Discrete Applied Mathematics 163 (2014) 247–257

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Text searching allowing for inversions and translocations of factors☆

Domenico Cantone [a], Simone Faro [a,*], Emanuele Giaquinta [a,b]

[a] Università di Catania, Dipartimento di Matematica e Informatica, Viale Andrea Doria 6, I-95125 Catania, Italy

[b] Department of Computer Science, University of Helsinki, Finland

ARTICLE INFO

Article history:
Received 13 December 2011
Received in revised form 15 February 2013
Accepted 17 May 2013
Available online 17 June 2013

Keywords:
Approximate string matching
Text processing
Computational biology
Inversions and translocations
Analysis of algorithms

ABSTRACT

The approximate string matching problem consists in finding all locations at which a pattern p of length m matches a substring of a text t of length n, after a finite number of given edit operations.

In this paper, we investigate such a problem when the edit operations are translocations of adjacent factors of equal length and inversions of factors. In particular, we first present an $\mathcal{O}(nm \max(\alpha, \beta))$-time and $\mathcal{O}(m^2)$-space algorithm, where α and β are respectively the maximum lengths of the factors which can be involved in any translocation and inversion, and show that under the assumptions of equiprobability and independence of characters our algorithm has a $\mathcal{O}(n \log_\sigma m)$ average time complexity, for an alphabet of size σ. We also present a very fast variant of a recently proposed algorithm for the same problem, based on an efficient filtering method, which has a $\mathcal{O}(n)$-time complexity in the average case, though in the worst case it retains the same $\mathcal{O}(nm \max(\alpha, \beta))$-time complexity.

1. Introduction

Retrieving information and teasing out the meaning of biological sequences are central problems in modern biology. Generally, basic biological information is stored in strings of nucleic acids (DNA, RNA) or amino acids (proteins).

With the availability of large amounts of DNA data, matching of nucleotide sequences has become an important application, and there is an increasing demand for fast computer methods for analysis and data retrieval. In recent years, much work has been devoted to the development of efficient methods for aligning strings and, despite sequence alignment seeming to be a well-understood problem (especially in the edit-distance model), the same cannot be said for the approximate string matching problem on biological sequences.

Approximate string matching is a fundamental problem in text processing; it consists in finding approximate matches of a pattern in a string. The closeness of a match is measured in terms of the sum of the costs of the edit operations necessary to convert the string into an exact match.

Most biological string matching methods are based on the *Levenshtein distance* [13], commonly referred to just as the *edit distance*, or on the *Damerau distance* [8]. The edit operations in the case of the Levenshtein distance are *insertions*, *deletions*, and *substitutions* of characters, whereas, in the case of the Damerau distance, *swaps* of characters, i.e., transpositions of two adjacent characters, are also allowed (for an in-depth survey on approximate string matching, see [14]). Both distances assume that changes between strings occur locally, i.e., only a small portion of the string is involved in the mutation event.

☆ The paper extends results that appeared in a preliminary form in [3].

* Corresponding author. Fax: +39 095 330094.
E-mail address: faro@dmi.unict.it (S. Faro).

0166-218X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.dam.2013.05.016

However, evidence shows that in some cases large-scale changes are possible [7,15]. For example, large pieces of DNA can be moved from one location to another (*translocations*), or replaced by their reversed complements (*inversions*).

In this paper, we investigate the approximate string matching problem under a string distance whose edit operations are translocations of equal length adjacent factors and inversions of factors. Such a problem can be solved naively in $\mathcal{O}(nm^2)$-time and $\mathcal{O}(m^2)$-space. In the first part of the paper, we present a $\mathcal{O}(nm\max(\alpha,\beta))$-time and $\mathcal{O}(m^2)$-space algorithm, where α and β are the maximum lengths of the factors that can be involved in a translocation and in an inversion, respectively. Our algorithm, called M-Sampling, is based on a dynamic-programming approach and makes use of the Directed Acyclic Word Graph of the pattern. We show that under the assumption of equiprobability and independence of characters in the alphabet, on average our algorithm has a $\mathcal{O}(n\log_\sigma m)$-time complexity, for an alphabet of size σ.

Next, in the second part of the paper, we present a very fast variant of the recently proposed GFG (Grabowski–Faro–Giaquinta) algorithm for the same problem (see [10]), based on an efficient permutation filtering method for locating candidate positions. As for the GFG algorithm and the M-Sampling algorithm presented in the first part of the paper, our algorithm, called Addition–Counting–Filter, achieves a $\mathcal{O}(nm\max(\alpha,\beta))$ worst-case time complexity but requires only $\mathcal{O}(\max(\alpha,\beta,\sigma))$ additional space. More interestingly, we will show that it has a $\mathcal{O}(n)$ average-case time complexity and that it is very fast in most practical cases. In fact, we conducted an extensive experimental evaluation to compare our proposed algorithms with two different implementations of the fast GFG algorithm [10]. Experimental results show that the Addition–Counting–Filter algorithm is up to 40% faster than the GFG algorithm.

The rest of the paper is organized as follows. In Section 2, we introduce some preliminary notions and definitions. Subsequently, in Section 3, we present our algorithm M-Sampling and analyze it both in the worst case and in the average case. In Section 4, we present our algorithm Addition–Counting–Filter and show that it has a linear complexity in the average case. Then, in Section 5, we examine the results of an experimental evaluation of our proposed algorithms (whose codes are reported in the Appendix) and compare them with two implementations of the GFG algorithm.

2. Basic notions and definitions

Let p be a string of length $m \geq 0$, over an alphabet Σ. We represent it as a finite array $p[0..m-1]$ of characters of Σ and write $|p| = m$. In particular, for $m = 0$, we have the empty string ε. We denote by $p[i]$ the $(i+1)$st character of p, for $0 \leq i < m$. Likewise, the substring or factor of p contained between the $(i+1)$st and the $(j+1)$st characters of p is indicated by $p[i..j]$, for $0 \leq i \leq j < m$. The set of factors of p is denoted by $Fact(p)$, and its size is $\mathcal{O}(m^2)$. Given another string p', we say that p' is a suffix of p (in symbols, $p' \sqsupseteq p$) if $p' = p[i..m-1]$, for some $0 \leq i < m$, and denote by $Suff(p)$ the set of the suffixes of p. Similarly, we say that p' is a prefix of p if $p' = p[0..i]$, for some $0 \leq i < m$. We also put $p_i \stackrel{\text{Def}}{=} p[0..i]$, for $0 \leq i < m$, and make the convention that p_{-1} denotes the empty string ε. In addition, we write pp' for the concatenation of p and p', and p^r for the reverse of the string p, i.e., $p^r \stackrel{\text{Def}}{=} p[m-1]p[m-2]\cdots p[0]$.

For $x \in Fact(p)$, we denote with $end\text{-}pos(x)$ the set of all positions in p where an occurrence of x ends; formally, we put $end\text{-}pos(x) \stackrel{\text{Def}}{=} \{i \mid x \sqsupseteq p_i\}$. For any given pattern p, we define an equivalence relation $\mathcal{R}_p$ by putting $x\mathcal{R}_p y \iff end\text{-}pos(x) = end\text{-}pos(y)$, for all $x, y \in \Sigma^*$, and denote with $\mathcal{R}_p(x)$ the equivalence class of the string x. For each equivalence class q of $\mathcal{R}_p$, we put $len(q) \stackrel{\text{Def}}{=} |val(q)|$, where $val(q)$ is the longest string x in the equivalence class q.

Example 1. Let $p = abcabccab$. Then we have $end\text{-}pos(ab) = \{1, 4, 8\}$ while $end\text{-}pos(bcc) = \{6\}$. Observe moreover that $\mathcal{R}_p(ab) = \{ab, b\}$. Similarly we have $\mathcal{R}_p(c) = \{abc, bc, c\}$. Thus $len(\mathcal{R}_p(ab)) = 2$ while $len(\mathcal{R}_p(c)) = 3$.

The Directed Acyclic Word Graph (DAWG) [2,5,6] of a pattern p is the deterministic automaton $\mathcal{A}(p) = (Q, \Sigma, \delta, root, F)$ whose language is $Fact(p)$, where $Q = \{\mathcal{R}_p(x) : x \in Fact(p)\}$ is the set of states, Σ is the alphabet of the characters in p, $root = \mathcal{R}_p(\varepsilon)$ is the initial state, $F = Q$ is the set of final states, and $\delta : Q \times \Sigma \to Q$ is the transition function defined by $\delta(\mathcal{R}_p(y), c) \stackrel{\text{Def}}{=} \mathcal{R}_p(yc)$, for all $c \in \Sigma$ and $yc \in Fact(p)$.

We define a failure function, $s\ell : Fact(p) \setminus \{\varepsilon\} \to Fact(p)$, called the *suffix link*, by putting $s\ell(x) \stackrel{\text{Def}}{=}$ "longest $y \in Suff(x)$such that $y \not\mathcal{R}_p x$", for any $x \in Fact(p) \setminus \{\varepsilon\}$. The function $s\ell$ has the property $x\mathcal{R}_p y \Longrightarrow s\ell(x) = s\ell(y)$. We extend the functions $s\ell$ and $end\text{-}pos$ to Q by putting $s\ell(q) \stackrel{\text{Def}}{=} \mathcal{R}_p(s\ell(val(q)))$ and $end\text{-}pos(q) \stackrel{\text{Def}}{=} end\text{-}pos(val(q))$, for each $q \in Q$.

A *distance* $d : \Sigma^* \times \Sigma^* \to \mathbb{R}$ is a function which associates to any pair of strings x and y the minimal cost of any finite sequence of edit operations which transforms x into y, if such a sequence exists, and ∞ otherwise. The *mutation distance* $md(x, y)$ is based on the following edit operations: *translocations*, where a factor of the form zw is transformed into wz, provided that $|z| = |w| > 0$; and *inversions*, where a factor z is transformed into z^r. Both operations are assigned unit cost.

Observe that, since equal length factors are involved in translocations, the maximum length of the factors involved in a translocation in a string x is $\lfloor |x|/2 \rfloor$, whereas the length of the factors involved in an inversion can be up to $|x|$.

Example 2. Let $x = gtgaccgtccag$ and $y = gtcgtgacccga$. Then $md(x, y) = 2$, since x can be transformed into y by translocating the substrings $x[2..4] = gac$ and $x[5..7] = cgt$, and inverting the substring $x[10..11] = ag$.

When $md(x, y) < \infty$, we say that x and y have an md-match. If x has an md-match with a suffix of y, we write $x \stackrel{md}{\sqsupseteq} y$.

3. An automaton-based approach: the M-SAMPLING algorithm

We present an efficient algorithm, called M-SAMPLING, which finds the *md*-matches of a given pattern p (of length m) in a text t (of length n). Our algorithm, based on the dynamic programming approach, has a $\mathcal{O}(nm\max(\alpha,\beta))$-time and $\mathcal{O}(m^2)$-space complexity, where $\alpha \leq \lfloor m/2 \rfloor$ and $\beta \leq m$ are bounds on the length of the factors that can be involved in any translocation and in any inversion, respectively.

Given p, t, m, n, α, and β as above, the M-SAMPLING algorithm iteratively computes, for $j = m-1, m, \ldots, n-1$, all the prefixes of p which have an *md*-match with a suffix of t_j, by exploiting information gathered at previous iterations. For this purpose, it is convenient to introduce the set $\mathcal{S}_j \stackrel{\text{Def}}{=} \{0 \leq i \leq m-1 \mid p_i \stackrel{md}{\sqsupseteq} t_j\}$, for $0 \leq j < n$. Thus, the pattern p has an *md*-match ending at position j of the text t if and only if $(m-1) \in \mathcal{S}_j$.

Since the allowed edit operations involve substrings of the pattern p, it is useful to consider also the set $\mathcal{F}_j^k$ of all the positions in p at which an occurrence of the suffix of t_j of length k ends. More precisely, for $1 \leq k \leq \alpha$ and $k-1 \leq j < n$, we put $\mathcal{F}_j^k \stackrel{\text{Def}}{=} \{k-1 \leq i \leq m-1 \mid t[j-k+1..j] \sqsupseteq p_i\}$. Observe that $\mathcal{F}_j^k \subseteq \mathcal{F}_j^h$, for $1 \leq h \leq k \leq m$.

Similarly, to handle inversions, it is convenient to define the set $\mathcal{I}_j^k$ of the positions in p at which an occurrence of the reverse of the suffix of t_j of length k ends. More precisely, for $1 \leq k \leq \beta$ and $k-1 \leq j < n$, we put $\mathcal{I}_j^k \stackrel{\text{Def}}{=} \{k-1 \leq i \leq m-1 \mid (t[j-k+1..j])^r \sqsupseteq p_i\}$.

Example 3. Let $p = cattcatgatcat$ be a pattern, and let $t = atcatgacttactgactta$ be the text. Then $\mathcal{F}_4^3$ is the set of all positions in p at which an occurrence of the suffix of t_4 of length 3 ends, namely, *cat*. Thus $\mathcal{F}_4^3 = \{2, 6, 12\}$. Similarly, we have that $\mathcal{F}_4^2 = \{2, 6, 9, 12\}$. Observe that $\mathcal{F}_4^3 \subseteq \mathcal{F}_4^2$.

Observe also that, since $t[2..4] = cat$ is the reverse string of $t[9..11] = tac$, we have $\mathcal{I}_{11}^3 = \mathcal{F}_4^3 = \{2, 6, 12\}$. Moreover, $\mathcal{I}_8^2 = \{4, 10\}$, since the reverse of the substring $t[7..8] = ct$ occurs in its reverse form at positions 3 and 9 of the pattern.

The sets $\mathcal{S}_j$ can then be computed by way of the recursive relations contained in the following elementary lemma.

Lemma 1. *Let t and p be a text of length n and a pattern of length m, respectively. Then $i \in \mathcal{S}_j$, for $0 \leq i < m$ and $i \leq j < n$, if and only if one of the following three facts holds:*

(a) $p[i] = t[j]$ *and* $(i-1) \in \mathcal{S}_{j-1} \cup \{-1\}$;
(b) $(i-k) \in \mathcal{F}_j^k$, $i \in \mathcal{F}_{j-k}^k$, *and* $(i-2k) \in \mathcal{S}_{j-2k} \cup \{-1\}$, *for some* $1 \leq k \leq \lfloor \frac{i+1}{2} \rfloor$;
(c) $i \in \mathcal{I}_j^k$ *and* $(i-k) \in \mathcal{S}_{j-k} \cup \{-1\}$, *for some* $1 \leq k \leq i+1$. □

Notice that conditions (b) and (c) in Lemma 1 refer to a translocation of adjacent factors of length k and to an inversion of a factor of length k, respectively.

Likewise, the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ can be computed according to the following lemma.

Lemma 2. *Let t and p be a text of length n and a pattern of length m, respectively. Then $i \in \mathcal{F}_j^k$ if and only if $p[i] = t[j]$ and either $k = 1$ or $(i-1) \in \mathcal{F}_{j-1}^{k-1}$, for $1 \leq k \leq \alpha$, $k-1 \leq j < n$, and $k-1 \leq i < m$.*

Similarly, $i \in \mathcal{I}_j^k$ if and only if $p[i-k+1] = t[j]$ and either $k = 1$ or $i \in \mathcal{I}_{j-1}^{k-1}$, for $1 \leq k \leq \beta$, $k-1 \leq j < n$, and $k-1 \leq i < m$. □

Based on the recurrence relations in Lemmas 1 and 2, a general dynamic programming algorithm can be readily constructed, characterized by an overall $\mathcal{O}(nm\max(\alpha,\beta))$-time and $\mathcal{O}(m^2)$-space complexity. However, the overhead due to the computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$ turns out to be quite relevant.

3.1. Efficient computation of the sets $\mathcal{F}_j^k$ and $\mathcal{I}_j^k$

An efficient method for computing the sets $\mathcal{F}_j^k$, for $1 \leq k \leq \alpha$ and $k-1 \leq j < n$, makes use of the DAWG of the pattern p and the function *end-pos*. Later we will also show how to compute efficiently the sets $\mathcal{I}_j^k$.

Let $\mathcal{A}(p) = (Q, \Sigma, \delta, root, F)$ be the DAWG of p. For each position j in t, let p' be the longest factor of p, of length at most α, which is a suffix of t_j; also, let q_j be the state of $\mathcal{A}(p)$ such that $\mathcal{R}_p(p') = q_j$, and let l_j be the length of p'. We call the pair (q_j, l_j) a *t-configuration* of $\mathcal{A}(p)$.

The idea is then to compute the t-configuration (q_j, l_j) of $\mathcal{A}(p)$, for each position j of the text, while scanning the text t. The set $\mathcal{F}_j^k$ computed at previous iterations does not need to be maintained explicitly; rather, it is enough to maintain only t-configurations. These are then used to compute efficiently the set $\mathcal{F}_j^k$ only when needed.

The longest factor of p ending at position j of t is computed in the same way as in the FORWARD-DAWG-MATCHING algorithm for the exact pattern matching problem (see [6]). We maintain the invariant that the current state of the automaton never corresponds to factors longer than α.

Let (q_{j-1}, l_{j-1}) be the t-configuration of $\mathcal{A}(p)$ at step $(j-1)$. Two cases must be distinguished.

Case $l_{j-1} < \alpha$: The new t-configuration (q_j, l_j) is set to $(\delta(q, t[j]), \mathit{length}(q)+1)$, where q is the first node in the suffix path $\langle q_{j-1}, s\ell(q_{j-1}), s\ell^{(2)}(q_{j-1}), \ldots\rangle$ of q_{j-1}, including q_{j-1}, having a transition on $t[j]$, if such a node exists; otherwise, (q_j, l_j) is set to $(\mathit{root}, 0)$.[1]

Case $l_{j-1} = \alpha$: We first compute the t-configuration corresponding to the factor $t[j-\alpha+1..j-1]$ of p of length $(\alpha-1)$ ending at position $j-1$ in t, namely, the t-configuration (q'_{j-1}, l'_{j-1}), where $(q'_{j-1}, l'_{j-1}) \equiv (s\ell(q_{j-1}), l_{j-1}-1)$, if $\mathit{length}(s\ell(q_{j-1})) = l_{j-1}-1$; otherwise, $(q'_{j-1}, l'_{j-1}) \equiv (q_{j-1}, l_{j-1}-1)$. Then we compute the new t-configuration (q_j, l_j) starting from (q'_{j-1}, l'_{j-1}) as in the previous case, observing that $l'_{j-1} = \alpha - 1$.

Before explaining how to compute the sets $\mathcal{F}_j^k$, it is convenient to introduce a partial function, $\phi : Q \times \mathbb{N} \to Q$, which, given a node $q \in Q$ and a length $k \le \mathit{length}(q)$, computes the state $\phi(q, k)$ whose corresponding set of factors contains the suffix of $\mathit{val}(q)$ of length k. Roughly speaking, $\phi(q, k)$ is the first node p in the suffix path of q such that $\mathit{length}(s\ell(p)) < k$.

In the preprocessing phase, the DAWG $\mathcal{A}(p) = (Q, \Sigma, \delta, \mathit{root}, F)$ together with the associated *end-pos* function is computed. Since for a pattern p of length m we have that $|Q| \le 2m+1$ and $|\mathit{end\text{-}pos}(q)| \le m$, for each $q \in Q$, we need only $\mathcal{O}(m^2)$ extra space (see [2,5]).

To compute the set $\mathcal{F}_j^k$, for $1 \le k \le l_j$, one can take advantage of the relation $\mathcal{F}_j^k = \mathit{end\text{-}pos}(\phi(q_j, k))$. Notice that, in particular, we have $\mathcal{F}_j^{l_j} = \mathit{end\text{-}pos}(q_j)$. The time complexity of the computation of $\phi(q, k)$ can be bounded by the length of the suffix path of node q. Specifically, since the sequence $\langle \mathit{len}(s\ell^{(0)}(q)), \mathit{len}(s\ell^{(1)}(q)), \ldots, 0\rangle$ of the lengths of the nodes in the suffix path from q is strictly decreasing, we can do at most $\mathit{len}(q)$ iterations over the suffix link, obtaining a $\mathcal{O}(m)$-time complexity.

According to Lemma 1, a translocation of length $2k$ at position j of the text t is possible only if factors of p of length at least k have been recognized at both positions j and $j-k$, namely if $l_j \ge k$ and $l_{j-k} \ge k$.

Let $\langle k_1, k_2, \ldots, k_r\rangle$ be the increasing sequence of all values k such that $1 \le k \le \min(l_j, l_{j-k})$. For each $1 \le i \le r$, condition (b) of Lemma 1 requires member queries on the sets $\mathcal{F}_j^{k_i}$ and $\mathcal{F}_{j-k_i}^{k_i}$.

Observe that, if we proceed for decreasing values of k, the sets $\mathcal{F}_j^k$, for $1 \le k \le l_j$, can be computed in constant time. Specifically, $\mathcal{F}_j^k$ can be computed in constant time from $\mathcal{F}_j^{k+1}$, for $k = 1, \ldots, l_j - 1$, with at most one iteration over the suffix link of the state $\phi(q_j, k+1)$. The computation of $\mathcal{F}_{j-k_r}^{k_r}$ has a $\mathcal{O}(\alpha)$-time complexity, since $\mathit{length}(q_{j-k_r}) \le \alpha$. To compute $\mathcal{F}_{j-k_i}^{k_i}$, for $i = r-1, r-2, \ldots, 1$, we distinguish the following two cases.

Case $k_{i+1} = k_i + 1$: Let $q' = \phi(q_{j-k_{i+1}}, k_{i+1})$. Given the node q' computed in the previous iteration, the node $\phi(q_{j-k_i}, k_i)$ can be computed in two steps: first, we look up the node corresponding to the suffix of length $k_{i+1}-2$ of the factor represented by q', with at most two iterations of the suffix link of q'; then, we perform a transition on $t[j-k_i]$ on the node so found. Formally,

$$\phi(q_{j-k_i}, k_i) = \delta(\phi(q', k_{i+1}-2), t[j-k_i]).$$

Case $k_{i+1} > k_i + 1$: Observe that $l_{j-s} \le s-1$ must hold, for each $s = k_{i+1}-1, \ldots, k_i+1$. In particular, we have $l_{j-(k_i+1)} \le k_i$, which implies that $l_{j-k_i} \le k_i + 1$, since $l_j \le l_{j-1} + 1$ always holds. Hence, the computation of $\phi(q_{j-k_i}, k_i)$ requires at most one iteration of the suffix link of q_{j-k_i}.

Thus the total complexity for computing all the sets $\mathcal{F}_{j-k_i}^{k_i}$ is $\mathcal{O}(\alpha)$.

Next, we show how to use the DAWG $\mathcal{A}(p^r)$ of p^r to compute efficiently the sets $\mathcal{I}_j^k$. Specifically, we compute the longest reversed factor ending at j and maintain the invariant that the current state of the automaton never corresponds to factors longer than β, using the algorithm Dawg-Delta reported in the Appendix, as in the case of the computation of the sets $\mathcal{F}_j^k$. Let (q_j^r, l_j^r) denote the t-configuration of $\mathcal{A}(p^r)$ after having read the character of t at position j, where l_j^r is the length of the longest reversed factor of p recognized. Then the sets $\mathcal{I}_j^k$ can be computed, for $2 \le k \le l_j^r$, by

$$\mathcal{I}_j^k = \{i \mid (m-i+k-2) \in \mathit{end\text{-}pos}(\phi(q_j^r, k))\}. \tag{1}$$

Indeed, $i \in \mathcal{I}_j^k$ iff $p[i-k+1..i] = (t[j-k+1..j])^r$ iff $p^r[(m-1)-i..(m-1)-(i-k+1)] = (p[i-k+1..i])^r = t[j-k+1..j]$. Thus (1) follows readily, since the latter equality is plainly equivalent to $(m-i+k-2) \in \mathit{end\text{-}pos}(\phi(q_j^r, k))$.

For each $k = 1, \ldots, l_j^r$, condition (3) of Lemma 1 requires member queries on the sets $\mathcal{I}_j^k$. As in the case of the sets $\mathcal{F}_j^k$, the set $\mathit{end\text{-}pos}(\phi(q_j^r, k))$ can be computed in constant time, in decreasing order of k, by iterating the suffix link on q_j^r. Although $\mathcal{I}_j^k$ is not equal to $\mathit{end\text{-}pos}(\phi(q_j^r, k))$, a member query on $\mathcal{I}_j^k$ can still be done in constant time using (1).

[1] We recall that $s\ell^{(0)}(q) \stackrel{\text{Def}}{=} q$ and, recursively, $s\ell^{(h+1)}(q) \stackrel{\text{Def}}{=} s\ell(s\ell^{(h)}(q))$, for $h \ge 0$, provided that $s\ell^{(h)}(q) \ne \mathit{root}$.

3.2. Worst-case time and space analysis

In this section, we present the worst-case time and space analysis of the M-Sampling algorithm. In particular, we will refer to the code of the M-Sampling algorithm reported in the Appendix.

First of all, observe that the main **for**-loop at line 6 is always executed n times. Moreover, we have $|\mathcal{S}_j| \leq m$, $l_j \leq \alpha$, and $l_j^r \leq \beta$, for all $0 \leq j < n$. For each iteration of the **for**-loop at line 22, the amortized cost of the two **while**-loops at lines 25 and 30 is $\mathcal{O}(1)$. Thus, at each iteration of the main **for**-loop, the internal **for**-loop at line 11 takes at most $\mathcal{O}(m)$-time, while the **for**-loops at lines 15 and 22 take at most $\mathcal{O}(m\beta)$-time and $\mathcal{O}(m\alpha)$-time, respectively. Summing up, the M-Sampling algorithm has a $\mathcal{O}(nm\max(\alpha, \beta))$ worst-case time complexity, which becomes $\mathcal{O}(nm^2)$-time when $\max(\alpha, \beta) = \Theta(m)$.

In order to evaluate the space complexity of the M-Sampling algorithm, we observe that, in the worst case, during the jth iteration of its main **for**-loop, the sets $\mathcal{F}_{j-k}^k$ and $\mathcal{S}_{j-2k}$, for $1 \leq k \leq \alpha$, and the sets $\mathcal{S}_{j-k}$, for $2 \leq k \leq \beta$, must be kept in memory to handle translocations and inversions, respectively. However, as explained before, we do not keep the values of $\mathcal{F}_{j-k}^k$ explicitly, but rather we maintain only their corresponding t-configurations of the automaton $\mathcal{A}(p)$. Thus, we need $\mathcal{O}(\alpha)$-space for the last α configurations of the automaton and $\mathcal{O}(m\max(\alpha, \beta))$-space to keep the last $\max(2\alpha, \beta)$ values of the sets $\mathcal{S}_{j-k}$, since the maximum cardinality of each set is m. Observe also that, although the size of the DAWG is linear in m, the *end-pos* $(\cdot)$ function can require $\mathcal{O}(m^2)$-space. Therefore, the total space complexity of the M-Sampling algorithm is $\mathcal{O}(m^2)$.

3.3. Average-case time analysis

Next, we evaluate the average time complexity of the M-Sampling algorithm, assuming a uniform distribution and independence of characters.

In our analysis, we do not include the time required for the computation of the DAWG and the *end-pos* $(\cdot)$ function, since they require $\mathcal{O}(m)$ and $\mathcal{O}(m^2)$ worst-case time, respectively, which turn out to be negligible if we assume m much smaller than n. We evaluate only the searching phase of the algorithm.

Given integers $1 \leq \alpha, \beta \leq m \leq n$ and an alphabet Σ of size $\sigma \geq 4$, for $j = 0, 1, \ldots, n-1$, we consider the following nonnegative random variables over the sample space of the pairs of strings $p, t \in \Sigma^*$ of length m and n, respectively:

- $X(j) \stackrel{\text{Def}}{=}$ the length $l_j \leq \alpha$ of the longest factor of p which is a suffix of t_j;
- $Y(j) \stackrel{\text{Def}}{=}$ the length $l_j^r \leq \beta$ of the longest factor of p^r which is a suffix of t_j;
- $Z(j) \stackrel{\text{Def}}{=} |\mathcal{S}_j|$, where we recall that $\mathcal{S}_j = \{0 \leq i \leq m-1 \mid t_i \stackrel{md}{\sqsupseteq} t_j\}$.

Then the run time of a call to the M-Sampling algorithm with parameters (p, t, α, β) is proportional to

$$\sum_{j=1}^{n-1} \left(Z(j-1) + \sum_{k=2}^{Y(j)} Z(j-k) + \left(\sum_{k=1}^{X(j)} Z(j-2k) + X(j) \right) \right), \tag{2}$$

where the external summation refers to the main **for**-loop (at line 6), and the three terms within it take care of the internal **for**-loops at lines 11, 15, and 22, in that order.

Let $E(\cdot)$ be the expectation function. The average-case complexity of the M-Sampling algorithm is thus the expectation of (2), which, by linearity, is equal to

$$\sum_{j=1}^{n-1} \left(E(Z(j-1)) + E\left(\sum_{k=2}^{Y(j)} Z(j-k) \right) + E\left(\sum_{k=1}^{X(j)} Z(j-2k) \right) + E(X(j)) \right). \tag{3}$$

Since $E(X(j)) \leq E(X(n-1))$, $E(Y(j)) \leq E(Y(n-1))$, $E(Z(j)) \leq E(Z(n-1))$, for $0 \leq j \leq n-1$,[2] and also $E(X(n-1)) = E(Y(n-1))$, by putting $X \stackrel{\text{Def}}{=} X(n-1)$ and $Z \stackrel{\text{Def}}{=} Z(n-1)$, expression (3) gets bounded from above by

$$\sum_{j=1}^{n-1} \left(E(Z) + E\left(\sum_{k=2}^{X} Z \right) + E\left(\sum_{k=1}^{X} Z \right) + E(X) \right). \tag{4}$$

Let Z_i and X_k be the indicator variables defined respectively, for $i = 0, \ldots, m-1$ and $k = 1, \ldots, m$, as

$$Z_i \stackrel{\text{Def}}{=} \begin{cases} 1 & \text{if } i \in \mathcal{S}_{n-1} \\ 0 & \text{otherwise} \end{cases} \quad \text{and} \quad X_k \stackrel{\text{Def}}{=} \begin{cases} 1 & \text{if } X \geq k \\ 0 & \text{otherwise,} \end{cases}$$

[2] In fact, for $j = m, \ldots, n-1$ all inequalities hold as equalities.

so that $Z = \sum_{i=0}^{m-1} Z_i$, $E(Z_i^2) = E(Z_i) = \Pr\{p_i \stackrel{md}{\sqsupseteq} t\}$, $X = \sum_{k=1}^{m} X_k$, and $E(X_k^2) = E(X_k) = \Pr\{X \geq k\}$. Then we have

$$\sum_{k=1}^{X} Z = XZ = \left(\sum_{k=1}^{m} X_k\right) \cdot \left(\sum_{i=0}^{m-1} Z_i\right) = \sum_{k=1}^{m} \sum_{i=0}^{m-1} X_k Z_i.$$

Therefore

$$E\left(\sum_{k=2}^{X} Z\right) \leq E\left(\sum_{k=1}^{X} Z\right) = \sum_{k=1}^{m} \sum_{i=0}^{m-1} E\left(X_k Z_i\right),$$

which yields the following upper bound for (4):

$$\sum_{j=1}^{n-1} \left(E(Z) + 2 \cdot \sum_{k=1}^{m} \sum_{i=0}^{m-1} E\left(X_k Z_i\right) + E(X)\right). \tag{5}$$

To estimate each of the terms $E(X_k Z_i)$ in (5), we use the well-known Cauchy–Schwarz inequality, which, in the context of expectations, assumes the form $|E(UV)| \leq \sqrt{E(U^2)E(V^2)}$, for any two random variables U and V such that $E(U^2)$, $E(V^2)$ and $E(UV)$ are all finite.

Then, for $1 \leq k \leq m$ and $0 \leq i \leq m-1$, we have

$$E(X_k Z_i) \leq \sqrt{E(X_k^2)E(Z_i^2)} = \sqrt{E(X_k)E(Z_i)}. \tag{6}$$

From (6), it then follows that (5) is bounded from above by

$$\sum_{j=1}^{n-1} \left(E(Z) + 2 \cdot \sum_{k=1}^{m} \sum_{i=0}^{m-1} \sqrt{E(X_k)E(Z_i)} + E(X)\right) = \sum_{j=1}^{n-1} \left(E(Z) + 2 \cdot \left(\sum_{k=1}^{m} \sqrt{E(X_k)}\right) \cdot \left(\sum_{i=0}^{m-1} \sqrt{E(Z_i)}\right) + E(X)\right). \tag{7}$$

To better understand (7), we evaluate the expectations $E(X)$ and $E(Z)$ and the sums $\sum_{k=1}^{m} \sqrt{E(X_k)}$ and $\sum_{i=0}^{m-1} \sqrt{E(Z_i)}$. To this purpose, it will be useful to estimate also the expectations $E(X_k) = \Pr\{X \geq k\}$, for $1 \leq k \leq m$, and $E(Z_i) = \Pr\{p_i \stackrel{md}{\sqsupseteq} t\}$, for $0 \leq i \leq m-1$.

Concerning $E(X_k) = \Pr\{X \geq k\}$, we reason as follows. Since $t[n-k..n-1]$ ranges uniformly over a collection of σ^k strings and there can be at most $\min(\sigma^k, m-k+1)$ distinct factors of length k in p, the probability $\Pr\{X \geq k\}$ that one of them matches $t[n-k..n-1]$ is at most $\min\left(1, \frac{m-k+1}{\sigma^k}\right)$, so that, for $k = 1, \ldots, m$, we have

$$E(X_k) \leq \min\left(1, \frac{m-k+1}{\sigma^k}\right). \tag{8}$$

Then, in view of (8), we have

$$E(X) = \sum_{i=0}^{m} i \cdot \Pr\{X = i\} = \sum_{i=1}^{m} \Pr\{X \geq i\} \leq \sum_{i=1}^{m} \min\left(1, \frac{m-i+1}{\sigma^i}\right). \tag{9}$$

Let $\bar{k}$ be the smallest integer $1 \leq k < m$ such that $\frac{m-k+1}{\sigma^k} < 1$. Then from (9) we have

$$\begin{aligned} E(X) &\leq \sum_{i=1}^{\bar{k}-1} 1 + \sum_{i=\bar{k}}^{m} \frac{m-i+1}{\sigma^i} \leq \bar{k} - 1 + (m-\bar{k}+1) \sum_{i=\bar{k}}^{m} \frac{1}{\sigma^i} \\ &< \bar{k} - 1 + \frac{\sigma}{\sigma-1} \cdot \frac{m-\bar{k}+1}{\sigma^{\bar{k}}} < \bar{k} - 1 + \frac{\sigma}{\sigma-1} < \bar{k} + 1. \end{aligned} \tag{10}$$

Since $\frac{m-(\bar{k}+1)+1}{\sigma^{\bar{k}+1}} \geq 1$, $\sigma^{\bar{k}+1} \leq m - (\bar{k}+1) + 1 \leq m-1$, so that $\bar{k}+1 < \log_\sigma m$. Then, from (10) and $\bar{k}+1 < \log_\sigma m$, we obtain

$$E(X) < \log_\sigma m. \tag{11}$$

Likewise, from (8) and $\bar{k}+1 < \log_\sigma m$, we have

$$\begin{aligned} \sum_{k=1}^{m} \sqrt{E(X_k)} &\leq \sum_{k=1}^{m} \sqrt{\min\left(1, \frac{m-k+1}{\sigma^k}\right)} = \sum_{k=1}^{\bar{k}-1} 1 + \sum_{k=\bar{k}}^{m} \sqrt{\frac{m-k+1}{\sigma^k}} \\ &\leq \bar{k} - 1 + \sqrt{m-\bar{k}+1} \cdot \sum_{k=\bar{k}}^{m} \frac{1}{\sqrt{\sigma^k}} < \bar{k} - 1 + \frac{\sqrt{\sigma}}{\sqrt{\sigma}-1} \cdot \sqrt{\frac{m-\bar{k}+1}{\sigma^{\bar{k}}}} \\ &< \bar{k} - 1 + \frac{\sqrt{\sigma}}{\sqrt{\sigma}-1} \leq \bar{k} + 1 < \log_\sigma m, \end{aligned} \tag{12}$$

where $\overline{k}$ is defined as above. Next, we estimate $E(Z_i) = \Pr\{p_i \overset{md}{\sqsupseteq} t\}$, for $0 \leq i \leq m-1$. Let us denote by $\mu(i)$ the number of distinct strings which have an *md*-match with a given string of length i and whose characters are pairwise distinct. Then $\Pr\{p_i \overset{md}{\sqsupseteq} t\} \leq \mu(i+1)/\sigma^{i+1}$. From the recursion

$$\begin{cases} \mu(0) = 1 \\ \mu(k+1) = \displaystyle\sum_{h=0}^{k} \mu(h) + \sum_{h=1}^{\lfloor \frac{k-1}{2} \rfloor} \mu(k-2h-1) \quad (\text{for } k \geq 0), \end{cases}$$

it is not hard to see that $\mu(i+1) \leq 3^i$, for $i = 0, 1, \ldots, m-1$, so that we have

$$E(Z_i) = \Pr\left\{p_i \overset{md}{\sqsupseteq} t\right\} \leq \frac{3^i}{\sigma^{i+1}}. \tag{13}$$

Then, concerning $E(Z)$, from (13), we have

$$E(Z) = E\left(\sum_{i=0}^{m-1} Z_i\right) = \sum_{i=0}^{m-1} E(Z_i) \leq \sum_{i=0}^{m-1} \frac{3^i}{\sigma^{i+1}} < \frac{1}{\sigma} \cdot \frac{1}{1-\frac{3}{\sigma}} = \frac{1}{\sigma-3} \leq 1 \tag{14}$$

(we recall that we have assumed that $\sigma \geq 4$). Likewise, from (13), we have

$$\sum_{i=0}^{m-1} \sqrt{E(Z_i)} \leq \sum_{i=0}^{m-1} \sqrt{\frac{3^i}{\sigma^{i+1}}} < \frac{1}{\sqrt{\sigma}} \cdot \frac{1}{1-\sqrt{\frac{3}{\sigma}}} = \frac{1}{\sqrt{\sigma}-\sqrt{3}} < 4. \tag{15}$$

From (14), (11), (12), and (15), it then follows that (7) is bounded from above by $(n-1) \cdot (9 \log_\sigma m + 1)$, yielding a $\mathcal{O}(n \log_\sigma m)$ average-time complexity for the M-Sampling algorithm.

4. A filter-based approach: the Addition–Counting–Filter algorithm

In this section, we present a very efficient algorithm for the approximate string matching problem allowing for inversions of factors and translocations of equal length adjacent factors, which works in $\mathcal{O}(nm \max(\alpha, \beta))$-worst case time complexity and $\mathcal{O}(n)$-average case time complexity, under the assumption of a uniform distribution and independence of characters.

Our algorithm, named Addition–Counting–Filter, improves the searching strategy introduced in the M-Sampling algorithm by making use of an efficient filter method along the same lines of the GFG algorithm, recently presented by Grabowski et al. [10]. Such a filtering technique, usually referred to as the *counting filter*, is well known [11,12,1,4] and has been used for the k-mismatches and k-differences string matching problem.

The idea behind the counting–filter technique is based upon the simple observation that (in our problem), if a pattern p has an occurrence (possibly involving inversions and translocations) starting at position i of a text t, then the $|p|$-substring $t[i..i+|p|-1]$ of the text is a permutation of the pattern p.

As before, let p and t be strings of length m and n, respectively, over a common alphabet $\Sigma = \{c_0, \ldots, c_{\sigma-1}\}$ of size σ. Much as the GFG algorithm, the Addition–Counting–Filter algorithm firstly identifies the set $\Gamma_{p,t}$ of all candidate positions i in the text such that the substring $t[i..i+m-1]$ is a permutation of p and, subsequently, for each such candidate position $i \in \Gamma_{p,t}$, it executes a verification procedure to check whether p and $t[i..i+m-1]$ match, up to non-overlapping inversions and translocations.

The GFG algorithm maintains in constant time the size δ of the symmetric difference of the multisets of the characters occurring in the current text window and of those occurring in the pattern, respectively. When $\delta = 0$, a candidate match is found. In contrast, the Addition–Counting–Filter algorithm maps the two multisets of our interest into natural numbers, using a hash function h that allows for very fast updates. In this case, a candidate match is found when the hash values associated to the current window and to the pattern are equal.

The hash function approach is based on the following elementary fact.

Lemma 3. *Let $\Sigma = \{c_0, c_1, \ldots, c_{\sigma-1}\}$ be an alphabet of size $|\Sigma| = \sigma$, let $m > 1$ be an integer, and let $h : \Sigma \to \mathbb{N}$ be the mapping $h(c_i) \overset{\text{Def}}{=} m^i$, for $i = 0, \ldots, \sigma-1$. Then, for any two distinct k-multicombinations (i.e., k-combinations with repetitions) φ_1 and φ_2 from the set Σ, with $1 \leq k \leq m$, we have*

$$\sum_{c \in \varphi_1} h(c) \neq \sum_{c \in \varphi_2} h(c). \tag{16}$$

Proof. Let $m > 1$ be a given integer, and let $\Sigma^{[k]}$ be the collection of all k-multicombinations from the set Σ, for $k \geq 1$. We prove (16) by induction on $k = 1, 2, \ldots, m$.

For the base case, let φ_1 and φ_2 be two distinct 1-multicombinations in $\Sigma^{[1]}$. Then we have $\varphi_1 = \{c\}$ and $\varphi_2 = \{c'\}$, with c, c' distinct characters in Σ. Plainly, $h(c) \neq h(c')$, so (16) holds in the base case.

Suppose now that (16) holds for $(k-1)$-multicombinations, with $2 \leq k \leq m$, and prove it for k-multicombinations. Thus, let φ_1 and φ_2 be two distinct k-multicombinations in $\Sigma^{[k]}$. First observe that, if $\varphi_1 \cap \varphi_2 \neq \emptyset$, then, for any $\overline{c} \in \varphi_1 \cap \varphi_2$, the multisets $\varphi_1 - \{\overline{c}\}$ and $\varphi_2 - \{\overline{c}\}$ are two distinct $(k-1)$-multicombinations from Σ, so that, by inductive hypothesis, we have

$$\sum_{c\in\varphi_1} h(c) = h(\overline{c}) + \sum_{c\in\varphi_1-\{\overline{c}\}} h(c) \neq h(\overline{c}) + \sum_{c\in\varphi_2-\{\overline{c}\}} h(c) = \sum_{c\in\varphi_2} h(c).$$

Thus we can assume that $\varphi_1 \cap \varphi_2 = \emptyset$. Let $\mu \stackrel{\text{Def}}{=} \max\{i : c_i \in \varphi_1\}$ and $\nu \stackrel{\text{Def}}{=} \max\{i : c_i \in \varphi_2\}$, and, without any loss in generality, assume that $\nu < \mu$. Then, since $k \leq m$, we have

$$\sum_{c\in\varphi_1} h(c) = h(c_\mu) + \sum_{c\in\varphi_1-\{c_\mu\}} h(c) > h(c_\mu) = m^\mu \geq \sum_{i=1}^{k} m^\nu \geq \sum_{c\in\varphi_2} h(c),$$

proving (16) also in the inductive case. □

Inspired by the previous lemma, the Addition–Counting–Filter algorithm precomputes a function $h : \Sigma \to \mathbb{N}$, defined as $h(c_i) \stackrel{\text{Def}}{=} m^i$, for $i = 0, \ldots, \sigma - 1$, where m is the length of the pattern.[3] The map h can be naturally extended to m-strings by putting $h(q) \stackrel{\text{Def}}{=} \sum_{i=0}^{m-1} h(q[i])$, for any string q of length m. The hash value $\lambda \stackrel{\text{Def}}{=} h(p)$ of the pattern p is then precomputed. Likewise, during the searching phase, the hash value $\gamma_i \stackrel{\text{Def}}{=} h(t[i..\, i+m-1])$ is computed for each window $t[i..\, i+m-1]$ of the text t, with $0 \leq i \leq n-m$. The set $\Gamma_{p,t}$ of all candidate positions in the text is then $\Gamma_{p,t} \stackrel{\text{Def}}{=} \{i : 0 \leq i \leq n-m \text{ and } \gamma_i = \lambda\}$. Observe that $\gamma_{i+1} = \gamma_i - h(t[s]) + h(t[s+m])$, so that γ_{i+1} can be computed in constant time from γ_i. Thus the set $\Gamma_{p,t}$ can be computed in $\mathcal{O}(n)$-time.

Example 4. Let $p = agcgt$ be an input pattern of length 5, and let $t = agacatgcgatgcc$ be a text, both over the alphabet $\Sigma = \{a, c, g, t\}$ of size 4. The algorithm precomputes the function $h : \Sigma \to \mathbb{N}$, defined as $h(a) = 1$, $h(c) = 5$, $h(g) = 25$ and $h(t) = 125$. Then the hash value of the pattern p $\lambda = h(p)$ is equal to $h(a) + h(g) + h(c) + h(g) + h(t) = 181$. The set $\Gamma_{p,t}$ of all candidate positions in the text is then $\Gamma_{p,t} = \{8, 9, 10, 11, 12\}$.

It has been observed experimentally that the collision problem for the hash function h is negligible. In fact, it turns out that in several practical cases the above hash function h is injective. As will be shown in Section 5, the Addition–Counting–Filter algorithm is up to 40% faster than GFG, since updating hash values requires fewer operations than updating the map δ of the GFG algorithm.

For each candidate position $i \in \Gamma_{p,t}$ (such that $\gamma_i = \lambda$), a verification procedure is run to check whether a match occurs at position i, up to translocations and inversions. The verification procedure used by the Addition–Counting–Filter algorithm is the same procedure Verify used by the GFG algorithm, which takes $\mathcal{O}(m \max(\alpha, \beta))$-time and $\mathcal{O}(\max(\alpha, \beta))$-space (see [10] for more details). Thus the total worst-case time complexity of the Addition–Counting–Filter algorithm is $\mathcal{O}(nm \max(\alpha, \beta))$, and it uses only $\mathcal{O}(\max(\alpha, \beta, \sigma))$ additional space.

4.1. Average-case time analysis

In the following analysis, we assume a uniform distribution and independence of characters.

In [10], Grabowski et al. proved a linear average-time bound for the GFG algorithm when $m = \omega(\sigma^{\mathcal{O}(1)})$ and $\sigma = \Omega(\log m / \log\log^{1-\varepsilon} m)$. Here, we establish a similar result for our algorithm, but with a less strict bound on σ.

Let $\Pr\{i \in \Gamma_{p,t}\}$ be the probability that position i is a candidate position to be verified, for a given random text t and pattern p of length n and m, respectively, and with $n \geq m$, over an alphabet of size σ. Since the preprocessing phase of our algorithm and each call to procedure Verify take $\mathcal{O}(\sigma)$-time and $\mathcal{O}(m^2)$-time, respectively, the average time complexity of the Addition–Counting–Filter algorithm can be expressed as

$$T(n, m, \sigma) = \mathcal{O}(\sigma) + \sum_{i=0}^{n-m} \Pr\{i \in \Gamma_{p,t}\} \cdot \mathcal{O}(m^2). \tag{17}$$

Thus, to obtain a linear average-time bound, it is enough to show that $\mathcal{O}(1/m^2)$ bounds the probability of finding a permutation of the pattern.

[3] It is understood that for an architecture, say, at 64 bits, all operations will take place modulo 2^{64}.

For simplicity, let us assume that σ divides m, and let $k \stackrel{\text{Def}}{=} m/\sigma$. For each text position i, with $0 \leq i \leq n-m$, the probability that the m-substring of the text, beginning at position i, is a permutation of the pattern p is exactly

$$\Pr\{i \in \Gamma_{p,t}\} = \frac{\binom{m}{f_0}\binom{m-f_0}{f_1}\binom{m-f_0-f_1}{f_2}\cdots\binom{f_{\sigma-1}}{f_{\sigma-1}}}{\sigma^m}, \tag{18}$$

where f_j is the number of occurrences of c_j in p, for $j = 0, 1, \ldots, \sigma-1$.

The right-hand side of (18), subject to the constraint $\sum_{j=0}^{\sigma-1} f_j = m$, attains its maximum when all f_j's are equal to k. Thus

$$\Pr\{s \in \Gamma_{p,t}\} \leq \frac{\binom{m}{k}\binom{m-k}{k}\binom{m-2k}{k}\cdots\binom{k}{k}}{\sigma^m} = \frac{m!}{(k!)^\sigma \sigma^m}.$$

We use Stirling's approximation for $m!$ and $k!$, recalling that $k = m/\sigma$:

$$\frac{m!}{(k!)^\sigma \sigma^m} = \Theta\left(\frac{\sqrt{2\pi m}(m/e)^m}{\left(\sqrt{2\pi m/\sigma}(m/(e\sigma))^{m/\sigma}\right)^\sigma \sigma^m}\right) = \Theta\left(\frac{\sqrt{2\pi m}}{\left(\sqrt{2\pi(m/\sigma)}\right)^\sigma}\right).$$

Since $\sqrt{2\pi}/(\sqrt{2\pi})^\sigma \leq 1$, we have

$$\frac{\sqrt{2\pi m}}{\left(\sqrt{2\pi(m/\sigma)}\right)^\sigma} = \mathcal{O}\left(\frac{\sqrt{m}}{\left(\sqrt{m/\sigma}\right)^\sigma}\right) = \mathcal{O}\left(\frac{\sigma^{\sigma/2}}{m^{(\sigma-1)/2}}\right).$$

Let $m \geq \sigma^c$ and $\sigma \geq \frac{5c}{c-1}$, for some constant $c > 1$. We have $\sigma \leq c(\sigma-5)$, so that $\sigma^\sigma \leq \sigma^{c(\sigma-5)} \leq m^{\sigma-5}$. Therefore $\sigma^{\sigma/2}/m^{(\sigma-1)/2} \leq 1/m^2$, since, plainly, $m^{(\sigma-5)/2} = m^{(\sigma-1)/2}/m^2$.

Thus, we have $\Pr\{i \in \Gamma_{p,t}\} = \mathcal{O}(1/m^2)$, for any $\sigma \geq \frac{5c}{c-1}$ and sufficiently large $m \geq \sigma^c$, with $c > 1$. Under such assumptions, from (17) we easily obtain

$$\begin{aligned} T(n,m,\sigma) &= \mathcal{O}(\sigma) + (n-m+1)\cdot\mathcal{O}(1/m^2)\cdot\mathcal{O}(m^2) \\ &= \mathcal{O}(\sigma+n) = \mathcal{O}(n), \end{aligned}$$

proving a linear average-time bound for the Addition–Counting–Filter algorithm.

5. Experimental evaluation

In this section, we present some experimental results which allow us to compare in terms of their running times the M-Sampling, GFG, and Addition–Counting–Filter algorithms (ACF). In our evaluation, in the case of the M-Sampling algorithm, we used an efficient bit-parallel implementation (BPMS) described in [3], which obtains better results in practice. Moreover, two variants of the GFG algorithm have been evaluated: the first one uses a verification phase based on a $\mathcal{O}(m^2)$-time dynamic programming procedure (GFG1), while the second one uses a verification phase based on the M-Sampling algorithm (GFG2).

All algorithms were implemented in the C programming language and were compiled with the `GNU C Compiler`, using the optimization options `-O3`. The tests were performed on a MacBook Pro with a 2 GHz Intel Core i7 processor, a 4 GB 1333 MHz DDR3 memory and a 64-bit word size. As input files, we used six random texts with a uniform distribution of characters and alphabet size $\sigma = 4, 8, 16, 32, 64, 128$, respectively. Moreover, we used a DNA sequence (dna) and a protein sequence (prot). All texts have size 1 Mb and were extracted from the smart tool [9]. For each input file, we generated sets of 500 patterns of fixed length m, randomly extracted from the text, for $m = 8, 16, 32, 64$. For each set of patterns, we calculated the mean over the running times of the 500 runs. Table 1 reports the running times obtained in our experimental results. In the case of the Addition–Counting–Filter algorithm, an asterisk symbol (*) indicates those runs in which hash collisions occurred.

From Table 1, it turns out that the filtering strategy is more effective than the BPM-Sampling algorithm and allows one to dramatically speed up the computation of the matches of a given pattern. Moreover, the Addition–Counting–Filter algorithm always obtains the best results and is up to 40% faster than the GFG algorithms. It also turns out that the collision problem is negligible. In particular, for each text position, the Addition–Counting–Filter algorithm incurs an average number of collisions less than 1.4×10^{-4}, in all cases, and less than 1.0×10^{-5}, in most of the cases.

Concerning the GFG algorithm, we observe that, for very small alphabets, the GFG2 variant, based on M-Sampling, is faster than the GFG1 variant, based on the dynamic programming verification, while in the other cases the two algorithms have much the same speed.

Finally, we observe that the rate of growth of the BPM-Sampling algorithm matches the estimated $\mathcal{O}(n \log_\sigma m)$ average-time complexity while the Addition–Counting–Filter algorithm exhibits the expected linear behavior.

Table 1
Running times of tested algorithms under various conditions. Running times are expressed in milliseconds. In the case of the Addition–Counting–Filter algorithm (ACF), an asterisk symbol (*) indicates those runs in which hash collisions occurred.

ALGO	m	**8**	**16**	**32**	**64**	m	**8**	**16**	**32**	**64**
BPMS	$\sigma = 4$	41.13	51.78	65.35	80.67	$\sigma = 32$	21.1	25.8	29.9	31.2
GFG1		16.41	22.58	24.03	24.94		7.57	7.59	7.68	7.86
GFG2		12.78	12.12	11.56	11.10		7.58	7.60	7.60	7.59
ACF		**9.40**	**8.70**	**8.17**	**7.73**		**4.24**	**4.29***	**4.41***	**4.82***
BPMS	$\sigma = 8$	31.7	37.1	44.8	54.7	$\sigma = 64$	18.4	21.0	25.7	29.9
GFG1		7.82	7.75	7.80	7.99		7.46	7.47	7.57	7.72
GFG2		7.87	7.79	7.71	7.73		7.48	7.49	7.51	7.51
ACF		**4.34**	**4.28**	**4.35**	**4.57***		**4.21**	**4.26***	**4.36***	**4.83***
BPMS	$\sigma = 16$	26.0	30.5	32.9	37.9	$\sigma = 128$	17.1	18.3	20.9	26.0
GFG1		7.63	7.66	7.79	7.91		7.40	7.42	7.49	7.63
GFG2		7.64	7.63	7.63	7.68		7.43	7.44	7.43	7.47
ACF		**4.22**	**4.26***	**4.36***	**4.58***		**4.21**	**4.26***	**4.35***	**4.71***
BPMS	**dna**	40.8	50.6	64.0	79.3	**prot**	25.1	29.9	32.6	36.9
GFG1		12.31	11.64	10.72	10.22		7.66	7.65	7.74	7.88
GFG2		16.16	20.91	20.33	19.86		7.63	7.61	7.62	7.66
ACF		**8.94**	**8.29**	**7.33**	**6.83**		**4.22**	**4.26***	**4.36***	**4.55***

Appendix. The codes

Below, we show the M-Sampling algorithm (on the left), and the DAWG state update algorithm Dawg-Delta and the filter-based algorithm Addition–Counting–Filter (on the right). For the code of the verification procedure Verify, the reader is referred to [10]. Notice that the function $s\ell^*$ in procedure Dawg-Delta denotes the improved suffix link [6].

```
M-Sampling (p, t, α, β, A, A')
 1. m ← |p|,   n ← |t|
 2. (q_0, l_0) ← Dawg-Delta(root_A, 0, α, t[0], A)
 3. (q'_0, l'_0) ← Dawg-Delta(root_A', 0, β, t[0], A')
 4. S_0 ← ∅
 5. if p[0] = t[0] then S_0 ← {0}
 6. for j ← 1 to n − 1 do
 7.    (q_j, l_j) ← Dawg-Delta(q_{j−1}, l_{j−1}, α, t[j], A)
 8.    (q'_j, l'_j) ← Dawg-Delta(q'_j, l'_j, β, t[j], A')
 9.    S_j ← ∅
10.    if p[0] = t[j] then S_j ← {0}
11.    for i ∈ S_{j−1} do
12.       if i < m − 1 and p[i + 1] = t[j] then
13.          S_j ← S_j ∪ {i + 1}
14.    u ← q'_j
15.    for k ← l'_j downto 2 do
16.       for i ∈ S_{j−k} ∪ {−1} do
17.          if (m − 2 − i) ∈ end-pos^r(u) then
18.             S_j ← S_j ∪ {i + k}
19.       if k = len(sℓ_A'(u)) + 1 then
20.          u ← sℓ_A'(u)
21.    last ← 0,   u ← q_j
22.    for k ← l_j downto 1 do
23.       if k ≤ j and k ≤ l_{j−k} then
24.          if last = k + 1 then
25.             while u' ≠ root_A
                and k − 1 ≤ len(sℓ_A(u')) do
26.                u' ← sℓ_A(u')
27.          u' ← δ_A(u', t[j − k])
28.       else
29.          u' ← q_{j−k}
30.          while k ≤ len(sℓ_A(u')) do
31.             u' ← sℓ_A(u')
32.       last ← k
33.       for i ∈ S_{j−2k} ∪ {−1} do
34.          if (i + k) ∈ end-pos(u)
             and (i + 2k) ∈ end-pos(u') then
35.             S_j ← S_j ∪ {i + 2k}
36.       if k = len(sℓ_A(u)) + 1
37.          then p ← sℓ_A(u)
38.    if (m − 1) ∈ S_j then Output(j)
```

```
Dawg-Delta(q, l, k, c, B)
 1. if l = k then
 2.    l ← l − 1
 3.    if len(sℓ_B(q)) = l
 4.       then q ← sℓ_B(q)
 5. if δ_B(q, c) = nil then
 6.    do
 7.       q ← sℓ*_B(q)
 8.    while q ≠ nil and δ_B(q, c) = nil
 9.    if q = nil then
10.       l ← 0, q ← root_B
11.    else l ← len(q) + 1
12.          q ← δ_B(q, c)
13. else l ← l + 1
14.       q ← δ_B(q, c)
15. return (q, l)
```

```
Addition–Counting–Filter (p, m, t, n, α, β)
1. for i ← 0 to σ − 1 do h[c_i] ← m^i
2. λ ← δ ← 0
3. for s ← 0 to m − 1 do
4.    λ ← λ + h[p[s]]
5.    δ ← δ + h[t[s]]
6. for s ← 0 to n − m do
7.    if δ = λ then
8.       Verify(p, m, t, s, α, β)
9.    δ ← δ − h[t[s]] + h[t[s + m]]
```

References

[1] R.A. Baeza-Yates, G. Navarro, New and faster filters for multiple approximate string matching, Random Struct. Algorithms 20 (1) (2002) 23–49.
[2] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, Theoret. Comput. Sci. 40 (1985) 31–55.
[3] D. Cantone, S. Faro, E. Giaquinta, Approximate string matching allowing for inversions and translocations, in: Proc. of the Prague Stringology Conference 2010, pp. 37–51.
[4] Domenico Cantone, Salvatore Cristofaro, Simone Faro, Efficient string-matching allowing for non-overlapping inversions, Theoret. Comput. Sci. 483 (2013) 85–95.
[5] M. Crochemore, Transducers and repetitions, Theoret. Comput. Sci. 45 (1) (1986) 63–86.
[6] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[7] P. Cull, T. Hsu, Recent Advances in the Walking Tree Method for Biological Sequence Alignment EUROCAST, 2003, pp. 349–359.
[8] F. Damerau, A technique for computer detection and correction of spelling errors, Commun. ACM 7 (3) (1964) 171–176.
[9] S. Faro, T. Lecroq, Smart: a string matching algorithm research tool, University of Catania and University of Rouen, 2011. http://www.dmi.unict.it/~faro/smart/.
[10] S. Grabowski, S. Faro, E. Giaquinta, String matching with inversions and translocations in linear average time (most of the time), Inform. Process. Lett. 111 (11) (2011) 516–520.
[11] R. Grossi, F. Luccio, Simple and efficient string matching with k mismatches, Inform. Process. Lett. 33 (3) (1989) 113–120.
[12] P. Jokinen, J. Tarhio, E. Ukkonen, A comparison of approximate string matching algorithms, Softw. Pract. Exp. 26 (12) (1996) 1439–1458.
[13] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions and reversals, Sov. Phys. Dokl. 10 (1966) 707–710.
[14] G. Navarro, A guided tour to approximate string matching, ACM Comput. Surv. 33 (1) (2001) 31–88.
[15] A.F. Vellozo, C.E.R. Alves, A. Pereira do Lago, Alignment with non-overlapping inversions in $O(n^3)$-time, in: WABI, 2006, pp. 186–196.