that preserve lexicographical comparison to all height-h substrings of columns of A. They can be assigned using the generalized suffix tree [18, 47] of the columns of A in $\mathcal{O}(n^2 \log n)$ time. Let B_h be an array such that $B_h[i, j]$ stores the identifier of $A[i \mathinner{..} i+h-1, j]$. To a subarray $W = A[i \mathinner{..} i+h-1, j \mathinner{..} j+w-1]$ we assign an *identifier* $\mathsf{id}(W) = B_h[i, j \mathinner{..} j+w-1]$. Then for any two subarrays W and W' of height h, $W = W'$ if and only if $\mathsf{id}(W) = \mathsf{id}(W')$. For every height $h = 1, \ldots, n$ and row i, we find all distinct squares in $B_h[i, 1], \ldots, B_h[i, n]$ in $\mathcal{O}(n)$ time [11, 22, 33]. This corresponds to the set of distinct tandems with top row i and bottom row $i + h - 1$. Finally, we assign identifiers from 2D-DBF of A to each of the tandems and use radix sort to sort them and enumerate distinct tandems. ◀

3 Improved Upper Bound for 2D-Runs

We introduce the framework that Amir et al. used for efficiently computing 2D-runs [5, 6].

We say that a subarray $B = A[i_1 \mathinner{..} i_2, j_1 \mathinner{..} j_2]$ of A is a *horizontal run* if it is horizontally periodic (that is, $\mathsf{hper}(B) \leq \mathsf{width}(B)/2$) and extending B by either of the columns $A[i_1 \mathinner{..} i_2, j_1 - 1]$ or $A[i_1 \mathinner{..} i_2, j_2 + 1]$ would result in a change of the smallest horizontal period. (Note that B does not have to be vertically periodic.)

For $k \in [1, \lfloor \log n \rfloor]$ and $i \in [1, n - 2^k + 1]$, let H_i^k be the string obtained by replacing the columns of array $A[i \mathinner{..} i + 2^k - 1, 1 \mathinner{..} n]$ with metasymbols such that $H_i^k[j] = H_i^k[j']$ if and only if $A[i \mathinner{..} i + 2^k - 1, j] = A[i \mathinner{..} i + 2^k - 1, j']$. Notice that each such horizontal run of height 2^k corresponds to a run in some H_i^k.

The following lemma will enable us to “anchor” each 2D-run R in the top-left or bottom-left corner of a horizontal run of “similar” height as R. It was proved in [6], but we provide a proof for completeness.

▶ **Lemma 7** (Lemma 7 in [6]). *Let R be a 2D-run whose height is in the range $[2^k, 2^{k+1})$. Then there is a horizontal run R' of height 2^k with $\mathsf{hper}(R') = \mathsf{hper}(R)$ and $\mathsf{width}(R') \geq \mathsf{width}(R)$ such that top-left or bottom-left corners of R and R' coincide (see Figure 4).*

Proof. Let $R = A[i_1 \mathinner{..} i_2, j_1 \mathinner{..} j_2]$ be the 2D-run in scope and let $k = \lfloor \log(i_2 - i_1 + 1) \rfloor$. We have to show that at least one of the two following statements holds.

- There is a run $R_1 = S[j_1 \mathinner{..} b]$ in $S = H_{i_1}^k$ with smallest period p and $b \geq j_2$.
- There is a run $R_2 = T[j_1 \mathinner{..} d]$ in $T = H_{i_2 - 2^k + 1}^k$ with smallest period p and $d \geq j_2$.

Since $\mathsf{vper}(R) \leq \mathsf{height}(R)/2$, all distinct rows of R are represented in each of $U = S[j_1 \mathinner{..} j_2]$ and $V = T[j_1 \mathinner{..} j_2]$ and hence $p = \mathsf{per}(U) = \mathsf{per}(V)$. Let $R_1 = S[a \mathinner{..} b]$ be the run that extends U and $R_2 = T[c \mathinner{..} d]$ be the run that extends V. Let us suppose towards a contradiction that $\max(a, c) < j_1$. Then, $A[i_1 \mathinner{..} i_2, j_1 - 1] = A[i_1 \mathinner{..} i_2, j_1 - 1 + p]$, which contradicts R being a run, since R and $B = A[i_1 \mathinner{..} i_2, j_1 - 1 \mathinner{..} j_2]$ have the same horizontal and vertical periods. ◀

The sum of the lengths of the runs in a string of length n can be $\Omega(n^2)$ as shown in [32]. However, we prove the following lemma, which is crucial for our approach. We will use it to obtain an overall bound on the possible widths of 2D-runs for our anchors.

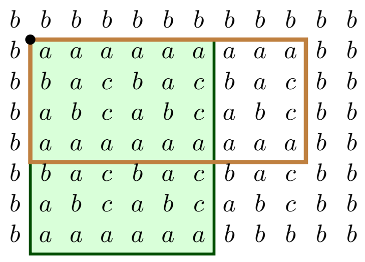


Figure 4 The shaded 7×6 subarray is a 2D-run R, with vertical period 3 and horizontal period $p = 3$. The other marked 4×9 rectangle encloses a horizontal run R' with the same top-left corner and the same horizontal period as R. We have $2 \cdot p \le width(R) \le width(R')$.

▶ **Lemma 8.** *For any string S of length n we have that*

$$\rho(S) \;:=\; \sum_{R \in \mathcal{R}(S)} (|R| - 2 \cdot \mathsf{per}(R) + 1) \;=\; \mathcal{O}(n \log n).$$

Proof. We consider for each run $R = S[i \,..\, j]$ of S the interval $I_R = [i, j - 2 \cdot \mathsf{per}(R) + 1]$. Note that $\rho(S) = \sum_{R \in \mathcal{R}(S)} |I_R|$.

Observe that for every $a \in I_R$ the string $S[a \,..\, a + \mathsf{per}(R) - 1]$ is primitive, since if it was of the form U^k for a string U and an integer $k > 1$, then $|U| < \mathsf{per}(R)$ would be a period of R, a contradiction. Hence, at each position $a \in I_R$ there is an occurrence of a primitively rooted square of length $2 \cdot \mathsf{per}(R)$.

A direct application of the Three Squares Lemma (Lemma 2) implies that at most $\mathcal{O}(\log n)$ primitively rooted squares can start at each position a. Each such square extends to a unique run. Thus, each position i belongs to $\mathcal{O}(\log n)$ intervals I_R for $R \in \mathcal{R}(S)$. This completes the proof. ◀

We are now ready to prove the main result of this section.

▶ **Theorem 9.** *There are $\mathcal{O}(n^2 \log^2 n)$ 2D-runs in an $n \times n$ array A.*

Proof. We will iterate over all horizontal runs $R' = A[i \,..\, i', j \,..\, j']$ whose height is a power of 2, i.e. $i' = i + 2^k - 1$ for some k. For each such horizontal run R', we consider the 2D-runs R with:

(a) top-left corner $A[i, j]$ or bottom-left corner $A[i', j]$,
(b) $\mathsf{hper}(R) = \mathsf{hper}(R')$, and
(c) $\mathsf{height}(R) \in [2^k, 2^{k+1})$.

For each such 2D-run R, we have $\mathsf{width}(R) \in [2 \cdot \mathsf{hper}(R'), \mathsf{width}(R')]$, else the horizontal period would break, i.e. property (b) would be violated. Let us notice that R' corresponds to a run $U = H_i^k[j \,..\, j'] \in \mathcal{R}(H_i^k)$. In particular, $\mathsf{width}(R) \in [2 \cdot \mathsf{per}(U), |U|]$.

Lemma 7 implies that each 2D-run is accounted for at least once in this manner. It is thus enough to bound the number of considered runs. We have n choices for i and $\log n$ choices for k. Further, due to Lemma 8, for each corresponding meta-string H_i^k we have $\mathcal{O}(n \log n)$ choices for a pair (j, c) such that $U = H_i^k[j \,..\, j'] \in \mathcal{R}(H_i^k)$ and $c \in [2 \cdot \mathsf{per}(U), |U|]$. In total, we thus have $\mathcal{O}(n^2 \log^2 n)$ choices for (i, k, j, c). We will complete the proof by showing that there is only a constant number of 2D-runs with top-left corner $A[i, j]$, width w and whose height is in the range $[2^k, 2^{k+1})$. (2D-runs with bottom-left corner $A[i', j]$ can be bounded symmetrically.)

▷ Claim 10 (cf. Lemma 10 in [6]). Let B be an $r \times c$ array with $r \in [2^k, 2^{k+1})$. Then, there are at most two integers $p > 2^{k-1}$ such that $p = \mathsf{vper}(B') \leq \mathsf{height}(B')/2$ for B' consisting of the top $\mathsf{height}(B') \geq 2^k$ rows of B.

Proof. Consider S to be the meta-string obtained by replacing the rows of B by single letters. Then, a direct application of the Three Squares Lemma (Lemma 2) to S yields the claimed bound. ◁

We apply Claim 10 to $B = A[i\,..\,\min(i + 2^{k+1} - 2, n), j\,..\,j + c - 1]$. If $\mathsf{vper}(R) \leq 2^{k-1}$, then $\mathsf{vper}(R) = \mathsf{vper}(R')$ by the Periodicity Lemma (Lemma 1) applied to the meta-string obtained by replacing the rows of the intersection of R' and B by single letters. Now Claim 10 implies that there are at most three choices to make for the vertical period: $\mathsf{vper}(R')$ and the two integers from the claim. Finally, for fixed top-left corner, width and vertical period we can have a single 2D-run. This concludes the proof. ◀

Amir et al. [6] presented the following algorithmic result.

▶ **Theorem 11** ([6]). *All 2D-runs in an $n \times n$ array can be computed in $\mathcal{O}(n^2 \log n + \mathsf{output})$ time, where output is the number of 2D-runs reported.*

By combining Theorems 9 and 11 we get the following corollary.

▶ **Corollary 12.** *All 2D-runs in an $n \times n$ array can be computed in $\mathcal{O}(n^2 \log^2 n)$ time.*

4 Upper Bound on the Number of Distinct Quartics

Fact 3 that originates from [7] shows that an $n \times n$ array A has $\mathcal{O}(n^2 \log^2 n)$ occurrences of primitively rooted quartics. This obviously implies that the number of distinct primitively rooted quartics is upper bounded by $\mathcal{O}(n^2 \log^2 n)$. Unfortunately, an array can contain $\Theta(n^4)$ occurrences of general quartics; this takes place e.g. for a unary array. In this section we show that $\mathcal{O}(n^2 \log^2 n)$ is also an upper bound for the number of *distinct* general quartics, i.e. subarrays of A of the form $W^{\alpha,\beta}$ for even $\alpha, \beta \geq 2$ and primitive W.

The following lemma and its corollary are the combinatorial foundation of our proofs. An array W with $\mathsf{height}(W) \in [2^a, 2^{a+1})$ and $\mathsf{width}(W) \in [2^b, 2^{b+1})$ will be called an *(a, b)-array.*

▶ **Lemma 13.** *Let a, b be non-negative integers and W, W' be different primitive (a, b)-arrays. If occurrences of $W^{2,3}$ and $(W')^{2,3}$ (of $W^{3,2}$ and $(W')^{3,2}$, respectively) in A share the same corner (i.e., top-left, top-right, bottom-left or bottom-right), then $\mathsf{width}(W) = \mathsf{width}(W')$ ($\mathsf{height}(W) = \mathsf{height}(W')$, respectively).*

Proof. Clearly it is sufficient to prove the lemma for $W^{2,3}$ and $(W')^{2,3}$. Assume w.l.o.g. that occurrences of $W^{2,3}$ and $(W')^{2,3}$ in A share the top-left corner and consider their overlap X.

Each of the rows of X has periods $\mathsf{width}(W)$ and $\mathsf{width}(W')$. Assume w.l.o.g. that $\mathsf{width}(W) \leq \mathsf{width}(W')$. Then

$$\mathsf{width}(X) = 3 \cdot \mathsf{width}(W) \geq \mathsf{width}(W) + 2^{a+1} \geq \mathsf{width}(W) + \mathsf{width}(W').$$

By the Periodicity Lemma (Lemma 1), $p = \gcd(\mathsf{width}(W), \mathsf{width}(W'))$ is a horizontal period of X.

The array X contains at least one occurrence of W and W' in its top-left corner. Hence, W and W' have a horizontal period p. If $\mathsf{width}(W) < \mathsf{width}(W')$, then $\mathsf{width}(W')$ cannot be a multiple of $\mathsf{width}(W)$, because then we would have $\mathsf{width}(W') > 2^{a+1}$. Hence, if $\mathsf{width}(W) < \mathsf{width}(W')$, we would have $p < \mathsf{width}(W)$ which by $p \mid \mathsf{width}(W)$ would mean that W is not primitive. This indeed shows that $\mathsf{width}(W) = \mathsf{width}(W')$. ◀

▶ **Corollary 14.** *Let a, b be non-negative integers and W, W' be different (a, b)-arrays. If occurrences of $W^{3,3}$ and $(W')^{3,3}$ in A share the same corner (i.e., top-left, top-right, bottom-left or bottom-right), then at least one of W, W' is not primitive.*

If $V^{2,2}$ is a non-primitively rooted quartic, then there exists a primitive array W such that $V = W^{\alpha,\beta}$ and at least one of α, β is greater than one. We will call the quartic $W^{2\alpha,2\beta}$ *thin* if $\alpha = 1$ or $\beta = 1$ for this decomposition, and *thick* otherwise. We refer to *points* in A as the $(n+1)^2$ positions where row and column delimiters intersect. Let us first bound the number of distinct thin quartics. For $\beta > 1$, we consider any rightmost occurrence of every such quartic, that is, any occurrence $A[i_1 \mathinner{.\,.} i_2, j_1 \mathinner{.\,.} j_2]$ that maximizes j_1.

▶ **Lemma 15.** *The total number of distinct thin quartics in A is $\mathcal{O}(n^2 \log^2 n)$.*

Proof. We give a proof for quartics of the form $W^{2,2\beta}$ for primitive W and $\beta > 1$; the proof for quartics of the form $W^{2\alpha,2}$ for $\alpha > 1$ is symmetric. We consider each pair of positive integers a, b and show that each point holds the top-left corner of at most two rightmost occurrences of $W^{2,2\beta}$ for primitive (a, b)-arrays W and $\beta > 1$.

Assume to the contrary that the rightmost occurrences of $W^{2,2\beta}$, $(W')^{2,2\beta'}$ and $(W'')^{2,2\beta''}$ share their top-left corner for primitive (a, b)-arrays W, W', W''. The arrays W, W', W'' are pairwise different, since otherwise one of the occurrences would not be the rightmost. By Lemma 13, we have $\mathsf{width}(W) = \mathsf{width}(W') = \mathsf{width}(W'')$. Assume w.l.o.g. that $\mathsf{height}(W) < \mathsf{height}(W') < \mathsf{height}(W'')$.

Let (i, j) denote the top-left corner of the three quartics. Let us consider three length-2ℓ strings formed of metacharacters that correspond to row fragments:

$$(A[i, j \mathinner{.\,.} j + w - 1]), \ldots, (A[i + 2\ell - 1, j \mathinner{.\,.} j + w - 1])$$

for $w = \mathsf{width}(W)$ and $\ell \in \{\mathsf{height}(W), \mathsf{height}(W'), \mathsf{height}(W'')\}$. All the three strings need to be primitively rooted squares. We apply the Three Squares Lemma (Lemma 2) to conclude that $\mathsf{height}(W'') > \mathsf{height}(W) + \mathsf{height}(W') > 2^{a+1}$, a contradiction. ◀

Now let us proceed to thick quartics. Unfortunately, in this case a single point can be the top-left corner of a linear number of rightmost occurrences of thick quartics; see the example in Figure 3. Let us consider an occurrence of $W^{\alpha,\beta}$ for even $\alpha, \beta > 2$ and primitive W, called a *positioned quartic*. It implies $\alpha \cdot \beta$ occurrences of W. Let us call all corners of all these occurrences of W *special points* of this positioned quartic. Each special point stores a direction in {top-left, top-right, bottom-left, bottom-right}. A special point has one of the directions if it is the respective corner of an occurrence of $W^{3,3}$ in this positioned quartic. Clearly, since $\alpha, \beta \geq 4$, for every special point in $W^{\alpha,\beta}$ except for the middle row if $\alpha = 4$ or middle column if $\beta = 4$, one can assign such a direction (if many directions are possible, we choose an arbitrary one); see Figure 5.

The quartics with primitive root W are called *W-quartics*. The set of all special points (with directions) of *all* positioned thick W-quartics for a given W is denoted by $\mathit{SpecialPoints}(W)$. Among W-quartics of the same height we distinguish the ones with maximal width, which we call *h-maximal* (horizontally maximal). Let us observe that each W-quartic is contained in an occurrence of some h-maximal W-quartic.

▶ **Theorem 16.** *The number of distinct quartics in an $n \times n$ array is $\mathcal{O}(n^2 \log^2 n)$.*

Proof. By Fact 3 and Lemma 15 it suffices to show that the total number of distinct thick quartics in A is $\mathcal{O}(n^2 \log^2 n)$. Let us fix non-negative integers a, b. It is enough to show that the number of distinct subarrays of A of the form $W^{\alpha,\beta}$ for even $\alpha, \beta > 2$ and any primitive (a, b)-array W is $\mathcal{O}(n^2)$.

The sets of special points have the following properties. Claim 17 follows from Corollary 14.

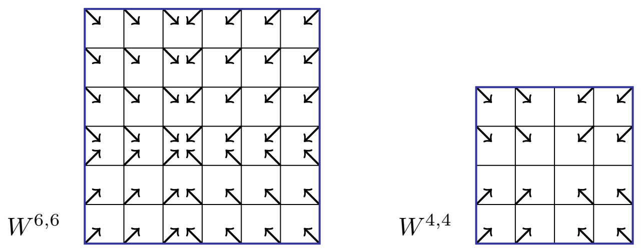


Figure 5 Special points of a positioned quartic with primitive root W with associated directions of four types. The arrow indicates the corner (four possibilities) of $W^{3,3}$ which is contained in the quartic. If several assignments of directions are possible, only one of them is chosen (it does not matter which one). In case of $W^{4,4}$ the middle row and column are not special.

▷ Claim 17. For primitive (a, b)-arrays $W \neq W'$, $\mathit{SpecialPoints}(W) \cap \mathit{SpecialPoints}(W') = \emptyset$.

For an array W, let us denote by $\mathit{ThickQuartics}(W)$ the total number of thick quartics in A with primitive root W.

▷ Claim 18. For a primitive (a, b)-array W, $\mathit{ThickQuartics}(W) < |\mathit{SpecialPoints}(W)|$.

Proof. For each $\alpha = 4, 6, \ldots$ in this order, we select one positioned h-maximal W-quartic U_α of height $\alpha \cdot \mathsf{height}(W)$. The number of distinct W-quartics in A of height $\alpha \cdot \mathsf{height}(W)$ is at most the number of special points in U_α in any of its rows. Note that this statement also holds if $U_\alpha = W^{\alpha,4}$; then there are still four special points in each (non-middle if $\alpha = 4$) row.

We describe a process of assigning distinct W-quartics to distinct special points in $\mathit{SpecialPoints}(W)$. Assume all points in this set are initially not marked. We choose any single row from U_α with all special points in this row still not marked. Then we mark all these special points. We can always choose a suitable row because the heights are increasing.

This way each W-quartic is assigned to only one special point from $\mathit{SpecialPoints}(W)$. ◁

By the claims, the total number of thick W-quartics for primitive (a, b)-arrays W is bounded by:

$$\sum_W \mathit{ThickQuartics}(W) < \sum_W |\mathit{SpecialPoints}(W)| \leq 4(n+1)^2,$$

where the sum is over all primitive (a, b)-arrays W. The conclusion follows. ◀

5 Algorithms for Computing Quartics

In this section we show algorithmic applications of 2D-runs related to quartics.

▶ **Theorem 19.** *All occurrences of primitively rooted quartics in an $n \times n$ array A can be computed in the optimal $\mathcal{O}(n^2 \log^2 n)$ time.*

Proof. Let us consider a 2D-run $R = A[i_1 \mathinner{..} i_2, j_1 \mathinner{..} j_2]$ with periods $\mathsf{hper}(R) = p$ and $\mathsf{vper}(R) = q$. It induces primitively rooted quartics of width $2p$ and height $2q$. The set of top-left corners of these quartics forms a rectangle $\hat{R} = [i_1, i_2 - 2p + 1] \times [j_1, j_2 - 2q + 1]$. We denote by $\mathcal{F}_{p,q}$ the family of such rectangles $\hat{R}$ over 2D-runs R with the same periods p, q.

Such rectangles for different 2D-runs may overlap, even when the dimensions of the quartic are fixed (see Observation 4). In order not to report the same occurrence multiple times, we need to compute, for every dimensions of a quartic, all points in the union of the corresponding rectangles. This could be done with an additional $\log n$-factor in the complexity using a standard line sweep algorithm [12]. However, we can achieve $\mathcal{O}(n^2 \log^2 n)$ total time using the fact that the total number of occurrences reported is $\mathcal{O}(n^2 \log^2 n)$.

$\triangleright$ Claim 20. Let $\mathcal{F}_1, \ldots, \mathcal{F}_k$ be families of 2D rectangles in $[1,n]^2$ and let $r = \sum_{i=1}^{k} |\mathcal{F}_i|$. We can compute k (not necessarily disjoint) sets of grid points $Out_i = \bigcup \mathcal{F}_i$ in $\mathcal{O}(n + r + \mathsf{output})$ total time, where $\mathsf{output} = \sum_i |Out_i|$ is the total number of reported points.

Proof. We design an efficient line sweep algorithm. We will perform a separate line sweep, left to right, for each family $\mathcal{F}_i$.

The sweep goes over horizontal (x) coordinates in a left-to-right manner. The broom stores vertical (y) coordinates of horizontal sides of rectangles that it currently intersects. They are stored in a sorted list L of pairs (y, c), where y is the coordinate, and c is the count of rectangles with bottom side at coordinate y minus the count of the rectangles with top side at coordinate y. Only pairs with non-zero second component are stored. Clearly, the second components of the list elements always sum up to 0.

A coordinate x is processed if L is non-empty before accessing it or there exist any vertical sides of rectangles at x. All vertical sides with the same y-coordinate are processed in a batch. For every such batch we want to guarantee that endpoints of all sides are stored in a list B in a top-down order.

A top (bottom) endpoint at vertical coordinate y is stored as $(y, +1)$ ($(y, -1)$, respectively).

Let us now describe how to process a horizontal coordinate x. Let us merge the list L that is currently in the broom with the list B of the batch by the first components. If there is more than one pair with the same first component, we merge all of them together, summing up the second components.

Let us denote by L' the resulting list. We iterate over all elements of L', keeping track of the partial sum of second components, denoted as s. For every element (y, c) of L', the point (x, y) is reported for $\bigcup \mathcal{F}_i$. Moreover, if the partial sum s before considering c was positive and the previous element of L' is (y', c'), all points $(x, y'+1), \ldots, (x, y-1)$ are reported to Out_i.

Finally, all pairs with second component equal to zero are removed from L' which becomes the new list L.

Let us now analyze the complexity of the algorithm. The line sweep makes n steps. The total size of lists B across all families $\mathcal{F}_i$ is $\mathcal{O}(r)$ and they can be constructed simultaneously in $\mathcal{O}(n + r)$ time via bucket sort.

Processing a batch with list B takes $\mathcal{O}(|L| + |B|)$ time plus the time to report points in Out_i. As we have already noticed, the sum of $\mathcal{O}(|B|)$ components is $\mathcal{O}(r)$. For every element (y, c) of the initial list L, a point with the vertical coordinate y is reported upon merging; hence, the sum of $\mathcal{O}(|L|)$ components is dominated by $\mathcal{O}(\mathsf{output})$. Overall we achieve time complexity $\mathcal{O}(n + r + \mathsf{output})$. $\triangleleft$

We apply the claim to the families $\mathcal{F}_{p,q}$. Then r and output are upper bounded by $\mathcal{O}(n^2 \log^2 n)$ by Theorem 9 and Fact 3, respectively. The optimality of our algorithm's complexity is due to the $\Omega(n^2 \log^2 n)$ lower bound on the maximum number of occurrences of primitively rooted quartics from [7]. ◀

We proceed to an efficient algorithm for enumerating distinct, not necessarily primitively rooted, quartics using 2D-runs. The solution for an analogous problem for 1-dimensional strings (computing distinct squares from runs) uses Lyndon roots of runs [22]. However, in 2 dimensions it is not clear if a similar approach could be applied efficiently, say, with the aid of 2D Lyndon words [40] as Lyndon roots of 2D-runs. We develop a different approach in which the workhorse is the following auxiliary problem related to the folklore nearest smaller value problem.

Let us consider a grid of height m in which every cell can be black or white. We say that the grid forms a *staircase* if the set of white cells in each row is nonempty and is a prefix of this row (see Figure 6). A staircase can be uniquely determined by an array $\mathit{Whites}[1\,..\,m]$ such that $\mathit{Whites}[i]$ is the number of white cells in the ith row. We consider *shapes* of white rectangles. Each shape is a pair (p,q) that represents the dimensions of the rectangle. These shapes (and corresponding rectangles) are partially ordered by: $(p,q) < (p',q') \;\Leftrightarrow\; (p,q) \neq (p',q') \;\wedge\; p \le p' \;\wedge\; q \le q'$.

MAX WHITE RECTANGLES
Input: An array $\mathit{Whites}[1\,..\,m]$ that represents a staircase.
Output: Shapes of all maximal white rectangles in this staircase.

▶ **Lemma 21.** *MAX WHITE RECTANGLES problem can be solved in $\mathcal{O}(m)$ time.*

Proof. Assume that $\mathit{Whites}[0] = \mathit{Whites}[m+1] = -1$. Let us define two tables of size m:

$$\mathit{NSVUp}[i] = \max\{j \,:\, j < i,\ \mathit{Whites}[j] < \mathit{Whites}[i]\},$$
$$\mathit{NSVDown}[i] = \min\{j \,:\, j > i,\ \mathit{Whites}[j] < \mathit{Whites}[i]\}.$$

They can be computed in $\mathcal{O}(m)$ time by a folklore algorithm for the nearest smaller value table; see e.g. [13]. Then the problem can be solved as in Algorithm 1 presented below. After the first for-loop, for each maximal white rectangle R we have $\mathit{MaxWidth}[\mathsf{height}(R)] = \mathsf{width}(R)$, but we could have redundant values for non-maximal rectangles. In order to filter out non-maximal rectangles, we process the candidates by decreasing height and remove the ones that are dominated by the previous maximal rectangle in the partial order of shapes. ◀

Algorithm 1 The first phase computes a set of shapes of type $(h, \mathit{MaxWidth}[h])$, at most one for each height h; see also Figure 6. In the second phase only inclusion-maximal shapes from this set are reported.

```
ComputeCandidates:
$MaxWidth[1..m] := (0,\dots,0)$
for $i := 1$ to $m$ do
    $h := NSVDown[i] - NSVUp[i] - 1$
    $MaxWidth[h] := \max(MaxWidth[h], Whites[i])$
ReportMaximal:
$mw := 0$
for $h := m$ down to $1$ do
    if $MaxWidth[h] > mw$ then
        Report the shape $(h, MaxWidth[h])$
        $mw := MaxWidth[h]$
```

▶ Remark 22. Note that the total area (and width) of a staircase can be large but the complexity of our algorithm is linear with respect to the number of rows, thanks to the small representation (array Whites).

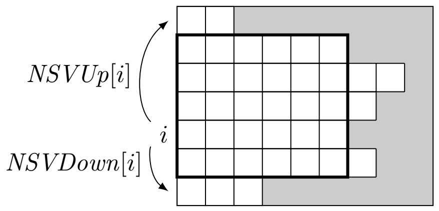


Figure 6 A maximal white rectangle containing row i is computed using the NSV tables for i.

Now our approach is graph-theoretic. The graph nodes correspond to occurrences of primitively rooted quartics. For a fixed primitively rooted quartic $W^{2,2}$ we consider the graph $G_W = (V, E)$, where V is the set of top-left corners of occurrences of $W^{2,2}$. Let $r = \mathsf{height}(W)$ and $c = \mathsf{width}(W)$. The edges in G connect vertex (i, j) with vertices $(i \pm r, j)$ and $(i, j \pm c)$, if they exist. See also Figure 7. This graph can be efficiently computed since we know its nodes due to Theorem 19.

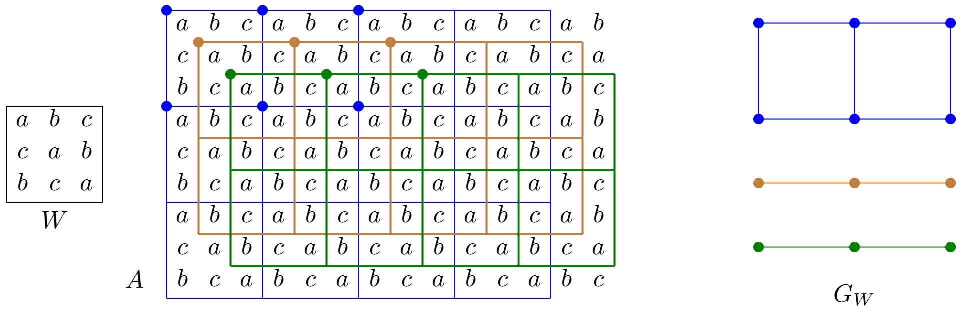


Figure 7 Graph G_W has 12 vertices that form two components with 3 vertices each (green and brown) and one component with 6 vertices (blue). Note the non-trivial occurrences of W in $W^{3,4}$.

▶ **Lemma 23.** *All graphs G_W, and their connected components, for all W which are primitive roots of quartics in A can be constructed in $\mathcal{O}(n^2 \log^2 n)$ time.*

Proof. We first compute all occurrences of primitively rooted quartics in A using Theorem 19. By Fact 3, there are $\mathcal{O}(n^2 \log^2 n)$ of them in total.

We can assign 2D-DBF identifiers (quadruples) to each of the occurrences and group the occurrences by distinct primitively rooted quartics via radix sort in $\mathcal{O}(n^2 \log^2 n)$ time. This gives us the vertices of G_W.

To compute the edges, we use an auxiliary $n \times n$ Boolean array D that will store top-left corners of occurrences of each subsequent primitively rooted quartic $W^{2,2}$.

Initially D is set to zeroes and after each W, all cells with ones are zeroed in $\mathcal{O}(|G_W|)$ time. Using this array and the positions of occurrences of $W^{2,2}$, the edges of G_W can be computed in $\mathcal{O}(|G_W|)$ time. It also allows to divide G_W into connected components via graph search in $\mathcal{O}(|G_W|)$ time. ◀

▶ **Theorem 24.** *All distinct quartics in an $n \times n$ array A can be computed in $\mathcal{O}(n^2 \log^2 n)$ time.*

Proof. We first apply Lemma 23. Now consider a fixed primitive W of height c and width r. Let us note that if $(i,j),(i',j')$ belong to the same connected component H of G_W, then $i \equiv i' \pmod{r}$ and $j \equiv j' \pmod{c}$. We say that a connected component H of G_W *generates* an occurrence of a power $W^{\alpha,\beta}$ if the $\alpha\beta$ occurrences of W that are implied by it belong to H. If $W^{\alpha,\beta}$ has an occurrence in A, then it is generated by some connected component H of G_W, unless $\min(\alpha,\beta)=1$.

We say that $W^{\alpha,\beta}$ is a *maximal* power if there is no other power $W^{\alpha',\beta'}$ in A such that $\alpha' \geq \alpha$, $\beta' \geq \beta$, and $(\alpha',\beta') \neq (\alpha,\beta)$. Similarly, we consider powers that are maximal among ones that are generated by a connected component H. Let $\mathit{MaxPowers}_W(H)$ be the set of maximal powers generated by a connected component H. It can be computed in linear time using Lemma 21 as shown in Algorithm 2, which we now explain.

For each vertex (i,j) in H, we insert four points to a set S, which correspond to the four occurrences of W underlying the occurrence of quartic $W^{2,2}$ at position (i,j). If S is treated as a set of white cells in a grid, then $W^{\alpha,\beta}$ for $\alpha > 1$ is a power generated by H if and only if the grid contains a white rectangle of shape (α,β). For a cell $(i,j) \in S$, we denote $R[i,j] = \min\{p \geq 0 : (i,j+p) \notin S\}$. Assuming that the cells of S are sorted by non-increasing second component, each value $R[i,j]$ can be computed from $R[i,j+1]$ in constant time, for a total of $\mathcal{O}(|S|)$ time. The sorting for all S can be done globally, using radix sort. Also, the array R can be stored globally and used for all S, cleared after each use. Finally, we process each maximal set of consecutive cells $(i,j),\ldots,(i+m-1,j) \in S$ that are located in the same column and apply Lemma 21 to solve the resulting instance of the MAX WHITE RECTANGLES problem. The total time required by this step is $\mathcal{O}(|S|)$.

Algorithm 2 Computing $\mathit{MaxPowers}_W(H)$ for a component H of G_W.

```
$S := \emptyset$
**foreach** $(i,j)$ **in** $V(H)$ **do**
    $a := \lfloor i/r \rfloor;\ \ b = \lfloor j/c \rfloor$
    $S := S \cup \{(a,b),(a+1,b),(a,b+1),(a+1,b+1)\}$
$R[0\,..\,n, 0\,..\,n] := (0,\ldots,0)$
**foreach** $(i,j)$ **in** $S$ *in non-increasing order of* $j$ **do**
    $R[i,j] := R[i,j+1]+1$
$\mathit{Result} := \emptyset$
**foreach** *maximal set* $\{(i,j),(i+1,j)\ldots,(i+m-1,j)\} \subseteq S$ **do**
    $\mathit{Whites}[1\,..\,m] := R[i\,..\,i+m-1,j]$
    $\mathit{Result} := \mathit{Result} \cup \textsc{MaxWhiteRectangles}(\mathit{Whites})$
remove redundant rectangles from $\mathit{Result}$
return $\mathit{Result}$
```

In the end we filter out the powers $W^{\alpha,\beta}$ that are not maximal in A similarly as in the proof of Lemma 21, using a global array $\mathit{MaxWidth}$. Let $W^{\alpha_1,\beta_1},\ldots,W^{\alpha_k,\beta_k}$ be the resulting sequence of maximal powers, sorted by increasing first component, and let $\alpha_0 = \beta_0 = 0$. Then the set of all quartics in A with primitive root W contains all $W^{2\alpha,2\beta}$ over $\alpha_{p-1} < 2\alpha \leq \alpha_p$, $1 \leq 2\beta \leq \beta_p$, for $p \in [2,k]$. They can be reported in $\mathcal{O}(n^2 \log^2 n)$ total time over all W due to the upper bound of Theorem 16. ◀

6 Final Remarks

We showed that the numbers of distinct runs and quartics in an $n \times n$ array are $\mathcal{O}(n^2 \log^2 n)$. This improves upon previously known estimations. We also proposed $\mathcal{O}(n^2 \log^2 n)$-time algorithms for computing all occurrences of primitively rooted quartics and all distinct quartics. A straightforward adaptation shows that for an $m \times n$ array these bounds and complexities all become $\mathcal{O}(mn \log m \log n)$.

We pose two conjectures for $n \times n$ 2D-strings:

- The number of 2D-runs is $\mathcal{O}(n^2)$.
- The number of distinct quartics is $\mathcal{O}(n^2)$.

References

1 Amihood Amir, Gary Benson, and Martin Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM Journal on Computing*, 23(2):313–323, 1994. doi:10.1137/S0097539792226321.

2 Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition detection in a dynamic string. In *27th Annual European Symposium on Algorithms, ESA 2019*, volume 144 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ESA.2019.5.

3 Amihood Amir, Ayelet Butman, Gad M. Landau, Shoshana Marcus, and Dina Sokol. Double string tandem repeats. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, volume 161 of *LIPIcs*, pages 3:1–3:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.3.

4 Amihood Amir and Martin Farach. Efficient 2-dimensional approximate matching of non-rectangular figures. In *Proceedings of the Second Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms*, pages 212–223. ACM/SIAM, 1991. URL: http://dl.acm.org/citation.cfm?id=127787.127829.

5 Amihood Amir, Gad M. Landau, Shoshana Marcus, and Dina Sokol. Two-dimensional maximal repetitions. In *26th Annual European Symposium on Algorithms, ESA 2018*, volume 112 of *LIPIcs*, pages 2:1–2:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.2.

6 Amihood Amir, Gad M. Landau, Shoshana Marcus, and Dina Sokol. Two-dimensional maximal repetitions. *Theoretical Computer Science*, 812:49–61, 2020. doi:10.1016/j.tcs.2019.07.006.

7 Alberto Apostolico and Valentin E. Brimkov. Fibonacci arrays and their two-dimensional repetitions. *Theoretical Computer Science*, 237(1-2):263–273, 2000. doi:10.1016/S0304-3975(98)00182-0.

8 Alberto Apostolico and Valentin E. Brimkov. Optimal discovery of repetitions in 2D. *Discrete Applied Mathematics*, 151(1-3):5–20, 2005. doi:10.1016/j.dam.2005.02.019.

9 Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4):533–541, 1978. doi:10.1137/0207043.

10 Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

11 Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, volume 78 of *LIPIcs*, pages 22:1–22:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.CPM.2017.22.

12 Jon Louis Bentley. Algorithms for Klee's rectangle problems. Unpublished notes, Computer Science Department, Carnegie Mellon University, 1977.

13 Omer Berkman, Baruch Schieber, and Uzi Vishkin. Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993. doi:10.1006/jagm.1993.1018.

14 Richard S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977. doi:10.1016/0020-0190(77)90017-5.

15 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, volume 161 of *LIPIcs*, pages 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.CPM.2020.8.

16 Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, volume 149 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ISAAC.2019.22.

17 Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981. doi:10.1016/0020-0190(81)90024-7.

18 Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on strings*. Cambridge University Press, 2007.

19 Maxime Crochemore and Lucian Ilie. Analysis of maximal repetitions in strings. In *Mathematical Foundations of Computer Science 2007, 32nd International Symposium, MFCS 2007*, volume 4708 of *Lecture Notes in Computer Science*, pages 465–476. Springer, 2007. doi:10.1007/978-3-540-74456-6_42.

20 Maxime Crochemore and Lucian Ilie. Maximal repetitions in strings. *Journal of Computer and System Sciences*, 74(5):796–807, 2008. doi:10.1016/j.jcss.2007.09.003.

21 Maxime Crochemore, Lucian Ilie, and Liviu Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011. doi:10.1016/j.tcs.2010.06.019.

22 Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

23 Maxime Crochemore and Robert Mercaş. On the density of Lyndon roots in factors. *Theoretical Computer Science*, 656:234–240, 2016. doi:10.1016/j.tcs.2016.02.015.

24 Maxime Crochemore and Wojciech Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. doi:10.1007/BF01190846.

25 Maxime Crochemore and Wojciech Rytter. *Jewels of stringology*. World Scientific, 2002. doi:10.1142/4838.

26 Antoine Deza, Frantisek Franek, and Adrien Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015. doi:10.1016/j.dam.2014.08.016.

27 Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.

28 Johannes Fischer, Stepan Holub, Tomohiro I, and Moshe Lewenstein. Beyond the runs theorem. In *String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015*, volume 9309 of *Lecture Notes in Computer Science*, pages 277–286. Springer, 2015. doi:10.1007/978-3-319-23826-5_27.

29 Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.

30 Frantisek Franek and Qian Yang. An asymptotic lower bound for the maximal number of runs in a string. *International Journal of Foundations of Computer Science*, 19(1):195–203, 2008. doi:10.1142/S0129054108005620.

31 Mathieu Giraud. Not so many runs in strings. In *Language and Automata Theory and Applications, Second International Conference, LATA 2008*, volume 5196 of *Lecture Notes in Computer Science*, pages 232–239. Springer, 2008. doi:10.1007/978-3-540-88282-4_22.

32 Amy Glen and Jamie Simpson. The total run length of a word. *Theoretical Computer Science*, 501:41–48, 2013. `doi:10.1016/j.tcs.2013.06.004`.

33 Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004. `doi:10.1016/j.jcss.2004.03.004`.

34 Lucian Ilie. A note on the number of squares in a word. *Theoretical Computer Science*, 380(3):373–376, 2007. `doi:10.1016/j.tcs.2007.03.025`.

35 Costas S. Iliopoulos, Dennis W. G. Moore, and William F. Smyth. A characterization of the squares in a Fibonacci string. *Theoretical Computer Science*, 172(1-2):281–291, 1997. `doi:10.1016/S0304-3975(96)00141-7`.

36 Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 756–767. ACM, 2019. `doi:10.1145/3313276.3316368`.

37 Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In Piotr Indyk, editor, *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. `doi:10.1137/1.9781611973730.36`.

38 Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999*, pages 596–604. IEEE Computer Society, 1999. `doi:10.1109/SFFCS.1999.814634`.

39 M. Lothaire. *Combinatorics on words, Second Edition*. Cambridge mathematical library. Cambridge University Press, 1997.

40 Shoshana Marcus and Dina Sokol. 2D Lyndon words and applications. *Algorithmica*, 77(1):116–133, 2017. `doi:10.1007/s00453-015-0065-z`.

41 Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara. New lower bounds for the maximum number of runs in a string. In *Proceedings of the Prague Stringology Conference 2008*, pages 140–145, 2008. URL: `http://www.stringology.org/event/2008/p13.html`.

42 Simon J. Puglisi, Jamie Simpson, and William F. Smyth. How many runs can a string contain? *Theoretical Computer Science*, 401(1-3):165–171, 2008. `doi:10.1016/j.tcs.2008.04.020`.

43 Wojciech Rytter. The number of runs in a string: Improved analysis of the linear upper bound. In *23rd Annual Symposium on Theoretical Aspects of Computer Science, STACS 2006*, volume 3884 of *Lecture Notes in Computer Science*, pages 184–195. Springer, 2006. `doi:10.1007/11672142_14`.

44 Wojciech Rytter. The number of runs in a string. *Information and Computation*, 205(9):1459–1469, 2007. `doi:10.1016/j.ic.2007.01.007`.

45 Jamie Simpson. Modified Padovan words and the maximum number of runs in a word. *The Australasian Journal of Combinatorics*, 46:129–146, 2010. URL: `http://ajc.maths.uq.edu.au/pdf/46/ajc_v46_p129.pdf`.

46 Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002. `doi:10.1016/S0304-3975(01)00121-9`.

47 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. `doi:10.1007/BF01206331`.

A Alternative Algorithm for the Proof of Lemma 21

An alternative, space efficient and more direct algorithm that does not use additional tables *NSVDown* and *NSVUp*, is shown below. The algorithm computes only the table *MaxWidth*. Then, we can use the second phase from Algorithm 1. We assume that the table *MaxWidth* is initially filled with zeros.

Algorithm 3 Alternative implementation of the first phase in Algorithm 1.

```
$Whites[0] := Whites[m+1] := 0$
$S :=$ empty stack; $push(S, 0)$
for $i := m$ down to $0$ do
    while $Whites[i] < Whites[top(S)]$ do
        $k := top(S)$; $h := top(S) - i - 1$
        $MaxWidth[h] := \max(MaxWidth[h], Whites[k])$
        $pop(S)$
    if $Whites[top(S)] = Whites[i]$ then $pop(S)$
    $push(S, i)$
```

The algorithm is a version of a folklore algorithm for the Nearest Smaller Values problem and correctness can be shown using the same arguments. If $Whites[i] < Whites[i+1]$, then the algorithm produces shapes of all Max White Rectangles anchored at $i+1$, otherwise $i+1$ is "nonproductive". Observe that $i+1 = top(S)$ when we start processing $i \geq 1$.

Let us analyze the time complexity of the algorithm. In total $m+2$ elements are pushed to the stack. Each iteration of the while-loop pops an element, so the total number of iterations of this loop is $\mathcal{O}(m)$. Consequently, the algorithm works in $\mathcal{O}(m)$ time. In the end one needs to filter out non-maximal rectangles as in the previous proof of Lemma 21.

Chapter 6

Efficient Enumeration of Distinct Factors Using Package Representations

Efficient Enumeration of Distinct Factors Using Package Representations

Panagiotis Charalampopoulos[1,2,⋆][0000−0002−6024−1557], Tomasz Kociumaka[3,⋆⋆][0000−0002−2477−1702], Jakub Radoszewski[2,⋆⋆⋆][0000−0002−0067−6401], Wojciech Rytter[2][0000−0002−9162−6724], Tomasz Waleń[2,⋆⋆⋆][0000−0002−7369−3309], and Wiktor Zuba[2,⋆⋆⋆][0000−0002−1988−3507]

[1] Department of Informatics, King's College London, UK,
panagiotis.charalampopoulos@kcl.ac.uk
[2] Institute of Informatics, University of Warsaw, Poland
{jrad,rytter,walen,w.zuba}@mimuw.edu.pl
[3] Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel
kociumaka@mimuw.edu.pl

Abstract. We investigate properties and applications of a new compact representation of string factors: families of *packages*. In a string T, each package (i, ℓ, k) represents the factors of T of length ℓ that start in the interval $[i, i+k]$. A family $\mathcal{F}$ of packages represents the set $\mathsf{Factors}(\mathcal{F})$ defined as the union of the sets of factors represented by individual packages in $\mathcal{F}$. We show how to efficiently enumerate $\mathsf{Factors}(\mathcal{F})$ and showcase that this is a generic tool for enumerating important classes of factors of T, such as powers and antipowers. Our approach is conceptually simpler than problem-specific methods and provides a unifying framework for such problems, which we hope can be further exploited.

We also consider a special case of the problem in which every occurrence of every factor represented by $\mathcal{F}$ is captured by some package in $\mathcal{F}$. For both applications mentioned above, we construct an efficient package representation that satisfies this property.

We develop efficient algorithms that, given a family $\mathcal{F}$ of m packages in a string of length n, report all distinct factors represented by these packages in $\mathcal{O}(n \log^2 n + m \log n + |\mathsf{Factors}(\mathcal{F})|)$ time for the general case and in the optimal $\mathcal{O}(n + m + |\mathsf{Factors}(\mathcal{F})|)$ time for the special case. We can also compute $|\mathsf{Factors}(\mathcal{F})|$ in $\mathcal{O}(n \log^2 n + m \log n)$ time in the general case and in $\mathcal{O}(n+m)$ time in the special case.

In particular, we improve over the state-of-the-art $\mathcal{O}(nk^4 \log k \log n)$-time algorithm for computing the number of distinct k-antipower factors, by providing an algorithm that runs in $\mathcal{O}(nk^2)$ time, and we obtain an alternative linear-time algorithm to enumerate distinct squares.

⋆ Partially supported by ERC grant TOTAL under the EU's Horizon 2020 Research and Innovation Programme (agreement no. 677651).

⋆⋆ Supported by ISF grants no. 1278/16 and 1926/19, by a BSF grant no. 2018364, and by an ERC grant MPM under the EU's Horizon 2020 Research and Innovation Programme (grant no. 683064).

⋆⋆⋆ Supported by the Polish National Science Center, grant no. 2018/31/D/ST6/03991.

Keywords: square in a string · antipower · longest previous factor array · string synchronising set

1 Introduction

There are many interesting subsets of factors of a given string T of length n which can be described very concisely (sometimes in $\mathcal{O}(n)$ space, even for subsets of quadratic size). In this paper, we consider compact descriptions, called *package representations*, defined in terms of weighted intervals: each interval $[i, i+k]$ gives starting positions of factors and the weight ℓ gives the common length of these factors. Formally, $\mathcal{F}$ is a set of triples (i, ℓ, k).

By $\mathsf{Factors}(\mathcal{F})$ we denote the set of factors in a given text T of length n that are represented by packages from $\mathcal{F}$. More formally,

$$\mathsf{Factors}(\mathcal{F}) \;=\; \{T[j\mathinner{.\,.} j+\ell) \;:\; j \in [i, i+k] \;\text{ and }\; (i, \ell, k) \in \mathcal{F}\}.$$

A package representation $\mathcal{F}$ is called *special* if it represents all occurrences of $\mathsf{Factors}(\mathcal{F})$. Formally, $\mathcal{F}$ is special if for every factor $F \in \mathsf{Factors}(\mathcal{F})$ and for every occurrence $T[j\mathinner{.\,.} j+\ell) = F$, there is a triple $(i, \ell, k) \in \mathcal{F}$ such that $j \in [i, i+k]$. Special representations describe *all* occurrences of factors with a given property.

We consider the following subsets of factors.

Powers. A square is a string of the form X^2. In general, for an integer $k > 1$, a k-power is a string of the form X^k. This notion can be generalized to rational exponents $\gamma > 1$, setting $X^\gamma = X^k X[1\mathinner{.\,.} r]$ for $\gamma = k + r/|X|$, where k and $r < |X|$ are non-negative integers.

Antipowers. A k-antipower (for an integer $k \geq 2$) is a concatenation of k pairwise distinct strings of the same length. Antipowers were introduced in [15] and have already attracted considerable attention [1,2,4,14,25].

Example 1. Consider a string T = `abababababa`. The squares in T can be represented by a set of packages $\mathcal{F} = \{(1, 4, 7), (1, 8, 3)\}$. The package $(1, 4, 7)$ represents all the squares of length 4 and the package $(1, 8, 3)$—those of length 8.

Our problem can be related to computing the *subword complexity* of the string T; see, e.g., [31]. Let us recall that the subword complexity is a function which gives, for every $\ell \in [1, n]$, the number of different factors of T of length ℓ. The subword complexity of a given string can be computed using the suffix tree in linear time. Our algorithm can be easily augmented to determine, for each length ℓ, the number of length-ℓ factors in $\mathsf{Factors}(\mathcal{F})$.

Our results. We compute $|\mathsf{Factors}(\mathcal{F})|$ in $\mathcal{O}(n \log^2 n + m \log n)$ time in the general case and in $\mathcal{O}(n + m)$ time in the special case, both for any length-n string T over an integer alphabet. The solution to the general case uses string synchronising sets and runs, whereas the solution to the special case is based on the longest previous factor array. Our algorithms for special package representations yield new simple algorithms for reporting and counting powers and antipowers. In particular, we present the first linear-time algorithms to count and enumerate distinct γ-powers for a given rational constant $\gamma > 1$; Crochemore et al. [11] showed how to do this for integer γ only. For k-antipowers, we improve the previously known best time complexity.

2 Algorithms for Special Package Representations

Let $T = T[1]\cdots T[n]$. The longest previous factor array $LPF[1\mathinner{.\,.}n]$ is defined as

$$LPF[i] = \max\{\ell \geq 0 \,:\, T[i\mathinner{.\,.}i+\ell) = T[j\mathinner{.\,.}j+\ell) \text{ for some } j \in [1, i-1]\}.$$

This array can be computed in $\mathcal{O}(n)$ time [9,10]. Let

$$\begin{aligned} U_\ell &= \{\, j \in [1,n] \,:\, LPF[j] \geq \ell \,\} \\ \mathsf{Pairs}(\mathcal{F}) &= \bigcup_{(i,\ell,k)\,\in\mathcal{F}} \{(j,\ell) \,:\, j \in [i, i+k] \setminus U_\ell\}. \end{aligned}$$

The algorithms are based on the following crucial observation that links the solution to the special case with the LPF table.

Observation 2. *If $\mathcal{F}$ is a special package representation, then* $\mathsf{Factors}(\mathcal{F}) = \{\, T[j\mathinner{.\,.}j+\ell) \,:\, (j,\ell) \in \mathsf{Pairs}(\mathcal{F}) \,\}$ *and* $|\mathsf{Factors}(\mathcal{F})| = |\mathsf{Pairs}(\mathcal{F})|$.

2.1 Reporting Distinct Factors

Due to Observation 2, reporting all distinct factors reduces to computing the set $\mathsf{Pairs}(\mathcal{F})$. We can assume that packages representing factors of the same length are disjoint; this can be achieved by merging overlapping packages in a preprocessing step that can be executed in $\mathcal{O}(n)$ time using radix sort.

The definition of $\mathsf{Pairs}(\mathcal{F})$ yields the following (inefficient) algorithm. It constructs the sets U_ℓ for all $\ell = n, \ldots, 1$ and, for each of them, generates all elements of the set $\mathsf{Pairs}(\mathcal{F})$ with the second component equal to ℓ.

Algorithm 1: High-level structure of the algorithm.

```
$U := \emptyset$; $\mathcal{P} := \emptyset$
for $\ell := n$ down to 1 do
    $U := U \cup \{j \,:\, LPF[j] = \ell\}$                      // $U = U_\ell$
    foreach $(i, \ell, k) \in \mathcal{F}$ do
        foreach $j \in [i, i+k] \setminus U$ do
            $\mathcal{P} := \mathcal{P} \cup \{(j, \ell)\}$        // Ultimately, $\mathcal{P} = \mathsf{Pairs}(\mathcal{F})$
```

Next, we describe an efficient implementation of Algorithm 1 based on the union-find data structure. In our algorithm, the elements of the data structure are $[1, n+1]$ and the sets stored in the data structure always form intervals. The operation `Find`(i) returns the rightmost element of the interval containing i, and the operation `Union`(i) joins the intervals containing elements i and $i-1$.

Algorithm 2: Implementation of Algorithm 1.

```
$\mathcal{P} := \emptyset$
**for** $i := 0$ **to** $n+1$ **do** Create set $\{i\}$
**for** $\ell := n$ **down to** $1$ **do**
    **foreach** $j$ *such that* $LPF[j] = \ell$ **do** Union($j$)
    **foreach** $(i, \ell, k) \in \mathcal{F}$ **do**
        $j :=$ Find($i-1$) $+ 1$
        **while** $j \le i+k$ **do**
            $\mathcal{P} := \mathcal{P} \cup \{(j, \ell)\}$
            $j :=$ Find($j$) $+ 1$
```

Theorem 3. *In the case of special package representations, all elements of* $\mathsf{Factors}(\mathcal{F})$ *can be reported (without duplicates) in* $\mathcal{O}(n + m + |\mathsf{Factors}(\mathcal{F})|)$ *time.*

Proof. We use Algorithm 2. The set U_ℓ is stored in the union-find data structure so that for each interval $[i, j]$ in the data structure, $i \notin U_\ell$ and $[i+1, j] \subseteq U_\ell$.

The elements of $\mathcal{F}$ are sorted by the second component using radix sort. The union-find data structure admits at most n union operations and $m + |\mathsf{Factors}(\mathcal{F})|$ find operations. We use a data structure for a special case of the union-find problem, where the sets of the partition have to form integer intervals at all times, so that each operation takes $\mathcal{O}(1)$ amortized time [18]. □

2.2 Counting Distinct Factors

Let us start with a warm-up algorithm. Recall that, in a preprocessing, we made sure that packages representing factors of the same length are disjoint.

By Observation 2, for each $(i, \ell, k) \in \mathcal{F}$, it suffices to count the number of elements in $LPF[i \mathinner{.\,.} i+k]$ that are smaller than ℓ. This can be done using range queries in time $O((n+m)\sqrt{\log n})$.

Let us proceed to a linear-time algorithm. We start with a simple fact.

Fact 4. *For every length-n text T, we have* $\sum_{i=1}^{n-1} |LPF[i+1] - LPF[i]| = \mathcal{O}(n)$.

Proof. The claim follows from the fact that $LPF[i+1] \ge LPF[i] - 1$ for $i \in [1, n-1]$. To prove this inequality, let $\ell = LPF[i]$. We have $T[i \mathinner{.\,.} i+\ell) = T[j \mathinner{.\,.} j+\ell)$ for some $j < i$. Hence, $T[i+1 \mathinner{.\,.} i+\ell) = T[j+1 \mathinner{.\,.} j+\ell)$, so $LPF[i+1] \ge \ell - 1$. □

We reduce the counting problem to answering off-line a linear number of certain queries. The off-line structure of the computation is crucial for efficiency.

Theorem 5. *In the case of special package representations,* $|\mathsf{Factors}(\mathcal{F})|$ *can be computed in* $\mathcal{O}(n+m)$ *time.*

Proof. Consider the following queries:

$$Q(i,\ell) = |\,[1,i] \setminus U_\ell\,| = |\{j \in [1,i] : LPF[j] < \ell\}|.$$

Then, the counting version of our problem reduces to efficiently answering such queries. Indeed, by Observation 2, we have

$$|\mathsf{Factors}(\mathcal{F})| \;=\; \sum_{(i,\ell,k)\in\mathcal{F}} Q(i+k,\ell) - Q(i-1,\ell).$$

Thus, we have to answer $\mathcal{O}(m)$ queries of the form $Q(i,\ell)$. An off-line algorithm answering q queries in $\mathcal{O}(n+q)$ time would be sufficient for our purposes.

We maintain an array $A[1\mathinner{.\,.} n]$ such that during the ith phase of the algorithm:

$$A[\ell] = \begin{cases} i - Q(i,\ell) & \text{if } \ell > LPF[i], \\ Q(i,\ell) & \text{otherwise.} \end{cases}$$

Since $LPF[1]=0$, the array needs to be filled with 1's for the first phase. Next, we observe that $i+1-Q(i+1,\ell) = i-Q(i,\ell)$ if $\ell > \max(LPF[i+1], LPF[i])$ and $Q(i+1,\ell) = Q(i,\ell)$ if $\ell \le \min(LPF[i+1], LPF[i])$. Hence, in the transition from the ith phase to the $(i+1)$th phase, we only need to update $\mathcal{O}(|LPF[i+1]-LPF[i]|)$ entries of A. By Fact 4, the cost of maintaining the array A for $i=1$ to n is $\mathcal{O}(n)$ in total. Each query $Q(i,\ell)$ can be answered in $\mathcal{O}(1)$ time during the ith phase.

Consequently, we can answer off-line q queries $Q(i,\ell)$ in $\mathcal{O}(n+q)$ time, assuming that the queries are sorted by the first component. Sorting can be performed in $\mathcal{O}(n+q)$ time using radix sort. □

3 Applications

In this section, we show three applications of special package representations.

3.1 Squares

It is known that a string of length n contains at most $\frac{11}{6}n$ distinct squares [12,17], and the same bound hold for γ-powers with $\gamma \ge 2$. Moreover, all the distinct square factors in a string over an integer alphabet can be reported in $\mathcal{O}(n)$ time [6,11,20]. The algorithm from [11] can report distinct string powers of a given integer exponent using a run-based approach via Lyndon roots. Hence, it can report distinct squares and cubes in particular. We show that our generic approach—which is also much simpler—applies to this problem.

A *generalised run* in a string T is a triple (i,j,p) such that:

- $T[i\mathinner{.\,.} j]$ has a period p (not necessarily the shortest) with $2p \le j-i+1$,
- $T[i-1] \ne T[i-1+p]$ if $i>1$, and $T[j+1] \ne T[j+1-p]$ if $j<n$.

A *run* is a generalised run for which p is the shortest period of $T[i\mathinner{.\,.} j]$. The number of runs and generalised runs is $\mathcal{O}(n)$ and they can all be computed in $\mathcal{O}(n)$ time; see [5,30].

Proposition 6. *All distinct squares in a string of length n can be computed in $\mathcal{O}(n)$ time.*

Proof. A generalised run (i, j, p) induces squares $T[k\,..\,k+2p)$ for all $k \in [i, j-2p+1]$. Moreover, each occurrence of a square is induced by exactly one generalised run. For every generalised run (i, j, p), we add package $(i, 2p, j-2p-i+1)$ to $\mathcal{F}$; see Fig. 1. Then, we solve the factors problem using Theorem 3. □

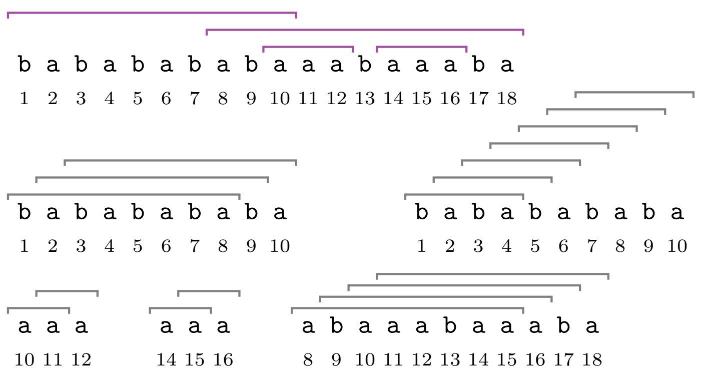


Fig. 1. Four runs (presented at the top) generate a package representation (below) of all squares as a set of five packages: $\{(1, 8, 2), (1, 4, 6), (10, 2, 1), (14, 2, 1), (8, 8, 3)\}$. One of the runs induces two generalised runs: with periods 2 and 4.

3.2 Powers with Rational Exponents

Proposition 6 can be easily generalised to powers of arbitrary exponent $\gamma \geq 2$. For exponents $\gamma < 2$, however, we need α-gapped repeats apart from the generalised runs. An *α-gapped repeat* (for $\alpha \geq 1$) in a string T is a quadruple (i_1, j_1, i_2, j_2) such that $i_1 \leq j_1 < i_2 \leq j_2$, and the factors $T[i_1\,..\,j_1] = T[i_2\,..\,j_2] = U$ and $T[j_1+1\,..\,i_2-1] = V$ satisfy $|UV| \leq \alpha|U|$. The two occurrences of U are called the *arms* of the α-gapped repeat and $|UV|$ is called the *period* of the α-gapped repeat. In other words, a gapped repeat is a string $S = T[i_1\,..\,j_2]$ associated with one of its periods larger than $\frac{1}{2}|S|$. Consequently, the same factor $T[i_1\,..\,j_2]$ can induce many α-gapped repeats.

An α-gapped repeat is called *maximal* if its arms cannot be extended simultaneously with the same character to either direction. The number of maximal α-gapped repeats in a string of length n is $\mathcal{O}(n\alpha)$ and they can all be computed in $\mathcal{O}(n\alpha)$ time assuming an integer alphabet [19].

Theorem 7. *For a given rational number $\gamma > 1$, all distinct γ-powers in a length-n string can be counted in $\mathcal{O}(\frac{\gamma}{\gamma-1}n)$ time and enumerated in $\mathcal{O}(\frac{\gamma}{\gamma-1}n + \mathsf{output})$ time.*

Proof. Each γ-power X^γ with $\gamma < 2$ is a $\frac{1}{\gamma-1}$-gapped repeat with period $|X|$, and therefore it is contained in a maximal $\frac{1}{\gamma-1}$-gapped repeat or in a generalised run with the same period; see [29]. Moreover, each γ-power X^γ with $\gamma \geq 2$ is contained in a generalised run with period $|X|$.

In other words, to generate all γ-powers, for each generalised run and $\frac{1}{\gamma-1}$-gapped repeat (if $\gamma < 2$) with period p, we need to consider all factors contained in it of length γp, provided that γp is an integer; see Fig. 2.

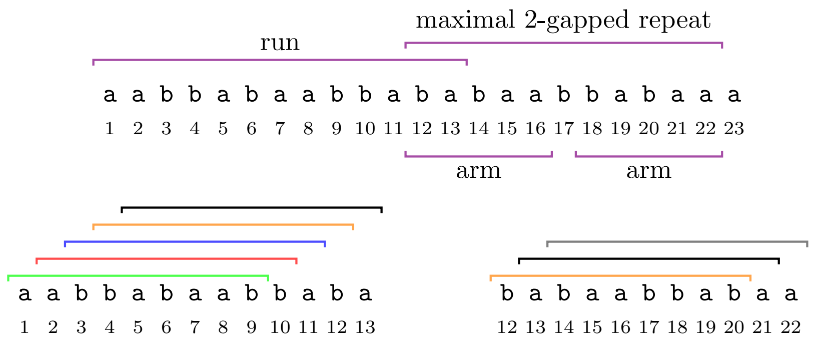


Fig. 2. A string with a (generalised) run with period 6 and a maximal $\frac{6}{5}$-gapped repeat (hence, also a maximal 1.5-gapped repeat) with period 6. The run and the gapped repeat generate 1.5-powers of length 9. Equal 1.5-powers are drawn with the same color; in total, the string contains 6 distinct 1.5-powers of length 9.

We proceed as follows. For each generalised run (i, j, p), if γp is an integer and $j - i + 1 \geq \gamma p$, then we insert $(i, \gamma p, j - i + 1 - \gamma p)$ to $\mathcal{F}$. Moreover, if $\gamma < 2$, then for each maximal $\frac{1}{\gamma-1}$-gapped repeat (i_1, j_1, i_2, j_2) with period $p = i_2 - i_1$, if γp is an integer, then we insert $(i_1, \gamma p, j_2 - i_1 + 1 - \gamma p)$ to $\mathcal{F}$. By the above discussion, the constructed family $\mathcal{F}$ is a special package representation of all γ-powers. The claim follows by Theorems 3 and 5. $\square$

Remark 8. For every fixed rational number $\gamma < 2$, strings of length n may contain $\Omega(n^2)$ distinct γ-powers. Specifically, if $\gamma = 2 - \frac{x}{y}$, where x and y are coprime positive integers, then the number of γ-powers in $\mathtt{a}^m\mathtt{b}\mathtt{a}^m$ is $\Theta(\frac{m^2 x}{y^2})$ [28].

3.3 Antipowers

In [25], it was shown how to report all occurrences of k-antipowers in $\mathcal{O}(nk \log k + \mathsf{output})$ time and count them in $\mathcal{O}(nk \log k)$ time. In [26], it was shown that the number of distinct k-antipower factors in a string of length n can be computed in $\mathcal{O}(nk^4 \log k \log n)$ time. Below, we show how to improve the latter result.

Theorem 9. *All distinct k-antipower factors of a string of length n can be reported in $\mathcal{O}(nk^2 + \mathsf{output})$ time and counted in $\mathcal{O}(nk^2)$ time.*

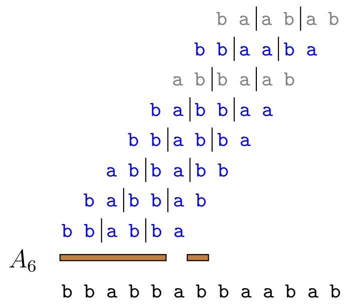


Fig. 3. Here, we consider 3-antipowers of length $\ell = 6$. The set of their starting positions is $A_6 = [1, 5] \cup [7, 7]$. Note that the first and the fourth antipower are the same, so we have only 5 distinct 3-antipowers of length 6. Interestingly $A_\ell = \emptyset$ for $\ell \neq 6$. Hence, the total number of distinct 3-antipowers equals 5.

Proof. The interval representation of a set $A \subseteq [1, n]$ is a collection of all maximal intervals in A. Let A_ℓ be the interval representation of the set of k-antipower factors of T of length ℓ (it can be non-empty only if k divides ℓ); see Fig. 3. In [25, Lemma 13], it was shown that the total size of the interval representations of sets $A_1, \ldots, A_n$ is $\mathcal{O}(nk^2)$. Moreover, they can be computed in $\mathcal{O}(nk^2)$ time.

For each ℓ and each interval $[i, j] \in A_\ell$, we insert $(i, \ell, j - i)$ to $\mathcal{F}$. The conclusion follows by Theorems 3 and 5. $\square$

4 Enumerating General Package Representations

For most of this section, we will focus on computing $|\mathsf{Factors}(\mathcal{F})|$. In the end, we will briefly explain how our solution can be adapted to enumerate $\mathsf{Factors}(\mathcal{F})$.

We consider highly periodic and non-highly-periodic factors separately (a precise definition follows). In both cases, we will employ the solution of Kociumaka et al. [26] for the so-called Path Pairs Problem, which we define below.

We say that $\mathcal{T}$ is a *compact tree* if it is a rooted tree with positive integer weights on edges. If an edge weight is $e > 1$, this edge contains $e - 1$ implicit nodes. A *path* in a compact tree is an upwards or downwards path that connects two explicit nodes.

Path Pairs Problem

Input: Two compact trees $\mathcal{T}$ and $\mathcal{T}'$ containing up to N explicit nodes each, and a set Π of M pairs (π, π') of equal-length paths, where π is a path going downwards in $\mathcal{T}$ and π' is a path going upwards in $\mathcal{T}'$.

Output: $|\bigcup_{(\pi,\pi')\in\Pi} \mathsf{Induced}(\pi, \pi')|$, where by $\mathsf{Induced}(\pi, \pi')$ we denote the set of pairs of (explicit or implicit) nodes (u, u') such that, for some i, the ith node on π is u and the ith node on π' is u'.

Lemma 10 ([26]). *The* Path Pairs Problem *can be solved in time $\mathcal{O}(N + M \log N)$ assuming that the weighted heights of the input trees do not exceed N.*

4.1 Non-Highly-Periodic Factors

Our solution uses the string synchronising sets recently introduced by Kempa and Kociumaka [22].

Informally, in the simpler case that T is cube-free, a τ-synchronising set of T consists in a small set of positions of T, called here *synchronisers*, such that each length-τ fragment of T contains at least one synchroniser, and the synchronisers within two long enough matching fragments of T are consistent.

Formally, for a string T and a positive integer $\tau \leq \frac{1}{2}n$, a set $S \subseteq [1, n-2\tau+1]$ is a *τ-synchronising set* of T if it satisfies the following two conditions:

1. If $T[i\,..\,i+2\tau) = T[j\,..\,j+2\tau)$, then $i \in S$ if and only if $j \in S$.
2. For $i \in [1, n-3\tau+2]$, $S \cap [i\,..\,i+\tau) = \emptyset$ if and only if $\mathsf{per}(T[i\,..\,i+3\tau-2]) \leq \frac{1}{3}\tau$.

Theorem 11 ([22]). *Given a string T of length n over an integer alphabet and a positive integer $\tau \leq \frac{1}{2}n$, one can construct in $\mathcal{O}(n)$ time a τ-synchronising set of T of size $\mathcal{O}(\frac{n}{\tau})$.*

As in [22], for a τ-synchronising set S, let $\mathsf{succ}_S(i) := \min\{j \in S \cup \{n-2\tau+2\} : j \geq i\}$ and $\mathsf{pred}_S(i) := \max\{j \in S \cup \{0\} : j \leq i\}$.

Lemma 12 ([22]). *If a factor U of T with $|U| \geq 3\tau - 1$ and $\mathsf{per}(U) > \frac{1}{3}\tau$ occurs at positions i and j in T, then $\mathsf{succ}_S(i) - i = \mathsf{succ}_S(j) - j \leq |U| - 2\tau$.*

By U^R we denote the reversal of a string U. We show the following result.

Lemma 13 (Aperiodic Lemma). *Assume that we are given a text T of length n, a positive integer $x \leq \frac{1}{3}n$, and a family $\mathcal{F}$ of m packages that represent factors of lengths in $[3x, 9x)$ and shortest periods greater than $\frac{1}{3}x$. Then, $|\mathsf{Factors}(\mathcal{F})|$ can be computed in $\mathcal{O}((n+m)\log n)$ time.*

Proof. We compute an x-synchronising set S of T in $\mathcal{O}(n)$ time using Theorem 11 and build the suffix trees $\mathcal{T}$ and $\mathcal{T}'$ of T and T^R, respectively, in $\mathcal{O}(n)$ time [13].

Let us now focus on all packages representing factors of a fixed length ℓ. By relying on Lemma 12, we will intuitively assign each factor to its first synchroniser.

Let us denote $\mathcal{A}_\ell = \bigcup\{[i, i+k] : (i, \ell, k) \in \mathcal{F}\}$. For each $j \in \mathcal{A}_\ell$, let $s = \mathsf{succ}_S(j)$ and consider $P_j = T[j\,..\,s]$ and $Q_j = [s+1\,..\,j+\ell)$; see Fig. 4.

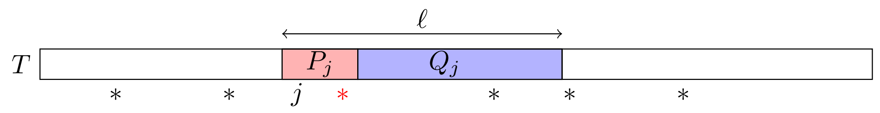


Fig. 4. The elements of an x-synchronising set S of string T are denoted by asterisks. The position j is an element of $[i, i+k]$ for some package (i, ℓ, k). The red asterisk denotes the synchroniser $s = \mathsf{succ}_S(j)$.

Note that, by Lemma 12, $s-j \le x$ and, as $j \le n-\ell+1$, we have $s \le n-2x+1$. Thus, $s \in S$. Hence, Lemma 12 implies that, for any $j, j' \in \mathcal{A}_\ell$ such that $T[j\,..\,j+\ell) = T[j'\,..\,j'+\ell)$, we have $P_j = P_{j'}$ and $Q_j = Q_{j'}$. Consequently, our problem reduces to computing the size of the set $\mathcal{P}_\ell = \{(P_j, Q_j) : j \in \mathcal{A}_\ell\}$. In turn, in our instance of the Path Pairs Problem, we want to count the pairs of nodes $u \in \mathcal{T}'$, $v \in \mathcal{T}$ such that $(\mathcal{L}(u)^R, \mathcal{L}(v)) \in \mathcal{P}_\ell$, where $\mathcal{L}(u)$ is the label of the path from the root to the node u. It remains to show how to compute path pairs that induce exactly these pairs of nodes. To this end, we design a line-sweeping algorithm.

We initialize an empty set Π that will eventually store the desired pairs of paths. We will scan the text T in a left-to-right manner with two fingers: f_p for packages and f_s for synchronisers, both initially set to 0. We maintain an invariant that $f_p \le f_s$. Whenever $f_p - 1 = f_s \le n-2x$, we set $f_s = \mathsf{succ}_S(f_s+1)$.

The finger f_p is repeatedly incremented until it reaches f_s. For each maximal interval $[i, j] \subseteq \mathcal{A}_\ell$ that f_p encounters, we do the following: If $j > f_s$, we split the interval into $[i, i+f_s]$ and $[f_s+1, j]$ and consider the first of them as $[i, j]$. Let

$$X_1 = T[i\,..\,f_s],\ X_2 = T[j\,..\,f_s],\ Y_1 = T[f_s+1\,..\,i+\ell),\ Y_2 = T[f_s+1\,..\,j+\ell).$$

For $k = 1, 2$, let u_k be the locus of X_k^R in $\mathcal{T}'$ and v_k be the locus of Y_k in $\mathcal{T}$. If either of these loci is an implicit node, we make it explicit. Finally, we add to Π the pair of paths u_1-to-u_2 in $\mathcal{T}'$ and v_1-to-v_2 in $\mathcal{T}$.

Let us denote the number of packages representing factors of length ℓ by m_ℓ. As there are $\mathcal{O}(\frac{n}{x})$ synchronisers, the line-sweeping algorithm can be performed in $\mathcal{O}(\frac{n}{x} + m_\ell)$ time. Thus, the number of paths (and extra explicit nodes) that we introduce in the two suffix trees is also $\mathcal{O}(\frac{n}{x} + m_\ell)$.

Over all $\ell \in [3x, 9x)$, we have

$$\mathcal{O}\left(x \cdot \frac{n}{x} + \sum_{\ell=3x}^{9x-1} m_\ell\right) = \mathcal{O}(n+m)$$

pairs of paths. The only operations that we need to explain how to perform efficiently are (a) computing the loci of strings in $\mathcal{T}$ and $\mathcal{T}'$ and (b) making all of them explicit. Part (a) can be implemented using an efficient algorithm for answering a batch of weighted ancestor queries from [24]. In part (b), we process the weighted ancestors in an order of non-decreasing weights, after globally sorting them using radix sort. The whole construction works in $\mathcal{O}(n+m)$ time. We obtain an instance of the Path Pairs Problem with $N, M = \mathcal{O}(n+m)$. The suffix trees are of weighted height $\mathcal{O}(n)$, so Lemma 10 completes the proof. $\square$

4.2 Highly Periodic Factors

A string U is called *periodic* if $2 \cdot \mathsf{per}(U) \le |U|$ and *highly periodic* if $3 \cdot \mathsf{per}(U) \le |U|$.

The *Lyndon root* of a periodic string U is the lexicographically smallest rotation of its length-$\mathsf{per}(U)$ prefix. If L is the Lyndon root of a periodic string

U, then U can be uniquely represented as (L, y, a, b) for $0 \le a, b < |L|$ such that $U = L[|L| - a + 1 \mathinner{.\,.} |L|]L^y L[1 \mathinner{.\,.} b]$. We call this the *Lyndon representation* of U.

In $\mathcal{O}(n)$ time, one can compute the Lyndon representations of all runs [11]. The unique run that extends a periodic factor of T can be computed in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing [5,27]. This allows computing its Lyndon representation in $\mathcal{O}(1)$ time.

For highly periodic factors, we will use *Lyndon roots* instead of synchronisers. The rest of this subsection is devoted to proving the following lemma.

Lemma 14 (Periodic Lemma). *Given a text T of length n and a set $\mathcal{F}$ of m packages of highly periodic factors, $|\mathsf{Factors}(\mathcal{F})|$ can be computed in $\mathcal{O}((n + m)\log n)$ time.*

Proof. For each $(i, \ell, k) \in \mathcal{F}$ and $j \in [i, i + k]$, the fragment $T[j \mathinner{.\,.} j + \ell)$ has a (unique) Lyndon representation (L, y, a, b) for some Lyndon root L. Let

$$P_{j,\ell} = L[|L| - a + 1 \mathinner{.\,.} |L|] \text{ and } Q_{j,\ell} = L^y L[1 \mathinner{.\,.} b]$$

Our problem consists in computing the size of the set

$$\mathcal{P} = \{(P_{j,\ell}, Q_{j,\ell}) : j \in [i, i + k],\ (i, \ell, k) \in \mathcal{F}\}$$

Let $\mathcal{T}$ and $\mathcal{T}'$ be the suffix trees of T and T^R, respectively. Then, we want to compute the number of pairs of nodes $u \in \mathcal{T}'$ and $v \in \mathcal{T}$ with $(\mathcal{L}(u)^R, \mathcal{L}(v)) \in \mathcal{P}$ We will show how this reduces to an instance of the Path Pairs Problem.

We have to appropriately define pairs of paths over $\mathcal{T}$ and $\mathcal{T}'$. Let us note that all the factors that each package of $\mathcal{F}$ represents have the same Lyndon root, since two strings with different periods at most $\frac{1}{3}\ell$ cannot overlap on $\ell - 1$ positions by the Fine and Wilf's periodicity lemma [16].

We initialize an empty set Π that will store pairs of paths. Let us consider a package $(i, \ell, k) \in \mathcal{F}$ such that $T[i \mathinner{.\,.} i + k + \ell)$ is represented by (L, y, a, b). By periodicity, we may focus on the factors starting in the first (at most) $|L|$ positions of $T[i \mathinner{.\,.} i + k + \ell)$.

To this end, let $t = \min\{|L|, k + 1\}$. We will insert at most two paths to Π, specified below:

– Let X_1 be the suffix of L of length a, X_2 be the suffix of L of length $a' = \max\{a - t, 0\}$, $Y_1 = L^\infty[1 \mathinner{.\,.} \ell - a]$ and $Y_2 = L^\infty[1 \mathinner{.\,.} \ell - a']$.

– If $t > a$, let X'_1 be the suffix of L of length $|L| - 1$ and X'_2 be the suffix of L of length $d = |L| + a - t$, $Y'_1 = L^\infty[1 \mathinner{.\,.} \ell - |L| + 1]$ and $Y'_2 = L^\infty[1 \mathinner{.\,.} \ell - d]$.

See Fig. 5 for an illustration. It can be readily verified that these pairs of paths induce exactly the required pairs of nodes.

For $k = 1, 2$, let u_k be the locus of X_k^R in $\mathcal{T}'$ and v_k be the locus of Y_k in $\mathcal{T}$. If either of these loci is an implicit node, we make it explicit. Finally, we add to Π the pair of paths u_1-to-u_2 in $\mathcal{T}'$ and v_1-to-v_2 in $\mathcal{T}$. Similarly for X'_ks and Y'_ks.

Let us consider the time complexity of the algorithm. The suffix trees $\mathcal{T}$ and $\mathcal{T}'$ can be computed in $\mathcal{O}(n)$ time [13]. Computing the loci of strings and making

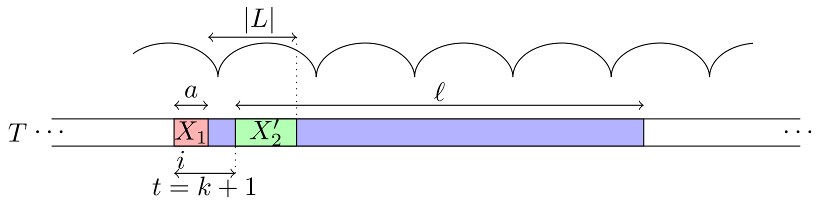


Fig. 5. The shaded part of the text denotes $T[i \mathinner{.\,.} i+k+\ell)$, which can be represented as (L, y, a, b), for some package $(i, \ell, k) \in \mathcal{F}_\ell$. X_1 is shaded in red, while X_2 is the empty string. We are in the case that $t = k+1 > a$; X_2' is shaded in green.

them explicit can be performed in $\mathcal{O}(n+m)$ time as in the proof of Lemma 13. We obtain an instance of the Path Pairs Problem with $N = \mathcal{O}(n+m)$ and $M \leq 2m$. Lemma 10 completes the proof. □

4.3 Wrap-up

Let $\mathcal{F}_\ell$ be the set of triples from $\mathcal{F}$ with the second component ℓ. Note that one can easily compute the contributions of all packages representing factors whose length is bounded by 2 in $\mathcal{O}(n)$ time using radix sort; we can thus assume that all packages represent factors of length at least 3.

We will iterate over $x = 3^j$ for all integers $j \in [1, \lfloor \log_3 n \rfloor - 1]$. For each $\ell \in [3x, 9x)$, we want to replace $\mathcal{F}_\ell$ by two—not too large—sets of packages:

- $\mathcal{F}_\ell^p$ representing factors with shortest period at most $x/3$, and
- $\mathcal{F}_\ell^a$ representing factors with shortest period greater than $x/3$,

such that $\mathsf{Factors}(\mathcal{F}_\ell) = \mathsf{Factors}(\mathcal{F}_\ell^p) \cup \mathsf{Factors}(\mathcal{F}_\ell^a)$.

Our aim is to decompose each package in $\mathcal{F}_\ell$ in pieces (i.e., decompose $[i, i+k]$ into subintervals), such that all factors represented by each piece either have shortest period at most $x/3$ or none of them does. We then want to group the resulting pieces into the two sets.

Let $\mathcal{R}_x$ denote the set of runs of T with length at least $3x$ and period at most $x/3$. As shown in [23, Section 4.4], $|\mathcal{R}_x| = \mathcal{O}(n/x)$. Further, $\mathcal{R}_x$ can be computed in $\mathcal{O}(n)$ time, by filtering out the runs that do not satisfy the criteria.

Lemma 15. *Given $\mathcal{R}_x$, we can compute sets $\mathcal{F}_\ell^p$ and $\mathcal{F}_\ell^a$ in $\mathcal{O}(n/x + |\mathcal{F}_\ell|)$ time.*

Proof. Initially, let $I = \emptyset$. For each run $R = T[a \mathinner{.\,.} b]$ with $\mathsf{per}(R) \leq x/3$ and $|R| \geq \ell$, we set $I := I \cup [a, b-\ell+1]$. There are $\mathcal{O}(n/x)$ such runs and hence our representation of I consists of $\mathcal{O}(n/x)$ intervals.

Recall that packages are pairwise disjoint. We decompose a package $(i, \ell, k) \in \mathcal{F}_\ell$ as follows.

For each maximal interval $[r, t]$ in $[i, i+k] \cap I$ we insert $(r, \ell, t-r)$ to $\mathcal{F}_\ell^p$, while for each maximal interval $[r', t']$ in $[i, i+k] \setminus I$ we insert $(r', \ell, t'-r')$ to $\mathcal{F}_\ell^a$. This can be done in $\mathcal{O}(n/x + |\mathcal{F}_\ell|)$ time with a standard line-sweeping algorithm. □

Now, let us put everything together. First of all, we compute $\mathcal{R}_x$ for each $x \in [1, \lfloor \log_3 n \rfloor - 1]$ in $\mathcal{O}(n \log n)$ total time. Then, for each ℓ, we replace $\mathcal{F}_\ell$ by $\mathcal{F}_\ell^p$ and $\mathcal{F}_\ell^a$ in $\mathcal{O}(n/\ell + |\mathcal{F}_\ell|)$ time, employing Lemma 15.

We process all $\mathcal{F}_\ell^p$'s together, as each factor U represented by them must be highly periodic; since $\mathsf{per}(U) \leq x/3$ and $|U| \geq 3x$ for some x, we surely have $3 \cdot \mathsf{per}(U) \leq |U|$. The total size of these sets is $\sum_{\ell=3}^{n} \mathcal{O}(n/\ell + |\mathcal{F}_\ell|) = \mathcal{O}(n \log n + m)$, and hence a call to Lemma 14 requires $\mathcal{O}(n \log^2 n + m \log n)$ time.

Then, we make a call to Lemma 13 for each $x \in [1, \lfloor \log_3 n \rfloor - 1]$, and the union of sets $\mathcal{F}_\ell^a$ for $\ell \in [3x, 9x)$. Again by Lemma 15, for each such ℓ, we have $|\mathcal{F}_\ell^a| = \mathcal{O}(n/x + |\mathcal{F}_\ell|)$.

The total time complexity required by the calls to Lemma 13 is:

$$\begin{aligned}\sum_{x=1}^{\lfloor \log_3 n \rfloor - 1} \mathcal{O}\left(n \log n + \sum_{\ell=3x}^{9x-1} |\mathcal{F}_\ell| \log n\right) &= \mathcal{O}(n \log^2 n) + \sum_{\ell=3}^{n} \mathcal{O}(|\mathcal{F}_\ell| \log n) \\ &= \mathcal{O}(n \log^2 n + m \log n).\end{aligned}$$

We have thus proved the main result of this section.

Theorem 16. $|\mathsf{Factors}(\mathcal{F})|$ *can be computed in* $\mathcal{O}(n \log^2 n + m \log n)$ *time.*

4.4 Reporting Factors

The reporting version of the Path Pairs Problem, where one is to output $\bigcup_{(\pi,\pi') \in \Pi} \mathsf{Induced}(\pi, \pi')$, can be solved in $\mathcal{O}(N + M \log N + \mathsf{output})$ time by a straightforward modification of the proof of Lemma 10.[1] We can also retrieve a pair of paths inducing each pair of nodes within the same time complexity (in order to be able to represent the relevant string as a factor of T).

Theorem 17. *All elements of* $\mathsf{Factors}(\mathcal{F})$ *can be reported (without duplicates) in* $\mathcal{O}(n \log^2 n + m \log n + \mathsf{output})$ *time.*

5 Final Remarks

Another natural representation of factors consists in a set of intervals $\mathcal{I}$, such that each $[i, j] \in \mathcal{I}$ represents all factors of $T[i\,..\,j]$. This problem is very closely related to the problem of property indexing [3,7,8,21]. Employing either of the optimal property indexes that were presented in [7,8], one can retrieve the (number of) represented factors in optimal time.

[1] The workhorse of Lemma 10 is computing the size of the union of certain 1D-intervals. For the reporting version, we simply have to report all elements of this union.

References

1. Alamro, H., Badkobeh, G., Belazzougui, D., Iliopoulos, C.S., Puglisi, S.J.: Computing the antiperiod(s) of a string. In: 30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019. LIPIcs, vol. 128, pp. 32:1–32:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.CPM.2019.32
2. Alzamel, M., Conte, A., Greco, D., Guerrini, V., Iliopoulos, C.S., Pisanti, N., Prezza, N., Punzi, G., Rosone, G.: Online algorithms on antipowers and antiperiods. In: String Processing and Information Retrieval - 26th International Symposium, SPIRE 2019. Lecture Notes in Computer Science, vol. 11811, pp. 175–188. Springer (2019). https://doi.org/10.1007/978-3-030-32686-9_13
3. Amir, A., Chencinski, E., Iliopoulos, C.S., Kopelowitz, T., Zhang, H.: Property matching and weighted matching. Theor. Comput. Sci. **395**(2-3), 298–310 (2008). https://doi.org/10.1016/j.tcs.2008.01.006
4. Badkobeh, G., Fici, G., Puglisi, S.J.: Algorithms for anti-powers in strings. Inf. Process. Lett. **137**, 57–60 (2018). https://doi.org/10.1016/j.ipl.2018.05.003
5. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. SIAM J. Comput. **46**(5), 1501–1514 (2017). https://doi.org/10.1137/15M1011032
6. Bannai, H., Inenaga, S., Köppl, D.: Computing all distinct squares in linear time for integer alphabets. In: 28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017. LIPIcs, vol. 78, pp. 22:1–22:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). https://doi.org/10.4230/LIPIcs.CPM.2017.22
7. Barton, C., Kociumaka, T., Liu, C., Pissis, S.P., Radoszewski, J.: Indexing weighted sequences: Neat and efficient. Inf. Comput. **270** (2020). https://doi.org/10.1016/j.ic.2019.104462
8. Charalampopoulos, P., Iliopoulos, C.S., Liu, C., Pissis, S.P.: Property suffix array with applications in indexing weighted sequences. ACM Journal of Experimental Algorithmics **25**(1) (2020). https://doi.org/10.1145/3385898
9. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. Cambridge University Press (2007)
10. Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Inf. Process. Lett. **106**(2), 75–80 (2008). https://doi.org/10.1016/j.ipl.2007.10.006
11. Crochemore, M., Iliopoulos, C.S., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: Extracting powers and periods in a word from its runs structure. Theor. Comput. Sci. **521**, 29–41 (2014). https://doi.org/10.1016/j.tcs.2013.11.018
12. Deza, A., Franek, F., Thierry, A.: How many double squares can a string contain? Discret. Appl. Math. **180**, 52–69 (2015). https://doi.org/10.1016/j.dam.2014.08.016
13. Farach, M.: Optimal suffix tree construction with large alphabets. In: 38th Annual Symposium on Foundations of Computer Science, FOCS 1997. pp. 137–143. IEEE Computer Society (1997). https://doi.org/10.1109/SFCS.1997.646102
14. Fici, G., Postic, M., Silva, M.: Abelian antipowers in infinite words. Adv. Appl. Math. **108**, 67–78 (2019). https://doi.org/10.1016/j.aam.2019.04.001
15. Fici, G., Restivo, A., Silva, M., Zamboni, L.Q.: Anti-powers in infinite words. J. Comb. Theory, Ser. A **157**, 109–119 (2018). https://doi.org/10.1016/j.jcta.2018.02.009
16. Fine, N.J., Wilf, H.S.: Uniqueness theorems for periodic functions. Proceedings of the American Mathematical Society **16**(1), 109–114 (1965). https://doi.org/10.2307/2034009

17. Fraenkel, A.S., Simpson, J.: How many squares can a string contain? J. Comb. Theory, Ser. A **82**(1), 112–120 (1998). https://doi.org/10.1006/jcta.1997.2843
18. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. J. Comput. Syst. Sci. **30**(2), 209–221 (1985). https://doi.org/10.1016/0022-0000(85)90014-5
19. Gawrychowski, P., I, T., Inenaga, S., Köppl, D., Manea, F.: Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes - finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets. Theory Comput. Syst. **62**(1), 162–191 (2018). https://doi.org/10.1007/s00224-017-9794-5
20. Gusfield, D., Stoye, J.: Linear time algorithms for finding and representing all the tandem repeats in a string. J. Comput. Syst. Sci. **69**(4), 525–546 (2004). https://doi.org/10.1016/j.jcss.2004.03.004
21. Hon, W., Patil, M., Shah, R., Thankachan, S.V.: Compressed property suffix trees. Inf. Comput. **232**, 10–18 (2013). https://doi.org/10.1016/j.ic.2013.09.001
22. Kempa, D., Kociumaka, T.: String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In: 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019. pp. 756–767. ACM (2019). https://doi.org/10.1145/3313276.3316368
23. Kociumaka, T.: Efficient Data Structures for Internal Queries in Texts. Ph.D. thesis, University of Warsaw (2018), `https://mimuw.edu.pl/~kociumaka/files/phd.pdf`
24. Kociumaka, T., Kubica, M., Radoszewski, J., Rytter, W., Waleń, T.: A linear-time algorithm for seeds computation. ACM Trans. Algorithms **16**(2) (2020). https://doi.org/10.1145/3386369
25. Kociumaka, T., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., Zuba, W.: Efficient representation and counting of antipower factors in words. In: 13th International Conference on Language and Automata Theory and Applications, LATA 2019. Lecture Notes in Computer Science, vol. 11417, pp. 421–433. Springer (2019). https://doi.org/10.1007/978-3-030-13435-8_31
26. Kociumaka, T., Radoszewski, J., Rytter, W., Straszyński, J., Waleń, T., Zuba, W.: Efficient representation and counting of antipower factors in words (2020), `https://arxiv.org/abs/1812.08101v3`
27. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. In: 26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015. pp. 532–551. SIAM (2015). https://doi.org/10.1137/1.9781611973730.36
28. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: String powers in trees. Algorithmica **79**(3), 814–834 (2017). https://doi.org/10.1007/s00453-016-0271-3
29. Kolpakov, R.: Some results on the number of periodic factors in words. Inf. Comput. **270** (2020). https://doi.org/10.1016/j.ic.2019.104459
30. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: 40th Annual Symposium on Foundations of Computer Science, FOCS 1999. pp. 596–604. IEEE Computer Society (1999). https://doi.org/10.1109/SFFCS.1999.814634
31. Lothaire, M.: Algebraic Combinatorics on Words. Cambridge University Press (2002). https://doi.org/10.1017/cbo9781107326019

Bibliography

[1] Amihood Amir, Gad M. Landau, Shoshana Marcus, and Dina Sokol. Two-dimensional maximal repetitions. In *26th Annual European Symposium on Algorithms, ESA 2018*, volume 112 of *LIPIcs*, pages 2:1–2:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ESA.2018.2.

[2] Amihood Amir, Gad M. Landau, Shoshana Marcus, and Dina Sokol. Two-dimensional maximal repetitions. *Theoretical Computer Science*, 812:49–61, 2020. doi:10.1016/j.tcs.2019.07.006.

[3] Alberto Apostolico and Valentin E. Brimkov. Fibonacci arrays and their two-dimensional repetitions. *Theoretical Computer Science*, 237(1-2):263–273, 2000. doi:10.1016/S0304-3975(98)00182-0.

[4] Alberto Apostolico and Valentin E. Brimkov. Optimal discovery of repetitions in 2D. *Discrete Applied Mathematics*, 151(1-3):5–20, 2005. doi:10.1016/j.dam.2005.02.019.

[5] Golnaz Badkobeh, Gabriele Fici, and Simon J. Puglisi. Algorithms for anti-powers in strings. *Information Processing Letters*, 137:57–60, 2018. doi:10.1016/j.ipl.2018.05.003.

[6] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

[7] Jon Louis Bentley. Algorithms for Klee's rectangle problems. Unpublished notes, Computer Science Department, Carnegie Mellon University, 1977.

[8] Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, volume 149 of *LIPIcs*, pages 22:1–22:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019. arXiv:1909.11577, doi:10.4230/LIPIcs.ISAAC.2019.22.

[9] Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Inf. Process. Lett.*, 106(2):75–80, 2008. doi:10.1016/j.ipl.2007.10.006.

[10] Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *J. Comb. Theory, Ser. A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.

[11] Paweł Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Tighter bounds and optimal algorithms for all maximal α-gapped repeats and palindromes - finding all maximal α-gapped repeats and palindromes in optimal worst case time on integer alphabets. *Theory of Computing Systems*, 62(1):162–191, 2018. doi:10.1007/s00224-017-9794-5.

[12] Tomohiro I and Dominik Köppl. Improved upper bounds on all maximal α-gapped repeats and palindromes. *Theoretical Computer Science*, 753:1–15, 2019. doi:10.1016/j.tcs.2018.06.033.

[13] Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient colored orthogonal range counting. *SIAM Journal on Computing*, 38(3):982–1011, 2008. doi:10.1137/070684483.

[14] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551. SIAM, 2015. doi:10.1137/1.9781611973730.36.