

# Efficient Representation and Counting of Antipower Factors in Words

Tomasz Kociumaka<sup>1,2</sup>, Jakub Radoszewski<sup>1</sup>, Wojciech Rytter<sup>1</sup>, Juliusz Straszyński<sup>1</sup>,  
Tomasz Waleń<sup>1</sup>, and Wiktor Zuba<sup>1</sup>

<sup>1</sup> Institute of Informatics, University of Warsaw, Poland,

{kociumaka,jrad,rytter,jks,walen,w.zuba}@mimuw.edu.pl

<sup>2</sup>Department of Computer Science, Bar-Ilan University, Ramat Gan, Israel

## 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 the time complexity of  $\mathcal{O}(n^2/k)$  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 from LATA 2019. In particular, all results about counting different antipowers factors are completely new compared with the LATA proceedings version.

## 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 regularity of words, based on diversity rather than on equality, that was recently introduced by Fici et al. in [11, 12]. Algorithmic study of antipowers was initiated by Badkobeh et al. [2]. Very recently, a related concept of antiperiods was considered by Alamro et al. [1].

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  $w[i..j]$  we denote a *fragment* of  $w$  composed of letters  $w[i], \dots, w[j]$ . The corresponding word  $w[i] \dots w[j]$  is called a *factor* of  $w$ . I.e., a fragment is a *positioned factor*. A fragment (and thus some occurrence of a factor) of  $w$  can be represented in  $\mathcal{O}(1)$  space by the indices  $i$  and  $j$ . Badkobeh et al. [2] considered fragments of a word that are antipowers and obtained the following result.

**Fact 1.1** ([2]). *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. [2] 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 [13, 9]. 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  $w[i..j]$  is a  $k$ -antipower. Badkobeh et al. [2] proposed the following data structures for answering such queries.

**Fact 1.2** ([2]). *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. 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  $\{1, \dots, n^{\mathcal{O}(1)}\}$ . Our first result is an algorithm that computes the number  $C$  of fragments of a word of length  $n$  that are  $k$ -antipowers in  $\mathcal{O}(nk \log k)$  time and reports all of 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  $r \in \{1, \dots, n\}$ , 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 [18].

**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  $i$ th letter. Let  $[i..j]$  denote the integer interval  $\{i, i + 1, \dots, j\}$  and  $[i..j)$  denote  $[i..j - 1]$ . By  $w[i..j]$  we denote the *fragment* of  $w$  between the  $i$ th and the  $j$ th letter, inclusively. Fragments are also called *positioned factors*. If  $i > j$ , the fragment is empty. Let us further denote  $w[i..j) = w[i..j - 1]$ . The word  $w[i] \cdots w[j)$  that corresponds to the fragment  $w[i..j)$  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  $i \in [0..|w| - p)$ .

An  $\alpha$ -gapped repeat  $\gamma$  (for  $\alpha \geq 1$ ) in a word  $w$  is a fragment of  $w$  of the form  $uvu$  such that  $|uv| \leq \alpha|u|$ . The two occurrences of  $u$  are called *arms* of the  $\alpha$ -gapped repeat and  $|uv|$ , denoted  $\text{per}(\gamma)$ , is called *the period* of the  $\alpha$ -gapped repeat. Note that an  $\alpha$ -gapped repeat is also an  $\alpha'$ -gapped repeat for every  $\alpha' > \alpha$ . An  $\alpha$ -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  $\alpha$ -gapped repeats  $\alpha$ -MGRs and the set of  $\alpha$ -MGRs in a word  $w$  is further denoted by  $\text{MGReps}_\alpha(w)$ . The first algorithm for computing  $\alpha$ -MGRs was proposed by Kolpakov et al. [20]. It was improved by Crochemore et al. [8], Tanimura et al. [23], and finally Gawrychowski et al. [14], who showed the following result.

**Fact 2.1** ([14]). *Given a word  $w$  of length  $n$  and a parameter  $\alpha$ , the set  $\text{MGReps}_\alpha(w)$  can be computed in  $\mathcal{O}(n\alpha)$  time and satisfies  $|\text{MGReps}_\alpha(w)| \leq 18\alpha n$ .*

A *run* (a maximal repetition) in a word  $w$  is a triple  $(i, j, p)$  such that  $w[i..j]$  is a fragment with the smallest period  $p$ ,  $2p \leq j - i + 1$ , 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 [19] showed that a word of length  $n$  has  $\mathcal{O}(n)$  runs, with sum of exponents  $\mathcal{O}(n)$ , and that they can be computed in  $\mathcal{O}(n)$  time. Bannai et al. [3] recently refined these combinatorial results.

**Fact 2.2** ([3]). *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.*

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

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

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

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

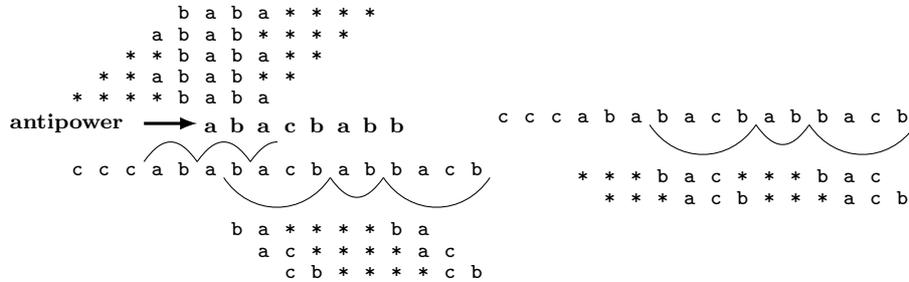


Figure 1: To the left: all weak  $(4, 2)$ -powers and one  $(4, 2)$ -antipower in a word of length 16. An asterisk denotes any character. The first five weak  $(4, 2)$ -powers are generated by the run  $\text{ababa}$  with period 2, and the last three are generated by the 1.5-MGR  $\text{bacbabacb}$ , 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.

An *interval representation* of a set  $X$  of integers is

$$X = [i_1..j_1] \cup [i_2..j_2] \cup \dots \cup [i_t..j_t],$$

where  $i_1 \leq j_1, j_1 + 1 < i_2, i_2 \leq j_2, \dots, j_{t-1} + 1 < i_t, i_t \leq j_t$ . We denote this representation by  $\mathcal{R}(X)$ . The value  $t$  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  $[0..n)$ . 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  $\mathcal{X}_1 \cap \mathcal{X}_2 \cap \dots \cap \mathcal{X}_r$  can be computed in  $\mathcal{O}(n+m)$  time.*

*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 [6]. Next, 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 it 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  $r$ .  $\square$

Let  $\mathcal{J}$  be a family of subintervals of  $[0..m)$ , 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 [21]). 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  $A = [0..m) \setminus \bigcup \mathcal{J}$ . We also show in Section 3 that a static range tree can 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. A *basic interval* is an interval of the form  $[a..a+2^i)$  that is a subinterval of  $[0..m-1)$  and such that  $i \geq 0$  is an integer and  $2^i \mid a$ . For example, the basic intervals for  $m = 8$  are  $[0..1), \dots, [7..8), [0..2), [2..4), [4..6), [6..8), [0..4), [4..8), [0..8)$ . In a *static range tree* (sometimes called a segment tree; see [21]) each node is identified with a basic interval. The children of a node  $J = [a..a+2^i)$ , for  $i > 0$ , are  $lchild(J) = [a..a+2^{i-1})$  and  $rchild(J) = [a+2^{i-1}..a+2^i)$ . Thus, a static range tree is a full binary tree of size  $\mathcal{O}(m)$ . The root of the tree, *root*, corresponds to  $[0..m)$ .

Every interval  $I \subseteq [0..m)$  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  $Decomp(I)$ ; see Fig. 2.

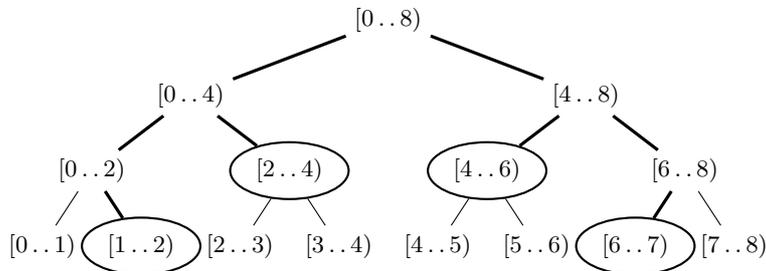


Figure 2: A static range tree for  $m = 8$  with the set of nodes that comprises  $Decomp([1..7))$ . The paths visited in the recursive decomposition algorithm are shown in bold.

Proofs of the lemmas from Section 2 follow.

Instead of Lemma 2.5, we show an equivalent lemma with an operation `count'` which returns  $|[0..m) \setminus \bigcup \mathcal{J}|$ .

**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  $[m..m')$  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 apply a static range tree. Every node  $J$  of the tree stores two values (see Fig. 3):

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

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

- If  $bi(J) > 0$ , then  $val(J) = 0$ .
- Otherwise, we define  $val(J) = 1$  if  $J$  is a leaf and  $val(J) = val(lchild(J)) + val(rchild(J))$  if it is not.

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

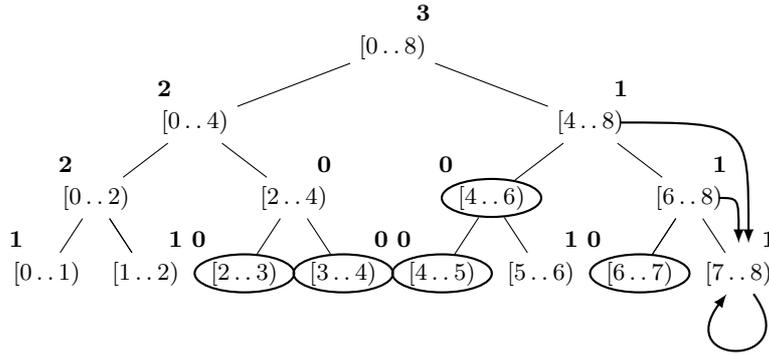


Figure 3: A static range tree for  $m = 8$  that stores the family  $\mathcal{J} = \{[2..3), [3..5), [4..7), [6..7)\}$ . The values  $val(J)$  are shown in bold. The arrows present selected *jump* pointers (cf. Lemma 2.6).

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. □

*Proof of Lemma 2.6.* 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 work in  $\Omega(|A| \log m)$  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 complexity.

Let us consider a subtree that is composed of all the nodes  $J$  with positive  $val_{root}(J)$ . 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.  $\square$

## 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  $uvu$  if the gapped repeat has period  $p = (q + 1)d$  and  $w[i..i + d]$ ,  $w[i + p..i + p + d]$  are contained in the first arm and in the second arm of the gapped repeat, respectively; cf. Fig. 4. In other words,  $u = u_1u_2u_3$ ,  $|u_2| = d$ ,  $|u_3vu_1| = qd$ , and  $uvu$  starts in the input word at position  $i - |u_1|$ .

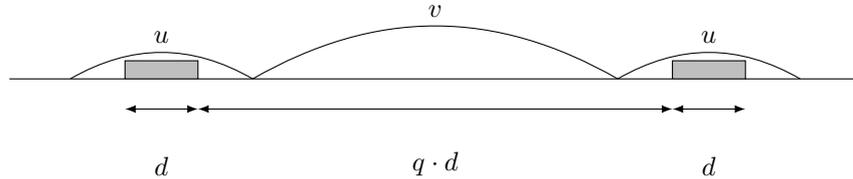


Figure 4: An occurrence of a gapped  $(q, d)$ -square generated by a gapped repeat with period  $(q + 1)d$ . Gray rectangles represent equal words.

Similarly, an occurrence in  $w$  of a  $(q, d)$ -square is *generated* by a generalized run with period  $p = (q + 1)d$  if it is fully contained in this generalized run. See Fig. 5 for a concrete example.

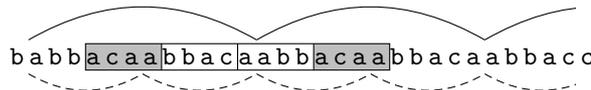


Figure 5: An occurrence of a gapped  $(2, 4)$ -square `acaa bbac aabb acaa` generated by a generalized run with period 12. Note that the generalized run has its origin in a (generalized) run with period 6 (depicted below) that *does not* generate this gapped square.

**Lemma 4.1.**

- (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  $(q + 1)d$ .
- (b) Each gapped repeat and each run  $\gamma$  with period  $(q + 1)d$  generates a single interval of positions where gapped  $(q, d)$ -squares occur, which is further denoted by  $Squares(q, d, \gamma)$  (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  $\gamma = w[i' .. j']$  the longest factor with period  $p$  that contains  $x$  (i.e., such that  $i' \leq i$  and  $i + \ell - 1 \leq j'$ ).

If  $|y| < 2p$ , then  $\gamma$  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  $d$ .

Otherwise (if  $|\gamma| \geq 2p$ ), the factor  $\gamma$  corresponds to a generalized run  $(i', j', p)$  that generates the gapped square  $x$ . In particular, this happens for  $q = 0$ .

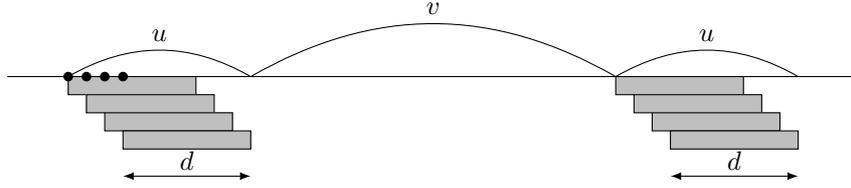


Figure 6: An interval, represented as a sequence of four consecutive positions (black dots), of starting positions of gapped  $(q, d)$ -square fragments generated by a gapped repeat with period  $(q + 1)d$ .

(b) Let  $\gamma$  be a gapped repeat or a generalized run with length  $\ell$  and period  $p = (q + 1)d$  that starts at position  $i$  in  $w$ . Then  $\gamma$  generates gapped  $(q, d)$ -squares that start at positions in  $[i .. i + \ell - (p + d)]$ .  $\square$

Let us denote

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

This definition can be extended to intervals  $I$ . To this end, let us introduce the operation

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

and define  $Chain_k(q, d, I) = I \cup (I \ominus d) \cup (I \ominus 2d) \cup \dots \cup (I \ominus (k - q - 2)d)$ . This set is further referred to as an *interval chain*; it can be stored in  $\mathcal{O}(1)$  space.

We denote by  $WeakPow_k(d)$  the set of starting positions in  $w$  of weak  $(k, d)$ -power fragments. A *chain representation* of a set of integers is its representation as a union of interval chains, limited to some base interval (in the case of weak  $(k, d)$ -powers, this will be  $[0 .. n - kd]$ ). The size of the chain representation is the number of chains. The following lemma shows how to compute small chain representations of the sets  $WeakPow_k(d)$ .

**Lemma 4.2.**

(a)  $WeakPow_k(d) = \bigcup_{q=0}^{k-2} \bigcup_{i \in Squares(q, d)} Chain_k(q, d, i) \cap [0 .. n - kd]$ .

(b)  $WeakPow_k(d) = \bigcup_{q=0}^{k-2} \bigcup \{Chain_k(q, d, I) : \gamma \in MGReps_{q+1}(w) \cup GRuns(w), \text{ where } \text{per}(\gamma) = (q + 1)d \text{ and } I = Squares(q, d, \gamma)\} \cap [0 .. n - kd]$ .

(c) For  $d = 1, \dots, \lfloor n/k \rfloor$ , the sets  $WeakPow_k(d)$  have chain representations of total size  $\mathcal{O}(nk^2)$  which can be computed in  $\mathcal{O}(nk^2)$  time. In particular,  $\sum_{d=1}^{\lfloor n/k \rfloor} \sum_{q=0}^{k-2} |\mathcal{R}(Squares(q, d))| = \mathcal{O}(nk^2)$  and all these interval representations can be computed in  $\mathcal{O}(nk^2)$  time.

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

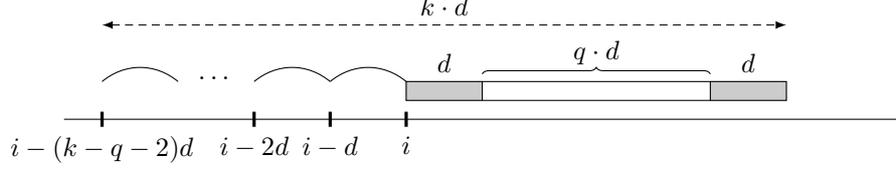


Figure 7: The fact that  $i \in \text{Squares}(q, d)$  is a witness of inclusion  $(\text{Chain}_k(q, d, i) \cap [0..n - kd]) \subseteq \text{WeakPow}_k(d)$ .

---

**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
     $p := \text{per}(\gamma)$ 
    if  $(q + 1) \mid p$  then
       $d := \frac{p}{q+1}$ 
       $I := \text{Squares}(q, d, \gamma)$ 
       $\mathcal{C}_d := \mathcal{C}_d \cup \{ \text{Chain}_k(q, d, I) \}$ 
   $\text{antipowers} := 0$ 
for  $d := 1$  to  $\lfloor n/k \rfloor$  do
   $\text{WeakPow}_k(d) := (\bigcup \mathcal{C}_d) \cap [0..n - kd]$ 
   $\text{antipowers} := \text{antipowers} + (n - kd + 1) - |\text{WeakPow}_k(d)|$ 
return  $\text{antipowers}$ 

```

---

Formula in (b) follows from point (a) by Lemma 4.1. Indeed, Lemma 4.1(a) shows that every gapped  $(q, d)$ -square fragment is generated by a  $(q + 1)$ -MGR with period  $(q + 1)d$  or a generalized run with period  $(q + 1)d$ . By Lemma 4.1(b), the starting positions of all such gapped squares that are generated by an MGR or a generalized run  $\gamma$  form an interval  $I = \text{Squares}(q, d, \gamma)$ . Hence, it yields an interval chain  $\text{Chain}_k(q, d, I)$  of starting positions of weak  $(k, d)$ -powers by point (a).

Finally, we obtain point (c) by applying the formula from point (b) to compute the chain representations of sets  $\text{WeakPow}_k(d)$  for all  $d = 1, \dots, \lfloor n/k \rfloor$ . This is also shown in the first part of the following SimpleCount algorithm, where the resulting chain representations are denoted as  $\mathcal{C}_d$ . The total number of interval chains in these representations is  $\mathcal{O}(nk^2)$  because, for each  $q \in [0..k - 2]$ , the number of  $(q + 1)$ -MGRs and generalized runs  $\gamma$  is bounded by  $\mathcal{O}(nk)$  due to Facts 2.1 and 2.2, respectively. Moreover, the desired interval representations of the sets  $\text{Squares}(q, d)$  can be computed from the intervals  $\text{Squares}(q, d, \gamma)$  in linear time using Lemma 2.4.  $\square$

Lemma 4.2 lets us count  $k$ -antipowers by computing the size of the complementary sets  $\text{WeakPow}_k(d)$ . Thus, we obtain the following preliminary result.

**Proposition 4.3.** *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 Lemma 4.2, points (b) and (c), to express the sets  $\text{WeakPow}_k(d)$  for all  $d = 1, \dots, \lfloor n/k \rfloor$  as a union of  $\mathcal{O}(nk^2)$  interval chains. 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  $\text{WeakPow}_k(d)$  in  $\mathcal{O}(nk^3)$  total time. Finally, the size of the complement of the set  $\text{WeakPow}_k(d)$  (in  $[0..n - kd]$ ) 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 processing the chains by their 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 4.2(c). By inspecting the structure of MGRs, we actually show that the formula from Lemma 4.2(b) generates only  $\mathcal{O}(nk)$  interval chains. A careful implementation lets us compute such a chain representation in  $\mathcal{O}(nk)$  time.

We say that an  $\alpha$ -MGR for integer  $\alpha$  with period  $p$  is *nice* if  $\alpha \mid p$  and  $p \geq 2\alpha^2$ . Let  $NMGReps_\alpha(w)$  denote the set of nice  $\alpha$ -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  $\alpha > 1$ ,  $|NMGReps_\alpha(w)| \leq 54n$ .*

*Proof.* Let us consider a partition of the word  $w$  into blocks of  $\alpha$  letters (the final  $n \bmod \alpha$  letters are not assigned to any block). Let  $uvu$  be a nice  $\alpha$ -MGR in  $w$ . We know that  $2\alpha^2 \leq |uv| \leq \alpha|u|$ , so  $|u| \geq 2\alpha$ . Now, let us fit the considered  $\alpha$ -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  $\alpha$ . 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  $u'$ .

Let us notice that  $|uv| = |u'v'|$ . Moreover,  $|u'| \geq \frac{1}{3}|u|$  since  $u$  encompasses at least one full block of  $w$ . Consequently,  $|u'v'| \leq 3\alpha|u'|$ .

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  $3\alpha$ -gapped repeat in  $t$ . It is also a  $3\alpha$ -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  $\alpha$ -MGR. This concludes that every nice  $\alpha$ -MGR in  $w$  has a corresponding  $3\alpha$ -MGR in  $t$ . Also, every  $3\alpha$ -MGR in  $t$  corresponds to at most one nice  $\alpha$ -MGR in  $w$ , as it can be translated into blocks of  $w$  and expanded in a single way to a  $3\alpha$ -MGR (that can happen to be a nice  $\alpha$ -MGR).

We conclude that the number of nice  $\alpha$ -MGRs in  $w$  is at most the number of  $3\alpha$ -MGRs in  $t$ . As  $|t| \leq n/\alpha$ , due to Fact 2.1 the latter is at most  $54n$ .  $\square$

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

*Proof.* The chain representations of sets  $WeakPow_k(d)$  are computed for  $d < 2k - 2$  and for  $d \geq 2k - 2$  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  $WeakPow_k(d)$  (and its trivial chain representation) in  $\mathcal{O}(n)$  time. Across all  $d < 2k - 2$ , this gives  $\mathcal{O}(nk)$  chains and  $\mathcal{O}(nk)$  time.

Henceforth we consider the case that  $d \geq 2k - 2$ . Let us note that if a gapped  $(q, d)$ -square with  $d \geq 2(q + 1)$  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  $p = (q + 1)d \geq 2(q + 1)^2$ . This observation lets us express the formula of Lemma 4.2(b) for  $d \geq 2k - 2$  equivalently using  $NMGReps_{q+1}(w)$  instead of  $MGReps_{q+1}(w)$ .

By Fact 2.2 and Lemma 5.1, for every  $q$  we have only  $|NMGReps_{q+1}(w) \cup GRuns(w)| = \mathcal{O}(n)$  MGRs and generalized runs to consider. Hence, the total size of chain representations of sets  $WeakPow_k(d)$  for  $d \geq 2k - 2$  is  $\mathcal{O}(nk)$  as well. The same applies to the total size of interval representations of sets  $Squares(q, d)$  for  $d \geq 2k - 2$ .

The last piece of the puzzle is the following claim.

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

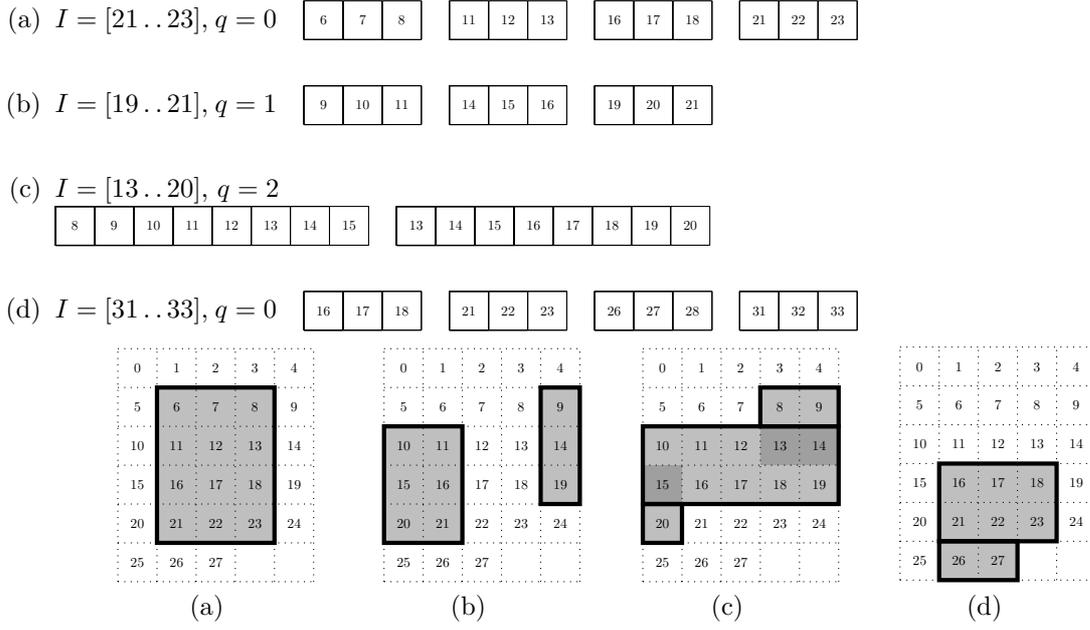


Figure 8: Examples of decompositions of various interval chains  $Chain_k(q, d, I)$  into orthogonal rectangles in the grid  $\mathcal{G}_d$  for  $d = 5, k = 5, n = 52$ .

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

$$\mathbf{next}_p[\alpha] = \min\{\alpha' \in [\alpha + 1 \dots k] : \alpha' \mid p\}.^1$$

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

This concludes the proof.  $\square$

## 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  $[0 \dots n - kd]$  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  $Chain_k(q, d, I)$  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.*

*Proof.* Translating the set  $Chain_k(q, d, I)$  onto our grid representation, it becomes a union of horizontal strips, each corresponding to an interval  $I \ominus ad$ , for  $a \in [0 \dots k - q - 2]$ , 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:

<sup>1</sup>We assume that  $\min \emptyset = \infty$ .

- 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 some cells that are not on the grid (negative numbers or numbers greater than  $n - kd$ ); see Fig. 8(d). In that case, we consider the first and the last included rows as individual rectangles; the remaining part of the decomposition corresponds to one of the cases mentioned before.  $\square$

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 (see Bentley [5]). 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  $[0..d]^2$  with integer coordinates are given and that each rectangle has height at most  $k$  or width exactly  $d$ . The area of their union can be computed in  $\mathcal{O}(r \log k + d)$  time and  $\mathcal{O}(r + d)$  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. The algorithm performs a sweep line 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  $\mathcal{O}(r + d)$  time using bucket sort [6].

For each strip, the sweep line stores a data structure that allows insertion and deletion of intervals with integer coordinates in  $[0..k]$  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  $\mathcal{O}(d)$  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- $d$  rectangles. Each of them induces a vertical interval on the second component. First, in  $\mathcal{O}(r + d)$  time the union  $S$  of these intervals represented as a union of pairwise disjoint maximal intervals can be 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  $2r + d/k$  additional intervals are inserted so the time complexity is still  $\mathcal{O}((r + d/k) \log k + d) = \mathcal{O}(r \log k + d)$ .  $\square$

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  $WeakPow_k(d)$  for  $d = 1, \dots, \lfloor n/k \rfloor$  as unions of  $\mathcal{O}(nk)$  interval chains. This takes  $\mathcal{O}(nk)$  time. Each chain is represented on the corresponding grid  $\mathcal{G}_d$  as the union of a constant number of rectangles using Lemma 5.4. This gives  $\mathcal{O}(nk)$  rectangles in total on all the grids  $\mathcal{G}_d$ , each of height at most  $k$  or width exactly  $d$ , for the given  $d$ .

As the next step, we renumber the components in the grids by assigning consecutive numbers to the components that correspond to rectangle vertices. This can be done in  $\mathcal{O}(nk)$  time, for all the grids simultaneously, using bucket sort [6]. The new components store the original values. After this transformation, rectangles with height at most  $k$  retain this property and rectangles with width  $d$  have maximal width. Let the maximum component in the grid  $\mathcal{G}_d$  after renumbering be equal to  $M_d$  and the number of rectangles in  $\mathcal{G}_d$  be  $R_d$ ; then  $\sum_d R_d = \mathcal{O}(nk)$  and  $\sum_d M_d = \mathcal{O}(nk)$ .

As the final step, we apply the algorithm of Lemma 5.5 to each grid to compute  $|WeakPow_k(d)|$  as the area of the union of the rectangles in the grid. One can readily verify that it can be adapted to compute the areas of the rectangles in the original components. The algorithm works in  $\mathcal{O}(\sum_d R_d \log k + \sum_d M_d) = \mathcal{O}(nk \log k)$  time. In the end, the number of  $(k, d)$ -antipower fragments equals  $n - kd + 1 - |WeakPow_k(d)|$ .  $\square$

## 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. [2] (see Fact 1.2). Let us recall that an antipower query  $(i, j, k)$  asks to check if a fragment  $w[i..j]$  of the word  $w$  is a  $k$ -antipower.

**Theorem 6.2.** *Assume that a word of length  $n$  is given. For every  $r \in [1..n]$ , 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  $r \in [1..n]$ . If an antipower query  $(i, j, k)$  satisfies  $k \leq r$ , 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  $w[i..j]$  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  $b \leq n/r$ . We group the length- $b$  fragments of  $w$  by the remainder modulo  $b$  of their starting position. For a remainder  $g \in [0..b-1]$  and index  $i \in [0.. \lfloor \frac{n-g}{b} \rfloor]$ , we store, as  $A_g^b[i]$ , the smallest index  $j > i$  such that  $w[jb+g..j(b+1)+g] = w[ib+g..i(b+1)+g]$  ( $j = \infty$  if it does not exist). We also store a data structure for range minimum queries over  $A_g^b$  for each group; it uses linear space, takes linear time to construct, and answers queries in constant time (see [4]). 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  $A_g^b$  for all  $b \in [1..m]$  and  $g \in [0..b-1]$  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  $[0..n)$  such that the 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  $\infty$ . For a given  $g$ , the indices  $i$  are considered in descending order. For each  $i$ , we take as  $x$  the identifier of the factor  $w[ib+g..i(b+1)+g]$ , 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  $\infty$ . For any  $b$  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

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

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 uncompact 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 [10].

**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 [15, 7]. 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 the following auxiliary data structure 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** ([17, Section 7.1]). *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  $y_0, \dots, y_{k-1}$ , 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  $0 \leq i < j < k$ .*

We denote by  $WeakPow_{k,i,j}(d)$  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.

**Lemma 7.4.** *For a given  $k$ , the sets  $WeakPow_{k,i,j}(d)$  for all  $d = 1, \dots, \lfloor n/k \rfloor$  and  $0 \leq i < j < k$  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  $a \in WeakPow_{k,i,j}(d)$  if and only if all the following conditions are satisfied:

1.  $a + i \cdot d \in Squares(j - i - 1, d)$

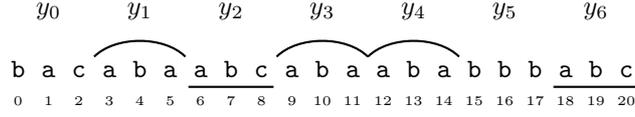


Figure 9: This weak  $(7, 3)$ -power is actually a weak  $(7, 1, 3, 3)$ -power. We have  $0 \in \text{WeakPow}_{7,1,3}(3)$ , since  $3 \in \text{Squares}(1, 3)$ ,  $3 \notin \text{Squares}(0, 3)$ , and  $0 \notin \text{Squares}(q, 3)$  for  $q \in [0..5]$ .

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

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  $y_i \neq y_{j'}$  for  $i < j' < j$ , and the third condition to  $y_{i'} \neq y_{j'}$  for  $i' < i$  and  $i' < j' < k$ .

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

$$\begin{aligned}
 A_{i,j}(d) &= \text{Squares}(j - i - 1, d) \ominus (i \cdot d), \\
 B_{i,j}(d) &= \bigcup_{q=0}^{j-i-2} (\text{Squares}(q, d) \ominus (i \cdot d)), \\
 C_{i,j}(d) &= \bigcup_{c=0}^{i-1} \bigcup_{q=0}^{k-c-2} (\text{Squares}(q, d) \ominus (c \cdot d)).
 \end{aligned}$$

By Lemma 4.2(c), the interval representations of all sets  $\text{Squares}(q, d)$  for  $0 \leq q \leq k - 2$  can be computed in  $\mathcal{O}(nk^2)$  time. By Lemma 5.2, the total size of interval representations of sets  $\text{Squares}(q, d)$  over all  $d \geq 2k - 2$  is  $\mathcal{O}(nk)$ . We further have:

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

*Proof.* The interval representation of the set  $\text{Squares}(q, d)$  has size  $\mathcal{O}(n/d)$ . Indeed, if  $a < b < a + d$  and  $a, b \in \text{Squares}(q, d)$ , then  $c \in \text{Squares}(q, d)$  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  $\mathcal{O}(k \sum_{d=1}^{2k-3} n/d) = \mathcal{O}(nk \log k)$ .  $\square$

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

For any  $i, j$ , and  $d$ ,

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

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

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

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).  $\square$

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  $\gamma$  if the  $(j - i - 1, d)$ -square  $y_i \cdots y_j$  is generated by  $\gamma$ . We denote by  $\text{WeakPow}_{k,i,j}(d, \gamma)$  the set of starting positions of weak  $(k, i, j, d)$ -powers generated by  $\gamma$ . It can be readily verified that the intervals generated in the above lemma can be labeled by the MGR or generalized run  $\gamma$  that generated them. This labelling is unique due to the following simple observation.

**Observation 7.6.** For any different MGRs or generalized runs  $\gamma_1, \gamma_2$ , the sets  $WeakPow_{k,i,j}(d, \gamma_1)$  and  $WeakPow_{k,i,j}(d, \gamma_2)$  are disjoint.

*Proof.* It suffices to note that for any  $q \leq k - 2$  and  $d$ , the sets  $Squares(q, d, \gamma_1)$  and  $Squares(q, d, \gamma_2)$  are disjoint.  $\square$

Let us first show how to count different weak  $(k, i, j, d)$ -powers for  $i > 0$ . The case of  $i = 0$  will be taken care of 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  $g(x) = q \in [0..kd]$  is a *synchronizer* if for every  $a \in WeakPow_{k,i,j}(d, \gamma)$ , the value  $a + g(w[a] \dots w[a + kd - 1])$  is the same.

Note that a synchronizer function is defined on factors of  $w$ , not on fragments; i.e., it admits 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  $\alpha = w[a..b]$  of  $w$ , let us denote  $\mathbf{start}(\alpha) = a$  and  $\mathbf{end}(\alpha) = b$ .

**Lemma 7.8.** A function *synch* that assigns to every weak  $(k, i, j, d)$ -power  $x$ , for  $i > 0$ , such that  $x = w[a] \dots w[a + kd - 1]$  and  $a \in WeakPow_{k,i,j}(d, \gamma)$ , the position  $\mathbf{start}(\gamma) - a$ , is a synchronizer.

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

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

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. Let  $w[a] \dots w[a + kd - 1] = y_0 \dots y_{k-1} = x$  be an occurrence of a weak  $(k, i, j, d)$ -power for equal-length words  $y_0, \dots, y_{k-1}$  and let  $\gamma$  be the MGR or generalized run that generates it. We have  $y_i = y_j$  and  $\gamma$  has period  $p = (j - i)d$ . Let  $r = \max\{b < i \cdot d : x[b] \neq x[b + p]\}$ . 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  $\gamma$ , i.e.,  $\mathbf{synch}(x) = \mathbf{start}(\gamma) - a = r + 1$ . Hence, indeed this value does not depend on the position  $a$  and  $\mathbf{synch}(x) \in [0, |x|]$ .  $\square$

### 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  $(\pi, \pi')$  of equal-length paths where  $\pi$  is a path going downwards in  $T$  and  $\pi'$  is a path going upwards in  $T'$ .

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

*Example 7.10.* Let us consider the instance of PATH PAIRS PROBLEM from Fig. 10. We have  $P = \{(\pi_1, \pi'_1), (\pi_2, \pi'_2), (\pi_3, \pi'_3)\}$ , where

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

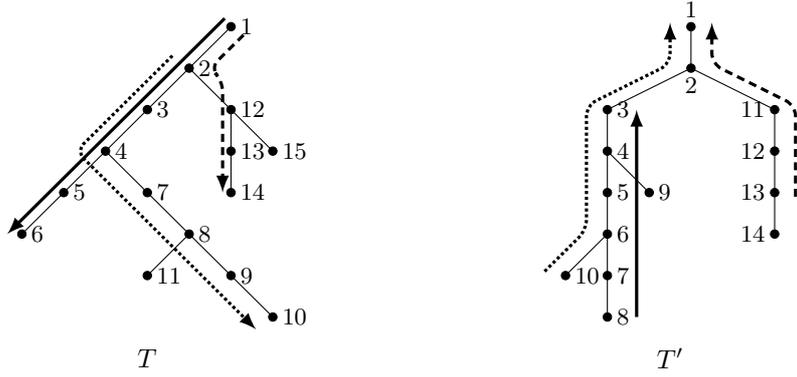


Figure 10: Illustration of PATH PAIRS PROBLEM and Example 7.10. For simplicity, the trees in this example do not contain implicit nodes.

Then

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

In total  $|\bigcup_{i=1}^3 \text{Induced}(\pi_i, \pi'_i)| = 16$  and  $\text{Induced}(\pi_1, \pi'_1) \cap \text{Induced}(\pi_2, \pi'_2) = \{(3, 6), (4, 5)\}$ .

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  $0 < i < j < k$ ,  $d \leq \frac{n}{k}$  in a word of length  $n$  reduces in  $\mathcal{O}(nk^4 \log k)$  time to a 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 every interval  $[a..b]$  in the interval representation of  $\text{WeakPow}_{k,i,j}(d)$ , let us denote  $q = a + \text{synch}(w[a..b])$ . Then we create a downwards path  $\pi$  in  $T$  that connects the loci of  $w[q..a+kd]$  and  $w[q..b+kd]$  and an upwards path  $\pi'$  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, we make the endpoints of the paths 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 a PATH PAIRS PROBLEM is equivalent to counting the number of different weak powers by the definition of a synchronizer.

By Lemma 7.4, the number of intervals in the interval representation of  $\text{WeakPow}_{k,i,j}(d)$  over all  $0 < i < j < k$  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.  $\square$

## 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 [22]. Here, we only consider explicit nodes of  $T$ . 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 of heavy edges; it includes the light edge going up from its topmost node provided that its topmost node is not the tree root. 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  $T$ .

In the solution to PATH PAIRS PROBLEM, we compute the heavy path decompositions of both trees  $T$  and  $T'$ . For each pair of paths  $(\pi, \pi')$  in  $P$ , we decompose each path  $\pi, \pi'$  into maximal fragments

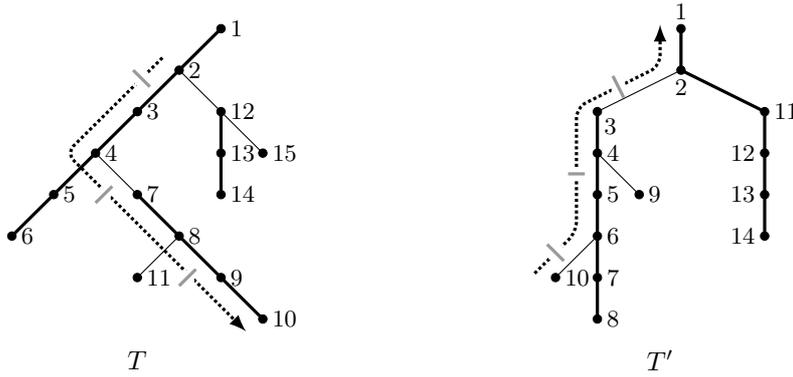


Figure 11: Partitioning of the second pair of paths from Example 7.10 into  $2, 3 \rightarrow 4, 7 \rightarrow 8, 9 \rightarrow 10$  and  $10, 6 \rightarrow 5, 4 \rightarrow 3, 2 \rightarrow 1$  along the heavy paths (drawn as thick edges).

belonging to different heavy paths. Note that the decomposition of the upwards path  $\pi'$  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  $\pi$  can be computed in  $\mathcal{O}(\log n)$  time by traversing  $\pi$  in the reverse direction. Then we further decompose the paths  $\pi, \pi'$  into maximal subpaths  $\pi = \pi_1, \dots, \pi_\ell, \pi' = \pi'_1, \dots, \pi'_\ell$  so that the lengths of  $\pi_i$  and  $\pi'_i$  are the same and each  $\pi_i$  and each  $\pi'_i$  is a fragment of one heavy path in  $T$  and in  $T'$ , respectively; we have  $\ell \leq 2 \log N$ . For an illustration, see Fig. 11. Finally, we create new pairs of paths  $(\pi_i, \pi'_i)$ , 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  $\mathcal{O}(M \log N)$ .

**Lemma 7.12.** *The answers to  $K$  special instances of 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  $0, 1, \dots, \mathcal{O}(N)$  in every instance. Then a path pair  $(\pi, \pi')$  such that  $\pi$  connects nodes with numbers  $i$  and  $j$  and  $\pi'$  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  $\text{minrot}(s)$  we denote a position  $i \in [0 \dots |s|)$  such that  $s[i \dots |s|]s[0 \dots i)$  is the lexicographically minimum cyclic shift of  $s$ . In case that there is more than one such position (i.e., that  $s$  is a power of a shorter word), we select as  $\text{minrot}(s)$  the first such position.

If  $i = 0$ , we partition every set  $A = \text{WeakPow}_{k,0,j}(d, \gamma)$  into four sets. Let

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



**Fact 7.16** ([16]). *One can preprocess a word  $w$  of length  $n$  in  $\mathcal{O}(n)$  time so that for any factor  $s$  of  $w$ ,  $\text{minrot}(s)$  can be computed in  $\mathcal{O}(1)$  time.*

Then the problem reduces to PATH PAIRS PROBLEM, as in the previous section.

**Lemma 7.17.** *Computing the number of different weak  $(k, 0, j, d)$ -powers for given  $k$  and all  $0 < j < k$ ,  $d \leq \frac{n}{k}$  in a word of length  $n$  reduces in  $\mathcal{O}(nk^3 \log k)$  time to a PATH PAIRS PROBLEM with  $M, N = \mathcal{O}(nk^3 \log k)$ .*

We finally arrive at the main result of this section.

**Theorem 7.18.** *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  $0 \leq i < j < k$ . 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 7.17 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] Hayam Alamro, Golnaz Badkobeh, Djamel Belazzougui, Costas S. Iliopoulos, and Simon J. Puglisi. Computing the antiperiod(s) of a string. In Nadia Pisanti and Solon P. Pissis, editors, *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019*, volume 128 of *LIPICs*, pages 32:1–32:11. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPICs.CPM.2019.32.
- [2] 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.
- [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] Michael A. Bender, Martin Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005. doi:10.1016/j.jalgor.2005.08.001.
- [5] Jon Louis Bentley. Algorithms for Klee’s rectangle problems. Unpublished notes, Computer Science Department, Carnegie Mellon University, 1977.
- [6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: <http://mitpress.mit.edu/books/introduction-algorithms>.
- [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] Maxime Crochemore, Roman Kolpakov, and Gregory Kucherov. Optimal bounds for computing  $\alpha$ -gapped repeats. In Adrian-Horia Dediu, Jan Janousek, Carlos Martín-Vide, and Bianca Truthe, editors, *Language and Automata Theory and Applications, LATA 2016*, volume 9618 of *Lecture Notes in Computer Science*, pages 245–255. Springer, 2016. doi:10.1007/978-3-319-30000-9\_19.
- [9] 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.

- [10] Martin Farach-Colton, Paolo Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.
- [11] Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q. Zamboni. Anti-powers in infinite words. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *Automata, Languages and Programming, ICALP 2016*, volume 55 of *LIPICs*, pages 124:1–124:9. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICs.ICALP.2016.124.
- [12] Gabriele Fici, Antonio Restivo, Manuel Silva, and Luca Q. Zamboni. Anti-powers in infinite words. *Journal of Combinatorial Theory, Series A*, 157:109–119, 2018. doi:10.1016/j.jcta.2018.02.009.
- [13] 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.
- [14] Paweł Gawrychowski, Tomohiro I, Shunsuke Inenaga, Dominik Köppl, and Florin Manea. Tighter bounds and optimal algorithms for all maximal  $\alpha$ -gapped repeats and palindromes - finding all maximal  $\alpha$ -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.
- [15] 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.
- [16] Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In Roberto Grossi and Moshe Lewenstein, editors, *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, volume 54 of *LIPICs*, pages 28:1–28:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPICs.CPM.2016.28.
- [17] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for seeds computation. *CoRR*, abs/1107.2422v2, 2019. arXiv:1107.2422v2.
- [18] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszypiński, Tomasz Waleń, and Wiktor Zuba. Efficient representation and counting of antipower factors in words. In Carlos Martín-Vide, Alexander Okhotin, and Dana Shapira, editors, *Language and Automata Theory and Applications - 13th International Conference, LATA 2019*, volume 11417 of *Lecture Notes in Computer Science*, pages 421–433. Springer, 2019. doi:10.1007/978-3-030-13435-8\_31.
- [19] Roman 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] Roman Kolpakov, Mikhail Podolskiy, Mikhail Posypkin, and Nickolay Khrapov. Searching of gapped repeats and subrepetitions in a word. *Journal of Discrete Algorithms*, 46-47:1–15, 2017. doi:10.1016/j.jda.2017.10.004.
- [21] Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In Gabriele Fici, Marinella Sciortino, and Rossano Venturini, editors, *String Processing and Information Retrieval - 24th International Symposium, SPIRE 2017, Proceedings*, volume 10508 of *Lecture Notes in Computer Science*, pages 290–303. Springer, 2017. doi:10.1007/978-3-319-67428-5\_25.
- [22] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.
- [23] Yuka Tanimura, Yuta Fujishige, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. A faster algorithm for computing maximal  $\alpha$ -gapped repeats in a string. In Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, editors, *String Processing and Information Retrieval, SPIRE 2015*, volume 9309 of *Lecture Notes in Computer Science*, pages 124–136. Springer, 2015. doi:10.1007/978-3-319-23826-5\_13.

This figure "fig\_rects.png" is available in "png" format from:

<http://arxiv.org/ps/1812.08101v3>