HAL
archives-ouvertes.fr

Finding the Anticover of a String

Mai Alzamel, Alessio Conte, Shuhei Denzumi, Roberto Grossi, Costas Iliopoulos, Kazuhiro Kurita, Kunihiro Wasa

▶ **To cite this version:**

Mai Alzamel, Alessio Conte, Shuhei Denzumi, Roberto Grossi, Costas Iliopoulos, et al.. Finding the Anticover of a String. 31st Annual Symposium on Combinatorial Pattern Matching (CPM 2020), 2020, Copenhagen, Denmark. 10.4230/LIPIcs.CPM.2020.2 . hal-02957658

HAL Id: hal-02957658
https://hal.inria.fr/hal-02957658

Submitted on 5 Oct 2020

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

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

Finding the Anticover of a String

Mai Alzamel
Department of Informatics,
King's College London, UK
Department of Computer Science,
King Saud University, KSA
mai.alzamel@kcl.ac.uk

Alessio Conte
Dipartimento di Informatica,
Università di Pisa, Italy
conte@di.unipi.it

Shuhei Denzumi
Graduate School of Information Science and Technology, The University of Tokyo, Japan
denzumi@mist.i.u-tokyo.ac.jp

Roberto Grossi
Dipartimento di Informatica,
Università di Pisa, Italy
grossi@di.unipi.it

Costas S. Iliopoulos
Department of Informatics,
King's College London, UK
costas.iliopoulos@kcl.ac.uk

Kazuhiro Kurita
IST, Hokkaido University, Sapporo, Japan
k-kurita@ist.hokudai.ac.jp

Kunihiro Wasa
National Institute of Informatics, Tokyo, Japan
wasa@nii.ac.jp

Abstract

A *k-anticover* of a string x is a set of pairwise distinct factors of x of equal length k, such that every symbol of x is contained into an occurrence of at least one of those factors. The existence of a k-anticover can be seen as a notion of non-redundancy, which has application in computational biology, where they are associated with various non-regulatory mechanisms. In this paper we address the complexity of the problem of finding a k-anticover of a string x if it exists, showing that the decision problem is NP-complete on general strings for $k \geq 3$. We also show that the problem admits a polynomial-time solution for $k = 2$. For unbounded k, we provide an exact exponential algorithm to find a k-anticover of a string of length n (or determine that none exists), which runs in $O^*(\min\{3^{\frac{n-k}{3}}, (\frac{k(k+1)}{2})^{\frac{n}{k+1}}\})$ time using polynomial space.

2012 ACM Subject Classification Mathematics of computing → Combinatorics on words

Keywords and phrases Anticover, String algorithms, Stringology, NP-complete

Digital Object Identifier 10.4230/LIPIcs.CPM.2020.2

1 Introduction

The notion of periodicity in strings is well studied in many fields like combinatorics on words, pattern matching, data compression and automata theory (see [16], [17]), because it is of paramount importance in several applications, not to talk about its theoretical aspects. Algorithms and data structures for finding repeating patterns or regularities in strings (see [9], [12]) are central to several fields of computer science including computational biology, pattern matching, data compression, and randomness testing. The nature and extent of periodicity in strings is also of immense combinatorial interest in its own right [17].

The notion of cover belongs to the area of quasiperiodicity, that is, a generalization of periodicity in which the occurrences of the period may overlap [3]. We call a proper factor u of a nonempty string y a *cover* of y, if every letter of y is within some occurrence of u in y. A cover u of y needs to be a border (i.e. a prefix and a suffix) of y. A cover of a string s is a string that covers all positions of s with its occurrences. Intuitively, s can be generated by overlapping/concatenating copies of its cover u. Covers in strings were already


© Mai Alzamel, Alessio Conte, Shuhei Denzumi, Roberto Grossi, Costas S. Iliopoulos, Kazuhiro Kurita, and Kunihiro Wasa;
licensed under Creative Commons License CC-BY

extensively studied. A linear-time algorithm finding the shortest cover of a string was given by Apostolico et al. [4] and later on improved into an on-line algorithm by Breslauer [20]. A linear-time algorithm computing all the covers of a string was proposed by Moore and Smyth [19]. Afterwards an on-line algorithm for the all-covers problem was given by Li and Smyth [15]. Similar combinatorial covering problems have been studied on graphs [8, 21], and other types of quasiperiodicities include seeds [13], as well as variants including approximate and partial covers and seeds.

A power of order k is defined by a concatenation of k identical blocks of symbols, where k is at least 2: it is evident how a covers are generalizations of powers. Powers in various forms later came to be important structures in computational biology, where they are associated with various regulatory mechanisms and play an important role in genomic fingerprinting (for further reading see, e.g., [14] and references therein). Antipowers are an orthogonal notion to that of powers, that were introduced recently by Fici et al. in [6, 10]. In contrast to powers, antipowers insist instead on the diversity of consecutive blocks: an *antipower (antiperiod)* of order k is a concatenation of k pairwise distinct strings of equal length. A linear algorithm for computing the antiperiods was given in [1], and online algorithms are given in [2].

We define an *anticover* as a generalization of the notion of antipower: an anticover of a string x is a set of pairwise distinct factors of x of equal length, such that every symbol of x is within some occurrence of one of those factors. Equivalently, x can be generated by overlapping/concatenating a set of pairwise distinct strings of equal length. Some practical motivation for this problem can be found in Mincu and Popa [18], that considers the similar problem of *partitioning* a string into distinct factors: they show that this problem is motivated by an application in the DNA compositions, a short DNA fragment can be obtained that can be self-united into the desired DNA structure. They present that to produce the wanted DNA structure, it is mandatory that no two fragments are equal.

In this paper we show that the computation of a 2-anticover of a string x of length n over an alphabet Σ, if it exists, can be done in $O(n|\Sigma|)$ time and space. For the general case, given $k \geq 3$, we show that checking whether a tring x has a k-anticover is NP-complete. Moreover, we provide an exact exponential algorithm to find a k-anticover of x (or determine that none exists), which runs in $O^*(\min\{3^{\frac{n-k}{3}}, (\frac{k(k-1)}{2})^{\frac{n}{k+1}}\})$ time using polynomial space.

In the literature Condon et al. [7] studied the complexity of partitioning problems for strings. In particular, they introduced the Equality-Free String Partition problem, which requires to partition a string x into factors $f_1 f_2 \cdots f_\ell$, each factor f_i of length *at most* k, so that factors are pairwise different $f_i \neq f_j$ for $i \neq j$. Among the results, they proved that this problem is NP-complete for $k = 2$ and unbounded alphabet. We observe that our notion of k-anticover requires the factors to be of length *exactly* k, and thus the problem of finding an anticover is different from Equality-Free String Partition problem. First, checking if a partition of factors of length k is equality-free can be trivially done in nearly linear time. Second, there are strings that admit a solution for one of the two problems, but not the other (e.g., *ababa* for $k = 3$ admits the equality-free partition $ab \cdot a \cdot ba$, but not an anticover).

2 Preliminaries

Let Σ be a finite ordered alphabet. A *string* is defined as a sequence of zero or more symbols from Σ. An *empty string* is a string of length 0, denoted by ε. A string x of length n is represented by the sequence $x = x_1 x_2 \cdots x_n$. We use the notation $x[i \ldots j]$ as a shorthand for $x_i x_{i+1} \cdots x_j$ and call it a *factor* or *substring* of x with length $j - i + 1$. We also say that a nonempty string s is a factor or substring of x with length $k \leq n$ if $s = x[i \ldots i + k - 1]$ for an integer $i \in [1, n - k + 1]$; in that case, s occurs in x at position i. The factor $x[1 \ldots j]$ is a *prefix* of x and the factor $x[j \ldots n]$ is a *suffix* of x.

▶ **Definition 1.** *Given an integer $k \geq 2$ and a string x of length $n \geq k$, let $S = \{i_1, i_2, \dots i_\ell\}$ be an ordered set of positions in x chosen from $\{1, 2, \dots, n-k+1\}$. We say that S is a* k-anticover *of x if*

(i) *any two factors $x[i_j \dots i_j + k - 1]$ and $x[i_h \dots i_h + k - 1]$ are different, for $j \neq h$, and*

(ii) *any position in x is covered, namely, $i_1 = 1$, $i_\ell = n - k + 1$, and $i_{j+1} - i_j \leq k$ for $1 \leq j \leq n - k$.*

▶ **Example 2.** For $x = abbbaaaaabab$ and $k = 3$, the ordered set $S = \{1, 3, 5, 9, 11\}$ denotes a 3-anticover of x: $\underline{ab}\overline{\underline{bb}}\underline{aaa}\overline{\underline{aa}}\underline{b}\overline{\underline{ab}}$. We remark that the indices $i_1 = 1$ and $i_\ell = n - k + 1$ must be part of any k-anticover, as they represent the only ways of covering the first and last symbol of x.

In this paper we consider the following problem.

k-AntiCover
Input: A string $x = x_1 x_2 \dots x_n$ and an integer $k \geq 2$, where $n \geq k$.
Output: Does a k-anticover S of x exist?

It is obvious that any k-length substring that only occurs *once* in x can be included "for free" in any k-anticover without risk of redundancy. We call *free factors* the corresponding factors, and remark that we can consider trivially covered the symbols that they span.

In the following, we identify $i_j \in S$ with its factor $x[i_j \dots i_j + k - 1]$, and sometimes we say that $x[i_j \dots i_j + k - 1]$ belongs to an anticover S, actually meaning that $i_j \in S$.

3 Hardness of k-Anticover for $k \geq 3$

In this section we show that solving k-AntiCover, namely, deciding whether a string x of length n has a k-anticover, is NP-complete for $k \geq 3$.

Firstly, observe that we can easily test in polynomial time whether S is a k-anticover, by checking that each pair of corresponding factors is distinct and, for each position $p \in \{1, \dots, n\}$, that S contains a factor that covers x (i.e., some $i_j \in \{p - k + 1, \dots, p\}$); thus k-AntiCover $\in$ NP.

We prove its completeness for $k = 3$ by a polynomial time reduction from 3-SAT to 3-AntiCover, i.e., given a 3-CNF boolean formula $\mathcal{F}$, we build a string $\mathcal{X}$ (in polynomial time) that admits a 3-anticover if and only if $\mathcal{F}$ is satisfiable. For $k > 3$, we remark that the techniques utilized could be adapted to reduce a k-SAT instance to k-AntiCover, although we omit this analysis for space reasons.

More in detail, we focus on a peculiar variant of 3-SAT, still NP-complete, where each literal in $\mathcal{F}$ is restricted to appear *at most* 3 times. This variant has been addressed in [22, Theorem 2.1], where it is shown that SAT "is NP-complete when restricted to instances with 2 or 3 variables per clause and at most 3 occurrences per variable". Hence 3-SAT with at most 3 occurrences per variable is NP-complete.[1]

In the following, let $C_1, \dots, C_l$ be the clauses of $\mathcal{F}$, and $v_1, \dots, v_m$ its variables. For a variable v_i, we refer to the 3 occurrences of its positive literal as v_i^1, v_i^2, and v_i^3, and the ones of its negative literal as $\neg v_i^1$, $\neg v_i^2$, and $\neg v_i^3$, meaning that each v_i^j (and each $\neg v_i^j$) appears at most once in $\mathcal{F}$.

[1] For the sake of completeness, we observe that 3-SAT with *exactly* 3 occurrences per variable is always satisfiable as consequence of [22, Theorem 2.4].

3.1 Overview of the reduction

We here introduce the structure and components of the reduction, which we then explain in detail.

The string $\mathcal{X}$ is divided in two parts:

- The first one models the clauses of $\mathcal{F}$, where literals correspond to specific factors, and using a factor in the cover of $\mathcal{X}$ means using the corresponding literal in $\mathcal{F}$. In essence, each clause contains three factors corresponding to the occurrences of its literals (where the three different occurrences of the same literal will correspond to different factors), and in order to cover all elements of a clause gadget we will need to use at least one of such factors.
- The second one contains *coherence gadgets*, which in essence enforce us to use factors in a way coherent with truth assignments (i.e., if factors i and j in the first part correspond to v_h and $\neg v_h$, then i and j cannot be used at the same time in the cover). Say we want to cover a clause using the factor corresponding to v_1^2: the coherence gadgets will force us to use the strings corresponding to $\neg v_1^1$, $\neg v_1^2$, and $\neg v_1^3$ to cover the *second* part of the string, meaning they cannot be used anymore in the first one (or they would break the non-redundancy constraint of the anticover).

We present $\mathcal{X}$ as a collection of smaller strings corresponding to gadgets, delimited by what we call *jolly characters*: these allow us to essentially ignore the order in which the pieces of the strings are re-combined and prevent any interaction between adjacent gadgets.

Jolly characters. To simplify the explanation, we use the jolly character "$\star$": in essence, each single $\star$ represents a unique character that does not appear anywhere else in the string, i.e., we can imagine that at the end of the reduction each $\star$ is then iteratively replaced with a unique distinct character not appearing in the string.

Jolly characters give us 2 useful properties for $k = 3$:

- All factors including $\star$ are free factors, so the $k-1$ symbols preceding and succeeding a $\star$ are trivially covered by free factors.
- In the string $A \star B$, then the k-length factors of A and B that are not free do not overlap: if $\star$ occurs at $\mathcal{X}[i]$, the right-most factor of A and left-most of B that could be non-free are, respectively, at positions $i-k+1$ and $i+1$.
- As a corollary, $A \star B$ and $B \star A$ have the same answer to k-AntiCover. More in general, if we have a collection of strings of the form $\star A \star$ (starting and ending in $\star$), and we want to append them to create a single string ($\mathcal{X}$), the order we chose does not impact the answer of k-AntiCover on the string.

3.2 The clauses part

We now detail the clause gadget of $\mathcal{X}$. Firstly, let p_j^i and n_j^i be symbols in Σ representing, respectively, the literals v_i^j and $\neg v_i^j$.

Let C_h be an arbitrary clause of $\mathcal{F}$, say, $(\neg v_1^3 \vee v_5^1 \vee v_7^1)$, corresponding to the third occurrence of the negative literal of v_1, and the first occurrences of the positive literals of v_5 and v_7. Then the corresponding gadget is

$$\mathcal{C}_h = \star \# \# hhn_1^3 \star \# \# hhp_5^1 \star \# \# hhp_7^1 \star$$

where

- n_1^3, p_5^1, and p_7^1 are the characters representing the corresponding occurrences of the literals, as described above.
- h is a character representing C_h (i.e., is different in every clause), whereas the character $\#$ is the same across all clauses, and $\star$ are jolly characters.

Now observe that the only characters we need to cover are the first h of each pair hh: Indeed the 2 symbols following and preceding each $\star$ can be trivially covered by free factors, as shown in the example below:

$$\underline{\star\#\#}h\overline{hn_1^3}\underline{\star\#\#}h\overline{hp_5^1}\underline{\star\#\#}h\overline{hp_7^1}\star$$

As $k = 3$, we have 3 possible factors we can use to cover each of these h symbols; in particular, the first with $\{\#\#h, \#hh, hhn_1^3\}$, the second with $\{\#\#h, \#hh, hhp_5^1\}$, and the third with $\{\#\#h, \#hh, hhp_7^1\}$.

As h is clause-specific, the two strings $\#\#h$ and $\#hh$ in each set only appear here, and no constraint is imposed on their usage. However, as the anticover can contain each string at most once, it will have to include at least one string among hhn_1^3, hhp_5^1, and hhp_7^1.

In essence, adding this occurrence of hhp_7^1 to the anticover will correspond to assigning "true" to v_7.

The first part of $\mathcal{X}$ will thus correspond to the gadgets corresponding to all clauses $C_1, \ldots, C_l$ appended after each other in sequence (as discussed above, the order is irrelevant).

In the second part of $\mathcal{X}$, we will then enforce coherence of assignments, i.e., using hhp_7^1 to cover $\mathcal{C}_h$ must forbid us from using the factors corresponding to the literal $\neg v_7$ in the rest of the clause part.

3.3 Auxiliary gadgets

In order to explain the coherence part, we first detail the auxiliary gadgets that compose it.

Gadget `forbid`(abc). Suppose we want to make sure that some string of length 3, say, abc, cannot be used to cover rest of the string. Then we can place the following gadget in $\mathcal{X}$:

$$\texttt{forbid}(abc) = \star\$abc\$\star\,\$abc\$\star\,\$abc\$\star$$

where again the $\star$ are jolly characters, but \$ is a gadget-specific character, i.e., each occurrence of a forbidding gadget has a unique character in place of the \$.

Similarly to above, we can observe that all characters are covered by free factors except the b characters in the middle, which can be covered by the strings $\$ab$, abc, $bc\$$: as we need to cover 3 characters, we must use all three of these strings. In turn, this means the string abc can *not* be used anywhere else in $\mathcal{X}$. We refer to this gadget as `forbid`(abc).

An important observation is that all factors used except for abc are either free factors, or contain the character \$, meaning those strings will not appear anywhere else and thus not affect our choices while covering the rest of the string.

Gadget `one-of`(abc, def). Suppose now we have two strings abc and def, and we want to make sure that at most one of the two may be used in the cover of the rest of the string. Then we can place the following gadget in $\mathcal{X}$:

$$\texttt{one-of}(abc, def) = \texttt{forbid}(c€d) \star bc€de \star abc€def\star$$

Here too, € is a gadget-specific character which is used only in this specific instance of this gadget (same as the $ above, we differentiate so as to avoid confusion with the $\texttt{forbid}(c€d)$ gadget).

Let's analyze it from left to right. Firstly, we forbid the string $c€d$ so it cannot be used in the rest of the string. Then, to cover the € in the central part $...\star bc€de\star...$, we must use either $bc€$ or $€de$. Finally, in the right part $...\star abc€def\star$ we need to cover the three symbols $...c€d...$: if $bc€$ was used in the central part, we cannot use it now, and since we cannot use $c€d$ either, to cover the c symbol we must use abc (while we can use $€de$ to cover the remaining two symbols). Symmetrically, if we used $€de$ in the central part of the gadget instead, we must use def to cover the right part.

It follows that to cover $\texttt{one-of}(abc, def)$ we must use either abc or def, meaning we can only use one of them in the rest of the string.

As above, all other factors used are either free or include €, so they do not impact the rest of the string.

Gadget $\texttt{amplifier}(abc, def)$. Finally, the amplifier gadget is the core of our coherence enforcement. It corresponds to the following string:

$$\texttt{amplifier}(abc, def) = \texttt{forbid}(cde)\star abcdef\star$$

Differently from the gadgets above, this one does affect strings other than the input ones: we call bcd the *trigger* of the amplifier. Specifically this is the word made from the last two characters of the first string abc, and the first of the second string def. Furthermore, note how the string cde made of the third character of the first string, and the first two of the second, becomes forbidden.[2]

Focus now on the right part: in $...\star abcdef\star$ only the symbols $...cd...$ are not covered by free factors. Since cde is forbidden, to cover them we have two ways:

- use the trigger string bcd.
- use both abc and def.

As the name suggests, this gadget *amplifies* the effects of using the trigger cde elsewhere in $\mathcal{X}$, as it will then force us to use both abc and def to cover $\texttt{amplifier}(abc, def)$. Note that this gadget does create and affect factors that are not free and may occur somewhere else, so we will analyze its usage carefully.

3.4 The coherence part

Let v_i be a variable of $\mathcal{F}$. In the clauses part, its literals can appear in up to six clauses. Let w.l.o.g. the symbols $1 \dots 6$ represent the identifiers of these clauses: the factors representing the literals will thus be $\{11p_i^1, 22p_i^2, 33p_i^3\}$ for the positive ones, and $\{44n_i^1, 55n_i^2, 66n_i^3\}$ for the negative.[3]

As explained above, say that the gadget of clause C_h contains an occurrence of the factor $22p_i^2$: we want to say that using this occurrence of the factor in the anticover means that v_i is set to true (thus C_h is satisfied by v_i). In order to enforce coherence, and make the anticover correspond to a satisfying assignment, we must then make it impossible to use a factor corresponding to a negative value of v_i anywhere in the clauses part.

---

[2] Note that $\texttt{amplifier}(def, abc)$ is a different gadget: it will forbid fab and its trigger will be efa.

[3] If a literal, say p_i^j, does not appear in a clause of $\mathcal{F}$, let it be represented by $\star\star p_i^j$.

More formally, we want to create a gadget for a variable v_i which, to be covered, forces us to use either all of $\{11p_i^1, 22p_i^2, 33p_i^3\}$, or all of $\{44n_i^1, 55n_i^2, 66n_i^3\}$ (this way, one set of strings is fully "burned" to cover this gadget, and only elements from the other set may be used in the rest of the string).

Gadget `enforce`(v_i). We do so by using the `one-of` gadget and a nested use of the `amplifier` gadget, with the following gadget made of 5 parts, which we call `enforce`(v_i):

1. `amplifier`$(11p_i^1, 2p_i^2 3)$
2. `amplifier`$(22p_i^2, 33p_i^3)$
3. `amplifier`$(44n_i^1, 5n_i^2 6)$
4. `amplifier`$(55n_i^2, 66n_i^3)$
5. `one-of`$(1p_i^1 2, 4n_i^1 5)$

Now, key observations are that $2p_i^2 3$ in gadget 1 is the trigger of gadget 2, while $5n_i^2 6$ in gadget 3 is the trigger of gadget 4, and finally, the arguments of gadget 5 are the triggers of gadgets 1 and 3.

In order to cover `one-of`$(1p_i^1 2, 4n_i^1 5)$ we must use (at least) one between $1p_i^1 2$ and $4n_i^1 5$. If we choose $1p_i^1 2$ to cover gadget 5, this *triggers* gadget 1, so to cover gadget 1 we must use both $11p_i^1$ and $2p_i^2 3$; in turn, this triggers gadget 2, forcing us to use both $22p_i^2$ and $33p_i^3$; on the other hand, gadgets 3 and 4 can be covered using their respective triggers, meaning that *all* strings corresponding to positive literals $\{11p_i^1, 22p_i^2, 33p_i^3\}$ are used by the cover, but it is not necessary to use any of the negative ones $\{44n_i^1, 55n_i^2, 66n_i^3\}$, which can be used in the clauses part. If, instead, we cover gadget 5 using $4n_i^1 5$, the situation is exactly symmetrical: we burn all the negative literals $\{44n_i^1, 55n_i^2, 66n_i^3\}$ on gadgets 3 and 4, but we may cover 1 and 3 using the triggers, and using the positive literals $\{11p_i^1, 22p_i^2, 33p_i^3\}$ in the clauses part.[4] We have thus proven the following result.[5]

▶ **Theorem 3.** *$\mathcal{F}$ is satisfiable if an only if $\mathcal{X}$ has a 3-anticover. As a consequence, problem k-AntiCover is* NP*-complete for $k \geq 3$.*

4 Polynomial-Time Algorithm for $k = 2$

In this section, we show that that 2-AntiCover can be reduced to 2-SAT in $O(n|\Sigma|)$ time and space, obtaining the following result.

▶ **Theorem 4.** *Problem 2-AntiCover can be solved in $O(n|\Sigma|)$ time and space.*

Proof. We run first a preliminary test to see if the input string x contains a factor of length 3 that occurs three or more times in it. A 2-anticover cannot exist a 2-AntiCover in this case, and we answer no. Indeed, let abc be a a factor that occurs three or more times in x. Since the position corresponding to b can be covered either by the factors of length 2, ab or bc, when we find the third occurrence of abc, we cannot use ab or bc as it would be chosen twice. Hence, there is no way to cover the position of b in the third occurrence of abc in any 2-AntiCover. Running this test can be easily done in $O(n \log |\Sigma|)$ time [12].

[4] For completeness, we remark that it is crucial to use `amplifier`$(11p_i^1, 2p_i^2 3)$ instead of `amplifier`$(2p_i^2 3, 11p_i^1)$: the latter one forbids the string 311, which does not contain $\star$ nor characters representing the literal and might affect other coherence gadgets. Instead `amplifier`$(11p_i^1, 2p_i^2 3)$ forbids $p_i^1 2p_i^2$, which is safe as it may not appear in other coherence gadgets.

[5] Again, we remark that all the gadgets in this reduction can be extended to any k rather than just 3, although we omit this for space reasons.

Instead, if this preliminary test is positive, we build a 2-SAT formula as follows. Let i be any position in the string x, and p_i the Boolean variable denoting whether or not the factor $x[i \dots i+1]$ is chosen in the 2-AntiCover.

We have a first group of clauses C_i, for $1 \leq i \leq n$, where C_i says that the first ($i = 1$) and last ($i = n$) positions must be covered by the only possible factors $x[1 \dots 2]$ and $x[n-1 \dots n]$, respectively, and any other position must be covered by the factor(s) of length $k = 2$ starting at position $i-1$ or i:

$$C_1 = (p_1) \tag{1}$$
$$C_n = (p_{n-1}) \tag{2}$$
$$C_i = (p_{i-1} \vee p_i) \qquad 2 \leq i \leq n-1 \tag{3}$$

Furthermore, when $x[i \dots i+1] = x[j \dots j+1]$ for $i \neq j$, we should take at most one of them, and so we cannot take both, giving the second group B_{ij} of clauses:

$$B_{ij} = (\neg p_i \vee \neg p_j) \qquad 1 \leq i < j \leq n \text{ such that } x[i \dots i+1] = x[j \dots j+1] \tag{4}$$

Let $\mathcal{F}$ be the 2-SAT formula obtained by putting the clauses C_i and B_{ij} in logical $\wedge$. We observe that $\mathcal{F}$ contains n clauses C_i and $O(n|\Sigma|)$ clauses B_{ij}. Recall that each factor of length 3 can occur at most twice in x. Thus, given any position i, we claim that there are at most $2|\Sigma|+1$ positions $j \neq i$ such that $x[i \dots i+1] = x[j \dots j+1]$. Indeed, any other occurrence of $s = x[i \dots i+1]$ is followed by a third symbol, say, c (unless that occurrence is a suffix of x). But $s\,c$ is a factor of length 3, and can appear at most twice. Since we have at most $|\Sigma|$ choices for c plus the end of string case, this gives the desired upper bound. As there are at most n positions i, and for each of them there are at most $2|\Sigma|+1$ positions j where the same factor of length 2 occurs, we conclude that there are $O(n|\Sigma|)$ clauses B_{ij}. Summing up, $\mathcal{F}$ has $O(n|\Sigma|)$ size and it can be built $O(n|\Sigma|)$ time.

It is straightforward to see that $\mathcal{F}$ is satisfied if and only if there is a 2-AntiCover for string x. Since 2-SAT can be solved in linear time in the size of the formula $\mathcal{F}$ [5], we obtain the bounds stated in the theorem. ◀

5 Exact Exponential-Time algorithms for $k \geq 3$

In this section we consider a better algorithm than a brute-force algorithm for solving k-AntiCover. The task of k-AntiCover is finding a subset of positions satisfying the given constraint. By trying all subsets of positions, we can solve k-AntiCover. Since the number of subset of positions is $O(2^{n-k})$, the brute-force algorithm runs in $O^*(2^{n-k})$ time, where the $O^*(\cdot)$ notation ignores $poly(n)$ factors. Note that $|\Sigma|$ and k is bounded by n. Thus, $O^*(\cdot)$ notation ignores $poly(|\Sigma|)$ and $poly(k)$ factors. In this section, we give two exponential time algorithms. The former algorithm breaks the trivial 2^{n-k}-barrier for any $k \geq 3$. The latter algorithm is clearly better than the brute-force algorithm and, in addition, it outperforms the former algorithm when $k > 9$.

5.1 Breaking the trivial barrier

Let x be a string with length n and k be an integer. We consider a set of positions of x from 1 to $n-k+1$. We partition this set as follows: Let $\mathcal{S} = \{S_1, \dots, S_\ell\}$ be a partition of positions $1, 2, \dots, n-k+1$. For any two substring y, y' with length k starting from position j and j' respectively, both j and j' are in S_i if and only if $y = y'$. That is, each partition corresponds to some substring with length k in x. For example, given a string *abcabca* and $k = 3$, $\mathcal{S} = \{\{1, 4\}, \{2, 5\}, \{3\}\}$.

We describe our proposed algorithm. Let $S_1 \in \mathcal{S}$ be the set containing position 1. We first pick position 1 to cover the 1-st character on x. Hence, after choosing 1, we need to solve at most $|S_1|$ subproblems. Note that for each subproblem, the first k letters are already covered, and thus, we have k options for covering the $(k+1)$th letter in each subproblem.

Since we cannot pick positions which already are picked, the time complexity $T(\cdot)$ of this algorithm satisfies the following inequality: $T(n-k+1) \leq cT(n-k+1-c)$, where c is the size of partition from which we pick a substring. Since the sum of the size of the partitions is $n-k$, the time complexity is $O^*(c^{\frac{n-k+1}{c}})$.

It is known that this formula takes its maximum when $c = 3$ [11]. Hence, the time complexity of this algorithm is $O^*(3^{\frac{n-k+1}{3}}) = O^*(3^{\frac{n-k}{3}})$ time.

▶ **Theorem 5.** *k-AntiCover can be solved in $O^*(3^{\frac{n-k}{3}})$ time and polynomial space.*

5.2 A better upper bound for large k

In this subsection, we give a faster algorithm when k is large. Now we first introduce some terminologies. A set $S = \{s_1, \ldots, s_\ell\}$ is a *k-cover* if $\bigcup_{i=1,\ldots,\ell} \{s_i, \ldots, s_i + k\} = \{1, \ldots, n\}$. Hence, a trivial k-cover is $\{1, 1+k, 1+2k, \ldots\}$. Note that each s_i corresponds to a position of x but a k-cover may have two positions which derives from the same substring. A k-cover S is *minimal* if there is no subset of S which is a k-cover. We say that s_i is *redundant* in S if $S \setminus \{s_i\}$ is also a k-cover.

▶ **Lemma 6.** *Let x be a string and k be an integer. Then, if x has a k-anticover, then there is a minimal k-cover S such that S is also a k-anticover.*

Proof. Let $S = \{s_1, \ldots, s_\ell\}$ be a k-anticover. Since S is a k-anticover, $S \setminus \{s_i\}$ is also k-anticover if $S \setminus \{s_i\}$ is a k-cover of x. Hence, S becomes a minimal cover by removing redundant elements one by one. Hence, the statement holds. ◀

From Lemma 6, we can determine whether x has a k-anticover by enumerating all minimal k-covers. Hence, in the following, we propose an enumeration algorithm for all minimal k-covers.

We firstly give an upper bound of the number of all minimal k-covers of substrings with length $k+1$ such that each minimal k-cover has no redundant positions. Assume that by concatenating these $\lceil n/(k+1) \rceil$ substrings, we can reconstruct the input string. Let us consider the following problem Cover(x, k): given a string x of length $k+1$, the task is to enumerate all minimal k-covers in it under the assumption, for $0 \leq i \leq k-1$, that we can select the length of a first interval s_1 between 1 to k and we can pick the last $k-1$ characters. For example, we consider an instance $x = abcd$ and $k = 3$. The subproblem Cover$(x, 3)$ has the following six solutions: $\underline{a}\overline{bcd}$, $\underline{ab}\overline{cd}$, $\underline{abc}\overline{d}$, $\underline{a}\overline{bc}\underline{d}$, $\underline{ab}\overline{c}\underline{d}$, and $\underline{abc}\overline{d}$. The next lemma shows the upper bound:

▶ **Lemma 7.** *Problem Cover(x, k) has at most $\frac{k(k+1)}{2}$ minimal k-covers.*

Proof. Let i be the length of a first interval. Since we cover all characters, we have to choose a position 1. In addition, we pick the second position between 2 and $i+1$. Since the length of x is $k+1$, then it is a minimal k-cover. Hence, we have i choices for each i. Therefore, we have $\sum_{1 \leq i \leq k} i = \frac{k(k+1)}{2}$ solutions and the statement holds. ◀

From the above lemma, the number of solutions in each subproblem is at most $\frac{k(k+1)}{2}$. In addition, the number of subproblems is $\frac{n}{k+1} + 1$. Now, we can obtain all minimal k-covers which have no redundant positions between $c(k+1)+1$ to $(c+1)(k+1)$ for any non-negative integer c by trying all the combinations of concatenating solutions of all the subproblems. Because any k-anticover has no redundant positions, the following theorem holds.

Table 1 The list of values of $\left(\frac{k(k+1)}{2}\right)^{\frac{1}{k+1}}$. We round the base of the exponent up to the fourth digit after the decimal point. Note that $3^{\frac{1}{3}}$ is approximately equal to 1.4423.

k	3	5	9	10	20	30
$\left(\frac{k(k+1)}{2}\right)^{\frac{1}{k+1}}$	1.5651	1.5705	1.4633	1.4396	1.2900	1.2192

▶ **Theorem 8.** *There is an algorithm solving k-AntiCover in $O^*\left(\left(\frac{k(k+1)}{2}\right)^{\frac{n}{k+1}}\right)$ time and polynomial space.*

From Theorem 8 and Table 1, the latter algorithm is better than the former algorithm if k is larger than 9. Combining two theorems, we obtain the following theorem.

▶ **Theorem 9.** *Problem k-AntiCover can be solved in $O^*\left(\min\left\{3^{\frac{n-k}{3}}, \left(\frac{k(k+1)}{2}\right)^{\frac{n}{k+1}}\right\}\right)$ time, using polynomial space.*

6 Concluding remarks

In this paper we proposed the k-AntiCover problem, a natural combinatorial problem on strings with applications to fields such as computational biology. We have shown that finding whether a string of length n can be covered by (possibly overlapping) distinct factors of length k is polynomial for $k = 2$, and NP-complete otherwise.

We have also shown how to design exact exponential algorithms for general k, which improve upon a trivial brute-force approach and get progressively more efficient for larger values of k.

A question that remains open is whether the proposed algorithms match the inherent computational complexity of the problem or whether faster solutions exist. Another is whether the problem remains NP-complete under natural restrictions, such as an alphabet of constant size.

References

1 Hayam Alamro, Golnaz Badkobeh, Djamal Belazzougui, Costas S Iliopoulos, and Simon J Puglisi. Computing the antiperiod (s) of a string. In *30th Annual Symposium on Combinatorial Pattern Matching (CPM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

2 Mai Alzamel, Alessio Conte, Daniele Greco, Veronica Guerrini, Costas Iliopoulos, Nadia Pisanti, Nicola Prezza, Giulia Punzi, and Giovanna Rosone. Online algorithms on antipowers and antiperiods. In Nieves R. Brisaboa and Simon J. Puglisi, editors, *String Processing and Information Retrieval*, pages 175–188, Cham, 2019. Springer International Publishing.

3 Alberto Apostolico and Andrzej Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theoretical Computer Science*, 119(2):247–265, 1993.

4 Alberto Apostolico, Martin Farach, and Costas S Iliopoulos. Optimal superprimitivity testing for strings. *Information Processing Letters*, 39(1):17–20, 1991.

5 Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Inf. Process. Lett.*, 8(3):121–123, 1979. doi:10.1016/0020-0190(79)90002-4.

6 Golnaz Badkobeh, Gabriele Fici, and Simon J Puglisi. Algorithms for anti-powers in strings. *Information Processing Letters*, 137:57–60, 2018.

7 Anne Condon, Ján Maňuch, and Chris Thachuk. The complexity of string partitioning. *J. Discrete Algorithms*, 32:24–43, 2015. doi:10.1016/j.jda.2014.11.002.

8 Alessio Conte, Roberto Grossi, and Andrea Marino. Large-scale clique cover of real-world networks. *Information and Computation*, 270:104464, 2020. Special Issue on 26th London Stringology Days & London Algorithmic Workshop (LSD & LAW). doi:10.1016/j.ic.2019.104464.

9 Maxime Crochemore and Wojciech Rytter. *Jewels of stringology: text algorithms*. World Scientific, 2002.

10 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.

11 Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Springer-Verlag, Berlin, Heidelberg, 1st edition, 2010.

12 Dan Gusfield. Algorithms on strings, trees, and sequences. 1997. *Computer Science and Computational Biology. New York: Cambridge University Press*, 1997.

13 Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear time algorithm for seeds computation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete algorithms*, pages 1095–1112. SIAM, 2012.

14 Roman Kolpakov, Ghizlane Bana, and Gregory Kucherov. mreps: efficient and flexible detection of tandem repeats in DNA. *Nucleic acids research*, 31(13):3672–3678, 2003.

15 Yin Li and William F Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.

16 M Lothaire. *Applied combinatorics on words*, volume 105. Cambridge University Press, 2005.

17 Monsieur Lothaire and M Lothaire. *Algebraic combinatorics on words*, volume 90. Cambridge university press, 2002.

18 Radu Stefan Mincu and Alexandru Popa. The maximum equality-free string factorization problem: Gaps vs. no gaps. In Alexander Chatzigeorgiou, Riccardo Dondi, Herodotos Herodotou, Christos Kapoutsis, Yannis Manolopoulos, George A. Papadopoulos, and Florian Sikora, editors, *SOFSEM 2020: Theory and Practice of Computer Science*, pages 531–543, Cham, 2020. Springer International Publishing.

19 Dennis Moore and W. F. Smyth. Computing the covers of a string in linear time. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '94, page 511–515, USA, 1994. Society for Industrial and Applied Mathematics.

20 Dennis Moore and William F Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994.

21 Robert Z Norman and Michael O Rabin. An algorithm for a minimum cover of a graph. *Proceedings of the American Mathematical Society*, 10(2):315–319, 1959.

22 Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85–89, 1984. doi:10.1016/0166-218X(84)90081-7.