String Matching with Involutions

Cristian Grozea[1], Florin Manea[2], Mike Müller[2], Dirk Nowotka[2]

[1] Fraunhofer Institute FIRST,
D-12489, Berlin
cristian.grozea@first.fraunhofer.de
[2] Christian-Albrechts-Universität zu Kiel, Institut für Informatik,
D-24098 Kiel, Germany.
{flm,mimu,dn}@informatik.uni-kiel.de

Abstract. In this note we propose a novel algorithm for locating in a text T every occurrence of a strings that can be obtained from a given pattern P by successively applying antimorphic involutions on some of its factors. When the factors on which these involutions are applied overlap, a linear time algorithm is obtained. When we apply the involutions to non-overlapping factors we obtain an algorithm running in $\mathcal{O}(|T||P|)$ time and $\mathcal{O}(|P|)$ space, in the worst case. We also improve the latter algorithm to achieve linear average running time, when the alphabet of the pattern is large enough.

1 Introduction

String matching is one of the most basic and well studied algorithmic problems: given a text T we are interested in finding all the occurrences of a pattern P as a factor of T. Besides many applications in text processing, data compression, cryptography, this problem has gained even more attention in the context of computational molecular biology. Starting from the fact that biological data can be represented as sequences of letters from a fixed alphabet, word processing algorithms, and especially string matching problems, are seen as central in the intersection between computer science and biology (see [1], and the references therein, for a presenation of the essential string-algorithmic problems related to biology).

A class of more general problems was derived from the aforementioned problem, namely the approximate string matching problems. In such a problem one has to find all the factors of the text T that can be obtained from the pattern P by a series of simple operations (again, [1] describes several basic approximate matching problems). Although in most of the classical cases these operations are local (for instance, string-editing operations like insertion, deletion, or substitution of letters), papers like [2–4] discuss also the case of (bio-inspired) operations that can affect the pattern on a larger scale, e.g., rotations of factors.

Starting from the two observations that a word over the DNA-alphabet $\{A, C, G, T\}$ encodes basically the same information as its Watson-Crick complementary and that the Watson-Crick complementarity relation can be formalized

as an antimorphic involution on words, we address two unconventional generalizations of the string matching problem, that may find applications in computational biology. More precisely, we want to locate in a text T all the occurrences of words that can be obtained from a given pattern P by applying successively an antimorphic involution f on some of its factors. In the first problem, we consider the case when the factors on which the involution f is applied may overlap; the operation that transforms a string into the other by applying f on one of its factors is called f-rotation and denoted $\Rightarrow_f$. In the second case, we consider the case when the factors on which the involution f is applied do not overlap. Formally, we present solutions to the following two problems:

Problem 1. Given two words T and P over V, and an antimorphic involution $f : V^* \to V^*$, identify all the factors P' of T such that $P \Rightarrow_f^* P'$.

Problem 2. Given two words T and P over V, and an antimorphic involution $f : V^* \to V^*$, identify all the factors P' of T that are obtained by non-overlapping f-rotations from P.

In the case of Problem 1, we first show that given two words P and P', both of length m, and an antimorphic involution $f : V^* \to V^*$, we can decide in $\mathcal{O}(m)$ time and $\mathcal{O}(|V|)$ space whether $P \Rightarrow_f^* P'$. The solution relies on the key remark that $P \Rightarrow_f^* P'$ if and only if $P' = f^{i_1}(a_1) \dots f^{i_m}(a_m)$, where $i_j \in \{0, 1\}$ and $a_1 \dots a_m$ is a word obtained by permuting the letters of P. By this simple observation, we can use a variation of the classical counting filter (see, for instance, [5]) to decide the aforementioned problem. Moreover this idea can be easily adapted to get a linear time and space $\mathcal{O}(|V)|)$ solution to Problem 1.

As an initial step in solving Problem 2, we show that given two words P and P', both of length m, and an antimorphic involution $f : V^* \to V^*$, we can decide in $\mathcal{O}(m)$ time and space whether P' can be obtained from P by non-overlapping f-rotations. This time, the solution is based on a greedy strategy. Building on this solution, we solve Problem 2 in a straight-forward, yet efficient, way: we test all the factors of length m of T to see whether they can be obtained by non-overlapping f-rotations from P. This yields a $\mathcal{O}(nm)$ time and $\mathcal{O}(m)$ space solution for Problem 2, where $n = |T|$ and $m = |P|$. However, when we first use the solution of Problem 1 to first detect the factors of T that may be obtained by f-rotations from P, and then analyse only these factors to see whether they are obtained by non-overlapping f-rotations from P, we get a solution of Problem 2 working in $\mathcal{O}(n)$ average time, when the size of the alphabet V is subject to a series of simple and natural constraints.

In order to identify the major contributions of this paper, we briefly recall the state of the art results regarding several closely related problems. To begin with, Problem 2 was considered so far only for f being the rotation (see [2], and the references therein). The paper [2] also provides the most efficient solution for this particular case of Problem 2, that we are aware of; it works in $\mathcal{O}(nm)$ time in the worst case, and has $\mathcal{O}(m^2)$ space complexity. In [3, 4] a more general problem was considered: one searches the factors of T that can be obtained

by non-overlapping rotations and translocations[3] (an operation that transforms a word xy into yx) from P; the respective solutions of this problem can be easily adapted to solve Problem 2 for $f = (\cdot)^R$. The solution of [3] is the better one: it has $\mathcal{O}(nm^2)$ time complexity, $\mathcal{O}(m)$ space complexity, and $\mathcal{O}(n)$ average time complexity; however, the average running time is obtained under a series of artificial restrictions on the alphabets of P and T. Finally, the design of an algorithm solving Problem 2 for rotations in linear average time, while preserving the $\mathcal{O}(nm)$ worst case time complexity and the $\mathcal{O}(m^2)$ space complexity, was still under investigation [2].

With respect to the above, our contributions are the following:

- We solve Problems 1 and 2 in a general setting, namely when f is an arbitrary antimorphic involution. Note, however, that our generalization allows the usage of a single general operation that can be applied on factors (not just the simple rotations), instead of several particular operations, like in [3, 4].
- Our solutions to Problems 1 and 2 can be seen as on-line algorithms, as they model a scenario where the letters of T are read one by one, and only the last m letters of T are memorized and processed at each step. Nevertheless, the solution of Problem 2 is based on independently analysing all the factors of length m of the text T. Therefore, it can be easily implemented as a parallel algorithm.
- Our solution of Problem 2 is based on novel strategy, completely different from the ones used in other related papers (that were based mainly on dynamic programming, see, for instance, [2, 3] and the references therein). The greedy solution we propose seems conceptually simpler to us.
- Our solution to Problem 2 matches the time complexity $\mathcal{O}(nm)$ of the most time-efficient solution obtained in the particular case of rotations [2], and the space complexity $\mathcal{O}(m)$ of the most space-efficient solution obtained in the same particular case [3]. Therefore, when compared to any other solution of the problem, our algorithm is either faster and uses the same amount of space, or is as fast but uses less space.
- Our solution to Problem 2 is shown to run in linear average time, provided that some simple and natural restrictions on the size of the pattern's alphabet holds; therefore, we solve the problem left open in [2]. The statistical analysis we use to compute the average running time of our algorithm overcomes many of the aforementioned artificial restrictions used in the similar analysis of the algorithm proposed in [3], which was the most-efficient algorithm, with respect to the average running-time, solving Problem 2 for rotations.

To conclude this section, note that this paper is not aimed towards modelling an exact biological situation in an algorithmic setting. In fact, we propose and solve efficiently two algorithmic problems, loosely based on principles of molecular biology, with the hope that our approach might become useful in real-life applications, as well as in theoretical bio-inspired computational approaches.

[3] Although it was not explicitly defined, in paper [3] the translocations and rotations are performed on non-overlapping factors. Otherwise, the respective string matching problem can be easily solved in $\mathcal{O}(n)$ time and $\mathcal{O}(m)$ space using a counting filter.

2 Preliminaries

In this section we give several basic definitions that are used throughout the paper. For more details on words and combinatorics on words the reader is referred to [6], while [1] is a good reference for algorithmics on words.

Let V be a finite alphabet. The *length* of a word $w \in V^*$ is denoted by $|w|$; the number of occurrences of a letter $a \in V$ in w is denoted $|w|_a$ and the number of occurrences of the letters of an alphabet U in w is denoted by $|w|_U = \sum_{a \in U} |w|_a$. The *empty word* is denoted by ε. Also, $alph(w)$ denotes the set of all letters occurring in w. In the algorithmic problems we discuss, when given an input word w of length n we assume that $alph(w) \subseteq \{1, \ldots, n\}$, and, consequently, w is seen as a sequence of integers. This is a common assumption in algorithmics on words (see, e.g., the discussion in [7]).

A word u is a *factor* of a word v, if $v = xuy$, for some words x, y. We say that u is a *prefix* of v, if $x = \varepsilon$ and a *suffix* of v if $y = \varepsilon$. We denote by $w[i]$ the i^{th} symbol of w, so $w = w[1] \ldots w[n]$; we denote by $w[i..j]$ the factor of w starting at position i and ending at position j, consisting of the catenation of the symbols $w[i], \ldots, w[j]$, for $1 \leq i \leq j \leq n$. A function $f : V^* \to V^*$ is an antimorphism if $f(uv) = f(v)f(u)$, for any words u and v over V. Note that, when we want to define an antimorphism it is enough to give the definitions of $f(a)$, for all $a \in V$. An antimorphism $f : V^* \to V^*$ is an antimorphic involution when $f^2(a) = a$ for all $a \in V$. A distinguished antimorphic involution, playing an important role in this paper, is the rotation, defined as $(\cdot)^R : V^* \to V^*$ with $(a)^R = a$ for all $a \in V$. Clearly, for a word w of length n we have $(w)^R = w[n]w[n-1] \ldots w[1]$.

The Parikh vector (or mapping) of a word w over $V = \{1, \ldots, n\}$ is the array with n elements A_w where $A_w[i] = |w|_i$.

Let f be an antimorphic involution over V, and let $\ell = |\{(i, j) \mid i \leq j, f(i) = j\}|$. We may assume, without loss of generality, that for $i \leq \ell$ we have either $f(i) = i$ or $f(i) > \ell$ (for this to hold, a relabelling of the letters of V may be used). We define the f-Parikh vector of a word w as the array A_w^f with ℓ elements, such that $A_w^f[i] = |w|_i + |w|_{f(i)}$ if $i \neq f(i)$ and $A_w^f[i] = |w|_i$, otherwise. Basically, the f-Parikh vector of w can be seen as the Parikh vector of the word obtained from w by seeing i and $f(i)$ as the same symbol; in the case of $f = (\cdot)^R$, the f-Parikh vector of a word coincides with its classical Parikh vector.

Let $P, P' \in V^*$ be two words, such that $|P| = |P'|$. Let f be an antimorphic involution on V. We say that P' *is obtained by an f-rotation from* P if $P = xuy$ with $x, u, y \in V^*$ and $P' = xf(u)y$; we write this as $P \Rightarrow_f P'$. We denote by $\Rightarrow_f^*$ the transitive closure of $\Rightarrow_f$. Further, we say that P' *is obtained by non-overlapping f-rotations from* P if $P = x_1u_1 \ldots x_ku_kx_{k+1}$ and $P' = x_1f(u_1) \ldots x_kf(u_k)x_{k+1}$. Note that when f is the rotation $(\cdot)^R$ defined above, we just say rotation instead of $(\cdot)^R$-rotation.

Before describing the basic data structures that we use, let us recall that the computational model on which our algorithms run is the unit-cost RAM model with logarithmic word-size; for a basic presentation of this model, [8, Section 2.2] is a good reference (see also the Appendix).

The basic data structures that we need are the following. For a string w of length n, over an alphabet $V \subseteq \{1, \ldots, n\}$, we define a suffix-array data structure that contains two arrays Suf, a permutation of $\{1, \ldots, n\}$, and LCP with n elements from $\{0, 1, \ldots, n-1\}$. Basically, Suf is defined such that $w[Suf[i]..n]$ is the i^{th} suffix of w in the lexicographical order. The array LCP is defined by $LCP[1] = 1$ and $LCP[r]$ is the length of the longest common prefix of $w[Suf[r-1]..n]$ and $w[Suf[r]..n]$. These data structures are constructed in time $\mathcal{O}(n)$. For more details, see [7], and the references therein. Moreover, one can process the array LCP in linear time $\mathcal{O}(n)$ in order to return in constant time the answer to queries "What is the length of the longest common prefix of $w[i..n]$ and $w[j..n]$?", denoted $LCPref(i, j)$. The idea is to first compute the inverse permutation of Suf, i.e., an array S that associates to each i the value $S[i] = \ell$ if and only if $i = Suf[\ell]$. Further we compute in linear time a range minimum query data structure for the array LCP (see [1]), and this enables us to return in constant time the answer to queries "What is the minimum number from $LCP[i], \ldots, LCP[j]$?". Now, $LCPref(i, j)$ is obtained as the minimum from $LCP[i'+1], \ldots, LCP[j']$, where $i' = \min\{S[i], S[j]\}$ and $j' = \max\{S[i], S[j]\}$.

3 Solution of Problem 1

We begin with a simple lemma (whose proof can be found in the Appendix):

Lemma 1. *Let $P = a_1 \ldots a_m$ and let $P' = f^{i_1}(a_{\sigma(1)}) \ldots f^{i_m}(a_{\sigma(m)})$, where $i_j \in \{0, 1\}$ and σ is a permutation of $\{1, \ldots, m\}$. Then $P \Rightarrow_f^* P'$.* □

Now it is immediate how we can test, for two words P and P', whether $P \Rightarrow_f^* P'$. We compute the f-Parikh vectors of these two words, namely A_P^f and $A_{P'}^f$, and check whether they are identical or not. If yes, then $P \Rightarrow_f^* P'$; otherwise, P' cannot be obtained from P by f-rotations. Clearly, this test takes linear time.

Before stating the solution of Problem 1 we note that we can assume that V (the input alphabet) does not contain letters that do not occur in P or $f(P)$. Otherwise, we can just split T in words that fulfil this assumption, and solve the problem for all those shorter words instead of T. Therefore, we may assume that the f-Parikh vectors of T and P have at most $|alph(P) \cup alph(f(P))|$ elements. Note that the size of these vectors is $\ell = |\{(i, j) \mid i \leq j, f(i) = j\}|$ and recall that we work under the assumption that either $i = f(i)$ or $i \neq f(i) > \ell$.

Similar to [5], we read the word T letter by letter, from left to right, and compute the following:

- the variable *count* which equals, after the k^{th} letter of T was read,

$$\sum_{i=1}^{\ell} |A_w^f[i] - A_P^f[i]|, \text{ where } k \geq m \text{ and } w = T[k-m+1..k];$$

- the array D with ℓ elements, such that $D[i] = A_w^f[i] - A_P^f[i]$, where w is defined just as above.

Computing *count* and D for $k = m$ takes $\mathcal{O}(m)$ time, and they can be updated in constant time when the k^{th} letter of T is read, for $k > m$. Indeed, if the k^{th} letter of T is i or $f(i)$ and the $(k-m)^{th}$ letter of T was j or $f(j)$ we decrease $D[j]$ by 1 and increase $D[i]$ by 1; the value of *count* is updated accordingly. Clearly, $P \Rightarrow_f^* T[k-m+1..k]$ if and only if $count = 0$ after the k^{th} letter of T was read. Therefore, detecting every factor of T that can be obtained by f-rotations from P takes $\mathcal{O}(n)$ time and $\mathcal{O}(|V|)$ space, where $n = |T|$.

Proposition 1. *Problem 1 can be solved in $\mathcal{O}(|T|)$ time and $\mathcal{O}(|V|)$ space.*

4 Solution of Problem 2

As in the previous case, we first prove a lemma:

Lemma 2. *Let P and P' be two words over V, such that P' is obtained by non-overlapping f-rotations from P. Then there exists a unique factorization $P = x_1u_1 \dots x_ku_kx_{k+1}$ with $k \geq 0$, $x_i, u_i \in V^*$ for $i \in \{1, \dots, k\}$, such that:*

1. *$P' = x_1f(u_1) \dots x_kf(u_k)x_{k+1}$;*
2. *u_i and $f(u_i)$ begin with a different letter, for all $i \in \{1, \dots, k\}$;*
3. *if y_i is a non-trivial prefix of u_i then $f(y_i)$ is not a prefix of $f(u_i)$, for all $i \in \{1, \dots, k\}$.*

Proof. Let us assume that $P \neq P'$. Otherwise, the statement of the lemma holds canonically (for $k = 0$). Since P' can be obtained from P by non-overlapping f-rotations, we get that there is a factorization $P = \alpha_1\beta_1 \dots \alpha_p\beta_p\alpha_{p+1}$, such that $P' = \alpha_1f(\beta_1) \dots \alpha_pf(\beta_p)\alpha_{p+1}$. We may assume, without loss of generality, that $\beta_i \neq f(\beta_i)$ (otherwise, that f-rotation cannot be detected and is, in fact, not needed).

Let us choose a value i with $1 \leq i \leq p$. Assume that $x \in V^+$ is the longest common prefix of β_i and $f(\beta_i)$. Then, it follows that both β_i and $f(\beta_i)$ end with $y = f(x)$. Clearly, if $|x| \geq |\beta_i|/2$ then $\beta_i = f(\beta_i)$, due to the fact that the prefixes of length $|x|$ of β_i and $f(\beta_i)$ are equal and the suffixes of length $|x|$ of β_i and $f(\beta_i)$ are equal, as well. As x is the longest common prefix of β_i an $f(\beta_i)$ we get that $x = \beta_i = f(\beta_i)$, a contradiction. Thus, $|x| < |\beta_i|/2$ and it follows that $\beta_i = x\beta_i'y$ and $f(\beta_i) = xf(\beta_i')y$. Therefore, we can obtain a new factorization $P = \alpha_1\beta_1 \dots \alpha_ix\beta_i'y \dots \alpha_p\beta_p\alpha_{p+1}$ such that $P' = \alpha_1f(\beta_1) \dots \alpha_ixf(\beta_i')y \dots \alpha_pf(\beta_p)\alpha_{p+1}$.

We can repeat this procedure for different values of i (thus, at most p times), and finally obtain a factorization $P = \alpha_1'\beta_1' \dots \alpha_p'\beta_p'\alpha_{p+1}'$ such that $P' = \alpha_1'f(\beta_1') \dots \alpha_p'f(\beta_p')\alpha_{p+1}'$ and u_i and $f(u_i)$ begin (and end, as well) with a different letter, for all $i \in \{1, \dots, k\}$.

Again, choose a value i with $1 \leq i \leq p$. Assume that there exists a non-trivial prefix y of β_i' such that $f(y)$ is a prefix of $f(\beta_i')$. If $|y| > |\beta_i|/2$ then $f(\beta_i') = f(y)f(z)$, for some word $z \in V^+$ with $|z| < |\beta_i|/2$. Therefore, $\beta_i' = zy$; but we also have $\beta_i' = yx$ for some word x, with $|x| = |z|$. Consequently, $zy = yx$ and it follows that there exist words $u, v \in V^*$ and a natural number t such that

$z = uv$, $y = (uv)^t u$, and $x = vu$. We can always assume that $u \neq \varepsilon$. We get that u is also a non-trivial prefix of β'_i, such that $f(u)$ is a prefix of $f(\beta'_i)$ and, this time, $|u| < |\beta'_i|/2$.

So, let us now analyse the general case when there exists a non-trivial prefix y of β'_i such that $f(y)$ is a prefix of $f(\beta'_i)$ with $|y| \leq |\beta_i|/2$. Moreover, take y to be the shortest such prefix of β'_i. As $f(y)$ is a prefix of $f(\beta'_i)$ it follows that $\beta'_i = yxy$ for some $x \in V^*$. Therefore, $f(\beta'_i) = f(y)f(x)f(y)$. Now, we have a new factorization of $P = \alpha'_1\beta'_1 \dots \alpha'_i y\beta'_i y \dots \alpha'_p\beta'_p\alpha'_{p+1}$ such that $P' = \alpha'_1 f(\beta'_1) \dots \alpha'_i f(y)f(\beta'_i)f(y) \dots \alpha'_p f(\beta'_p)\alpha'_{p+1}$.

We can repeat this procedure finitely many times (at most $(\sum_{i=1}^{p} |\beta'_i|)/2$ times) and get, in the end, a factorization $P = x_1u_1 \dots x_k u_k x_{k+1}$, that fulfils the conditions stated in our lemma.

It remains to show that there is exactly one such factorization. Indeed, it is not hard to show, by induction, that in a factorization that fulfils the conditions of the lemma, $P = x'_1u'_1 \dots x'_q u'_q x'_{q+1}$, the following hold, for $i \leq q$, $\ell_i = \sum_{1 \leq j < i}(|x'_i| + |u'_i|)$ and $m = |P|$:

- x'_i is the longest common prefix of $P[\ell_i..m]$ and $P'[\ell_i..m]$
- u'_i is the shortest prefix of $P[\ell_i + |x'_i|..m]$ such that $f(u'_i)$ is a prefix of $P'[\ell_i + |x'_i|..m]$.

Moreover, x'_{q+1} is the longest common prefix of $P[\ell_q + |x_q| + |u_q|..m]$ and $P'[\ell_q + |x_q| + |u_q|..m]$

Thus, $x'_1 = x_1$, $u'_1 = u_1$, and, using induction once again, we get that $k = q$ and $x'_i = x_i$ and $u_i = u'_i$ for all $i \in \{1, \dots, k\}$. □

Now it remains to be seen how this Lemma helps us test whether a word P' can be obtained from P by non-overlapping f-rotations. The solution, depicted in Algorithm 4.1, is based on the construction of the unique factorization of P given by Lemma 2. As in the case of the previous problem, we can assume that V equals $alph(P) \cup alph(f(P))$; if P' contains other letters, then it cannot be obtained by f-rotations from P.

Algorithm 4.1 works as follows. In a preprocessing phase, it constructs the word $w = P'f(P)$ and data structures that permit us answering $LCPref$-queris in w in constant time. Then, it starts with $i = 1$, and tries to see whether $P'[i..m]$ can be obtained by non-overlapping f-rotations from $P[i..m]$. Following the ideas from the proof of Lemma 2, it first detects the longest common prefix of $P[i..m]$ and $P'[i..m]$, and computes the value ℓ such that $P[i..\ell - 1]$ is this longest common prefix. By Lemma 2, we should now test whether $P[\ell..m]$ can be obtained from $P'[\ell..m]$ by non-overlapping f-rotations. If $\ell > m$, the conclusion is trivial, so let us assume that $\ell \leq m$. Clearly, $P[\ell] \neq P'[\ell]$, and now the algorithm checks simultaneously the letters of the two words, trying to find the smallest $j \geq \ell$ such that the longest common prefix of $w[\ell..2m] = P'[\ell..m]f(P)$ and $w[2m - j + 1..2m] = f(P[j])f(P[j-1]) \dots f(P[1])$ is longer than or equal to $j - \ell + 1$. This condition means that $w[\ell..j]$ is a common prefix of $w[\ell..j]$ and $f(P[\ell..j])$, i.e., $w[\ell..j] = f(P[\ell..j])$. If no such j smaller than or equal to m exists, then P cannot be factorized as in Lemma 2, so P' cannot be obtained by

Algorithm 4.1 $Test(P, P', f)$: decides whether P' can be obtained from P by non-overlapping f-rotations

```
1: Compute data structures that permit us to answer in constant time LCPref queries
   for the string w = P'f(P);
2: Let m = |P| and i = 1;
3: while i ≤ m do
4:    Let ℓ = i and found = false;
5:    while P[ℓ] = P'[ℓ] and ℓ ≤ m do
6:       Let ℓ = ℓ + 1;
7:    end while
8:    Set j = ℓ + 1 and ℓ = ℓ + 1;
9:    if l > m then
10:      Halt the algorithm and return "Yes".
11:   end if
12:   while found = false and j ≤ m do
13:      if LCPref(w[ℓ..2m], w[2m − j + 1..2m]) ≥ j − ℓ + 1 then
14:         Set found = true and i = j + 1;
15:      else
16:         Set j = j + 1;
17:      end if
18:   end while
19:   if j > m then
20:      Halt the algorithm and return "No".
21:   end if
22: end while
23: Return "Yes".
```

non-overlapping f-rotations from P. Otherwise, if a j was found, the procedure described above is restarted for $i = j + 1$. If all the letters of P and P' were checked, and the algorithm did not return "No", then P admits a factorization like the one we searched for, so the answer is "Yes".

The overall time complexity of the algorithm is $\mathcal{O}(m)$ as the needed data structures can be computed in linear time and, further, for each letter of P a constant number of operations are done. The space complexity is $\mathcal{O}(m)$, as well (see the Appendix for a more detailed analysis of the complexity).

We can directly use this algorithm to solve Problem 2. We just have to test every factor $T[i..i + m - 1]$ of length m of the word T to see whether it can be obtained by non-overlapping f-rotations from P or not. This takes $\mathcal{O}(nm)$ time, and $\mathcal{O}(m)$ space, as the space needed to perform the tests can be reused.

Proposition 2. *Problem 2 can be solved in $\mathcal{O}(|T||P|)$ time and $\mathcal{O}(|P|)$ space.*

5 An average-time efficient solution of Problem 2

A basic tool in our analysis is the Γ function, defined by Euler (see, for instance, [9]) as an extension of the factorial function to complex and real number

arguments. In the case of complex numbers with a positive real part, we have

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt.$$

In the following we will only see Γ as a function defined on positive real numbers, greater or equal to 1. Note first that $\Gamma(n) = (n-1)!$ for n a positive natural number. Also, the Γ function has the property that it can be asymptotically approximated using the Stirling formula. That is, there exist two positive constants K_1 and K_2 such that:

$$K_1 \left(\frac{r}{e}\right)^r \sqrt{2\pi r} \leq \Gamma(r+1) \leq K_2 \left(\frac{r}{e}\right)^r \sqrt{2\pi r}, \text{ for } r \text{ large enough.}$$

Moreover, when defined on positive values, Γ is logarithmically convex (that is, $\log(\Gamma)$ is a convex function; see also the Bohr-Mollerup Theorem presented in [9]). Finally, for simplicity, Gauss used the (more natural) notation $\Pi(r) = \Gamma(r+1)$; note that, with the new notation, $\Pi(n) = n!$ for any natural number n. Of course, Π remains logarithmically convex.

In order to obtain an algorithm efficient in average, we need to split the discussion in several cases, depending on the antimorphic involution f. In all these cases we work under the assumption that the letters of the alphabet $P \cup f(P)$ occur with equal probability.

The case of rotations: We begin with this case because it was already discussed in previous papers [2, 3], and we can compare our results to the ones obtained in the aforementioned papers. This case occurs when $f(a) = a$ for all $a \in V$; in fact, $f = (\cdot)^R$.

There is one simple idea that one can use in order to obtain a faster solution of Problem 2, without using extra space. We note that a factor $T[i..i+m-1]$ of T can be obtained from P by non-overlapping rotations only if $P \Rightarrow^*_{(\cdot)^R} T[i..i+m-1]$. That is, $T[i..i+m-1]$ of T can be obtained from P by non-overlapping rotations only if $T[i..i+m-1]$ and P have the same Parikh vector. So, we first identify the factors of T that can be obtained from P by rotations, and then check which of them can be actually obtained from P by non-overlapping f-rotations. Basically, we use the solution of Problem 1 to identify the positions k of the word T such that $T[k-m+1..k]$ can be obtained from P by successive rotations and then check, using the solution of Problem 2 just presented, whether $T[k-m+1..k]$ can be obtained from P by non-overlapping rotations.

The following analysis shows that this strategy leads to a solution of Problem 2 that has $\mathcal{O}(n)$ average running time and still $\mathcal{O}(m)$ space complexity.

Assume that $|P| = m$, $|alph(P)| = s$ and $alph(P) = \{a_1, \ldots, a_s\}$; denote by $k_i = |P|_{a_i}$, for $i \in \{1, \ldots, s\}$. We can assume that $s \geq 2$ (otherwise, the problem we solve is trivial).

Let us first count the number N_P of words over the alphabet $alph(P)$ that have the same Parikh vector as P has. It is rather plain to see that

$$N_P = \binom{m}{k_1}\binom{m-k_1}{k_2}\cdots\binom{m-(\sum_{i=1}^{s-1} k_i)}{k_s} = \frac{m!}{k_1!\cdots k_s!} = \frac{m!}{\Pi(k_1)\cdots\Pi(k_s)}$$

As Π is logarithmically convex, we get that:

$$\frac{\log(\Pi(k_1)\cdots\Pi(k_s))}{s} = \sum_{i=1}^{s} \frac{\log(\Pi(k_i))}{s} \geq \log\left(\Pi\left(\frac{k_1+\ldots+k_s}{s}\right)\right)$$

Therefore, we get that:

$$\log(\Pi(k_1)\cdots\Pi(k_s)) \geq \log\left(\Pi\left(\frac{k_1+\ldots+k_s}{s}\right)^s\right) = \log\left(\Pi\left(\frac{m}{s}\right)^s\right)$$

It follows immediately that $\Pi(k_1)\cdots\Pi(k_s) \geq \Pi\left(\frac{m}{s}\right)^s$, and, as a consequence, we get that $N_P \leq \frac{m!}{\Pi\left(\frac{m}{s}\right)^s}$

Recall that we work under the assumption that $m \geq |V|$, and we will estimate N_P for arbitrarily long patterns over the same alphabet. That is, we use the Stirling formula to get that there exists a constant K such that:

$$N_P \leq \frac{m!}{\Pi\left(\frac{m}{s}\right)^s} \leq K \cdot \frac{\left(\frac{m}{e}\right)^m \sqrt{2\pi m}}{\left(\left(\frac{m}{se}\right)^{m/s}\sqrt{2\pi(m/s)}\right)^s} = K \cdot \frac{s^m\sqrt{2\pi m}}{(\sqrt{2\pi(m/s)})^s}.$$

Once we have an upper bound for N_P we can also find an upper bound for the probability q that a word P' of length m has the same Parikh vector as P. That is:

$$q = \frac{N_P}{s^m} \leq K\frac{\sqrt{2\pi m}}{(\sqrt{2\pi(m/s)})^s} = \frac{K}{m}\cdot\frac{s^{s/2}\sqrt{2\pi}}{(\sqrt{2\pi})^s m^{(s-3)/2}}.$$

If $s \geq 3$ we have that $m^{(s-3)/2} \geq s^{(s-3)/2}$, so

$$q \leq \frac{K}{m}\cdot\frac{s^{s/2}\sqrt{2\pi}}{(\sqrt{2\pi})^s s^{(s-3)/2}}.$$

It follows that $q \leq \frac{1}{m}\cdot K'$ where K' is constant, as $\frac{s^{3/2}\sqrt{2\pi}}{(\sqrt{2\pi})^s} \to 0$, when $s \to \infty$.

When $s = 2$ we easily obtain that $q \leq \frac{1}{\sqrt{m}}\cdot K''$, for a constant K''.

Therefore, the average running time of our solution to Problem 2 is $\mathcal{O}(\frac{nm}{m})$ when the pattern contains at least three different letters and $\mathcal{O}(\frac{nm}{\sqrt{m}})$ when the pattern is a binary word.

Proposition 3. *Problem 2 for $f = (\cdot)^R$ can be solved in $\mathcal{O}(|T|)$ average running time and $\mathcal{O}(|P|)$ space when $|alph(P)| \geq 3$ and $\mathcal{O}(|T|\sqrt{|P|})$ average running time and $\mathcal{O}(|P|)$ space, when $|alph(P)| = 2$.*

This result can be compared with the one obtained in [3], where an algorithm running in linear average-time for a more general problem was obtained. There, the aim was to find the factors of T that can be obtained by translocations and rotations from P. However, the extra-operation played no role in the analysis of the average running time of the algorithm, and the linear average running time was obtained under artificial and restrictive hypotheses (for example, s divided

m and s was lower bounded by $\log(m)/\log\log^{1-\epsilon}(m)))$; the claims of the authors that these restriction do not affect the generality of the problem do not seem to hold canonically and, unfortunately, are not supported by arguments. In the paper [2], where exactly Problem 2 for rotations was addressed, no discussion on the average running time of the proposed algorithm was made; instead, finding a solution with linear average running time and good worst case time-complexity and space complexity was announced as future work.

Finally, we leave open the problem of finding an algorithm runing in average linear time for the case of binary alphabets. It is easy to see that there are patterns for which the probability q is in $\Theta(\frac{1}{\sqrt{m}})$ (e.g., a word P over $\{1,2\}$ with $|P|_1 = |P|_2$), so the average running time is in $\Theta(n\sqrt{m})$.

The case when $f(a) \neq a$, for all $a \in V$. This case can be seen as opposite to the previous one: the restriction of f to V has no fixed point. However, the strategy we use is pretty similar. We first identify the factors of T that can be obtained by f-rotations from P (i.e., have the same f-Parikh vector as P has) and then we check which of them can be actually obtained from P by non-overlapping f-rotations. The analysis is pretty much similar.

We take $|P| = m$ and $\{a_1, \ldots, a_s\} = \{i \in alph(P) \cup alph(f(P)) \mid i < f(i)\}$ (recall that the letters of P are assumed to be natural numbers); denote by $k_i = |P|_{a_i} + |P|_{f(a_i)}$, for $i \in \{1, \ldots, s\}$. We can assume that $s \geq 2$ (otherwise, every factor of length m of T is obtained from P by f-rotations).

In this case, we denote by N_P the number of words that have the same f-Parikh vector as P has, and get:

$$N_P = \binom{m}{k_1} 2^{k_1} \binom{m-k_1}{k_2} 2^{k_2} \cdots \binom{m-(\sum_{i=1}^{s-1} k_i)}{k_s} 2^{k_s} = \frac{2^m m!}{\Gamma(k_1)\cdots\Gamma(k_s)}.$$

Using the same strategy as above, we obtain an upper bound for the probability q that a word P' of length m has the same f-Parikh vector as P. That is, there exists a constant K such that:

$$q = \frac{N_P}{(2s)^m} \leq K \cdot \frac{\sqrt{2\pi m}}{(\sqrt{2\pi(m/s)})^s} = \frac{K}{m} \cdot \frac{s^{s/2}\sqrt{2\pi}}{(\sqrt{2\pi})^s m^{(s-3)/2}}.$$

As in the former case, if $s \geq 3$ (i.e., $alph(P) \cup alph(f(P))$ has at least 6 elements) it follows that $q \leq \frac{1}{m} \cdot K'$, where K' is a constant.

When $s = 2$ (i.e., $alph(P) \cup alph(f(P))$ has 4 elements) we obtain again that $q \leq \frac{1}{\sqrt{m}} \cdot K''$, for a constant K''.

Therefore, the average running time of our solution to Problem 2 is $\mathcal{O}(\frac{nm}{m})$ when $alph(P) \cup alph(f(P)) \geq 6$ and $\mathcal{O}(\frac{nm}{\sqrt{m}})$ when $alph(P) \cup alph(f(P)) = 4$.

Proposition 4. *Let f be an antimorphic involution such that $f(a) \neq a$ for all $a \in V$. In this setting, Problem 2 can be solved in $\mathcal{O}(|T|)$ average running time and $\mathcal{O}(|P|)$ space when $alph(P) \cup alph(f(P)) \geq 6$ and $\mathcal{O}(|T|\sqrt{|P|})$ average running time and $\mathcal{O}(|P|)$ space, when $alph(P) \cup alph(f(P)) = 4$.*

It would be interesting to see whether a faster algorithm can be obtained for the case when $alph(P) \cup alph(f(P)) = 4$, which modells the DNA-alphabet with f being the Watson-Crick complementarity. However, note that in some biological problems several consecutive symbols occurring in the DNA-sequence are grouped together and the sequence is seen as being over a greater alphabet (e.g., the Amino Acid alphabet, obtained by gropuing together each three consecutive letters of the DNA-sequence); in such cases, our algorithm runs in linear time.

The general case. In this case there exists $V' \subseteq alph(P)$ such that $f(a) = a$ for all $a \in V'$ and $V'' \subseteq alph(P)$ such that $f(a) \neq a$, for all $a \in V''$. Using a strategy similar to the above, we are able to design algorithms that solve Problem 2 efficiently with respect to average time complexity. However, in this case, the restrictions on the alphabet are stronger than in the previous cases; fortunately, the nature of this restriction remains similar: tha size of the alphabet must be greater than a given constant. The results obtained in this case are summarized in the following proposition (whose proof is given in the Appendix).

Proposition 5. *Let f be an antimorphic involution such that $f(a) = a$ for all $a \in V'$ and $f(a) \neq a$ for all $a \in V''$.*
If $|P|_{V'} \geq \frac{|P|}{2}$ then Problem 2 can be solved in $\mathcal{O}(|T|)$ average running time and $\mathcal{O}(|P|)$ space when $alph(P) \cap |V'| \geq 5$ and $\mathcal{O}(|T|\sqrt{|P|})$ average running time and $\mathcal{O}(|P|)$ space, when $alph(P) \cap |V'| = 4$.
If $|P|_{V''} > \frac{|P|}{2}$ then Problem 2 can be solved in $\mathcal{O}(|T|)$ average running time and $\mathcal{O}(|P|)$ space when $|\{i \mid i < f(i), i \in alph(P)\}| \geq 5$ and $\mathcal{O}(|T|\sqrt{|P|})$ average running time and $\mathcal{O}(|P|)$ space, when $|\{i \mid i < f(i), i \in alph(P)\}| = 4$.

References

1. Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York, NY, USA (1997)
2. Cantone, D., Cristofaro, S., Faro, S.: Efficient matching of biological sequences allowing for non-overlapping inversions. In: CPM. Volume 6661 of Lecture Notes in Computer Science., Springer (2011) 364–375
3. Grabowski, S., Faro, S., Giaquinta, E.: String matching with inversions and translocations in linear average time (most of the time). Inf. Process. Lett. **111**(11) (2011) 516–520
4. Cantone, D., Faro, S., Giaquinta, E.: Approximate string matching allowing for inversions and translocations. In Holub, J., Žďárek, J., eds.: 2010. (2010) 37–51
5. Baeza-Yates, R.A., Navarro, G.: New and faster filters for multiple approximate string matching. Random Struct. Algorithms **20**(1) (2002) 23–49
6. Lothaire, M.: Combinatorics on Words. Cambridge University Press (1997)
7. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM **53** (2006) 918–936
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)
9. Artin, E.: The gamma function. A thena series: Selected topics in mathematics. Holt, Rinehart and Winston (1964)

Appendix

We provide some basic information on the computational model we use: the unit-cost RAM (Random Access Machine) with logarithmic word size. For a more detailed explanation, [8] is a good reference (see the explanations in Section 2.2, Analyzing Algorithms). In this model (which is generally used in the analysis of algorithms) we assume that each memory cell can store $\mathcal{O}(\log n)$ bits, or, in other words, that *the machine word size* is $\mathcal{O}(\log n)$; the constant hidden by the $\mathcal{O}$-notation is at least 1. The instructions are executed one after another, with no concurrent operations. The model contains common instructions: arithmetic (add, subtract, multiply, divide, remainder, shifts and bitwise operations, etc.), data movement (load the content of a memory cell, store a number in a memory cell, copy the content of a memory cell to another), and control (conditional and unconditional branch, subroutine call and return). Each such instruction takes a constant amount of time. Note that testing the equality of two numbers is also assumed to take a constant amount of time. It is also a common assumption, made when working with the unit-cost RAM with logarithmic word size, that basic operations on arrays containing a polynomial (in n) number of $\mathcal{O}(\log n)$-bit integer elements, are carried out in constant time; such operations are accessing the value stored in a cell of an array, using it in arithmetic operations, updating it, but not sorting the array, extracting the minimum or the maximum, etc. Basically, this model allows us to measure the number of instructions executed in an algorithm, making abstraction of the time spent to execute each of the basic instructions.

Proof of Lemma 1:
Assume first that $\sigma(1) = i$. Then, if $i_1 = 1$ we immediately obtain that:

$$P = a_1 \dots a_i a_{i+1} \dots a_m \Rightarrow_f f(a_i)f(a_{i-1}) \dots f(a_1)a_{i+1} \dots a_m.$$

If $i_1 = 0$, we can apply a further rotation to the word obtained above:

$$f(a_i)f(a_{i-1}) \dots f(a_1)a_{i+1} \dots a_m \Rightarrow_f a_i f(a_{i-1}) \dots f(a_1)a_{i+1} \dots a_m.$$

Therefore, for both possible values of i_1 we can obtain by f-rotations a word that begins with $f^{i_1}(a_{\sigma(1)})$. Let us assume that we can obtain by f-rotations, from P, a word that begins with $f^{i_1}(a_{\sigma(1)}) \dots f^{i_j}(a_{\sigma(j)})$. Clearly, by the same strategy as the above, by applying at most two f-rotations, we can obtain from this word a word that begins with $f^{i_1}(a_{\sigma(1)}) \dots f^{i_{j+1}}(a_{\sigma(j+1)})$. By this inductive argument, the statement of our lemma is shown. □

The complexity of Algorithm 4.1:
Recall that $m = |P|$. To begin with, note that step 1 takes $\mathcal{O}(m)$ time, according to the discussions on data structures, made in the Preliminaries section. Further, in one execution of the cycle in steps $3 - 19$, for an we make $i \leq m$ exactly ℓ comparisons to find the longest prefix of $P[i..m]$ and $P'[i..m]$, provided that this longest prefix has exactly the length ℓ, and, then, another p comparisons to find

the shortest word such that $P'[i+\ell+1..i+\ell+p] = f(P[i+\ell+1..i+\ell+p])$; then, the cycle in steps $3-19$ is executed again for a new value of i equal to $i+\ell+p+1$ (that is, i is updated to have this new value, according to its previous value). It is rather plain that the total execution of this cycle takes $\mathcal{O}(m)$ time. Note that the comparison in step 10 needs only $\mathcal{O}(1)$ time, as it basically consists in checking the result to a $LCPref$-query on w, and for this we use the data structures constructed in step 1.

According to the above, the overall time complexity of Algorithm 4.1 is $\mathcal{O}(m)$. The space complexity of this algorithm is, clearly, $\mathcal{O}(m)$; indeed, the only data structures that we use are those computed in step 1, and they are stored in linear space.

The Proof of Proposition 5:
For $m = |P|$, we have either $|P|_{V'} \geq \frac{m}{2}$ or $|P|_{V''} > \frac{m}{2}$.

Let us first look at the case when $|P|_{V'} \geq \frac{m}{2}$. Let m' be the length of the word P' obtained from P by deleting all the letters of V''. Also, let T' be the word obtained from T by deleting the occurrences of letters of V''. Our solution is to identify first all the factors $T'[i..i+m'-1]$ such that $P' \rightarrow_f^* T'[i..i+m'-1]$. Further, if j is the position of T that corresponds to $T'[i]$ (that is, when we deleted the letters in V'' from T, the letter on position j became the letter on position i in T'), we test whether the strings $T[\ell \ldots \ell+m-1]$ with $j-(m-m')+1 \leq \ell \leq j$ can be obtained from P by non-overlapping f-rotations. This solution is clearly sound, as $T[\ell..\ell+m-1]$ can be obtained by non-overlapping f-rotations from P only if the string obtained from $T[\ell..\ell+m-1]$ by deleting the letters in V'' can be obtained by f-rotations from P'.

We now analyse its average running time. As in the previous cases, we obtain an upper bound on the number $N_{P'}$ of strings over V' that have the same Parikh vector as P' has. More precisely, for $s = |V'|$, we have that there exists a constant K such that:

$$N_{P'} \leq K \cdot \frac{s^{m'}\sqrt{2\pi m'}}{(\sqrt{2\pi(m'/s)})^s}.$$

Recall that $m' \geq \frac{m}{2}$. Thus, the probability q that a word P'' of length m' over V' has the same Parikh vector as P' can be upper bounded as follows:

$$q = \frac{N_P}{s^{m'}} \leq K \cdot \frac{\sqrt{2\pi m'}}{(\sqrt{2\pi(m'/s)})^s} \leq \frac{4K}{m^2} \cdot \frac{s^{s/2}\sqrt{2\pi}}{4(\sqrt{2\pi})^s m^{(s-5)/2}}.$$

When $s \geq 5$ (which means that $|V| \geq 7$), we get that $q \leq \frac{1}{m^2}K'$, for a constant K'. When $s = 4$ (which means that $|V| \geq 6$) we have that $q \leq \frac{1}{m\sqrt{m}}K''$, for a constant K''. Therefore, our strategy produces, in this case, an algorithm that runs in average better than the non-optimized algorithm solving Problem 2 only when $s \geq 4$.

Further, we analyse the case when $|w|_{V''} > \frac{m}{2}$. The strategy we use is exactly as in the previous case, with the only difference that now we delete the letters in V' from T and P to obtain T' and, respectively, P', and search the factors

of T' that can be obtained from P' by f-rotations. Again, denote by m' the length of P' (and we have that $m' > \frac{m}{2}$) and by s the cardinality of the set $\{i \in alph(P') \cup alph(f(P')) \mid i < f(i)\}$. In this setting, the probability q that a word P'' of length m' over V'' has the same f-Parikh vector as P' can be upper bounded as follows. There exists a constant K such that:

$$q \leq \frac{4K}{m^2} \cdot \frac{s^{s/2}\sqrt{2\pi}}{4(\sqrt{2\pi})^s m^{(s-5)/2}}.$$

When $s \geq 5$ (which means that $|V| \geq 11$), we get that $q \leq \frac{1}{m^2}K'$, for a constant K'. When $s = 4$ (i.e., $|V| \geq 9$) we have that $q \leq \frac{1}{m\sqrt{m}}K''$, for a constant K''. Therefore, our strategy produces, in this case, an algorithm that runs in average better than the non-optimized algorithm solving Problem 2 only when $s \geq 4$. □