Efficient construction of maximal and minimal representations of motifs of a string ⋆

François Nicolas [a], Veli Mäkinen [a] and Esko Ukkonen [a]

[a] *Department of Computer Science,*
P. O. Box 68 (Gustaf Hällströmin katu 2b)
FIN-00014 University of Helsinki

Abstract

Two substrings of a given text string are called synchronous (occurrence-equivalent) if their sets of occurrence locations are translates of each other. Linear time algorithms are given for the problems of finding a shortest and a longest substring that is synchronous with a given substring. We also introduce approximate variants of the motif discovery problem and give polynomial time algorithms for finding longest and shortest substrings whose suitably translated occurrence location set contains or, respectively, is contained in a given set of locations. The FFT technique used here also leads to an $O(n \log n)$ algorithm for finding the maximum-content gapped motif that is synchronous with a given set of locations; the previously known algorithm for this problem is only quadratic.

1 Introduction

Discovery of repetitive patterns or *motifs* of a given string of symbols $\mathbf{t}$ is a central task in combinatorial analysis of sequences, with numerous applications on various areas such as information retrieval from texts, and biological sequence analysis [6].

To economically represent such motifs, say, as an index structure, it is of interest to find a sparse set of representatives for all motifs [1,8,10]. Two

⋆ Supported by the Academy of Finland under grants 21196 (From Data to Knowledge) and 7523004 (Algorithmic Data Analysis).

Email addresses: nicolas@cs.helsinki.fi (François Nicolas), vmakinen@cs.helsinki.fi (Veli Mäkinen), ukkonen@cs.helsinki.fi (Esko Ukkonen).

motifs are considered equivalent ('synchronous') if their occurrence locations in $\mathbf{t}$ are translates of each other. The maximal (longest) and minimal (shortest) motifs in an equivalence class are obvious candidates for representing the class. For example, string `AAXBYCCCZAAUBVCCCA` has substring motif `AA` that occurs twice. The longest substring motif with the same translated occurrences is `CCC` and the shortest is `B`. In this paper, we give efficient algorithms for finding such representatives for substring and gapped motifs as well as consider some approximate variants with relaxed synchronicity requirements.

For substring motifs (*i.e.*, motifs without gaps) the suffix-tree of the original string is the well-known full index which can be constructed in linear time and from which the occurrence locations of any substring motif can be found in time linear in the length of the motif and the number of its occurrences. The suffix-tree also helps in finding a longest and a shortest synchronous substring motif for a given motif. In fact, if we require that the shortest motif should be a substring of the given motif and, similarly, that the longest motif should be a superstring of the given motif, then the shortest and longest motifs can be quite easily found using suffix-tree techniques in linear time [11,12]. In Section 4 of this paper we complement this result by giving a linear-time algorithm for finding a shortest and a longest representative motif without the substring/superstring restriction.

We also consider the following problem of *approximate* motif discovery. Given some set of locations of $\mathbf{t}$, it is possible that no substring motif has (after any translation) exactly this set of occurrence locations. Then it is of interest to find a substring motif that has this same pattern of occurrences in some approximate sense.

Two such approximations will be introduced. The first one is a longest substring motif whose translated occurrence set contains the given set of locations (longest super-synchronous motif). It is shown in Section 5 that such a motif can be found using FFT in $O(|\mathbf{t}| \log |\mathbf{t}|)$ time. This same technique can interestingly be applied also on the discovery of gapped motifs (*i.e.*, motifs that can contain so-called don't care symbols that match any symbol). We obtain an $O(|\mathbf{t}| \log |\mathbf{t}|)$ algorithm for constructing the maximum-content gapped motif, *i.e.*, the motif with largest number of non-gap symbols, that is synchronous to a given motif (or to a given set of occurrence locations). This improves on the earlier, quadratic-time algorithm of [2,9–12]. It was also shown recently, that finding a smallest such gapped motif is NP–complete [11,12].

Finally, we propose in Section 6 another approximate motif, namely shortest substring motif whose translated occurrence set is contained in the given set of locations (shortest sub-synchronous motif). Such a motif can be found using combined suffix-tree and dynamic programming techniques in time $O(|\mathbf{t}|^2)$.

2 Synchronous motifs

2.1 Synchronicity

A *string* $\mathbf{w}$ (over Σ) is a finite sequence of elements (called *letters* or *symbols*) drawn from a finite *alphabet* (Σ). The *length* of $\mathbf{w}$ is denoted $|\mathbf{w}|$. If not otherwise stated, we assume for simplicity that Σ is the integer alphabet, *i.e.*, $\Sigma = [0, |\mathbf{w}| - 1]$ where $[0, |\mathbf{w}| - 1]$ denotes the interval of all integers from 0 to $|\mathbf{w}| - 1$.

String concatenation is denoted multiplicatively. The letter of $\mathbf{w}$ that occurs at position i is denoted $\mathbf{w}[i]$: $\mathbf{w} = \mathbf{w}[0]\mathbf{w}[1]\mathbf{w}[2]\cdots\mathbf{w}[|\mathbf{w}| - 1]$. The substring of $\mathbf{t}$ between locations i and j is denoted $\mathbf{w}[i, j]$: $\mathbf{w}[i, j] = \mathbf{w}[i]\mathbf{w}[i+1]\cdots\mathbf{w}[j-1]\mathbf{w}[j]$.

The set of all occurrence locations of $\mathbf{x}$ in $\mathbf{t}$ is denoted $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$: $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$ is the set of all $i \in [0, |\mathbf{t}| - |\mathbf{x}|]$ such that $\mathbf{t}[i, i + |\mathbf{x}| - 1] = \mathbf{x}$. For example, if $\mathbf{t} = 01001001010$ then $\mathrm{Loc}_{\mathbf{t}}(10) = \mathrm{Loc}_{\mathbf{t}}(1) = \{1, 4, 7, 9\}$, $\mathrm{Loc}_{\mathbf{t}}(11) = \emptyset$, $\mathrm{Loc}_{\mathbf{t}}(0) = \{0, 2, 3, 5, 6, 8, 10\}$, and $\mathrm{Loc}_{\mathbf{t}}(010) = \mathrm{Loc}_{\mathbf{t}}(01) = \{0, 3, 6, 8\}$.

We say that $\mathbf{x}$ and $\mathbf{y}$ are *synchronous* in $\mathbf{t}$ if $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$ and $\mathrm{Loc}_{\mathbf{t}}(\mathbf{y})$ are translates of each other, *i.e.*, if there exists an integer d such that $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x}) = \mathrm{Loc}_{\mathbf{t}}(\mathbf{y}) + d$. Synchronism is an equivalence relation on the substrings of $\mathbf{t}$. In our example $\mathbf{t} = 01001001010$, substrings 1 and 010 are synchronous as $\mathrm{Loc}_{\mathbf{t}}(1) = \mathrm{Loc}_{\mathbf{t}}(010) + 1$.

The synchronicity relation of substrings and other motifs has earlier been considered (under various names and notations), *e.g.*, in [1,2,8–10].

2.2 Problems on synchronous and approximately synchronous substrings

This paper tackles the four problems listed below. In each case $\mathbf{t}$ is a string (the *text*) over Σ, and $\mathbf{x}$ is a non-empty substring of $\mathbf{t}$.

Problem 1 (Longest Synchronous Substring) *Find a longest string* $\mathbf{x}^*$ *such that* $\mathbf{x}$ *and* $\mathbf{x}^*$ *are synchronous in* $\mathbf{t}$.

Problem 2 (Shortest Synchronous Substring) *Find a shortest string* $\mathbf{x}_*$ *such that* $\mathbf{x}$ *and* $\mathbf{x}_*$ *are synchronous in* $\mathbf{t}$.

In the following two problems we introduce relaxed variants of synchronicity. The motivating situation is such that we are given some set of locations of

$\mathbf{t}$ and want to find a motif that is associated with them. As it may happen that no substring of $\mathbf{t}$ is exactly synchronous with the given locations, we only require that the given locations, when suitably translated, are contained in or contain the occurrence locations of the motif to be discovered. In what follows, L is a set of locations of $\mathbf{t}$, *i.e.*, a subset of $[0, |\mathbf{t}| - 1]$; for example, L could be $L = \mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$ for some $\mathbf{x}$ or just a set of somehow interesting locations of $\mathbf{t}$ for which we want to find a motif.

Problem 3 (Longest Super-Synchronous Substring) *Find a longest string* $\mathbf{x}^*$ *such that there exists an integer* d *satisfying* $L + d \subseteq \mathrm{Loc}_{\mathbf{t}}(\mathbf{x}^*)$.

Problem 4 (Shortest Sub-Synchronous Substring) *Find a shortest string* $\mathbf{x}_*$ *such that there exists an integer* d *satisfying* $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x}_*) + d \subseteq L$.

Linear-time algorithms for Problems 1 and 2 are presented in Section 4. Note that solutions $\mathbf{x}^*$ of Problem 1 are not necessarily superstrings of $\mathbf{x}$. In the same way, solutions $\mathbf{x}_*$ of Problem 2 are not necessarily substrings of $\mathbf{x}$. It is known [11,12] that the longest superstring of $\mathbf{x}$ and the shortest substring of $\mathbf{x}$ that are synchronous to $\mathbf{x}$ in $\mathbf{t}$ can be found in linear time. An $O(|\mathbf{t}| \cdot \log |\mathbf{t}| \cdot \log \sigma)$ algorithm for Problem 3 and a quadratic algorithm for Problem 4 are presented in Sections 5 and 6, respectively. Here σ denotes the cardinality of the alphabet of $\mathbf{t}$, *i.e.*, the number of distinct letters that occur in $\mathbf{t}$. Note that σ is not greater than $|\mathbf{t}|$ so the algorithm for Problem 3 is $O\left(|\mathbf{t}| \cdot \log^2 |\mathbf{t}|\right)$. The algorithms for Problems 1, 2 and 4 rely on suffix-trees. The algorithm for Problem 3 relies on Fast Fourier Transform (FFT).

3 Preliminaries on suffix-trees

3.1 Basics of suffix-trees

Let us clarify some terminology related to directed graphs (*digraphs*) in order to properly deal with the suffix-tree of $\mathbf{t}$. In a digraph, a *root* is a node from which every other node of the digraph is reachable. Define a *tree* as a rooted acyclic digraph such that every non-root node has in-degree one: all edges of a tree are directed away from its root. A node is called a *leaf* if its out-degree is zero, and *internal* otherwise. A tree is called *branching* if no internal node is of out-degree one.

The *suffix-tree* [14,6] of $\mathbf{t}$, denoted $T_{\mathbf{t}}$, is the leaf- and edge-labeled branching tree (actually a compacted *trie* representing all suffixes of $\mathbf{t}$) satisfying the following:

- each edge is labeled with a non-empty substring of $\mathbf{t}\$$ where $\$$ is a symbol that does not occur in $\mathbf{t}$,
- no two edges leaving a node have their labels beginning with the same letter,
- the leaves are bijectively labeled with $[0, |\mathbf{t}| - 1]$, and
- for each $i \in [0, |\mathbf{t}| - 1]$, the path from the root to leaf number i spells out $(\mathbf{t}\$)[i, |\mathbf{t}|]$, *i.e.*, the edge labels concatenated along the path make the suffix of $\mathbf{t}\$$ starting at position i.

Notice that, as a branching tree on $|\mathbf{t}|$ leaves, the suffix-tree of $\mathbf{t}$ has at most $2\,|\mathbf{t}| - 1$ nodes. Each edge-label $\mathbf{x}$ is encoded with a pair (i, j) of indices with $0 \leq i \leq j \leq |\mathbf{t}|$ such that $\mathbf{x} = (\mathbf{t}\$)[i, j]$; each leaf is labeled with one integer; internal nodes are unlabeled. Hence, the suffix-tree $T_{\mathbf{t}}$ is of size $O(|\mathbf{t}|)$.

Theorem 5 ([14,3,4]) *Let $\mathbf{t}$ be a string over integers $[0, |\mathbf{t}| - 1]$. The suffix-tree $T_{\mathbf{t}}$ of $\mathbf{t}$ can be constructed in $O(|\mathbf{t}|)$ time.*

For any leaf-labeled tree T and node x of T, let $\mathrm{Lab}_T(x)$ denote the set of all leaf-labels that are reachable from x in T.

The fundamental property of suffix-trees is:

Lemma 6 ([6]) *Let $T_{\mathbf{t}}$ be the suffix-tree of $\mathbf{t}$, and let $\mathbf{x}$ be a substring of $\mathbf{t}$. There exists a unique node x of $T_{\mathbf{t}}$ such that $\mathrm{Lab}_{T_{\mathbf{t}}}(x) = \mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$. Moreover, x can be found from $\mathbf{x}$ and $T_{\mathbf{t}}$ in the following way: follow the unique path from the root that in its concatenated edge labels spells out $\mathbf{x}$, until $\mathbf{x}$ is exhausted; node x is the head of the edge on which the last match occurs.*

It follows from this lemma that if the number of distinct letters that occur in $\mathbf{t}$ is bounded then x can be computed from $\mathbf{x}$ and $T_{\mathbf{t}}$ in $O(|\mathbf{x}|)$ time.

3.2 Candidate solutions in the suffix-tree

In this section, we notice that the optimum solutions of our problems can only be found at very particular places in the suffix-tree of the input text.

3.2.1 Maximization problems

For every node v of the suffix-tree $T_{\mathbf{t}}$, let $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(v)$ denote the string that labels the path from the root of $T_{\mathbf{t}}$ to node v. The next lemma is a consequence of Lemma 6.

Lemma 7 *Let $\mathbf{x}$ be a substring of $\mathbf{t}$ that occurs in $\mathbf{t}$ at least twice, and let x be the node of $T_{\mathbf{t}}$ such that $\mathrm{Lab}_{T_{\mathbf{t}}}(x) = \mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$. The following two assertions*

hold: $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x}) = \mathrm{Loc}_{\mathbf{t}}(\boldsymbol{\lambda}_{T_{\mathbf{t}}}(x))$ *and* $\mathbf{x}$ *is a prefix of* $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(x)$.

In other words, Lemma 7 states that $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(x)$ is the longest string with the same occurrence locations in $\mathbf{t}$ as $\mathbf{x}$. Note that if $\mathbf{x}$ occurs only once in $\mathbf{t}$ then $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(x)$ ends with the special end symbol \$, and thus $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(x)$ does not occur in $\mathbf{t}$. It follows from Lemma 7 that any optimum solution $\mathbf{x}^*$ of the Longest (Super-)Synchronous Substring problem has to be of the form $\mathbf{x}^* = \boldsymbol{\lambda}_{T_{\mathbf{t}}}(x^*)$ for some internal node x^* of $T_{\mathbf{t}}$.

3.2.2 Minimization problems

For every non-root node v of the suffix-tree $T_{\mathbf{t}}$, let $\boldsymbol{\lambda}'_{T_{\mathbf{t}}}(v) := \boldsymbol{\lambda}_{T_{\mathbf{t}}}(u)a$, where u denotes the parent node of v and a denotes the first symbol of the string that labels the edge from u to v.

Lemma 8 *Let* $\mathbf{x}$ *be a non-empty substring of* $\mathbf{t}$*, and let* x *be the node of* $T_{\mathbf{t}}$ *such that* $\mathrm{Lab}_{T_{\mathbf{t}}}(x) = \mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$*. The following two assertions hold:* $\mathrm{Loc}_{\mathbf{t}}(\boldsymbol{\lambda}'_{T_{\mathbf{t}}}(x)) = \mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$ *and* $\boldsymbol{\lambda}'_{T_{\mathbf{t}}}(x)$ *is a prefix of* $\mathbf{x}$.

PROOF. Follows from Lemma 6. □

In other words, Lemma 8 states that $\boldsymbol{\lambda}'_{T_{\mathbf{t}}}(x)$ is the shortest string with the same occurrence locations in $\mathbf{t}$ as $\mathbf{x}$. It follows from Lemma 8 that every optimum solution $\mathbf{x}_*$ of the Shortest (Sub-)Synchronous Substring problem is of the form $\mathbf{x}_* = \boldsymbol{\lambda}_{T_{\mathbf{t}}}(x_*)$ for some non-root node x_* of $T_{\mathbf{t}}$.

4 Longest and shortest synchronous motifs

The aim of this section is to prove that Problem 1 (Longest Synchronous Substring) and Problem 2 (Shortest Synchronous Substring) can be solved in linear time under the integer alphabet hypothesis. The proof combines Lemma 9 below and the discussion presented in Section 3.2.

Lemma 9 *Let* L *be a subset of the integer interval* $[0, n-1]$ *for some* n*, and let* T *be a branching tree whose leaves are bijectively labeled with a subset of* $[0, n-1]$*. The set of all nodes* v *of* T *such that* $\mathrm{Lab}_T(v)$ *is a translate of* L *is computable in* $O(n)$ *time.*

PROOF. Let k denote the cardinality of L, and let X denote the set of all nodes v of T such that $\mathrm{Lab}_T(v)$ has cardinality k. Clearly, set X can be found

in $O(n)$ time because:

- T has at most $2n-1$ nodes, and
- the cardinalities of all sets of form $\mathrm{Lab}_T(v)$, where v is a node of T, can be evaluated from T in a bottom-up fashion in linear time.

Since two sets are translates of each other only if they have the same cardinality, it remains to select the elements $x \in X$ such that $\mathrm{Lab}_T(x)$ is a translate of L. The trick is to realize that $(\mathrm{Lab}_T(x))_{x\in X}$ is a family of *pairwise disjoint* subsets of $[0, n-1]$. Therefore, the rule of sum ensures that the cardinality of X, denoted h, satisfies $hk \leq n$. We can now finish the computations in $O(n)$ time as follows.

(1) *For each $x \in X$, construct a (possibly unsorted) list of all elements of* $\mathrm{Lab}_T(x)$.
(2) *Bucket sort simultaneously all the lists from Step 1, as well as the list of all elements of L.*
(3) *For each $x \in X$, check whether* $\mathrm{Lab}_T(x)$ *is a translate of L by comparing the corresponding sorted lists.*

For each node v of T, $\mathrm{Lab}_T(v)$ can be formed from v and T in a time proportional to the size of $\mathrm{Lab}_T(v)$, and thus Step 1 takes $O(hk)$ time, which is also $O(n)$. At Step 2, there are $hk + k$ integers to sort using n buckets, and thus this step takes $O(n)$ time. Finally, each of the h list comparisons at Step 4 takes $O(k)$ time, giving total time $O(n)$. $\square$

Theorem 10 *There exists an algorithm that, given a string $\mathbf{t}$ over $[0, |\mathbf{t}|-1]$ and a non-empty substring $\mathbf{x}$ of $\mathbf{t}$, finds in $O(|\mathbf{t}|)$ time a longest (resp. shortest) string synchronous to $\mathbf{x}$ in $\mathbf{t}$.*

PROOF. The algorithm for the Longest Synchronous Substring problem is as follows.

(1) *Construct the suffix-tree $T_{\mathbf{t}}$ of $\mathbf{t}$.*
(2) *Construct the set, denoted Y, of all nodes v of $T_{\mathbf{t}}$ such that* $\mathrm{Lab}_{T_{\mathbf{t}}}(v)$ *is a translate of* $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$.
(3) *Find a node $x^* \in Y$ with maximum $|\boldsymbol{\lambda}_{T_{\mathbf{t}}}(x^*)|$ and return $\mathbf{x}^* := \boldsymbol{\lambda}_{T_{\mathbf{t}}}(x^*)$.*

It follows from Section 3.2.1 that the algorithm is correct provided that $\mathbf{x}$ occurs in $\mathbf{t}$ at least twice.

Step 1 can be implemented in $O(|\mathbf{t}|)$ time by Theorem 5. Moreover, $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x})$ is computable in $O(|\mathbf{t}|)$ time either by examining the suffix-tree $T_{\mathbf{t}}$ or, directly, by applying the KMP algorithm [7]. Therefore, Step 2 can be accomplished in $O(|\mathbf{t}|)$ time by Lemma 9. Finally, the lengths of all strings $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(v)$, where v is

a node of $T_{\mathbf{t}}$, can be evaluated from $T_{\mathbf{t}}$ in a top-down fashion in linear time. Therefore, Step 3 can also be implemented in $O(|\mathbf{t}|)$ time. We have thus shown that the Longest Synchronous Substring problem can be solved in linear time.

To obtain a linear-time algorithm for the Shortest Synchronous Substring problem, replace Step 3 with:

(3) *Find a node* $x_* \in Y$ *with minimum* $\left|\boldsymbol{\lambda}'_{T_{\mathbf{t}}}(x_*)\right|$ *and return* $\mathbf{x}_* := \boldsymbol{\lambda}'_{T_{\mathbf{t}}}(x_*)$.

□

Note that the set Y constructed in the above proof represents all substrings that are synchronous to $\mathbf{x}$. In fact, $\mathbf{y}$ is synchronous to $\mathbf{x}$ if, and only if, there is a node $v \in Y$ such that $\boldsymbol{\lambda}'_{T_{\mathbf{t}}}(v)$ is a prefix of $\mathbf{y}$ and $\mathbf{y}$ is a prefix of $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(v)$.

It also follows from Theorem 10 that a collection of substrings that contains a longest and a shortest representative for every class of the synchronism equivalence can be constructed in time $O(|\mathbf{t}|^2)$. This is because the number of such classes is $O(|\mathbf{t}|)$ as the strings $\boldsymbol{\lambda}_{T_{\mathbf{t}}}(v)$ for internal nodes v of $T_{\mathbf{t}}$ are representing all classes.

5 Longest super-synchronous and maximum-content gapped motifs

The aim of this section is to prove that Problem 3 (Longest Super-Synchronous Substring) can be solved in sub-quadratic time.

Definition 11 *For any integer sets* F *and* G*, define* $\Delta(F, G)$ *as the set of all integers* d *such that* $F + d \subseteq G$.

For instance, if $F = \{-2, 1, 3\}$ and $G = \{-3, 0, 2, 5, 7, 8, 11, 13\}$ then $\Delta(F, G) = \{-1, 4, 10\}$. The next lemma (that is based on the FFT technique from [5]) is essentially contained in [6, Section 4.3.2]. Its proof is given for the sake of completeness.

Lemma 12 *Let* n *be a positive integer and let* F *and* G *be two subsets of integers from* $[0, n-1]$*. The integer set* $\Delta(F, G)$ *is computable in* $O(n \log n)$ *time.*

Note that $\Delta(F, G)$ is a subset of $[1-n, n-1]$, and thus $\Delta(F, G)$ can be encoded in $O(n)$ space as a bit vector or as a sorted list.

PROOF. Let m denote the smallest element of F. Since for every integer d, $\Delta(F-d,G)=\Delta(F,G)-d$, we may replace F with $F-m$ without loss of generality. Now, $\Delta(F,G)$ is a subset of $[0,n-1]$, and thus $\Delta(F,G)$ equals the set of all $d\in[0,n-1]$ such that $(F+d)\cap G$ has the same cardinality as F. Let us explain how to compute the cardinality of $(F+d)\cap G$ for every $d\in[0,n-1]$ in $O(n\log n)$ time.

For any set E, let χ_E denote the indicator function of E: $\chi_E(p)=1$ for every $p\in E$ and $\chi_E(p)=0$ for every $p\notin E$. Define two polynomials $f(z)$ and $g(z)$ by:

$$f(z):=\sum_{p=0}^{n-1}\chi_F(n-1-p)z^p \qquad \text{and} \qquad g(z):=\sum_{p=0}^{n-1}\chi_G(p)z^p\,.$$

Then, for every $d\in[0,n-1]$, the coefficient of z^{n-1+d} in the product $f(z)g(z)$ equals the cardinality of $(F+d)\cap G$. Since the product of two polynomials with degrees less than n is computable in $O(n\log n)$ time using FFT [13], the lemma holds. □

For clarity reasons, we first consider the binary alphabet case of our problem and thereafter the general integer-alphabet case.

Theorem 13 *There exists an algorithm that, given a string* $\mathbf{t}$ *over* $\{0,1\}$ *and a set* $L\subseteq[0,|\mathbf{t}|-1]$ *of locations of* $\mathbf{t}$*, finds in* $O(|\mathbf{t}|\cdot\log|\mathbf{t}|)$ *time a longest string* $\mathbf{x}^*$ *such that* $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x}^*)$ *contains a translate of* L*.*

PROOF. The algorithm for the Longest Super-Synchronous Substring problem in the binary alphabet case is as follows.

(1) *Compute* $D:=\Delta\left(L,\mathrm{Loc}_{\mathbf{t}}(0)\right)\cup\Delta\left(L,\mathrm{Loc}_{\mathbf{t}}(1)\right)$.
(2) *Find two integers* i *and* j *such that* $[i,j]$ *is the largest run of consecutive integers included in* D*, and return* $\mathbf{x}^*:=\mathbf{t}[p+i,p+j]$*, where* p *is some element of* L*.*

Let us first study the time complexity of the algorithm. The computations of $\mathrm{Loc}_{\mathbf{t}}(0)$ and $\mathrm{Loc}_{\mathbf{t}}(1)$ are straightforward. After that, $\Delta\left(L,\mathrm{Loc}_{\mathbf{t}}(0)\right)$ and $\Delta\left(L,\mathrm{Loc}_{\mathbf{t}}(1)\right)$ are computable in $O(|\mathbf{t}|\cdot\log|\mathbf{t}|)$ time by Lemma 12. Hence, Step 1 is computable in $O(|\mathbf{t}|\cdot\log|\mathbf{t}|)$. Clearly, Step 2 can be performed in $O(|\mathbf{t}|)$ time.

Let us now prove the correctness of the algorithm.

Claim 14 *For any integers* i *and* j *with* $i\le j$*,* $[i,j]$ *is a subset of* D *if, and only if,* $\mathbf{t}[p+i,p+j]=\mathbf{t}[q+i,q+j]$ *for all* $p,\,q\in L$*.*

PROOF. Let d be an integer. For each $a \in \{0,1\}$, d belongs to $\Delta(L, \mathrm{Loc}_{\mathbf{t}}(a))$ if, and only if, $\mathbf{t}[p+d] = a$ for all $p \in L$. Therefore, d belongs to D if, and only if, $\mathbf{t}[p+d] = \mathbf{t}[q+d]$ for all p, $q \in L$. Since d can run from i to j, the claim holds. □

It is easy to see that a non-empty string $\mathbf{x}^*$ is a feasible solution of the problem if, and only if, there exist two integers i and j with $i \leq j$ such that $\mathbf{x}^* = \mathbf{t}[p+i, p+j]$ for every $p \in L$. Hence, Claim 14 means that the runs of consecutive integers that are included in D are in one-to-one correspondence with the feasible solutions of the problem. More precisely, each subset of D of the form $[i,j]$ with $i \leq j$ corresponds to a solution string with length $j-i+1$. It follows that the algorithm is correct. □

Let us now state the main result of the section.

Theorem 15 *There exists an algorithm that, given a string* $\mathbf{t}$ *over* $[0, |\mathbf{t}|-1]$ *and a subset* $L \subseteq [0, |\mathbf{t}|-1]$*, finds a longest string* $\mathbf{x}^*$ *such that* $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x}^*)$ *contains a translate of* L*. The running time of the algorithm is* $O(|\mathbf{t}| \cdot \log|\mathbf{t}| \cdot \log\sigma)$ *where* σ *denotes the number of distinct letters that occur in* $\mathbf{t}$*.*

PROOF. The trick is to encode the input text $\mathbf{t}$ into a binary string by means of a uniform letter-to-word substitution. Otherwise, the idea is the same as in the proof Theorem 13. More precisely, the algorithm for the Longest Super-Synchronous Substring problem is as follows.

(1) *Compute the set, denoted* Σ*, of all letters that occur in* $\mathbf{t}$*.*
(2) *Construct an injective function* $\boldsymbol{\gamma} : \Sigma \to \{0,1\}^m$*, where* $m := \lceil \log\sigma \rceil$*.*
(3) *Construct the binary string* $\mathbf{t}' := \boldsymbol{\gamma}(\mathbf{t}[0])\boldsymbol{\gamma}(\mathbf{t}[1]) \cdots \boldsymbol{\gamma}(\mathbf{t}[|\mathbf{t}|-1])$*.*
(4) *Construct* $D := \Delta(L', \mathrm{Loc}_{\mathbf{t}'}(0)) \cup \Delta(L', \mathrm{Loc}_{\mathbf{t}'}(1))$*, where* $L' := \{mp : p \in L\}$*.*
(5) *Compute two integers* i *and* j *with maximum* $j-i$ *such that* $[mi, mj+m-1] \subseteq D$ *and return* $\mathbf{x}^* := \mathbf{t}[p+i, p+j]$*, where* p *is some element of* L*.*

The most time consuming step is Step 4. By Lemma 12, it takes $O(|\mathbf{t}'| \cdot \log|\mathbf{t}'|)$ time, and since $|\mathbf{t}'| = m|\mathbf{t}|$, we have $|\mathbf{t}'| \cdot \log|\mathbf{t}'| = O(|\mathbf{t}| \cdot \log|\mathbf{t}| \cdot \log\sigma)$. Note that Step 1 can be performed in $O(|\mathbf{t}|)$ time because we assume that the letters of $\mathbf{t}$ are integers from $[0, |\mathbf{t}|-1]$. Steps 2, 3 and 5 can clearly be achieved in $O(m|\mathbf{t}|)$ time.

The proof of correctness is similar to the binary case because for every integer i, $[mi, mi+m-1]$ is a subset of D if, and only if, $\mathbf{t}[p+i] = \mathbf{t}[q+i]$ for all p, $q \in L$. □

We conclude by applying the FFT technique of this section to a problem on motifs with gaps. Such motifs may contain *joker* (don't care) symbols `?` that match any symbol. In our example string `AAXBYCCCZAAUBVCCCA`, gapped motifs `AA?B` and `B???C` are synchronous to substring `AA`.

Continuing our theme on maximal representatives, we now consider the question of finding the gapped motif that is synchronous to a given set L of locations of $\mathbf{t}$ and has the largest possible *content*. By the content we mean the number of non-joker symbols in the motif. In our example, for locations $L = \{0, 9\}$ the gapped motif with largest possible content is `AA?B?CCC` (assuming that no jokers are allowed at the beginning or at the end).

A well-known alignment algorithm [2,9–12] finds a maximum-content gapped motif in quadratic time by:

- making k copies of $\mathbf{t}$ where k denotes the cardinality of L,
- aligning them such that the locations in L become on top of each other making one column, and
- reading a consensus motif from the columns of the alignment.

Theorem 16 *Given a string* $\mathbf{t}$ *over* $[0, |\mathbf{t}| - 1]$ *and a subset* $L \subseteq [0, |\mathbf{t}| - 1]$*, a maximum-content gapped motif* $\mathbf{g}^*$ *such that* $L + d \subseteq \mathrm{Loc}_{\mathbf{t}}(\mathbf{g}^*)$ *for some integer* d *can be found in time* $O(|\mathbf{t}| \cdot \log |\mathbf{t}| \cdot \log \sigma)$ *where* σ *denotes the number of distinct letters that occur in* $\mathbf{t}$*.*

PROOF. The algorithm is the same as in the proof of Theorem 15 but now we use the set D in a different way. We should include into the gapped motif a representative of each location of $\mathbf{t}$ whose letter is repeated in $\mathbf{t}$ as required by L, *i.e.*, when aligned according to L, the corresponding column would contain only this letter which therefore should appear in the maximum-content motif.

Such letters can be read from D. Replace Step 5 of the algorithm of Theorem 15 by the following.

(5) *Let* G *be the set of all indices* i *such that* $[mi, mi + m - 1] \subseteq D$*, and let* r *denote the smallest and* s *the largest element of* G*. Then the maximum-content motif* $\mathbf{g}^*$ *is* $\mathbf{g}^* := g_r g_{r+1} \ldots g_{s-1} g_s$ *where for some fixed element* p *of* L *and every* $i \in [r, s]$*,* $g_i := \mathbf{t}[p + i]$ *if* $i \in G$*, and* $g_i :=$ `?` *otherwise.*

Construction of G and $\mathbf{g}^*$ obviously takes time $O(m\,|\mathbf{t}|)$, hence the time-bound of the algorithm stays the same as in Theorem 15. The simple details of the correctness proof are left to the reader. □

6 Shortest sub-synchronous motifs

The aim of this section is to prove that Problem 4 (Shortest Sub-Synchronous Substring) can be solved in quadratic time. The proof relies on the discussion presented in Section 3.2.2 and on the two lemmas below.

We first make the following simple remark on replacing an arbitrary alphabet by integers.

Remark 17 *Let* $\mathbf{t}$ *be a string. If the equality between any two letters of* $\mathbf{t}$ *is decidable in constant time then it is possible to compute from* $\mathbf{t}$*, in* $O(|\mathbf{t}|^2)$ *time, a string* $\tilde{\mathbf{t}}$ *over the integer alphabet* $[0, |\mathbf{t}| - 1]$ *such that for any indices* i *and* j*,* $\tilde{\mathbf{t}}[i] = \tilde{\mathbf{t}}[j]$ *if, and only if,* $\mathbf{t}[i] = \mathbf{t}[j]$*.*

It follows from Remark 17 that the general Shortest Sub-Synchronous Substring problem reduces in quadratic-time to its restriction to instances where the text $\mathbf{t}$ and its substring $\mathbf{x}$ are over the integer alphabet $[0, |\mathbf{t}| - 1]$. As our solution algorithm will be quadratic, we can also afford this quadratic-time alphabet transformation.

Lemma 18 *Let* L *be a set of integers, let* W *be a non-empty set, and let* $(K_w)_{w \in W}$ *be a family of integer sets. Then*

$$\Delta\left(\bigcup_{w \in W} K_w, L\right) = \bigcap_{w \in W} \Delta(K_w, L)\,.$$

PROOF. The simple proof is left to the reader. □

The main machinery of the algorithm for Shortest Sub-Synchronous Substring is described in the proof of the next lemma.

Lemma 19 *Let* L *be a subset of* $[0, n-1]$ *for some* n*, and let* T *be a branching tree whose leaves are bijectively labeled with a subset of* $[0, n-1]$*. The set of all nodes* v *of* T *such that a translate of* $\mathrm{Lab}_T(v)$ *is included in* L *is computable in* $O(n^2)$ *time.*

PROOF. Let k denote the cardinality of L. For each node v of T, define $D_v := \Delta(\mathrm{Lab}_T(v), L)$ (see Definition 11). Then a translate of $\mathrm{Lab}_T(v)$ is included in L if, and only if, $D_v \neq \emptyset$. It follows from Lemma 12 that all D_v's can be constructed in $O(n^2 \log n)$ time. However, we can do better: we can compute the sorted list of all elements of D_v for every node v of T in a bottom-up fashion in $O(kn)$ time using the following two properties.

Property 1 *For each leaf node v, $D_v = L - i$, where i denotes the label of v.*

Property 2 *For each internal node v, $D_v = \bigcap_{w \in W} D_w$, where W denotes the set of all children of v.*

Property 1 is trivial. To prove Property 2, apply Lemma 18 with $K_w := \mathrm{Lab}_T(w)$ for every $w \in W$ and remark that $\bigcup_{w \in W} K_w = \mathrm{Lab}_T(v)$.

Let us examine the time complexity. Bucket sorting L takes $O(n)$ time. After that, D_v is computable in $O(k)$ time for any leaf node v by Property 1. Furthermore, given d sorted lists of at most k integers each, it is possible to compute their sorted intersection in $O(dk)$ time. Thus, it follows from Property 2 that for every internal node v, D_v is computable in $O(dk)$ time, where d denotes the out-degree of v, provided that D_w has already been computed for every child w of v. Hence, the total time requirement is $O(kn)$, which is also $O(n^2)$. □

We can now prove the main result of the section.

Theorem 20 *There exists an algorithm that, given a string $\mathbf{t}$ and a set $L \subseteq [0, |\mathbf{t}| - 1]$ of locations of $\mathbf{t}$, finds in $O(|\mathbf{t}|^2)$ time a shortest string $\mathbf{x}_*$ such that L contains a translate of $\mathrm{Loc}_{\mathbf{t}}(\mathbf{x}_*)$.*

PROOF. The algorithm for the Shortest Sub-Synchronous Substring problem can be sketched as follows.

(1) *Construct the suffix-tree $T_{\mathbf{t}}$ of $\mathbf{t}$.*
(2) *Construct the set, denoted Y, of all nodes v of $T_{\mathbf{t}}$ such that a translate of $\mathrm{Lab}_{T_{\mathbf{t}}}(v)$ is included in L.*
(3) *Find a node $x_* \in Y$ with minimum $\left|\boldsymbol{\lambda}'_{T_{\mathbf{t}}}(x_*)\right|$ and return $\mathbf{x}_* := \boldsymbol{\lambda}'_{T_{\mathbf{t}}}(x_*)$.*

It follows from Section 3.2.2 that the algorithm is correct if the trivial cases $L = [0, |\mathbf{t}| - 1]$ (the root of $T_{\mathbf{t}}$ belongs to Y) and $L = \emptyset$ (Y is empty) are disregarded.

Combining Remark 17 and Theorem 5, we obtain that Step 1 needs $O(|\mathbf{t}|^2)$ time. Step 2 can be implemented in $O(|\mathbf{t}|^2)$ time by Lemma 19. Step 3 can also be naively implemented within the same time bound. □

7 Conclusion

Efficient algorithms for discovery of maximal and minimal representative motifs were presented. The approximate variants of the problem seem to deserve further study; we have just made some initial remarks.

References

[1] A. Apostolico and L. Parida. Incremental paradigms of motif discovery. *J. of Computational Biology*, 11(1):15–25, 2004.

[2] H. Arimura and T. Uno. An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. *J. of Combinatorial Optimization*, 13(3):243–262, 2007.

[3] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. of the 38th Annual Symposium on Foundations of Computer Science (FOCS 1997)*, pages 137–143, 1997.

[4] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *J. of the ACM*, 47(6):987–1011, 2000.

[5] M. J. Fischer and M. S. Paterson. String matching and other products. In R. M. Karp, editor, *Complexity of Computation*, volume VII of *SIAM–AMS Proceedings*, pages 113–125, 1974.

[6] D. Gusfield. *Algorithms on Strings, Trees, and Sequences–Computer Science and Computational Biology*. Cambridge University Press, 1997.

[7] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(2):323–350, 1977.

[8] L. Parida, I. Rigoutsos, A. Floratos, D. E. Platt, and Y. Gao. Pattern discovery on character sets and real-valued data: linear bound on irredundant motifs and an efficient polynomial time algorithm. In *Proc. of the 11th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 297–308, 2000.

[9] J. Pelfrêne, S. Abdeddaïm, and J. Alexandre. Extracting approximate patterns. *J. of Discrete Algorithms*, 3(2–4):293–320, 2005.

[10] N. Pisanti, M. Crochemore, R. Grossi, and M.-F. Sagot. Bases of motifs for generating repeated patterns with wild cards. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2(1):40–50, 2005.

[11] E. Ukkonen. Structural analysis of gapped motifs of a string. In *Proc. of the 32nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2007)*, volume 4708 of *LNCS*, pages 681–690, 2007.

[12] E. Ukkonen. Maximal and minimal representations of gapped and non–gapped motifs of a string. *Theoretical Computer Science*, 2009. In press.

[13] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, 1999.

[14] P. Weiner. Linear pattern matching algorithms. In *Proc. of the 14th Annual Symposium on Switching and Automata Theory (Foundations of Computer Science, FOCS)*, pages 1–11, 1973.