Theoretical Computer Science 408 (2008) 106–115

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Incremental discovery of the irredundant motif bases for all suffixes of a string in $O(n^2 \log n)$ time☆

Alberto Apostolico[a,b,*], Claudia Tagliacollo[a]

[a] *Accademia Nazionale dei Lincei, Rome, Italy*

[b] *College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30318, USA*

ARTICLE INFO

Keywords:
Design and analysis of algorithms
Pattern matching
Motif discovery
Irredundant motif
Base

ABSTRACT

Compact bases formed by motifs called "irredundant" and capable of generating all other motifs in a sequence have been proposed in recent years and successfully tested in tasks of biosequence analysis and classification. Given a sequence s of n characters drawn from an alphabet Σ, the problem of extracting such a base from s had been previously solved in time $O(n^2 \log n \log \mid \Sigma \mid)$ and $O(\mid \Sigma \mid n^2 \log^2 n \log \log n)$, respectively, using the FFT-based string searching by Fischer and Paterson. More recently, a solution on binary strings taking time $O(n^2)$ without resorting to the FFT was also proposed. In the present paper, we considered the problem of incrementally extracting the bases of all suffixes of a string. This problem was solved in a previous work in time $O(n^3)$. A much faster incremental algorithm is described here, which takes time $O(n^2 \log n)$ for binary strings. Although this algorithm does not make use of the FFT, its performance is comparable to the one exhibited by the previous FFT-based algorithms involving the computation of only one base. The implicit representation of a single base requires $O(n)$ space, whence for finite alphabets the proposed solution is within a $\log n$ factor from optimality.

1. Introduction

In [10,12], innovative notions of pattern saturation and irredundancy have been formulated that conjugate statistical and syntactic descriptors in a tightly intertwined fashion, thereby reducing the roster of candidates to be extracted and evaluated in applications of pattern discovery (see, e.g., [13]). A central tool for these developments is the algebraic-flavored notion of a base of "irredundant" motifs, which denotes a compact subset of the set of all patterns that can be generated by appropriate combination of any other pattern in the set. To date, few algorithms have been produced for the extraction of a base from a sequence. Algorithms based on the landmark, FFT-driven string searching algorithm by Fischer and Paterson [7] were given in [11] and (for a slightly different notion) [9], and exhibit time bounds of $O(n^2 \log n \log |\Sigma|)$ and $O(|\Sigma| n^2 \log^2 n \log \log n)$, respectively, for an input sequence s of n characters drawn from an alphabet Σ. More recently, a solution taking time $O(n^2)$ on a binary string without resorting to the FFT has emerged [5]. Here we consider the problem of extracting the bases of all suffixes of a string incrementally. This problem was solved in previous work in time $O(n^3)$ [3]. A faster incremental algorithm is described here, which takes time $O(n^2 \log n)$ on a binary string. Albeit in a non-trivial way, the algorithm builds on a criterion developed in [5] for testing the occurrences of certain "autocorrelations" of the input string, as well as on bounds on the sizes of the occurrence lists of motifs in a base. Although this algorithm does not make use of the FFT, its performance is comparable to the one exhibited by the previous FFT-based algorithms for computing only one base. As a by-product of

☆ An extended abstract related to this work appears in the proceedings of WABI 07.

* Corresponding author at: Accademia Nazionale dei Lincei, Rome, Italy.
E-mail addresses: axa@cc.gatech.edu, axa@dei.unipd.it (A. Apostolico).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.08.002

possible independent interest, a bookkeeping is maintained of the occurrence lists of all suffixes of the autocorrelations of s – a $\Theta(n^3)$ size collection – within the $O(n^2 \log n)$ time bound. Finally, we note that while the explicit description of a base may require $\Omega(n^2)$ worst-case space, its implicit description only requires space $O(n)$, whence with finite alphabets the algorithm described in the present paper is within a $\log n$ factor from optimality when this kind of description is adopted.

The paper is organized as follows. Upon presenting basic definitions and properties, we highlight our constant-time occurrence testing. We then describe a basic tool used in our algorithms, which speeds up the computation of the occurrence lists of all patterns needed in the computation of a base. The application of these constructs to the algorithms are then detailed. The paper is self-contained and notations largely conform to those adopted in [4,3,5].

2. Preliminaries

Let Σ be a finite alphabet of *solid* characters, and let '$\bullet$' $\notin \Sigma$ denote a don't-care character, that is, a wildcard matching any of the characters in $\Sigma \cup \{\bullet\}$. A *pattern* is a string over $\Sigma \cup \{\bullet\}$ containing at least one *solid* character. We use σ to denote a generic character from Σ. For characters σ_1 and σ_2, we write $\sigma_1 \preceq \sigma_2$ if and only if σ_1 is a don't care or $\sigma_1 = \sigma_2$.

Given two patterns p_1 and p_2 with $|p_1| \leq |p_2|$, $p_1 \preceq p_2$ holds if $p_1[j] \preceq p_2[j]$, $1 \leq j \leq |p_1|$. We also say in this case that p_1 is a *sub-pattern* of p_2, and that p_2 *implies* or *extends* p_1. If, moreover, the first characters of p_1 and p_2 are matching solid characters, then p_1 is also called a *prefix* of p_2. For example, let $p_1 = ab \bullet \bullet e$, $p_2 = ak \bullet \bullet e$ and $p_3 = abc \bullet e \bullet g$. Then $p_1 \preceq p_3$, and $p_2 \not\preceq p_3$. Note that the $\preceq$ relation is transitive. The following operators are further introduced.

Definition 1 ($\oplus$). Let $\sigma_1, \sigma_2 \in \Sigma \cup \{\bullet\}$.

$$\sigma_1 \oplus \sigma_2 = \begin{cases} \sigma_1, & \text{if } \sigma_1 = \sigma_2 \\ \bullet, & \text{if } \sigma_1 \neq \sigma_2. \end{cases}$$

Definition 2 (*Extended* $\oplus$). Given patterns p_1 and p_2, $p_1 \oplus p_2 = p_1[i] \oplus p_2[i]$, $\forall i$, $1 \leq i \leq \min\{|p_1|, |p_2|\}$).

Definition 3 (*Consensus, Meet*). Given the patterns p_1, p_2, the *consensus* of p_1 and p_2 is the pattern $p = p_1 \oplus p_2$. Deleting all leading and trailing don't cares from p yields the *meet* of p_1 and p_2, denoted by $[p_1 \oplus p_2]$.

For instance, $aac\bullet tgcta \oplus caact\bullet cat = \bullet a\bullet\bullet t\bullet c\bullet\bullet$, and $[aac\bullet tgcta \oplus caact\bullet cat] = a\bullet\bullet t\bullet c$. Note that a meet may be the empty word. Let now $s = s_1 s_2 \ldots s_n$ be a sequence of n over Σ. We use suf_i to denote the suffix $s_i s_{i+1} \ldots s_n$ of s.

Definition 4 (*Autocorrelation*). A pattern p is an *autocorrelation* of s if p is the meet of s and one of its suffixes, i.e., if $p = [s \oplus suf_i]$ for some $1 < i \leq n$.

For instance, the autocorrelations of $s = acacacacabaaba$ are: $\overline{m}_1 = s \oplus suf_2 = s \oplus suf_{11} = s \oplus suf_{14} = a$, $\overline{m}_2 = s \oplus suf_3 = acacaca\bullet a\bullet\bullet a$, $\overline{m}_3 = s \oplus suf_4 = aba$, $\overline{m}_4 = s \oplus suf_5 = acaca\bullet a$, $\overline{m}_5 = s \oplus suf_6 = s \oplus suf_9 = s \oplus suf_8 = s \oplus suf_{10} = s \oplus suf_{12} = a\bullet a$, $\overline{m}_6 = s \oplus suf_7 = aca\bullet a$.

Definition 5 (*Motif*). For a sequence s and positive integer k, $k \leq |s|$, a k-*motif* of s is a pair $(m, \mathcal{L}_m)$, where m is a pattern such that $|m| \geq 1$ and $m[1]$, $m[|m|]$ are solid characters, and $\mathcal{L}_m = (l_1, l_2, \ldots, l_q)$ with $q \geq k$ is the exhaustive list of the starting position of all occurrences of m in s.

Note that both components concur in this definition: two distinct location lists correspond to two distinct motifs even if the pattern component is the same; conversely, motifs that have different location lists are considered to be distinct. In what follows, we will denote motifs by their pattern component alone, when this causes no confusion. Consider $s = abcdabcd$. Using the definition of motifs, the different 2-motifs are as follows: $m_1 = ab$ with $\mathcal{L}_{m_1} = \{1, 5\}$, $m_2 = bc$ with $\mathcal{L}_{m_2} = \{2, 6\}$, $m_3 = cd$ with $\mathcal{L}_{m_3} = \{3, 7\}$, $m_4 = abc$ with $\mathcal{L}_{m_4} = \{1, 5\}$, $m_5 = bcd$ with $\mathcal{L}_{m_5} = \{2, 6\}$ and $m_6 = abcd$ with $\mathcal{L}_{m_6} = \{1, 5\}$.

Given a motif m, a *sub-motif* of m is any motif m' that may be obtained from m by (i) changing one or more solid characters into don't care, (ii) eliminating all resulting don't-cares that precede the first remaining solid character or follow the last one, and finally (iii) updating $\mathcal{L}_m$ in order to produce the (possibly, augmented) list $\mathcal{L}_{m'}$. We also say that m is a *condensation* for any of its sub-motifs.

We are interested in motifs for which any condensation would disrupt the list of occurrences. A motif with this property has been called *maximal* or *saturated*. In intuitive terms, a motif m is maximal or saturated if we cannot make it more specific while retaining the cardinality of the list $\mathcal{L}_m$ of its occurrences in s. More formally, in a saturated motif m, no don't-care of m can be replaced by a solid character that appears in all the locations in $\mathcal{L}_m$, nor can m be expanded by a pattern prefix or suffix without affecting the cardinality of $\mathcal{L}_m$.

A motif $(m, \mathcal{L}_m)$ is redundant if m and its location list $\mathcal{L}_m$ can be deduced from the other motifs *without* knowing the input string s. Trivially, every unsaturated motif is redundant. As it turns out, however, saturated motifs may be redundant, too. More formally:

Definition 6. A saturated motif $(m, \mathcal{L}_m)$, is *redundant* if there exist saturated motifs $(m_i, \mathcal{L}_{m_i})$ $1 \leq i \leq t$, such that

$$\mathcal{L}_m = (\mathcal{L}_{m_1} + d_1) \cup (\mathcal{L}_{m_2} + d_2) \cup \cdots \cup (\mathcal{L}_{m_p} + d_t)$$

with $0 \leq d_j < |m_j|$.

Here and in the following, $(\mathcal{L}+d)$ is used to denote the list that is obtained by adding a uniform offset d to every element of $\mathcal{L}$. For instance, the saturated motif $m_1 = a\bullet a$ is redundant in $s = acacacacabaaba$, since $\mathcal{L}_{m_1} = \{1, 3, 5, 7, 9, 12\} = (\mathcal{L}_{m_2}) \cup (\mathcal{L}_{m_3}) \cup (\mathcal{L}_{m_4} + 1)$ where $m_2 = acac$, $m_3 = aba$ and $m_4 = ca\bullet a$.

Saturated motifs enjoy some special properties.

Property 1. *Let $(m_1, \mathcal{L}_{m_1})$ and $(m_2, \mathcal{L}_{m_2})$ be saturated motifs. Then,*

$$m_1 = m_2 \;\Leftrightarrow\; \mathcal{L}_{m_1} = \mathcal{L}_{m_2}.$$

We also know that, given a generic pattern m, it is always possible to determine its occurrence list in any sequence s. With a saturated motif m, however, it is possible in addition to retrieve the structure of m from the sole list $\mathcal{L}_m$ in s, simply by taking:

$$m = \left[\bigoplus_{i\in\mathcal{L}_m} suf_i\right].$$

We also have:

Property 2. *Let $(m_1, \mathcal{L}_{m_1})$, $(m_2, \mathcal{L}_{m_2})$ be motifs of s. Then,*

$$m_1 \preceq m_2 \Leftrightarrow \mathcal{L}_{m_2} \subseteq \mathcal{L}_{m_1}.$$

Similarly:

Property 3. *Let $(m, \mathcal{L}_m)$ be a saturated motif of s. Then $\forall L \subseteq \mathcal{L}_m$ it is*

$$m \preceq \left[\bigoplus_{k\in L} suf_k\right].$$

Let now $suf_i(m)$ denote the ith suffix of m.

Definition 7 (*Coverage*). The occurrence at j of m_1 is *covered* by m_2 if $m_1 \preceq suf_i(m_2), j \in \mathcal{L}_{m_2} + i - 1$ for some $suf_i(m_2)$.

For instance, $\overline{m}_6 = aca\bullet a$ with $\mathcal{L}_{\overline{m}_6} = \{1, 3, 5, 7\}$ is covered at position 5 by $\overline{m}_2 = acacaca\bullet a\bullet\bullet a$, $\mathcal{L}_{\overline{m}_2} = \{1, 3\}$. In fact, let m' be ith suffix of $\overline{m}_3$ with $i = 5$, that is, $m' = aca\bullet a\bullet\bullet a$. Then $5 \in \mathcal{L}_{\overline{m}_2} + 4$ and $\overline{m}_6 \prec m'$, which together lead to conclude that $\overline{m}_6$ is covered at 5 by $\overline{m}_2$. An alternate definition of the notion of coverage can be based solely on occurrence lists:

Definition 8 (*Coverage*). The occurrence at j of m_1 is *covered* by m_2 if there is i such that $\mathcal{L}_{m_2} + i \subseteq \mathcal{L}_{m_1}, j \in \mathcal{L}_{m_2} + i$.

In terms of our running example, we have: $5 \in \mathcal{L}_{\overline{m}_2} + 4$ and $\mathcal{L}_{\overline{m}_2} + 4 = \{5, 7\} \subset \mathcal{L}_{\overline{m}_6} = \{1, 3, 5, 7\}$.

A maximal motif that is not redundant is called an *irredundant motif*. Hence a saturated motif $(m, \mathcal{L}_m)$ is irredundant if the components of the pair $(m, \mathcal{L}_m)$ cannot be deduced by the union of a number of other saturated motifs.

We use $\mathcal{B}_i$ to denote the set of irredundant motifs in suf_i. Set $\mathcal{B}_i$ is called the *base* for the motifs of suf_i. In particular, $\mathcal{B}$ is used to denote the base of s, which coincides with $\mathcal{B}_1$.

Definition 9 (*Base*). Given a sequence s on an alphabet Σ, let $\mathcal{M}$ be the set of all saturated motifs on s. A set of saturated motifs $\mathcal{B}$ is called a *base* of $\mathcal{M}$ iff the following hold: (1) for each $m \in \mathcal{B}$, m is irredundant with respect to $\mathcal{B} - \{m\}$, and, (2) let $\mathbf{G}(\mathcal{X})$ be the set of all the redundant maximal motifs generated (in the sense of Definition 6) by the set of motifs $\mathcal{X}$, then $\mathcal{M} = \mathbf{G}(\mathcal{B})$.

In general, $|\mathcal{M}| = \Omega(2^n)$. However, the base of 2-motifs has size linear in $|s|$. This follows immediately from the known result (see, e.g., [3]):

Theorem 1. *Every irredundant motif is the meet of s and one of its suffixes.*

In the remainder of this paper, treatment will be restricted to 2-motifs. Recall now that in order for a motif to be irredundant it must have at least one occurrence that cannot be deduced from occurrences of other motifs. In [3], such an occurrence is called *maximal* and the motif is correspondingly said to be *exposed* at the corresponding position. Clearly, every motif with a maximal occurrence is saturated. However, not every saturated motif has a maximal occurrence. In fact, the set of irredundant motifs is precisely the subset of saturated motifs with a maximal occurrence. The following known definitions and properties (see, e.g., [3,11,5]) are listed for future reference.

Definition 10 (*Maximal Occurrence*). Let $(m, \mathcal{L}_m)$ be a motif of s and $j \in \mathcal{L}_m$. Position j is a *maximal occurrence* for m if for no $d' \geq 0$ and $(m', \mathcal{L}_{m'})$ we have $\mathcal{L}_{m'} \subseteq (\mathcal{L}_m - d')$ with $(j - d') \in \mathcal{L}_{m'}$.

For a given $m \in \mathcal{B}$, let $\mathcal{L}_m^{max}$ denote the list of maximal occurrences of m.

Lemma 1. $m \in \mathcal{B} \Leftrightarrow |\mathcal{L}_m^{max}| > 0$.

Lemma 2. *If $m \in \mathcal{B}$, then $j \in \mathcal{L}_m^{max} \Leftrightarrow [s \oplus suf_{(max\{j,k\}-min\{j,k\})}] = m, \forall k \in \mathcal{L}_m$.*

Lemma 3. $\sum_{m\in\mathcal{B}} |\mathcal{L}_m| < 2n$.

Lemma 2 shows that in order to check whether a position i is a maximal occurrence for an assigned motif $(m, \mathcal{L}_m)$, it suffices to check the condition $[suf_i \oplus suf_k] = m, \forall k \in \mathcal{L}_m$. Also Lemma 3 [11], which poses a counter-intuitive linear bound on the cumulative size of the occurrence lists in a base, will play an important role in our construction.

3. The incremental management of motif occurrences

Any approach to the extraction of bases of irredundant motifs must solve the problem of finding the occurrences of the autocorrelations or meets of the input string s or of its suffixes. This evokes the notable variant of *approximate* string searching featuring don't cares (see, e.g., [1,2,8]), which admits of a classical $O(n \log m \log |\Sigma|)$ time solution based on the FFT [7] (see also [6] for a more recent $O(n \log n)$ speedup). Such an FFT-based solution is the one adopted in [11,9], resulting in an overall time $O(n^2 \log n \log |\Sigma|)$. The incremental approach in [3] proceeds instead by computing those lists directly and for consecutively increasing suffixes of each autocorrelation. This produces the base for each suffix of s, at the overall cost of $O(n^3)$ time. In [5], the fact is exploited that all patterns being sought come from the set of autocorrelations of the *same* string. In a nutshell, an occurrence of $m = suf_i \oplus suf_j$ at some position k in s induces strong interdependencies among the number of don't-cares in each of the three patterns $m = suf_i \oplus suf_j$, $m' = suf_i \oplus suf_k$ and $m'' = suf_j \oplus suf_k$. For binary alphabets, with d_x denoting the number of don't cares in x and $pref_i(x)$ the prefix of x of length i, the following holds.

Lemma 4 (*[5]*). *Let* $m = [suf_i \oplus suf_j]$, $m' = pref_{|m|}(suf_i \oplus suf_k)$ *and* $m'' = pref_{|m|}(suf_j \oplus suf_k)$.

$$k \in \mathcal{L}_m \;\Leftrightarrow\; d_m = d_{m'} + d_{m''}.$$

Thus, following an $O(n^2)$ preprocessing of the input string s in which the number of don't cares in every suffix of each autocorrelation of s is counted, it is possible to answer in constant time whether any meet occurs at any position of s, just by checking the balance of don't cares. We display the proof of Lemma 4 in the Appendix in order to convey the flavor of these constructions, which are summarized in the following.

Theorem 2. *Let s be a binary string of n characters, and m the meet of any two suffixes of s. Following an $O(n^2)$ time preprocessing of s, it is possible to decide for any assigned position k whether or not k is an occurrence of m in constant time.*

3.1. Bottlenecks

We now concentrate on designing an algorithm that produces the bases of all suffixes of an input string s. Following an initial preparation, the algorithm will proceed incrementally on suffixes of increasing length, along the lines of a paradigm introduced in [3]. At the generic iteration $n - i$ the algorithm builds the base $\mathcal{B}_i$ relative to suf_i. This base is formed in part by selecting the elements of $\mathcal{B}_{i+1}$ that are still irredundant in suf_i, in part by identifying and discarding, from the set of new candidate motifs consisting of the meets of suf_i, those motifs that are covered by others. Since the elements in any base will come from meets of some of the suffixes of s, a bottleneck for the procedure is represented by the need to compute the occurrences of all such meets. Before entering the details of our construction, we need to examine more closely the challenge posed by the incremental management of such meets.

Our algorithm must build the sets $\mathcal{M}_i = \left\{[suf_i \oplus suf_j], \forall j > i\right\}$ and $\mathcal{B}_i \subseteq \mathcal{M}_i$ as i goes from $n - 1$ down to 1 through the main cycle. For the generic iteration, this entails, in particular, to update the lists of occurrences of $\mathcal{M}_{i+1} = \left\{[suf_{i+1} \oplus suf_j], \forall j > i + 1\right\}$ and $\mathcal{B}_{i+1} \subseteq \mathcal{M}_{i+1}$ in order to produce those of $\mathcal{M}_i$ as well as $\mathcal{B}_i \subseteq \mathcal{M}_i$. As there are possibly $O(n^2)$ occurrences to update at each of the $n - 1$ iterations, this task is a major potential source of inefficiency, even though it is not difficult to see that the lists do not need to be built from scratch at each iteration [3]. In fact, consider a generic motif $m = [suf_i \oplus suf_j]$ and let $m' = [suf_{i+d} \oplus suf_{j+d}]$, $m' \in \mathcal{M}_{i+d}$, be the motif such that $m = \sigma\,\{\bullet\}^{d-1}\,m'$. Then, the set of occurrences of m is determined by scanning the list of occurrences of m' and verifying the condition $s[i] = s[k - d]$ for every $k \in \mathcal{L}_{m'}$. This is accomplished in constant time per update. Still, for any of the sets $\mathcal{M}$ under consideration

$$\sum_{m \in \mathcal{M}} |\mathcal{L}_m| = O(n^2).$$

Thus, the method costs $O(n^2)$ per iteration, and $O(n^3)$ in total. Our goal is to set up a more prudent organization of the data, leading to a global cost $O(n^2 \log n)$, amortized over all iterations. This seems counterintuitive, since there is no way around listing all occurrences in all lists in less than cubic space. However, we can take advantage of the dynamics undergone by our list and make do with a partially implicit representation. In order to proceed, we need some auxiliary developments.

3.2. Earliest index and the persistence of an occurrence

It is a crucial consequence of Theorem 2 that once the don't cares have been tallied for all suffixes of each meet of s then it takes only constant time to determine whether or not an arbitrary position k is an occurrence of m, m being the meet of an arbitrary pair of suffixes of s. Although the same could be done on-the-fly with no penalty, we will assume for simplicity that a trivial, $O(n^2)$ pre-processing phase has already been performed to determine the number (and individual "pedigree") of don't cares in each $[s \oplus suf_i]$ $(i = 1, 2, \ldots, n)$, and concentrate on computing the occurrences of every pairwise suffix meet.

Definition 11 (*Earliest Index*). Let m be a meet of s and $1 \le j \le |s|$, and let $suf_k(m)$ indicate as usual the k-th suffix of m. The *earliest index* I_j^m *of* m *at* j is $I_j^m = min\{k : (j - |m| + k) \in \mathcal{L}_{suf_k(m)}\}$.

That is, starting at some occurrence j of the last solid character of m, the index I_j^m is k if $suf_k(m)$ is the longest suffix of m ending at j ($m[k] \neq s[j-|m|+k]$).

Consider now a generic meet $m = [s \oplus suf_i]$. Knowing the earliest index relative to m at every position j of s, we also know that for $l \geq I_j^m$, position $j-|m|+l$ must be included in the list of occurrences of $suf_l(m)$, whereas for $l < I_j^m$ the position $j-|m|+l$ is not an occurrence of $suf_l(m)$.

Lemma 5. *Let $m = [s \oplus suf_i]$ and $1 \leq j \leq |m|$. Computing I_j^m, the earliest index of j relative to m, takes time $O(\log n)$.*

Proof. The computation is carried out by straightforward binary search. At the generic step, we check that $j-|m|+k$ is an occurrence of $suf_k(m)$; if this is the case, we proceed with the next longer suffix in the recursion, otherwise with the next shorter one. The cost of the step is that of determining whether a given position is an occurrence for a certain meet of two suffixes of s, which Theorem 2 affords in constant time. Through the $O(\log n)$ steps, the computation of I_j^m takes thus $O(\log n)$. □

This immediately yields:

Corollary 1. *Computing the earliest indices of all meets of s at all positions $1, 2, \ldots, |s| = n$ takes time $O(n^2 \log n)$.*

Corollary 1 is a crucial handle for our speedup, which nevertheless requires a few additional observations. First, recall that the motifs that survive each round of updates are essentially a subset of the current version of $\mathcal{M}$: their respective lists are sublists of the original ones. Upon updating, each surviving occurrence in a list will retain its original starting position, up to an offset which is *uniform* for all fellow survivors. By making the convention that the elements in a list are represented by their *ending*, rather than their *starting* position and keeping track of lengths we will never need to rename the survivors. Moreover, the occurrences that do not survive will never be readmitted to any list. Finally, because the base $\mathcal{B}_i$ comes only from meets of suf_i, then at iteration $n-i$, we only need, in addition to $\mathcal{B}_{i+1}$, the lists in the set: $\mathcal{M}_i = \left\{[suf_i \oplus suf_j], \forall j > i\right\}$. We will see next that, under our conventions, these lists can be made readily available throughout, at a total cost of $O(n^2 \log n)$. In fact, a stronger construct can be established, whereby *all* sets $\mathcal{M}_i^j = \{[suf_i \oplus suf_k], \forall k \neq i, k \geq j\}$ can be implicitly maintained with their individual lists throughout, for each $j \leq i$, at the overall cost of $O(n^2 \log n)$ instead of $O(n^3)$. The remainder of the section is devoted to substantiate this claim.

3.3. Monitoring the life span of an occurrence: Panpipes

At the iteration for suf_i the list for the generic meet m of s will appear as partitioned into sections, as follows. The currently *open* section contains the ending positions in suf_i of occurrences of suffixes of m that fall still short of their respective earliest indices. The remaining sections are called *closed* and assigned to various lengths, as follows: the section assigned to *length* ℓ stores the ending positions, if any exist, of the occurrences in suf_i of suffixes of m of length ℓ that cannot be prolonged into occurrences of length $\ell' > \ell$. A list will be initialized as soon as the rightmost two replicas in s of the last solid character of its meet are found. Let these positions be k and $h > k$, respectively. These two entries k and h are dubbed *open* and appended to the open list of name $j = (h-k)$. New entries are added to the open list while longer and longer suffixes of the input string s are considered. At the iteration for suf_i, i is added to the open section of the all lists of meets having $s[i]$ as their last character. At that point, a "sentinel" pointer is also issued from the positions $i-|m|+I_i^m$ of s to this entry in the list. The role of each sentinel is to gain access to its corresponding entry when the latter "decays" at iteration $k = i-|m|+I_i^m$, as a consequence of the corresponding occurrence becoming "too short" to survive. At that point, the entry i is taken out of the open section of the list and moved to the closed section assigned to length $m-I_i^m$. In conclusion, the list assigned to m undergoes "refresh cycles" as longer and longer extensions provoke the defection of more and more entries from the open to the closed length sublists and new, shorter suffix occurrences are discovered and added to the open list.

For each meet m, the list assigned to m is partitioned into sublists arranged in order of decreasing length, with the length of the open list set conventionally equal to n, and the items inside each sublist are in turn sorted in order of ascending position. The collective list will be referred to as the *panpipes* of m, after the ancient musical instrument it resembles, sketched in Fig. 1. For any integer $\ell \leq n-1$, tallying the current size of all the occurrences of suffixes of m not longer than ℓ is like stabbing the set of degrading pipes with an orthogonal stick, striking at a height of ℓ from the base, and then counting how many were hit. Standard balanced tree implementation of the list with its subsections supports each of:

- INSERTION of an element in the open section;
- DEMOTION of an element to the closed section of a given length;
- LINE STABBING at any height, or tallying elements of a given minimum length;

in $O(\log n)$ time each.

Theorem 3. *Maintaining the panpipes of all distinct meets of s consecutively at suf_i, for $i = n-1, n-2, \ldots, 1$ takes overall time $O(n^2 \log n)$.*

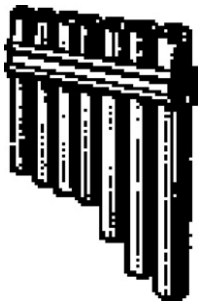

Fig. 1. Panpipes: the longest pipe stores the open section of the list, then shorter and shorter pipes hold shorter and shorter suffixes of the same meet.

Proof. It takes $O(n^2 \log n)$ computation to determine l_j^m for all j's and meets of s. Then, refer to the preceding description for the updates. Each one of the $O(n)$ candidate occurrences of each of the $O(n)$ meets is inserted in the open sub-list exactly once, and then possibly moved from there to a specific length list once and forever. This accounts for $O(n^2)$ panpipes primitives in total, at an individual cost of $O(\log n)$ each, which yields a total complexity of $O(n^2 \log n)$. □

Corollary 2. *The sequence of sets* $\mathcal{M}_i = \left\{[suf_k \oplus suf_j], \forall k \geq i, j > k\right\}$ $(i = n-2, n-3, \ldots, 1)$*, each with its occurrence lists and individual list cardinalities can be consecutively generated one from the other in overall time* $O(n^2 \log n)$.

Proof. Following all necessary preprocessing, and with m denoting the generic meet of s, just let $suf_i(m)$ first "inherit" the whole list of $suf_{i+1}(m)$ (that is, $\mathcal{L}_{suf_i(m)} = \mathcal{L}_{suf_{i+1}(m)} - 1$), and then use the sentinels at i to access and eliminate from that list all occurrences $j - |m| + i$ such that $l_j^m = i$. □

4. Computing the bases of all suffixes of a string

We are ready to detail the generic iteration of the algorithm. Iteration $n - i$ will determine the base $\mathcal{B}_i$ for suf_i, so that in particular the base of s itself will be available after n iterations. The input for this iteration is as follows:

- The set $\mathcal{M}_{i+1}$ of meets of suf_{i+1} each with its individual occurrence list.
- The base $\mathcal{B}_{i+1}$, represented by the collection of patterns on $\Sigma \cup \{\bullet\}$ each with its list of occurrences in suf_{i+1}, with maximal occurrences tagged.

The output of the iteration are $\mathcal{M}_i$ and $\mathcal{B}_i$, in the same representation.

Recall that at any time the collective size of all lists in any given set $\mathcal{B}_i$ is linear in $n - i$, by virtue of Lemma 3. This is not necessarily true of the collective size of the lists of $\mathcal{M}_i$. However, these sets possess each at most $n - 1$ meets. Each iteration of the main cycle consists of the two phases:

- Phase 1: extract from $\mathcal{B}_{i+1}$ the motifs that are still irredundant in suf_i;
- Phase 2: identify all *new* irredundant motifs.

We describe these two phases in succession.

4.1. Phase 1 — Computing $\mathcal{B}_{i+1} \cap \mathcal{B}_i$

This phase consists of identifying the motifs of $\mathcal{B}_{i+1}$ that are still irredundant in suf_i. Two distinct events may lead a motif m in $\mathcal{B}_{i+1}$ to become redundant:

(1) m is covered by a new motif discovered at the current iteration;
(2) m is covered by the occurrence starting at i of some other element of $\mathcal{B}_{i+1}$.

It is convenient to single out from $\mathcal{B}_{i+1}$ the motifs that exhibit a new occurrence starting at i, and handle them separately from the rest. This enables us to search for the motifs of $\mathcal{B}_{i+1}$ that are still irredundant in suf_i, among motifs:

- [1(a)] with an occurrence starting at i;
- [1(b)] without an occurrence starting at i.

These two cases differ on the basis of how a maximal occurrence is covered. If a motif m that becomes redundant in suf_i has an occurrence starting at i and maximal occurrence j in suf_{i+1}, this means that $m \prec m'$ for some $m'$with $j \in \mathcal{L}_{m'}$. In the second case, as it shall be seen later in detail, such a motif becomes redundant because some other motif extends its maximal occurrence j by adding a solid character $\sigma = s[i]$ at position $j - i + 1$.

Since the two phases operate on distinct sets of motifs (respectively with and without an occurrence at i), they can be handled independently upon separating their respective inputs. Alternatively, the entire $\mathcal{B}_{i+1}$ is fed as input to Phase 1(a)

and the output of this phase will be the input of Phase 1(b). Whereas the preliminary separation reduces the input size for either phase, deciding for each motif of $\mathcal{B}_{i+1}$ whether or not it has an occurrence at i induces an extra cost $O(|\mathcal{B}_{i+1}|)$. The second approach also requires some partitioning, but this can be performed on a smaller input in between phases or at the end. The approach described next uses preliminary partitioning. The second approach is left for an exercise.

4.1.1. Phase 1(a)

Since we know the name (meet-id, list-head) and length of the motif in $\mathcal{B}_{i+1}$ to be checked, we can compute the position at which an occurrence at i would end, and then check (or compute from scratch) the earliest index of that position relative to the meet name. Therefore, separating from $\mathcal{B}_{i+1}$ the motifs with an occurrence at i takes at most $O(|\mathcal{B}_{i+1}|)$. With $\mathcal{B}^i_{i+1}$ denoting the subset of $\mathcal{B}_{i+1}$ containing such motifs, the goal is then that of determining $\mathcal{B}^i_{i+1} \cap \mathcal{B}_i$. This set exhibits some important properties, which are derived next.

Lemma 6. *Let $\mathcal{B}^j$ be the set of irredundant motifs with an occurrence at j, and $\mathcal{M}^j$ the set of meets $[suf_j \oplus suf_k]$, $\forall k \neq j$. Then $\mathcal{B}^j \subseteq \mathcal{M}^j$.*

Proof. Let m be an element of $\mathcal{B}^j$. From Lemma 1, m must have at least one maximal occurrence k. If $k = j$ then $m = [suf_j \oplus suf_l]$, $\forall l \in \mathcal{L}_m$. If this is not the case, it follows from the maximality of the occurrence at k that $m = [suf_k \oplus suf_l]$, $\forall l \in \mathcal{L}_m$, which holds in particular for $l = j$. □

Lemma 6 is useful when searching irredundant motifs of which a specific occurrence is known, since it restricts the set of candidates to a linear subset of all pairwise suffix meets. In particular, the lemma can be used to determine which ones among the old motifs having an occurrence at i conserve their irredundancy in suf_i.

Corollary 3. *Let, as earlier, $\mathcal{M}_i$ denote the set of meets $[suf_i \oplus suf_k]$, $\forall k > i$. Then*

$$\left(\mathcal{B}^i_{i+1} \cap \mathcal{B}_i\right) \subseteq \mathcal{M}_i.$$

Proof. From Lemma 6. □

In order for a motif in $\mathcal{B}_{i+1}$ to stay irredundant in the transition from suf_{i+1} to suf_i, at least one of its maximal occurrences in suf_{i+1} must be preserved also in suf_i.

Lemma 7. *Let $m \in \mathcal{B}_{i+1}$. Then*

$$m \in \mathcal{B}_i \Leftrightarrow \exists k \neq i : k \in \mathcal{L}^{max}_m.$$

Proof. This holds clearly for a motif of $\mathcal{B}_{i+1}$ with no occurrence at i, since irredundancy presupposes a maximal occurrence. Assume then a motif of $\mathcal{B}_{i+1}$ having i as its sole maximal occurrence. Then, $m = [suf_i \oplus suf_k]$, $\forall k \in \mathcal{L}_m$. Let $k \in \mathcal{L}_m$ be a maximal occurrence of m in suf_{i+1}. Since $k \in \mathcal{L}_m$, we have $m = [suf_i \oplus suf_k]$, so that k is a maximal occurrence of m in suf_i as well. □

Therefore, no motif of the old base can preserve its irredundancy by having i as its *sole* maximal occurrence. On the other hand, preserving irredundancy for an old motif does not necessarily require a maximal occurrence at i. These properties suggest that the redundancy of a motif $m \in \mathcal{B}_{i+1}$ can be assessed by scanning maximal occurrences of m and deciding which ones among them are still maximal in suf_i. If the maximal occurrences of m are already known in suf_{i+1}, all that is left is to check maximality with respect to the new occurrence at i.

Lemma 8. *$m \in \mathcal{B}_{i+1}$, $i \in \mathcal{L}_m$, and $m \in \mathcal{B}_i \Leftrightarrow \exists k \in \mathcal{L}^{max}_m$ such that $m = [suf_i \oplus suf_k]$.*

Proof. Immediate from Lemma 7. □

In conclusion, the bulk of Phase 1(a) consists of scanning the maximal occurrences of each $m \in \mathcal{B}_{i+1}$ also occurring at i and determining whether at least one such occurrence stays maximal. Maximal occurrence j stays maximal in suf_i iff $[suf_i \oplus suf_j] = m$, a condition that can be tested by comparing the number of don't cares respectively in m and $[suf_i \oplus suf_j]$, given that m occurs at i and j. Alternatively, this condition can be checked by comparing the occurrence lists of m and $[suf_i \oplus suf_j]$. In fact, since both i and j are in $\mathcal{L}_m$, it must be that

$$m \preceq [suf_i \oplus suf_j] \Leftrightarrow \mathcal{L}_{[suf_i \oplus suf_j]} \subseteq \mathcal{L}_m.$$

Hence, in order to check whether $[suf_i \oplus suf_j]$ and m coincide it suffices to check the condition: $|\mathcal{L}_{[suf_i \oplus suf_j]}| = |\mathcal{L}_m|$, Note that, as a by-product, either method inductively maintains knowledge of the maximal occurrences for all motifs in the sets $\mathcal{B}$.

Lemma 9. *Phase* 1(a) *takes time $O(n)$.*

Proof. For any $j > i$, we identify $[suf_i \oplus suf_j]$ from our knowledge of i, j, and of their difference. In fact, the latter identifies a specific meet of s. Let now $m \in \mathcal{B}^i_{i+1}$. For every maximal occurrence j of m it takes constant time to compare don't cares or list sizes for m and $[suf_i \oplus suf_j]$. By Lemma 3, the size of all lists of in a base cumulates to less than $2n$, whence the total number of occurrences that need to be checked is $O(n)$. □

4.1.2. Phase 1(b)

Recall that the task of this phase is the identification of the motifs that stay irredundant in suf_i among the elements of $\mathcal{B}_{i+1}$ with no occurrence at i. The identification of these motifs is rather straightforward once it is observed that the only way in which such a motif m may become redundant in suf_i is for it to be covered, in its maximal occurrences in suf_{i+1}, by a motif $m' = \sigma(\bullet)^d m$ with an occurrence at i.

Lemma 10. *If $m' \in \mathcal{M}_i$ covers $m \in \mathcal{B}_{i+1}$, $i \notin \mathcal{L}_m$, then $m' = \sigma(\bullet)^d m$ where $\sigma = s[i]$ and $d \geq 0$.*

Proof. Occurrence $j \in \mathcal{L}_m^{max}$ loses maximality if $\exists k \in \mathcal{L}_m$ such that $[suf_i \oplus suf_{i+k-j}] \succ m$, where it is assumed w.l.o.g. $k > j$. Since j is a maximal occurrence of m in suf_{i+1}, then $[suf_k \oplus suf_j] = m$ and the only possibility is $[suf_i \oplus suf_{i+k-j}] = s[i](\bullet)^d m$, where $d = j - i$. □

The elimination from $\mathcal{B}_{i+1}$ of the motifs $m \notin \mathcal{B}_i$ without an occurrence at i is done by checking for every maximal occurrence of m whether it can be extended in such a way as to lose maximality. The procedure terminates as soon as an occurrence that stays maximal is met, or when all maximal occurrences have been obliterated.

Lemma 11. *Phase* 1(b) *takes time $O(n)$.*

Proof. Since each m to be checked is in $\mathcal{B}_{i+1}$, then the total number of motif occurrences of which the possible extension into i needs to be checked is $O(n)$. Checking for extensibility of an occurrence is easily done in constant time. □

4.2. Phase 2 — Identifying the new irredundant motifs

Recall that by *new* irredundant motifs we mean those elements of $\mathcal{B}_i$ that did not belong to $\mathcal{B}_{i+1}$. Lemma 6 prescribes that these motifs are to be identified among the elements of $\mathcal{M}_i = \{[suf_i \oplus suf_j], \forall j > i\}$. Indeed, to be irredundant these motifs must have a single maximal occurrence at position i in case they already had multiple occurrences in suf_{i+1}; otherwise, they must have precisely two occurrences, both occurrences being maximal. Let then

$$\widetilde{B} = \mathcal{M}_i - (\mathcal{B}_{i+1} \cap \mathcal{B}_i)$$

be the set of the candidate new irredundant motifs. Since i is the only possible maximal occurrence of any motif $m \in \widetilde{B}$, we must check which ones among old and new motifs with an occurrence at i can cover this occurrence of m. The way this is done is based on the following properties.

Lemma 12. *Let $m_1, m_2 \in \widetilde{B}$ and let $j \neq i \in L_{m_1} \cap L_{m_2}$, $|L_{m_1}| < |L_{m_2}|$. Then, $m_2 \notin B_i$.*

Proof. Observe first that it is impossible for both motifs to be irredundant in suf_i. In fact, since they do not belong to the old base they would have to have maximal occurrences at i. But if this holds for m_1 then $[suf_i \oplus suf_j] = m_1, \forall k \in L_{m_1}$, whence, in particular, $[suf_i \oplus suf_j] = m_1$. Likewise, it must be $[suf_i \oplus suf_k] = m_2, \forall k \in L_{m_2}$, hence $[suf_i \oplus suf_j] = m_2 = m_1$. Assume then w.l.o.g. that only m_2 is irredundant. Then, $[suf_i \oplus suf_k] = m_2, \forall k \in L_{m_2}$ and thus $[suf_i \oplus suf_j] = m_2$. Since $i, j \in L_{m_1}$ we have $[suf_i \oplus suf_j] \succeq m_1$ and thus $|L_{m_2}| \leq |L_{m_1}|$, which contradicts the hypothesis. □

Lemma 13. *Let $m_{new} \in \widetilde{B}$, $m_{old} \in \mathcal{B}_{i+1}^i \cap \mathcal{B}_i$, and $j \neq i \in L_{m_{new}} \cap L_{m_{old}}$, $|L_{m_{old}}| < |L_{m_{new}}|$. Then, $m_{new} \notin B_i$.*

Proof. As was already argued, if $m_{new} \in B_i$ then its occurrence at i must be maximal, hence $[suf_i \oplus suf_j] = m_{new}$. We have then again $m_{new} \succeq m_{old}$ which generates the contradiction $|L_{m_{new}}| \leq |L_{m_{old}}|$. □

In conclusion, in order to check for irredundancy of the elements of $\widetilde{B}$, it must be checked for every such motif m whether it is covered by another motif of $\widetilde{B}$ or by some old irredundant motif which is still irredundant in suf_i. Let $m_1, m_2, \ldots, m_l$ ($l \leq n$) be the motifs to verify. They all come in the form $[suf_i \oplus suf_k]$ for some $k > i$. Considering $m_1 = [suf_i \oplus suf_{k_1}]$ with $L_{m_1} = \{i, k_1, k_2, \ldots, k_r\}$, the motifs that can possibly obliterate m_1 are $[suf_i \oplus suf_j], j \in L_{m_1}, j \neq i, j \neq k_1$. Taking $m' = [suf_i \oplus suf_{k_2}]$ as the first motif to be considered, we check the condition $|L_{m_1}| \geq |L_{m'}|$. Note that having chosen m' as $[suf_i \oplus suf_{k_2}]$ where both i and k_2 are occurrences for m_1, we must have $|L_{m_1}| \geq |L_{m'}|$. If $|L_{m_1}| = |L_{m'}|$, then $m_1 = m'$ and m' is excluded from further analysis. If $|L_{m_1}| > |L_{m'}|$, then m_1 is obliterated by m' and thus m must be eliminated since it is redundant. The procedure is repeated with the surviving motif until all redundant motifs have been eliminated.

Lemma 14. *Phase* 2 *correctly identifies all new irredundant motifs.*

Proof. We need first to establish that the described approach is correct. Indeed, assume that the k-th iteration is handling the motif m with an occurrence at j, and that the pair $m, m' = [suf_i \oplus suf_j]$ is checked, where m' had been already considered during some previous iteration $h < k$. Two situations are possible:

(1) m' had been eliminated at iteration h. In this case, there must be a motif covering m' at i, whence that motif will also cover the occurrence at i of m. Thus, m can be eliminated at the current iteration.
(2) m' has been previously checked but not eliminated. This means that $m' \in B_i$. Since m and m' share an occurrence other than i it must be $m \notin B_i$, so that also in this case m can be eliminated at the current iteration. □

Lemma 15. *Phase* 2 *takes time $O(n)$.*

Proof. Following each one of the comparisons, the procedure eliminates a distinct meet of suf_i from further consideration. Since there are $O(n)$ such meets, we also have $O(n)$ iterations, each requiring constant time to compare the cardinalities of two lists. □

Theorem 4. *The irredundant motif bases of all suffixes of a binary string can be computed incrementally in time* $O(n^2 \log n)$.

Proof. By the preceding properties and discussion. □

5. Concluding remarks

Several issues are still open. Notable among them are the existence of an optimal algorithm for general alphabets, and of an optimal incremental algorithm for alphabets of constant or unbounded size.

Acknowledgements

The first author's work was supported in part by the Italian Ministry of University and Research under the Bi-National Project FIRB RBIN04BYZ7, and by the Research Program of Georgia Tech. The work was carried out in part while visiting the Institute for Mathematical Sciences, National University of Singapore in 2006, and the Shanghai CAS-MPG Partner Institute for Computational Biology between the Chinese Academy of Sciences and the German Max Planck Society in 2007, with support provided by those Institutes. The second author's work was carried out in part while visiting the College of Computing of the Georgia Institute of Technology.

Appendix. Proof of Lemma 4

We first show that if m has an occurrence at k it implies the claim. Under such hypotheses, we have $i, j, k \in \mathcal{L}_m$, whence, by Property 3, also $m \preceq m' = pref_{|m|}(suf_i \oplus suf_k)$. Similarly, it must be $m \preceq m''$. Considering then homologous positions in m, m' and m'', the following holds:

- If $m[l] = \sigma$ then, from $m \preceq m'$ and $m \preceq, m''$, we get $m'[l] = m''[l] = \sigma$.
- If $m[l] = \bullet \Leftrightarrow suf_i[l] \neq suf_j[l]$, and one of the following two cases is possible:
  (1) $m'[l] = \sigma \Leftrightarrow suf_i[l] = suf_k[l] \Leftrightarrow suf_j[l] \neq suf_k[l] \Leftrightarrow m''[l] = \bullet$.
  (2) $m'[l] = \bullet \Leftrightarrow suf_i[l] \neq suf_k[l] \Leftrightarrow suf_j[l] = suf_k[l] \Leftrightarrow m''[l] = \sigma$.
  The last one is summarized by $m[l] = \bullet \Leftrightarrow m'[l] \neq m''[l]$. Note that, since m' and m'' both result from a meet of suf_k with some other suffix of s, then $m'[l] \neq m''[l]$ implies $m'[l] = \bullet$ or $m''[l] = \bullet$.

Thus, in correspondence with every don't-care of m, only one of the patterns m' and m'' will have a don't-care. Since every solid character of m must also appear in homologous positions in both m' and m'', we have that the total number of don't cares in m' and m'' equals the don't cares in m.

To prove the converse, we show that if k is not an occurrence of m then this infringes the claimed relationship. Assume then $k \notin \mathcal{L}_m$. Hence, $\exists l$ such that $m[l] = \sigma$ and $suf_k[l] \neq \sigma$. Since $m[l] = \sigma$, it must be $suf_i[l] = \sigma$ and $suf_j[l] = \sigma$, whence $m'[l] = \bullet = m''[l]$. Upon re-examining the distribution of don't cares in m' and m'' with respect to m, we have the following cases:

- $m[l] = \sigma$. This splits into:
  (1) $m'[l] = \sigma \Leftrightarrow m''[l] = \sigma$.
  (2) $m'[l] = \bullet \Leftrightarrow m''[l] = \bullet$.
- $m[l] = \bullet$. There is no change with respect to the first part of the proof.

We see thus that the difference with respect to the assumption $k \in \mathcal{L}_m$ is posed by some solid characters in m that become don't care in m' and m''. Every don't-care in m is balanced by corresponding don't cares in m' and m''. However, we must now add to the equation a positive contribution that amounts to twice the number of positions of suf_k that cause a *mismatch* with m. In other words, when $k \notin \mathcal{L}_m$ we have $d_m < d_{m'} + d_{m''}$, hence $d_m \neq d_{m'} + d_{m''}$. □

References

[1] Alfred V. Aho, Algorithms for finding patterns in strings, in: Handbook of Theoretical Computer Science (vol. A): Algorithms and Complexity, 1990, pp. 255–300.
[2] Alberto Apostolico, Zvi Galil, Pattern Matching Algorithms, Oxford University Press, New York, 1997.
[3] Alberto Apostolico, Laxmi Parida, Incremental paradigms of motif discovery, Journal of Computational Biology 11 (1) (2004) 15–25.
[4] Alberto Apostolico, Pattern discovery and the algorithmics of surprise, in: P. Frasconi, R. Shamir (Eds.), Artificial Intelligence and Heuristic Methods for Bioinformatics, IOS Press, 2003, pp. 111–127.
[5] Alberto Apostolico, Claudia Tagliacollo, Optimal extraction of irredundant motif bases, in: G. Lin (Ed.), Proceedings of COCOON 07, in: Springer Lecture Notes in Computer Science, LNCS, vol. 4598, 2007, pp. 360–371.
[6] Richard Cole, Ramesh Hariharan, Verifying candidate matches in sparse and wildcard matching, in: STOC '02: Proceedings of the Thiry-Fourth Annual ACM Symposium on Theory of Computing, 2002, pp. 592–601.

[7] Michael J. Fischer, Michael S. Paterson, String matching and other products, in: R. Karp (Ed.), Proceedings of the SIAM–AMS Complexity of Computation, American Mathematical Society, Providence, RI, 1974, pp. 113–125.
[8] Gonzalo Navarro, A guided tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.
[9] Johann Pelfrêne, Saïd Abdeddaïm, Joël Alexandre, Extracting approximate patterns, Journal of Discrete Algorithms 3 (2–4) (2005) 293–320.
[10] Laxmi Parida, Algorithmic techniques in computational genomics, Ph.D. Thesis, Department of Computer Science, New York University, 1998.
[11] Nadia Pisanti, Maxime Crochemore, Roberto Grossi, Marie-France Sagot, Bases of motifs for generating repeated patterns with wild cards, IEEE/ACM Transactions on Computational Biology and Bioinformatics 2 (1) (2005) 40–50.
[12] Laxmi Parida, Isidore Rigoutsos, Aris Floratos, Dan Platt, Yuan Gao, Pattern discovery on character sets and real-valued data: Linear bound on irredundant motifs and an efficient polynomial time algorithm, in: Symposium on Discrete Algorithms, 2000, pp. 297–308.
[13] Jason T.L. Wang, Bruce A. Shapiro, Dennis Elliot Shasha, Pattern Discovery in Biomolecular Data: Tools, Techniques and Applications, Oxford University Press, 1999.