Motif Trie: An Efficient Text Index for Pattern Discovery with Don't Cares[☆]

Roberto Grossi

Dipartimento di Informatica, Università di Pisa, ERABLE *Team INRIA*

Giulia Menconi

Dipartimento di Informatica, Università di Pisa

Nadia Pisanti

Dipartimento di Informatica, Università di Pisa, ERABLE *Team INRIA*

Roberto Trani

Dipartimento di Informatica, Università di Pisa

Søren Vind

DTU Compute, Technical University of Denmark

Abstract

We introduce the *motif trie* data structure, which has applications in pattern matching and discovery in genomic analysis, plagiarism detection, data mining, intrusion detection, spam fighting and time series analysis, to name a few. Here the extraction of recurring patterns in sequential and textual data is one of the main computational bottlenecks. For this, we address the problem of extracting *maximal* patterns with at most k don't care symbols and at least q occurrences, according to a maximality notion we define. We apply the motif trie to this problem, also showing how to build it efficiently. As a result, we give the first algorithm that attains a stronger notion of output-sensitivity, where the cost for an input sequence of n symbols is proportional to the *actual* number of occurrences of each pattern, which is at most n (much smaller in practice). This avoids the best-known cost of $O(n^c)$ per pattern, for constant $c > 1$, which is otherwise impractical for massive sequences with large n.

[☆] A preliminary version of the results has been presented at FSTTCS 2014 [22]. The first and third authors are partially supported by the Italian MIUR under PRIN 2012C4E3KT national research project AMANDA. The last author is supported by a grant from the Danish National Advanced Technology Foundation.

Email addresses: grossi@di.unipi.it (Roberto Grossi), menconigiulia@gmail.com (Giulia Menconi), pisanti@di.unipi.it (Nadia Pisanti), tranir@cli.di.unipi.it (Roberto Trani), sovi@dtu.dk (Søren Vind)

1. Introduction

In *pattern discovery*, the task is to extract the "most important" and frequently occurring patterns from sequences of "objects" such as log files, time series, text documents, datasets or DNA sequences. Each individual object can be as simple as a character from $\{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ or as complex as a `json` record from a log file. What is of interest to us is the potentially very large set of all possible different objects, which we call the *alphabet* Σ, and sequence S built with n objects drawn from Σ.

We define the occurrence of a pattern in S as in *pattern matching* but its importance depends on its statistical relevance, namely, if the number of occurrences is above a certain threshold. However, pattern discovery is not to be confused with pattern matching. The problems may be considered inverse of each other: the former gets an input sequence S from the user, and extracts patterns P and their occurrences from S, where both are unknown to the user; the latter gets S and a given pattern P from the user, and searches for P's occurrences in S, and thus only the pattern occurrences are unknown to the user.

Many notions of patterns exist, reflecting the diverse applications of the problem [11, 4, 19, 21]. We study a natural variation allowing the special don't care character $\star$ in a pattern to mean that the position inside the pattern occurrences in S can be ignored (so $\star$ matches any single character in S). For example, $\mathtt{TA} \star \mathtt{C} \star \mathtt{ACA} \star \mathtt{GTG}$ is a pattern for DNA sequences.

A *motif* is a pattern of *any* length with *at most k don't cares* occurring *at least q times* in S. In this paper, we consider the problem of determining the *maximal* motifs, where any attempt to extend them or replace their $\star$'s with symbols from Σ causes a loss of significant information (where the number of occurrences in S changes). We denote the family of all motifs by M_{qk}, the set of maximal motifs $\mathcal{M} \subseteq M_{qk}$ (dropping the subscripts in $\mathcal{M}$) and let $\text{occ}(m)$ denote the number of occurrences of a motif m inside S. It is well known that M_{qk} can be exponentially larger than $\mathcal{M}$ [15].

1.1. Our results

We show how to efficiently build an index that we call a *motif trie* which is a trie that contains all prefixes, suffixes and occurrences of $\mathcal{M}$, and we show how to extract $\mathcal{M}$ from it. The motif trie is built level-wise, using an oracle $\textsc{Generate}(u)$ that reveals the children of a node u efficiently using properties of the motif alphabet and a bijection between new children of u and intervals in the ordered sequence of occurrences of u. We are able to bound the resulting running time with a strong notion of *output-sensitive* cost, borrowed from the analysis of data structures, where the cost is proportional to the *actual* number $\text{occ}(m)$ of occurrences of each maximal motif m.

Theorem 1. *Given a sequence S of n objects over an alphabet Σ, and two integers $q > 1$ and $k \geq 0$, there is an algorithm for extracting the maximal motifs $\mathcal{M} \subseteq M_{qk}$ and their occurrences from S in $O\Big(n(k+\log\Sigma)+k^3\times\sum_{m\in\mathcal{M}}\operatorname{occ}(m)\Big)$ time.*

Our result may be interesting for several reasons. First, observe that this is an optimal listing bound when the maximal number of don't cares is $k = O(1)$, which is true in many practical applications. The resulting bound is $O(n\log\Sigma + \sum_{m\in\mathcal{M}}\operatorname{occ}(m))$ time, where the first additive term accounts for building the motif trie and the second term for discovering and reporting all the occurrences of each maximal motif.

Second, our bound provides a strong notion of output-sensitivity since it depends on how many times each maximal motif occurs in S. In the literature for enumeration, an output-sensitive cost traditionally means that there is polynomial cost of $O(n^c)$ per pattern, for a constant $c > 1$. This is infeasible in the context of big data, as n can be very large, whereas our cost of $\operatorname{occ}(m) \leq n$ compares favorably with $O(n^c)$ per motif m, and $\operatorname{occ}(m)$ can be actually much smaller than n in practice. This has also implications in what we call "the `CTRL-C` argument," which ensures that we can safely stop the computation for a *specific* sequence S if it is taking too much time[1]. Indeed, if much time is spent with our solution, too many results to be really useful may have been produced. Thus, one may stop the computation and refine the query (change q and k) to get better results. On the contrary, a non-output-sensitive algorithm may use long time without producing any output: It does not indicate if it may be beneficial to interrupt and modify the query.

Third, our analysis improves significantly over the brute-force bound: M_{qk} contains pattern candidates of lengths p from 1 to n with up to $\min\{k,p\}$ don't cares, and so has size $\sum_p |\Sigma|^p \times (\sum_{i=1}^{\min\{k,p\}} \binom{p}{i}) = O(|\Sigma|^n n^k)$. Each candidate can be checked in $O(nk)$ time (e.g. string matching with k mismatches), or $O(k)$ time if using a data structure such as the suffix tree [19]. In our analysis we are able to remove both of the nasty exponential dependencies on $|\Sigma|$ and n in $O(|\Sigma|^n n^k)$. In the current scenario where implementations are fast in practice but skip worst-case analysis, or state the latter in pessimistic fashion equivalent to the brute-force bound, our analysis could explain why several previous algorithms are fast in practice. (We have implemented a variation of our algorithm that is very fast in practice.)

1.2. Applications

Although the motifs discovery problem has found immediate applications in stringology and computational biology, it is highly multidisciplinary and spans a vast number of applications in different areas. This situation is similar to the one for the edit distance problem and dynamic programming. We here give a short

[1] Such an algorithm is also called an anytime algorithm in the literature.

survey of some significant applications, but others are no doubt left out due to the difference in terminology used (see [1] for further references). Computer security researches use patterns in log files to perform intrusion detection and find attack signatures based on their frequencies [9], while commercial anti-spam filtering systems use pattern discovery to detect and block SPAM [18]. In the data mining community pattern discovery is used extensively [13] as a core method in web page content extraction [7]. A core building block of time series analysis is to use pattern discovery on events that occur over time [17, 20]. In plagiarism detection finding recurring patterns across a (large) number of documents is a core primitive to detect if significant parts of documents are plagiarized [6] or duplicated [5, 8]. And finally, in data compression extraction of the common patterns enables a compression scheme that competes in efficiency with well-established compression schemes [3]. In computational biology, motif discovery in biological sequences identifies areas of interest [19, 21, 11, 1]. Being the analysis of biological sequences our target application, in Section 1.3 we will give an overview of methods and problem variants for motifs discovery in this field.

As the motif trie is an index, we believe that it may be of independent interest for storing similar patterns across similar strings. Our result easily extends to real-life applications requiring a solution with two thresholds for motifs, namely, on the number of occurrences in a sequence and across a minimum number of sequences.

1.3. Related work

Finding motifs in biological sequences has many possible problem formulations and applications. These all share the requirement that the motif occurrences should do not have to be identical, but rather just similar: because of sequencing errors that may have taken place, and because of mutations that can be observed in homologous sequences. This is what makes the problem challenging from the algorithmic point of view. The problem formulation varies in crucial parameters such as (i) the frequency required for the motif; (ii) the type of differences that can be observed in different occurrences: bases substitutions only, insertions or deletions of single nucleotides or of short fragments, or possibly of long ones; (iii) the *conservation* of the motif, that is, the amount of such differences that are allowed. Applications range from the detection of transcription factor binding sites (typically identified as short and well conserved motifs without insertion or deletions), to the search for mobile elements or whole genes (long repetitions with larger amount of insertion and deletions and repeated but not necessarily frequently), passing through objects of medium frequency and size with a limited amount of short inserted or deleted fragments, representing for example individual genomic variants within species (polymorphisms).

The literature of algorithmic approaches and software tools for finding motifs and repetitions is vaste, as the variability of the problem formulations leads to a variability of algorithmic strategiess, and often to combinations of them. For finding long repetitions [40], for example, a preprocessing with an efficient and effective filtering [24, 23, 28] turns out to be the only possible combinatorial approach. For short motifs there are several enumerative pattern-driven algorithms [19, 30, 31, 35]. In order to deal with the possible explosion and redundancy of the output, several notion of maximality have been designed as well as algorithms for selecting maximal motifs only [11, 10, 35]. Pushing further the notion of maximality, there are approaches that suggest a notion of *bases* for the motifs, that is a limited size set of motifs that can generate all the others [27, 25, 15, 35]. Other methods resort to indexing techniques [19, 30]. Furthermore, there are algorithms that are designed to detect specific type of motifs such as satellites or tandem repeats [28, 35], palindromes and mirrors [33], gapped or structured motifs [29, 26, 37, 30] and their flexible variants [34, 39] where don't care symbols may represent any number of bases, circular patterns or texts [36, 38], etc.

This paper addresses the problem variant motifs with don't cares, and combines the ideas of indexing, using a maximality notion that we exploit in a twofold manner by bounding the output as well as the intermediate explosion of candidates during the inference phase.

Although the literature on pattern discovery is vast and spans many different fields of applications with various notation, terminology and variations, we could not find time bounds explicitly stated obeying our stronger notion of output-sensitivity, even for pattern classes different from ours. Output-sensitive solutions with a polynomial cost per pattern have been previously devised for slightly different notions of patterns. For example, Parida et al. [16] describe an enumeration algorithm with $O(n^2)$ time per maximal motif plus a bootstrap cost of $O(n^5 \log n)$ time. [2] Arimura and Uno obtain a solution with $O(n^3)$ delay per maximal motif where there is no limitations on the number of don't cares [4]. Similarly, the MadMX algorithm [11] reports dense motifs, where the ratio of don't cares and normal characters must exceed some threshold, in time $O(n^3)$ per maximal dense motif. Our stronger notion of output-sensitivity is borrowed from the design and analysis of data structures, where it is widely employed. For example, searching a pattern P in S using the suffix tree [14] has cost proportional to P's length and its number of occurrences. A one-dimensional query in a sorted array reports all the wanted keys belonging to a range in time proportional to their number plus a logarithmic cost. Therefore it seemed natural to us to extend this notion to enumeration algorithms also.

1.4. Reading guide

Our solution has two natural parts, after the preliminaries in Section 2. In Section 3 we define the *motif trie*, which is an index storing all maximal motifs and their prefixes, suffixes and occurrences. We show how to report only the maximal motifs in time linear in the size of the trie. That is, it is easy

[2] The set intersection problem (SIP) in appendix A of [16] requires polynomial time $O(n^2)$: The recursion tree of depth $\leq n$ can have unary nodes, and each recursive call requires $O(n)$ to check if the current subset has been already generated.

String	TACTGACACTGCCGA
Quorum	$q = 2$
Don't cares	$k = 1$

(a) Input and parameters for running example.

Maximal Motif	Occurrence List
A	2, 6, 8, 15
AC	2, 6, 8
ACTG⋆C	2, 8
C	3, 7, 9, 12, 13
G	5, 11, 14
GA	5, 14
G⋆C	5, 11
T	1, 4, 10
T⋆C	1, 10

(b) Output: Maximal motifs found (and their occurrence list) for the given input.

Figure 1: Example 1: Maximal Motifs found in string

to extract the maximal motifs from the motif trie—the difficulty is to build the motif trie without knowing the motifs in advance. In Section 4 we begin to describe an efficient algorithm for constructing the motif trie and bound its construction time by the number of occurrences of the maximal motifs, thereby obtaining an output-sensitive algorithm. We build the motif trie topdown, starting from the root and expanding each level of nodes u using a suitable procedure Generate(u), described in Sections 5–6, which is at the heart of the computation. Its correctness, along with the total complexity, is discussed in Section 7.

2. Preliminaries

2.1. Strings

We let Σ be the alphabet of the input string $S \in \Sigma^*$ and $n = |S|$ be its length. For $1 \leq i \leq j \leq n$, $S[i, j]$ is the substring of S between index i and j, both included. $S[i, j]$ is the empty string ϵ if $i > j$, and $S[i] = S[i, i]$ is a single character. Letting $1 \leq i \leq n$, a prefix or suffix of S is $S[1, i]$ or $S[i, n]$, respectively. The *longest common prefix* $lcp(x, y)$ is the longest string such that $x[1, |\, lcp(x, y)|] = y[1, |\, lcp(x, y)|]$ for any two strings $x, y \in \Sigma^*$.

2.2. Tries

A trie T over an alphabet Π is a rooted, labeled tree, where each edge (u, v) is labeled with a symbol from Π. All edges to children of node $u \in T$ must be labeled with distinct symbols from Π. We may consider node $u \in T$ as a string generated over Π by spelling out characters from the root on the path towards u. We will use u to refer to both the node and the string it encodes, and $|u|$ to denote its string length. A property of the trie T is that for any string $u \in T$, it also stores all prefixes of u. A compacted trie is obtained by compacting chains of unary nodes in a trie, so the edges are labeled with substrings: the suffix tree for a string is special compacted trie that is built on all suffixes of the string [14].

2.3. Motifs

A motif $m \in \Sigma\,(\Sigma \cup \{\star\})^*\,\Sigma$ consist of symbols from Σ and *don't care characters* $\star \notin \Sigma$. We let the length $|m|$ denote the number of symbols from $\Sigma \cup \{\star\}$ in m, and let $\mathrm{dc}(m)$ denote the number of $\star$ characters in m. Motif m *occurs* at position p in S iff $m[i] = S[p + i - 1]$ or $m[i] = \star$ for all $1 \leq i \leq |m|$. The number of occurrences of m in S is denoted $\mathrm{occ}(m)$. Note that appending $\star$ to either end of a motif m does not change $\mathrm{occ}(m)$, so we assume that motifs starts and ends with symbols from Σ. A *solid block* is a maximal (possibly empty ϵ) substring from Σ^* inside m.

We say that a motif m can be *extended* by adding don't cares and characters from Σ to either end of m. Similarly, a motif m can be *specialized* by replacing a don't care $\star$ in m with a symbol $c \in \Sigma$. An example is shown in Figure 1.

2.4. Maximal motifs

Given an integer quorum $q > 1$ and a maximum number of don't cares $k \geq 0$, we define a family of motifs M_{qk} containing motifs m that have a limited number of don't cares $\mathrm{dc}(m) \leq k$, and occurs frequently $\mathrm{occ}(m) \geq q$. A *maximal motif* $m \in M_{qk}$ cannot be extended or specialized into another motif $m' \in M_{qk}$ such that $\mathrm{occ}(m') = \mathrm{occ}(m)$. Note that extending a maximal motif m into motif $m'' \notin M_{qk}$ may maintain the occurrences (but have more than k don't cares). We let $\mathcal{M} \subseteq M_{qk}$ denote the *set of maximal motifs.*

Motifs $m \in M_{qk}$ that are *left-maximal* or *right-maximal* cannot be specialized or extended on the left or right without decreasing the number of occurrences, respectively. They may, however, be prefix or suffix of another (possibly maximal) $m' \in M_{qk}$, respectively.

(Running) Example 2. *As a running example, consider a frequency quorum $q = 2$, the simple text `TACTGACACTGCCGA`, and $k = 1$ as maximum number of allowed don't cares symbols (Figure 1(a)). The set of maximal motif $\mathcal{M}$ is shown in Figure 1(b) with their occurrences, while the set of all the motifs is $M_{qk} = M_{21} = \mathcal{M} \cup \{\mathtt{CT}, \mathtt{TG}, \mathtt{ACT}, \mathtt{CTG}, \mathtt{ACTG}, \mathtt{A{\star}T}, \mathtt{CTG{\star}C}, \mathtt{TG{\star}C}, \mathtt{C{\star}GA}, \mathtt{AC{\star}G}, \mathtt{A{\star}TG}\}$. All motifs in $\mathcal{M}$ are by definition both right-maximal and left-maximal. Among the other motifs in M_{21}, we have that $\{\mathtt{ACT}, \mathtt{A{\star}T}, \mathtt{ACTG}, \mathtt{C{\star}GA}, \mathtt{AC{\star}G}, \mathtt{A{\star}TG}\}$ are left maximal, and $\{\mathtt{CTG{\star}C}, \mathtt{TG{\star}C}, \mathtt{C{\star}GA}, \mathtt{AC{\star}G}, \mathtt{A{\star}TG}\}$ are right maximal.*

Fact 3. *If motif $m \in M_{qk}$ is right-maximal (resp. left-maximal), then it is a suffix (resp. a prefix) of a maximal motif.*

3. Motif Tries and Pattern Discovery

This section introduces the *motif trie*. This trie is not used for searching but its properties are exploited to orchestrate the search for maximal motifs in $\mathcal{M}$ and obtain a strong output-sensitive cost.

3.1. Efficient representation of motifs

We first give a few simple observations that are key to our algorithms. Consider a suffix tree built on S over the alphabet Σ, which can be done in $O(n \log |\Sigma|)$ time. It is shown in [21, 10] that when a motif m is maximal, its solid blocks correspond to nodes in the suffix tree for S, matching their substrings from the root[3]. For this reason, we introduce a new alphabet, the *maximal solid block alphabet* Π of size at most $2n$, consisting of the strings stored in all the suffix tree nodes.

We can write a maximal motif $m \in M_{qk}$ as an alternating sequence of $\leq k+1$ elements of Π and $\leq k$ don't cares, starting and ending with solid blocks. The possibility of having the empty string rather than a solid block stands for the case of possible consecutive don't cares.

Thus we represent m as a sequence of $\leq k+1$ strings from Π since the don't cares are implicit. By traversing the suffix tree nodes in *preorder* we assign integers to the strings in Π, allowing us to assume that $\Pi \subseteq [0, \ldots, 2n]$ where ϵ is represented by 0. Hence each motif $m \in M_{qk}$ is actually represented as a sequence of $\leq k+1$ integers from 0 to $2n$.

(Running) Example 4. *For our running example of Figure 1, the array A of sorted solid blocks is $[\mathtt{A}, \mathtt{AC}, \mathtt{ACTG}, \mathtt{C}, \mathtt{G}, \mathtt{GA}, \mathtt{T}]$, with blocks numbered from 1 to 7 in this order. Hence $|\Pi| = 8$ because we are including also ϵ, which is always encoded as 0. In this way, $m = \mathtt{ACTG} \star \mathtt{C}$ is compactly represented by the integer sequence 3, 4.*

Note that the order on the integers in Π shares the following grouping property with the strings over Σ.

Lemma 5. *Let A be an array storing the sorted alphabet Π. For any string $x \in \Sigma^*$, the solid blocks represented in Π and sharing x as a common prefix, if any, are grouped together in A in a contiguous segment $A[i, j]$ for some $1 \leq i \leq j \leq |\Pi|$.*

When it is clear from its context, we will use the shorthand $x \in \Pi$ to mean equivalently a string x represented in Π or the integer x in Π that represents a string stored in a suffix tree node. We observe that the set of strings represented in Π is *closed* under the longest common prefix operation: for any $x, y \in \Pi$, $lcp(x, y) \in \Pi$ and it may be computed in constant time after augmenting the suffix tree for S with a lowest common ancestor data structure [12].

Summing up, the above relabeling from Σ to Π only requires the string $S \in \Sigma^*$ and its suffix tree augmented with lowest common ancestor information.

[3] The proofs in [21, 10] can be easily extended to our notion of maximality.

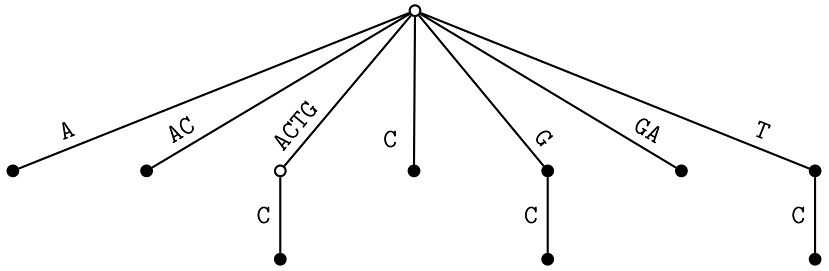

Figure 2: Motif trie for the running example, where ϵ-arcs are not shown.

.

3.2. Motif tries

We now give a sense to the machinery on alphabets described in Section 3.1. For the input sequence S, consider the family M_{qk} defined in Section 2, where each m is seen as a string $m = m[1, \ell]$ of $\ell \leq k{+}1$ integers from 0 to $2n$. Although each m can contain $O(n)$ symbols from Σ, we get a benefit from treating m as a short string over Π: unless specified otherwise, the prefixes and suffixes of m are respectively $m[1, i]$ and $m[i, \ell]$ for $1 \leq i \leq \ell$, where $\ell = \mathrm{dc}(m) + 1 \leq k + 1$. This helps with the following definition as it does not depend on the $O(n)$ symbols from Σ in a maximal motif m but it solely depends on its $\leq k{+}1$ length over Π.

Definition 6 (Motif Trie). *A* motif trie T *is a trie over alphabet* Π *which stores all maximal motifs* $\mathcal{M} \subseteq M_{qk}$ *and their suffixes.*

As a consequence of being a trie, T implicitly stores all prefixes of all the maximal motifs and edges in T are labeled using characters from Π. Hence, all sub-motifs of the maximal motifs are stored in T, and the motif trie can be essentially seen as a generalized suffix trie[4] storing $\mathcal{M}$ over the alphabet Π. From the definition, T has $O((k+1) \cdot |\mathcal{M}|)$ leaves, the total number of nodes is $|T| = O((k+1)^2 \cdot |\mathcal{M}|)$, and the height is at most $k + 1$.

We may consider a node u in T as a string generated over Π by spelling out the $\leq k + 1$ integers from the root on the path towards u. To decode the motif stored in u, we retrieve these integers in Π and, using the suffix tree of S, we obtain the corresponding solid blocks over Σ and insert a don't care symbol between every pair of consecutive solid blocks. When it is clear from the context, we will use u to refer to *(1)* the node u or *(2)* the string of integers from Π stored in u, or *(3)* the corresponding motif from $(\Sigma \cup \{\star\})^*$. We reserve the notation $|u|$ to denote the length of motif u as the number of characters from $\Sigma \cup \{\star\}$. For each node $u \in T$, we denote by L_u the list of occurrences of motif u in S, i.e. u occurs at p in S for every position $p \in L_u$.

[4] As it will be clear later, a compacted motif trie does not give any advantage in terms of the output-sensitive bound compared to the motif trie.

Since child edges for $u \in T$ are labeled with solid blocks, the child edge labels may be prefixes of each other, and one of the labels may be the empty string ϵ (which corresponds to having two neighboring don't cares in the decoded motif).

(Running) Example 7. *The motif tree for our running example is shown in Figure 2. The black nodes are maximal motifs (with their occurrence lists shown in the right column of Figure 1(b))*

3.3. Reporting maximal motifs using motif tries

We now describe how motif tries facilitate the discovery of maximal motifs with don't cares. Suppose we are given a motif trie T but we do not know which nodes of T store the maximal motifs in S. We can identify and report the maximal motifs in T in $O(|T|) = O((k+1)^2 \cdot |\mathcal{M}|) = O(k^2 \cdot \sum_{m \in \mathcal{M}} \text{occ}(m))$ time as follows.

We first identify the set R of nodes $u \in T$ that are right-maximal motifs. A characterization of right-maximal motifs in T is relatively simple: we choose a node $u \in T$ if *(i)* its parent edge label is not ϵ, and *(ii)* u has no descendant v with a non-empty parent edge label such that $|L_u| = |L_v|$. In other words, u cannot be extened with a solid block to its right while keeping the same set of occurrences as L_u. By performing a bottom-up traversal of nodes in T, computing for each node the length of the longest list of occurrences for a node in its subtree with a non-empty edge label, it is easy to find R in time $O(|T|)$ and by Fact 3, $|R| = O((k+1) \cdot |\mathcal{M}|)$.

Next we perform radix sort on the set of pairs $\langle |L_u|, \text{reverse}(u) \rangle$, where $u \in R$ and $\text{reverse}(u)$ denotes the reverse of the string u, to select the motifs that are also left-maximal (and thus are maximal). In this way, the suffixes of the maximal motifs become prefixes of the reversed maximal motifs. By Lemma 5, those motifs sharing common prefixes are grouped together consecutively. However, there is a caveat, as one maximal motif m' could be a suffix of another maximal motif m and we do not want to drop m' when reversed: fortunately, in that case, we have that $|L_m| \neq |L_{m'}|$ by the definition of maximality. Hence, after sorting, we consider consecutive pairs $\langle |L_{u_1}|, \text{reverse}(u_1) \rangle$ and $\langle |L_{u_2}|, \text{reverse}(u_2) \rangle$ in the order, and eliminate u_1 iff $|L_{u_1}| = |L_{u_2}|$ and u_1 is a suffix of u_2 in time $O(k+1)$ per pair (i.e. prefix under reverse). The remaining motifs are maximal (Figure 2).

4. Building Motif Tries

The rest of the paper is devoted to efficiently build the motif trie T discussed in Section 3.2. Suppose without loss of generality that enough new terminator symbols are prepended and appended to the sequence S to avoid border cases.

We proceed in top-down and level-wise fashion by employing an *oracle* that is invoked on each node u on the last level of the partially built trie, and which reveals the future children of u. The oracle is executed many times to generate T level-wise starting from its root u with $L_u = \{1, \ldots, n\}$, and stopping at level $k+1$ or earlier for each root-to-node path. Interestingly, this sounds like the wrong way to do anything efficiently, e.g. it is usually a slow way to build a suffix tree, however the oracle allows us to amortize the total cost to construct the trie.

The oracle is implemented by the GENERATE(u) procedure that generates the children $u_1, \ldots, u_d$ of u. We ensure that *(i)* GENERATE(u) operates on the $\leq k+1$ length motifs from Π, and *(ii)* GENERATE(u) avoids generating the motifs in $M_{qk} \setminus \mathcal{M}$ that are not suffixes or prefixes of maximal motifs. This is crucial, as otherwise we cannot guarantee output-sensitive bounds because M_{qk} can be exponentially larger than $\mathcal{M}$. Algorithm 1 implements the construction.

Algorithm 1: Breadth-First Construction of the Motif-Trie

```
Input  : Text of length $n$.
Output : Motif trie for the input text.

$u \leftarrow$ create the root
$L_u \leftarrow \{1, 2, \ldots, n\}$
$Q \leftarrow$ create an empty queue
ENQUEUE($Q, u$)
while $Q \neq \emptyset$ do
    $u$ = DEQUEUE($Q$)
    if $|L_u| > 1$ and GENERATE($u$) $\neq$ null then
        $\langle u_1, b_1\rangle, \ldots, \langle u_d, b_d\rangle$ = GENERATE($u$) // cumulative for same-level $u$'s
        for $i = 1, 2, \ldots, d$ do
            ENQUEUE($Q, u_i$)
            $(u, u_i) \leftarrow$ new arc labeled with solid block $b_i$
```

In Sections 5–6 we will show how to implement GENERATE(u) and prove:

Lemma 8. *Algorithm* GENERATE(u) *correctly produces the children of* u *and can be implemented in time* $O(\mathrm{sort}(L_u) + (k+1) \cdot |L_u| + \sum_{i=1}^{d} |L_{u_i}|)$.

By summing the cost to execute GENERATE(u) for all nodes $u \in T$, we now bound the construction time of T. Observe that when summing over T the formula stated in Lemma 8, each node exists as u once in the first two terms and as u_i once in the third term, so the latter can be ignored when summing over T (as it is dominated by the other terms)

$$\sum_{u \in T} (\mathrm{sort}(L_u) + (k+1) \cdot |L_u| + \sum_{i=1}^{d} |L_{u_i}|) = O\left(\sum_{u \in T} (\mathrm{sort}(L_u) + (k+1) \cdot |L_u|) \right) . \tag{1}$$

We bound

$$\sum_{u \in T} \mathrm{sort}(L_u) = O\left(n(k+1) + \sum_{u \in T} |L_u| \right) \tag{2}$$

by running a single cumulative radix sort for all the instances over the several nodes u at the same level, allowing us to amortize the additive cost $O(n)$ of the radix sorting among nodes at the same level (and there are at most $k+1$ such levels).

To bound $\sum_{u \in T} |L_u|$, we observe $\sum_i |L_{u_i}| \geq |L_u|$ (as trivially the ϵ extension always maintains the number of occurrences of its parent). Consequently we can charge each leaf u by the cost of its $\leq k$ ancestors, so

$$\sum_{u \in T} |L_u| = O\left((k+1) \times \sum_{\text{leaf } u \in T} |L_u|\right) \quad . \tag{3}$$

Finally, from Section 3.2 there cannot be more leaves than maximal motifs in $\mathcal{M}$ and their suffixes, and the occurrence lists of maximal motifs dominate the size of the non-maximal ones in T, which allows us to bound:

$$\sum_{\text{leaf } u \in T} |L_u| = O\left((k+1) \times \sum_{m \in \mathcal{M}} \text{occ}(m)\right) \quad . \tag{4}$$

By replacing the terms in the total cost of (1) with the upper bounds in (2)–(4), and adding the $O(n \log \Sigma)$ cost for the suffix tree and the LCA ancestor data structure of Section 3.1, we can obtain our main result.

Theorem 9. *Given a sequence S of n objects over an alphabet Σ and two integers $q > 1$ and $k \geq 0$, a motif trie containing the maximal motifs $\mathcal{M} \subseteq M_{qk}$ and their occurrences* occ(m) *in S for $m \in \mathcal{M}$ can be built by Algorithm 1 in time and space $O\Big(n(k + \log \Sigma) + (k+1)^3 \times \sum_{m \in \mathcal{M}} \text{occ}(m)\Big)$.*

5. Generate(u): Motif Trie Nodes as Maximal Intervals

We now discuss the central part of our construction, showing how to implement Generate(u) in the time bounds stated by Lemma 8. The idea is summarized in Algorithm 2.

Algorithm 2: Generate(u)

```
if $\text{dc}(u) \geq k$ then return null
$E_u \leftarrow$ permutation of $L_u$ for the corresponding suffixes in lexicographic order
$\mathcal{I}_u \leftarrow$ MaximalIntervals($E_u$) // see Section 6
$d \leftarrow |\mathcal{I}_u|$ // let $\mathcal{I}_u = \{I_1, \ldots, I_d\}$
for $i = 1, 2, \ldots, d$ do
    $u_i \leftarrow$ create new node
    $b_i \leftarrow \text{LCP}(I_i)$ // longest common prefix of the suffixes in $I_i$
return $\langle u_1, b_1 \rangle, \ldots, \langle u_d, b_d \rangle$
```

We first obtain E_u, which is an array storing the occurrences in L_u, sorted lexicographically according to the suffix associated with each occurrence. We can then show that there is a bijection between the children $u_1, \ldots, u_d$ of u (labeled by solid blocks $b_1, \dots, b_d$) and the set of maximal intervals in E_u: informally speaking, these intervals are maximal under inclusion, as long as the longest common prefix of the suffixes represented by the occurrences is preserved. (As we will see, each solid block b_i is one such longest common prefix.) By exploiting the properties of these intervals, we are able to find them efficiently through $O(1)$ scans of E_u. The bijection implies that we thus efficiently obtain the new children of u. The key point in the efficient implementation of the oracle $\textsc{Generate}(u)$ is to relate each node u and its future children to some suitable intervals that represent their occurrence lists $L_u, L_{u_1}, \dots, L_{u_d}$.

Though the idea of using intervals for representing trie nodes is not new (e.g. in [2]), we use intervals to expand the trie rather than merely representing its nodes. Not all intervals generate children as not all solid blocks that extend u necessarily generate a child. Also, some of the solid blocks $b_1, \dots, b_d$ can be prefixes of each other and one of the intervals can correspond to the empty string ϵ. To select them carefully, we need some definitions and properties.

5.1. Extensions and intervals

Consider a motif u and its list L_u of occurrences: these occurrences match the solid blocks in u, while the characters in S corresponding to the don't cares in u, and the character preceding and succeding u, specialize each occurrence as a substring of S (border cases are easily handled). In our example of Figure 1, motif $u = \texttt{ACGT}\star\texttt{C}$ has two occurrences $L_u = \{2, 8\}$ in S, with one don't care and, clearly, one preceding and one succeeding character for each occurrence $p \in L_u$: for $p = 8$, the preceding character is $\texttt{C}$, then the don't care matches $\texttt{C}$, and the succeeding character is $\texttt{G}$. This motivates the definition of *skipped characters* $\text{skip}(p)$ at position $p \in L_u$, which are the closest $d = \text{dc}(u) + 2$ characters in S that specialize u: formally, $\text{skip}(p) = \langle c_0, c_1, \dots, c_{d-1} \rangle$ where $c_0 = S[p-1]$, $c_{d-1} = S[p + |u|]$, and $c_i = S[p + j_i - 1]$, for $1 \le i \le d-2$, where $u[j_i] = \star$ is the ith don't care in u. In our example, $\text{skip}(p) = \langle \texttt{C}, \texttt{C}, \texttt{G} \rangle$.

Now, the children of u must extend u in the characters following u plus a don't care. Hence these characters should be taken from S after an occurrence of u and its next character, motivating the following definition. For an occurrence $p \in L_u$, we define its *extension* as the suffix $\text{ext}(p, u) = S[p+|u|+1, n]$ that starts at the position after p with an offset equivalent to skipping the prefix matching u plus one character (for the don't care). In our example, $\text{ext}(p, u) = \texttt{GA}$ for $p = 8$. We may write $\text{ext}(p)$, omitting the motif u if it is clear from the context.

Recalling that each suffix $\text{ext}(p)$ can be seen as an integer in Π (see Section 3.1), we denote by E_u the list L_u sorted using as keys $\text{ext}(p)$ where $p \in L_u$. By Lemma 5 consecutive positions in E_u share common prefixes of their extensions. Lemma 10 below states that these prefixes are the candidates for being correct edge labels for expanding u in the trie.

Lemma 10. *Let u_i be a child of node u, b_i be the label of edge (u, u_i), and $p \in L_u$ be an occurrence position. If position $p \in L_{u_i}$ then b_i is a prefix of* $\text{ext}(p, u)$.

Proof Assume otherwise, so $p \in L_u \cap L_{u_i}$ but b_i is not a prefix of $\mathrm{ext}(p, u)$. Then there is a mismatch of solid block b_i in $\mathrm{ext}(p, u)$, since at least one of the characters in b_i is not in $\mathrm{ext}(p, u)$. But this means that u_i cannot occur at position p, and consequently $p \notin L_{u_i}$, which is a contradiction. □

Lemma 10 states a necessary condition, so we have to filter the candidate prefixes of the extensions. We use the following notion of intervals to facilitate this task. We call $I \subseteq E_u$ an *interval* of E_u if I contains consecutive entries of E_u. With an abuse of notation, we write $I \equiv [i, j]$ to actually mean $I = E_u[i], E_u[i+1], \ldots, E_u[j]$. The *longest common prefix* of an interval is defined as $\mathrm{LCP}(I) = \min_{p_1,p_2 \in I} \mathit{lcp}(\mathrm{ext}(p_1), \mathrm{ext}(p_2))$, which is a solid block in Π as discussed at the end of Section 3.1. By Lemma 5, $\mathrm{LCP}(I) = \mathit{lcp}(\mathrm{ext}(E_u[i]), \mathrm{ext}(E_u[j]))$ can be computed in $O(1)$ time, where $E_u[i]$ is the first and $E_u[j]$ the last element in I.

5.2. Maximal and quasi-maximal intervals

Central to our algorithms is the following notion of maximality. An interval $I \subseteq E_u$ is *maximal* if

(1) there are at least q positions in I (i.e. $|I| \geq q$),

(2) motif u cannot be specialized with the same skipped character in $\mathrm{skip}(p)$ simultaneously for all $p \in I$,

(3) any interval $I' \subseteq E_u$ such that $I' \supset I$, has a shorter common prefix (i.e. $|\mathrm{LCP}(I')| < |\mathrm{LCP}(I)|$).

We denote by $\mathcal{I}_u$ the *set of all maximal intervals* of E_u. While conditions *(1)* and *(3)* are intuitive, as we want the largest intervals with $\geq q$ positions that cannot be extended, condition *(2)* is less intuitive but has a dramatic effect on the complexity: it is needed to avoid the enumeration of either motifs from $M_{qk} \setminus \mathcal{M}$ or duplicates from $\mathcal{M}$, recalling that the size of M_{qk} can be exponentially larger than that of $\mathcal{M}$. Condition *(2)* can be equivalently stated by defining C_I as the minimum number of different characters covered by any skipped character in $\mathrm{skip}(p)$ for all $p \in I$, and observing that $C_I \geq 2$ (as otherwise a skipped character in u could be specialized to as single symbol, thus extending a block).

The next lemma establishes a useful bijection between maximal intervals $\mathcal{I}_u$ and children of u, motivating why we use intervals to expand the motif trie.

Lemma 11. *Let u_i be a child of a node u. Then the occurrence list L_{u_i} is a permutation of a maximal interval $I \in \mathcal{I}_u$, and vice versa. The label on edge (u, u_i) is the solid block $b_i = \mathrm{LCP}(I)$. No other children or maximal intervals have this property with u_i or I.*

Proof We prove the statement by assuming that the motif trie T has been built, and that the maximal intervals have been computed for nodes $u \in T$.

We first show that given a maximal interval $I \in \mathcal{I}_u$, there is a single corresponding child $u_i \in T$ of u. Let $b_i = \mathrm{LCP}(I)$ denote the longest common prefix of occurrences in I, and note that b_i is distinct among the maximal intervals in $\mathcal{I}_u$. Also, since b_i is a common prefix for all occurrence extensions in I, the motif

$u \star b_i$ occurs at all locations in I (as we know that u occurs at those locations). Since $|I| \geq q$ and $u \star b_i$ is an occurrence at all $p \in I$, there must be a child u_i of u, where the edge (u, u_i) is labeled b_i and where $I \subseteq L_{u_i}$. From the definition of tries, there is at most one such node. There can be no $p' \in L_{u_i} - I$, since that would mean that an occurrence of $u \star b_i$ was not stored in I, contradicting the maximality assumption of I. Finally, because $C_I \geq 2$ and b_i is the longest common prefix of all occurrences in I, not all occurrences of u_i can be extended to its left using one symbol from Σ. Thus, u_i is a prefix or suffix of a maximal motif.

We now prove the other direction, that given a child $u_i \in T$ of u, we can find a single maximal interval $I \in \mathcal{I}_u$. First, denote by b_i the label on the (u, u_i) edge. From Lemma 10, b_i is a common prefix of all extensions of the occurrences in E_{u_i}. Since not all occurrences of u_i can be extended to its left using a single symbol from Σ, b_i is the longest common prefix satisfying this, and there are at least two different skipped characters of the occurrences in L_{u_i}. Now, we know that $u_i = u \star b_i$ occurs at all locations $p \in L_{u_i}$. Observe that L_{u_i} is a (jumbled) interval of E_u (since otherwise, there would be an element $p' \in E_u$ which did not match u_i but had occurrences from L_{u_i}, contradicting the grouping of E_u). All occurrences of u_i are in L_{u_i} so L_{u_i} is a (jumbled) maximal interval of E_u. We just described a maximal interval with a distinct set of occurrences, at least two different skipped characters and a common prefix, so there must surely be a corresponding interval $I \in \mathcal{I}_u$ such that $\mathrm{LCP}(I) = b_i$, $C_I \geq 2$ and $L_{u_i} \subseteq I$. There can be no $p' \in I - L_{u_i}$, as $p' \in L_u$ and b_i is a prefix of $\mathrm{ext}(p', u)$ means that $p' \in L_{u_i}$. □

An interval that satisfies only conditions *(2)* and *(3)* is called a *quasi-maximal* interval. We do not require that $|I| \geq q$ for any such interval I, as we need it when building larger maximal intervals (see Section 6.3). Since a maximal interval is quasi-maximal, we will refer most of the properties to the latter unless explicitly mentioned. In particular, we show that the set of quasi-maximal intervals, and thus its subset $\mathcal{I}_u$, form a tree covering of E_u. A similar lemma for intervals over the LCP array of a suffix tree was given in [2].

Lemma 12. *Let I_1, I_2 be two quasi-maximal intervals, where $I_1 \neq I_2$ and $|I_1| \leq |I_2|$. Then either I_1 is contained in I_2 with a longer common prefix (i.e. $I_1 \subset I_2$ and $|\mathrm{LCP}(I_1)| > |\mathrm{LCP}(I_2)|$) or the intervals are disjoint (i.e. $I_1 \cap I_2 = \emptyset$).*

Proof Let $I_1 \equiv [i, j]$ and $I_2 \equiv [i', j']$. Assume partial overlaps are possible, $i' \leq i \leq j' < j$, to obtain a contradiction. Since $|\mathrm{LCP}(I_1)| \geq |\mathrm{LCP}(I_2)|$, the interval $I_3 \equiv [j', j]$ has a longest common prefix $|\mathrm{LCP}(I_3)| \geq |\mathrm{LCP}(I_2)|$, and so I_2 could have been extended and was not quasi-maximal, giving a contradiction. The remaining cases are symmetric. □

6. MaximalIntervals(E_u): Finding all maximal intervals $\mathcal{I}_u$ in E_u

We now give the technical details to find all maximal intervals $\mathcal{I}_u$ in E_u, where each interval $I \in \mathcal{I}_u$ corresponds exactly to a distinct child u_i of u. The interval $I = E_u$ corresponding to the solid block ϵ is trivial to find, so we focus on the rest. We assume $|E_u| > 1$ and $\mathrm{dc}(u) < k$, as otherwise we are already done with u. We describe the steps summarized in Algorithm 3 to achieve our goal. The first three steps guarantee conditions *(2)* and*(3)*, thus they find the quasi-maximal intervals; the fourth step enforces condition *(1)*, thus obtaining the maximal intervals.

Algorithm 3: MAXIMALINTERVALS(E_u)

```
1 $\mathcal{R}_u \leftarrow \{[i, R(i)] : R(i)$ exists$\}$, where $R(i) > i$ is the smallest with $C_{[i,R(i)]} \geq 2$
2 $\mathcal{H}_u \leftarrow \{$quasi-maximal intervals with handles$\}$, obtained from $\mathcal{R}_u$'s intervals
3 $\mathcal{H}_u \leftarrow \mathcal{H}_u \cup$ $\{$composite quasi-maximal intervals$\}$, obtained from $\mathcal{H}_u$'s intervals
4 $\mathcal{I}_u \leftarrow \{I \in \mathcal{H}_u : |I| \geq q\}$
5 return $\mathcal{I}_u$ // it always contains $E_u$
```

Let $R(i) > i$ the smallest index in E_u such that $C_{[i,R(i)]} \geq 2$. That is, there are at least two different characters from Σ hidden by each of the skipped characters in the interval. Note that $R(1)$ is always defined while $R(i)$ does not necessarily exists for $i > 1$. We denote the set of these intervals by $\mathcal{R}_u = \{[i, R(i)] : 1 \leq i < |E_u| \text{ and } R(i) \text{ exists}\}$, which are the starting point of our computation.

Lemma 13. *For each quasi-maximal interval $I \equiv [i, j]$, there exists $R(i) \leq j$, and thus $[i, R(i)]$ is an initial portion of I.*

Starting from $\mathcal{R}_u$, we want to find all the quasi-maximal intervals in E_u. To this end, we introduce *handles*. For each $p \in E_u$, its *interval domain* $D(p)$ is the set of intervals $I' \subseteq E_u$ such that $p \in I'$ and $C_{I'} \geq 2$. We let ℓ_p be the length of the solid block that is the longest shared prefix b_i over $D(p)$, namely, $\ell_p = \max_{I' \in D(p)} |\mathrm{LCP}(I')|$. For a quasi-maximal interval I, if $|\mathrm{LCP}(I)| = \ell_p$ for some $p \in I$ we call p a *handle* on I.

Lemma 14. *A position $p \in E_u$ can be the handle for at most one quasi-maximal interval.*

Proof If p is not a handle, the claim is true. If it is so, let I and I' be two distinct quasi-maximal intervals for which p is a handle. Observe that $p \in I \cap I'$. This implies by transitivity that $|\mathrm{LCP}(I)| = |\mathrm{LCP}(I')| = |\mathrm{LCP}(I \cup I')|$, and thus I and I' cannot be quasi-maximal as the interval obtained from $I \cup I'$ cause them to violate condition *(3)*. □

Handles are relevant for the following reason, which motivates the definition of quasi-maximal intervals.

Lemma 15. *For each maximal interval $I \in \mathcal{I}_u$, either there is a handle $p \in E_u$ on I, or I is fully covered by ≥ 2 adjacent quasi-maximal intervals with handles.*

Proof From Lemma 12, any maximal interval $I \in \mathcal{I}_u$ is either fully contained in some other maximal interval, or completely disjoint from other maximal intervals. Partial overlaps of maximal intervals are impossible.

Now, assume there is no handle $p \in L_u$ on I. If so, all $p' \in I$ have $\ell_{p'} \neq |\operatorname{LCP}(I)|$ (since otherwise $p' \in I$ and $\ell_{p'} = |\operatorname{LCP}(I)|$ and thus p' was a handle on I). Clearly for all $p' \in I$, $|\operatorname{LCP}(I)|$ is a lower bound for $\ell_{p'}$. Thus, it must be the case that $\ell_{p'} > |\operatorname{LCP}(I)|$ for all $p' \in I$. This can only happen if I is completely covered by ≥ 2 quasi-maximal intervals, each with a longest common prefix that is larger than $|\operatorname{LCP}(I)|$. From Lemma 12, a single quasi-maximal interval I' is not enough because I' is properly contained (or completely disjoint) in I. □

Lemma 15 gives a clear indication on how to proceed. Let $\mathcal{H}_u$ denote the set of quasi-maximal intervals that have an handle. We first compute $\mathcal{H}_u$. From the definition, a handle on a quasi-maximal interval I' requires $C_{I'} \geq 2$, which is exactly what the intervals in $\mathcal{R}_u$ satisfy. As the LCP value can only drop when extending an interval, these are the only candidates for $\mathcal{H}_u$. Note that from Lemma 13, $\mathcal{R}_u$ contains a prefix for all of the quasi-maximal intervals. Furthermore, $|\mathcal{R}_u| = O(|E_u|)$, since only one $R(i)$ is calculated for each starting position. Among the intervals $[i, R(i)] \in \mathcal{R}_u$, we have to find those with maximum LCP for all p (i.e. where the LCP value equals ℓ_p) that can be expanded. After having computed $\mathcal{H}_u$, we compute the *composite* intervals, namely, those fully covered by ≥ 2 adjacent intervals from $\mathcal{H}_u$ as suggested by Lemma 15. We detail the steps.

6.1. Details of step 1

For each skipped character position, we find all indices where a maximal run of equal characters ends: $R(i)$ is the maximum index for the given i. This helps us because for any index i inside such a block of equal characters, $R(i)$ must be on the right of where the block ends (otherwise $[i, R(i)]$ would cover only one character in that block). Using this to calculate $R(i)$ for all indices $i \in E_u$ from left to right, we find each answer in time $O(k+1)$, and $O((k+1) \cdot |E_u|)$ total time.

Lemma 16. *Step 1 takes $O(\operatorname{sort}(L_u) + (k+1) \cdot |L_u|)$ time.*

6.2. Details of step 2

We show how to find the set $\mathcal{H}_u$ among the intervals of E_u. Observe that for each occurrence $p \in E_u$, we must find the interval I' with the largest $\operatorname{LCP}(I')$ value among all intervals containing p. This is unique by Lemma 14 and, moreover, $|\mathcal{H}_u| \leq |E_u|$. We use an idea similar to that used in Section 3.3 to filter maximal motifs from the right-maximal motifs. We sort the intervals $I' \equiv [i, R(i)] \in \mathcal{R}_u$ in decreasing lexicographic order according to the pairs $\langle |\operatorname{LCP}(I')|, -i \rangle$ (i.e. decreasing LCP values but increasing indices i), to obtain the sequence $\mathcal{D}_u$. Thus, if considering the intervals left to right in $\mathcal{D}_u$, we consider intervals with larger LCP values, from left to right in S for the same value, before moving to smaller LCP values.

Thus we scan $\mathcal{D}_u$ in its order, and consider the generic interval $I \equiv [i, R(i)] \in \mathcal{D}_u$. We define the following intuitive procedure for I, to expand it maximally to the left and right. Consider a border of I, let $a \in I$ be a border occurrence and $b \notin I$ be its neighboring occurrence in E_u (if any, otherwise it is trivial). For example, initially we have $a = i$ and $b = i - 1$ (if any) for the left border of I, and $a = R(i)$ and $b = R(i) + 1$ for the right border. We use pairwise *lcp* queries on the border of the interval: if $|\,lcp(a, b)| < |\,\mathrm{LCP}(I)|$, the interval cannot be expanded to span b; otherwise, we include b in the interval, so that b becomes the new border, and repeat the task. When the above expansion is completed, the resulting I is a quasi-maximal interval in $\mathcal{H}_u$.

During the expansion by the above procedure, we also maintain an array P_u of $|E_u|$ positions, initialized to null before scanning $\mathcal{D}_u$. When the expansion is completed, all elements in the resulting I are marked by writing $P_u[p] = I$ for each ccurrence $p \in I$.

However, before running the above machinery, we have to determine if the resulting I has already been added to $\mathcal{H}_u$ by some previously processed quasi-maximal interval. This avoids duplication as it might be expensive. Since quasi-maximal intervals must be fully contained in each other (from Lemma 12), we determine if $I \equiv [i, R(i)]$ is already fully covered by previously expanded intervals (with larger LCP values), and thus avoid the cost of its expansion, as follows.

- If either i or $R(i)$ is not included in any previous expansions (i.e. $P_u[i]$ or $P_u[R(i)]$ is null), we must expand I.
- If both i and $R(i)$ are part of a single previous expansion I_q (i.e. $P_u[i] = P_u[R(i)] = I_q$), I should be discarded.
- If i and $R(i)$ are part of two different expansions I_q and I_r (i.e. $P_u[i] = I_q \neq I_r = P_u[R(i)]$), I is discarded (even if it could give a new interval).

A comment is in order for the third item, as it is non obvious. It could be that there exists $i' \in I$ with $i < i' < R(i)$, such that $P_u[i']$ is null, thus potentially missing the interval $I' \in \mathcal{H}_u$ with handle i'. However, due to the ordering in $\mathcal{D}_u$, we eventually get $[i', R(i')]$ that is expanded to I' because it satisfies the condition in the first item above. Thus at the end of the scan of $\mathcal{D}_u$, the set $\mathcal{H}_u$ is correctly built.

Lemma 17. *Step 2 takes $O(\mathrm{sort}(L_u) + \sum_{i=1}^{d} |L_{u_i}|)$ time.*

6.3. Details of step 3

A *composite maximal interval* must be the union of a sequence of two or more adjacent intervals from $\mathcal{H}_u$.

For the sake of discussion, suppose first that the intervals in $\mathcal{H}_u$ are disjoint and their union gives E_u, thus $\mathcal{H}_u$ is an ordered partition of E_u where the interval order is the natural one given by their endpoints. Since the intervals induce a tree by Lemma 12 and 14, we can pictorially visualize this situation as

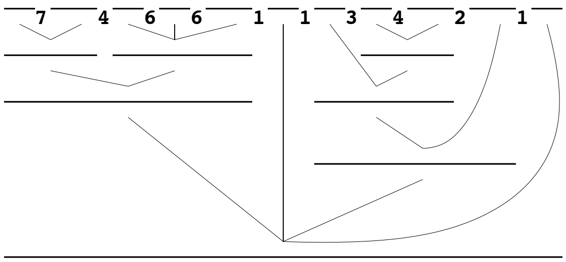

Figure 3: A possible scheme for step 3.

shown in Figure 3. The leaves in the first row are the intervals of $\mathcal{H}_u$: for any two adjacent intervals I and I' we store $|\,\mathrm{LCP}(I \cup I')|$, which can be computed in constant time by Lemma 5.

The composite quasi-maximal intervals are the internal nodes in the next rows, observing that each node has at least two children. The generation is by a simple greedy method: initialize $X = \mathcal{H}_u$ and, while X contains two or more adjacent intervals, take adjacent $I_1, I_2, \ldots, I_r$ from X for the largest r such their value $|\,\mathrm{LCP}(I_i \cup I_{i+1})|$ is maximum and equal for $1 \leq i < r$: replace $I_1, I_2, \ldots, I_r$ in X by their union $I_1 \cup I_2 \cup \cdots \cup I_r$.

In the scheme of Figure 3, we can represent each interval in X by a dash (`-`) and see X as sequence of dashes intermixed with the corresponding values $|\,\mathrm{LCP}(I \cup I')|$. The entries of X change as follows, $\boxed{\texttt{-7-}}$`4-6-6-1-1-3-4-2-1-`, `-4`$\boxed{\texttt{-6-6-}}$`1-1-3-4-2-1-`, $\boxed{\texttt{-4-}}$`1-1-3-4-2-1-`, `-1-1-3`$\boxed{\texttt{-4-}}$`2-1-`, `-1-1`$\boxed{\texttt{-3-}}$`2-1-`, `-1-1`$\boxed{\texttt{-2-}}$`1-`, $\boxed{\texttt{-1-1-1-}}$, `-`, where each box represents the union of two or more intervals.

In the general case for the intervals in $\mathcal{H}_u$, we have a nested situation in place of the first row in Figure 3. But the mechanism is the same: each time we choose adjacent intervals with the maximum LCP value, and replace them by their union. In this way we are exploiting the implicit tree structure of the quasi-maximal intervals.

We implement efficiently our mechanism by an idea similar to that used in Section 6.2. For each interval $I \in \mathcal{H}_u$ that is not the rightmost, check if its adjacent interval I' exists on its right: I and I' must be consecutive in E_u, and if more candidate intervals exist starting at the same position for I' (one prefix of another), choose the longest one by Lemma 12. Associate the value $|\,\mathrm{LCP}(I \cup I')|$ with I. We sort these intervals I in decreasing lexicographic order according to the pairs $\langle |\,\mathrm{LCP}(I \cup I')|, -i \rangle$: the intervals with the largest LCP value come first and it is easy to find those consecutive with the same LCP value. Consequently, scanning this order gives the order for which we make the union of intervals as in Figure 3, namely, starting from the leaves of the implicit tree of the quasi-maximal intervals towards the root. We maintain X as an ordered list of the above intervals.

Lemma 18. *After sorting X in decreasing lexicographic order, the cost of identifying intervals $I_1, I_2, \ldots, I_r$ in X and updating X with their union is $O(r)$ time.*

Proof First we identify adjacent intervals $I_1, I_2, \ldots, I_r$ with the maximum LCP value as the first $r-1$ ones, $I_1, I_2, \ldots, I_{r-1}$, occurring at the beginning of X. We remove these $r-1$ intervals from the beginning of X. Also, it is easy to locate I_r in X as it was associated with I_{r-1} during the sorting: in general we can have some bookkeeping, so that given I_{r-1} we find its associated I_r and vice versa. Let I denote the union $I_1 \cup I_2 \cup \cdots \cup I_r$.

Consider an interval I^* that precedes and is adjacent to I_1 in E_u. Let $\ell^* = |\operatorname{LCP}(I^* \cup I_1)|$ be its LCP value. We prove that $\ell^* = |\operatorname{LCP}(I^* \cup I)|$. Let $\ell = |\operatorname{LCP}(I_1, I_2)| = |\operatorname{LCP}(I)|$ and observe that the extensions of any two positions in I have at least the first ℓ characters equal. Also, $\ell \geq \ell^*$ as $I_1, I_2, \ldots, I_r$ are at the beginning of X. By definition of ℓ^*, the extensions of positions in I^* share the first ℓ characters equal with the extensions of positions in I_1. Since $\ell \geq \ell^*$, by transitivity the extensions of positions in I^* share the first ℓ characters equal with the extensions of positions in I, thus proving our claim. We can safely replace I_1 with I in the bookkeeping, as the interval associated with I^* in the decreasing lexicographic order, because its LCP value does not change.

We also replace I_r by I in X, observing that I inherits the LCP value from I_r. Moreover, this replacement preserves the order in X. Letting i be the starting position of I, and $i_r > i$ that of I_r, the intervals after I_r in X and with the same LCP value also follow I in the decreasing lexicographic order. Consider now an interval I_0 before I_r in X and with the same LCP value, and let $i_0 < i_r$ be its starting position (with I_0 different from $I_1, I_2, \ldots, I_r$). We prove that $i_0 < i$, and thus I_0 precedes also I in the decreasing lexicographic order. Suppose by contradiction that $i_0 \geq i$. Then $I_0 \subset I_j$ for a value of $j \in [1, r]$; its companion interval I'_0 must be $I'_0 \subset I$ as I'_0 cannot occur after I_r in E_u by Lemma 12. But then $|\operatorname{LCP}(I_0 \cup I'_0)| \leq |\operatorname{LCP}(I)|$ with $I_0 \cup I'_0 \subset I$ (properly contained by the hypothesis), which is a contradiction to Lemma 12. In summary, replacing I_r with I in X is correct. □

The total cost of step 3 is dominated by the initial sorting cost $O(\operatorname{sort}(L_u))$ plus the cost of making the union of intervals by Lemma 18. When taken over all the unions, the latter cost is proportional to the number of nodes and leaves in the implicit tree induced by all the quasi-maximal intervals. Since the number of leaves is upper bounded by $|\mathcal{H}_u| \leq |E_u|$, and the number of internal nodes cannot be larger than the number of leaves, as each node has at least two children, we obtain a total of $O(|E_u|)$ nodes and leaves, thus bounding the cost, recalling that $|E_u| = |L_u| = O(\operatorname{sort}(L_u))$.

Lemma 19. *Step 3 takes $O(\operatorname{sort}(L_u))$ time.*

6.4. Details of step 4

We can get all the maximal intervals by filtering the $O(|E_u|)$ quasi-maximal ones using condition *(1)* of Section 5.2. This takes additional $O(|E_u|)$ time.

Lemma 20. *Step 4 takes* $O(\mathrm{sort}(L_u))$ *time.*

7. Correctness and Complexity

By analyzing the algorithm described, one can prove the following two lemmas showing that the motif trie T is generated correctly. While Lemma 21 states that ϵ-extensions may be generated (i.e. a sequence of $\star$ symbols may be added to suffixes of maximal motifs), a simple bottom-up cleanup traversal of T is enough to remove these.

Lemma 21 (Soundness). *Each motif stored in* T *is a prefix or an* ϵ*-extension of some suffix of a maximal motif (encoded using alphabet* Π *and stored in* T*).*

Proof The property to be shown for motif $m \in T$ is: *(1)* m is a prefix of some suffix of a maximal motif $m' \in \mathcal{M}$ (encoded using alphabet Π), or *(2)* m is the suffix of some maximal motif $m' \in \mathcal{M}$ extended by at most k ϵ's (and don't cares).

Note that we only need to show that Generate(u) can only create children of $u \in T$ with the desired property. We prove this by induction. In the basis, u is the root and Generate(u) produce all motifs such that adding a character from Σ to either end decreases the number of occurrences: this is ensured by requiring that there must be more than two different skipped characters in the occurrences considered, using the LCP of such intervals and only extending intervals to span occurrences maintaining the same LCP length. Since there are no don't cares in these motifs they cannot be specialized and so each of them must be a prefix or suffix of some maximal motif.

For the inductive step, we prove the property by construction, assuming $\mathrm{dc}(u) < k$. Consider a child u_i generated by Generate(u) by extending with solid block b_i: it must not be the case that, without losing occurrences, *(a)* u_i can be specialized by converting one of its don't cares into a solid character from Σ, or *(b)* u_i can be extended in either direction using only characters from Σ. If either of these conditions is violated, u_i can clearly not satisfy the property (in the first case, the generalization u_i is not a suffix or prefix of the specialized maximal motif). However, these conditions are sufficient, as they ensure that u_i is encoded using Π and cannot be specialized or extended without using don't cares. Thus, if $b_i \neq \epsilon$, u_i is either a prefix of some suffix of a maximal motif (since u_i ends with a solid block it may be maximal), or if $b_i = \epsilon$, u_i may be an ϵ-extension of u (or a prefix of some suffix if some descendant of u_i has the same number of occurrences and a non-ϵ parent edge).

By the induction hypothesis, u satisfies *(1)* or *(2)* and u is a prefix of u_i. Furthermore, the occurrences of u have more than one different character at all locations covered by the don't cares in u (otherwise one of those locations in u could be specialized to the common character). When generating children, we ensure that *(a)* cannot occur by forcing the occurrence list of generated children to be large enough that at least two different characters is covered by each don't care. That is, u_i may only be created if it cannot be specialized in any location. Condition *(b)* is avoided by ensuring that there are at least two different skipped characters for the occurrences of u_i and forcing the extending block b_i to be maximal under that condition. □

Lemma 22 (Completeness). *If $m \in \mathcal{M}$ then T stores m and its suffixes.*

Proof We summarize the proof that GENERATE(u) is correct and the correct motif trie is produced. From Lemma 15, we create all intervals in GENERATE(u) by expanding those with handles, and expanding all composite intervals from these. By Lemma 11 the intervals found correspond exactly to the children of u in the motif trie. Thus, as GENERATE(u) is executed for all $u \in T$ when $\text{dc}(u) \leq k-1$, all nodes in T is created correctly until depth $k+1$.

Now clearly T contains $\mathcal{M}$ and all the suffixes: for a maximal motif $m \in \mathcal{M}$, any suffix m' is generated and stored in T as *(1)* $\text{occ}(m') \geq \text{occ}(m)$ and *(2)* $\text{dc}(m') \leq \text{dc}(m)$. □

As for the complexity, the whole process of pattern discovery goes as follows. First, we build the motif trie using steps 1–3 of GENERATE(u): Lemma 16, 17, 19 and 20 prove the claimed bound of Lemma 8. Using GENERATE(u) to expand the nodes of the motif trie from the root to the leaves, we obtain the cost of Theorem 9 proved in Section 4 by adding the $O(n \log \Sigma)$ cost for the suffix tree and the LCA ancestor data structure of Section 3.1. Finally, we report the maximal motifs as described in Section 3.3, yielding the final cost of $O(n(k + \log \Sigma) + (k+1)^3 \times \sum_{m \in \mathcal{M}} \text{occ}(m))$ stated in Theorem 1.

Acknowledgements

We are grateful to Giulia Punzi for having read a preliminary version of this paper.

8. Conclusions

In this paper we introduced the motif trie to address the problem of extracting the maximal motifs with don't cares in a sequence. The motif trie is a data structure of independent interest that might find other applications in pattern matching and discovery. It would be interesting to find an efficient algorithm to build the motif tries for flexbile motifs, where the don't care symbol is replace by a Kleene star symbol that matches any sequence, rather than a single character.

[1] M. I. Abouelhoda and M. Ghanem. String mining in bioinformatics. In *Scientific Data Mining and Knowledge Discovery*, pages 207–247, 2010.

[2] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[3] A. Apostolico, M. Comin, and L. Parida. Bridging lossy and lossless compression by motif pattern discovery. In *General Theory of Information Transfer and Combinatorics*, pages 793–813, 2006.

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

[5] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proc. 2nd WCRE*, pages 86–95, 1995.

[6] S. Brin, J. Davis, and H. Garcia-Molina. Copy detection mechanisms for digital documents. In *Proc. ACM SIGMOD*, 24(2):398–409, 1995.

[7] C.-H. Chang, C.-N. Hsu, and S.-C. Lui. Automatic information extraction from semi-structured web pages by pattern discovery. *Decis Support Syst*, 34(1):129–147, 2003.

[8] X. Chen, B. Francia, M. Li, B. Mckinnon, and A. Seker. Shared information and program plagiarism detection. *IEEE Transaction Information Theory*, 50(7):1545–1551, 2004.

[9] H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[10] M. Federico and N. Pisanti. Suffix tree characterization of maximal motifs in biological sequences. *Theoretical Computer Science*, 410(43):4391–4401, 2009.

[11] R. Grossi, A. Pietracaprina, N. Pisanti, G. Pucci, E. Upfal, and F. Vandin. MADMX: A strategy for maximal dense motif extraction. *Journal of Computational Biology*, 18(4):535–545, 2011.

[12] D. Harel and R. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[13] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys*, 43(1):3, 2010.

[14] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of ACM*, 23(2):262–272, 1976.

[15] 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. 11th SODA*, pages 297–308, 2000.

[16] L. Parida, I. Rigoutsos, and D. E. Platt. An Output-Sensitive Flexible Pattern Discovery Algorithm. In *Proc. 12th CPM*, pages 131–142, 2001.

[17] L. Pichl, T. Yamano, and T. Kaizoji. On the symbolic analysis of market indicators with the dynamic programming approach. In *Proc. 3rd ISNN*, pages 432–441, 2006.

[18] I. Rigoutsos and T. Huynh. Chung-Kwei: a Pattern-discovery-based System for the Automatic Identification of Unsolicited E-mail Messages. In *Proc. 1st CEAS*, 2004.

[19] M.-F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *Proc. 3rd LATIN*, pages 374–390. 1998.

[20] R. Sherkat and D. Rafiei. Efficiently evaluating order preserving similarity queries over historical market-basket data. In *Proc. 22nd ICDE*, pages 19–19, 2006.

[21] E. Ukkonen. Maximal and minimal representations of gapped and non-gapped motifs of a string. *Theoretical Computer Science*, 410(43):4341–4349, 2009.

[22] R. Grossi, G. Menconi, N. Pisanti, R. Trani, and S. Vind. Output-Sensitive Pattern Extraction in Sequences. In *Proc. 34th FSTTCS*, pages 303–314, 2014.

[23] P. Peterlongo, G.A. Tominaga Sacomoto, A. Pereira do Lago, N. Pisanti, and M.-F. Sagot. Lossless filter for multiple repeats with bounded edit distance. *Algorithms for Molecular Biology*, 4(3):1–20, 2009.

[24] P. Peterlongo, N. Pisanti, F. Boyer, A. Pereira do Lago, and M.-F. Sagot. Lossless filter for multiple repetitions with Hamming distance. *Journal of Discrete Algorithms*, 6(3):497–509, 2008.

[25] N. Pisanti, M. Crochemore, R. Grossi, and M.-F. Sagot. Bases of Motifs for Generating Repeated Patterns with Wild Cards. *IEEE/ACM Transaction Computational Biology Bioinformatics*, 2(1):40–50, 2005.

[26] C.S. Iliopoulos, J.A.M. McHugh, P. Peterlongo, N. Pisanti, W. Rytter, and M.-F. Sagot. A first approach to finding common motifs with gaps. *International Journal Foundation of Computer Science*, 16(6):1145–1154, 2005.

[27] N. Pisanti, M. Crochemore, R. Grossi, and M.-F. Sagot A Basis of Tiling Motifs for Generating Repeated Patterns and Its Complexity for Higher Quorum. In *Proc. 28th MFCS*, pages 622–631, 2003.

[28] M.-F. Sagot, and G. Myers. Identifying Satellites and Periodic Repetitions in Biological Sequences. *Journal of Computational Biology*, 5(3):539–553, 1998.

[29] M. Crochemore, C.S. Iliopoulos, M. Mohamed, and M.-F. Sagot. Longest repeats with a block of k don't cares. *Theoretical Computer Science*, 362(1-3):248–254, 2006.

[30] E. Eskin, and P.A. Pevzner. Finding composite regulatory patterns in DNA sequences. In *Proc. 10th ISMB*, pages 354–363, 2002.

[31] R.M. Karp, R.E. Miller, and A.L. Rosenberg. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. In *Proc. 4th STOC*, pages 125–136, 1972.

[32] E. Eskin, U. Keich, M.G. Gelfand, and P.A. Pevzner. Genome-Wide Analysis of Bacterial Promoter Regions. In *Proc. 8th PSB*, pages 29–40, 2003.

[33] M.-F. Sagot, and A. Viari. Flexible Identification of Structural Objects in Nucleic Acid Sequences: Palindromes, Mirror Repeats, Pseudoknots and Triple Helices. In *Proc. 8th CPM*, pages 224–246, 1997.

[34] A. Apostolico, M. Comin, and L. Parida. VARUN: Discovering Extensible Motifs under Saturation Constraints. *IEEE/ACM Transaction Computational Biology Bioinformatics*, 7(4):752–762, 2010.

[35] L. Parida, C. Pizzi, and S.E. Rombo. Irredundant tandem motifs. *Theoretical Computer Science*, 525: 89–102, 2014.

[36] J. Lin, Y. Jiang, and D. Adjeroh. Circular pattern discovery *Computer Journal*, 58(5): 1061–1073, 2013.

[37] A. Apostolico, C. Pizzi, and E. Ukkonen. Efficient algorithms for the discovery of gapped factors. *Algorithms for Molecular Biology*, 6(1):5, 2011.

[38] R. Grossi, C.S. Iliopoulos, R. Mercas, N. Pisanti, S.P. Pissis, A. Retha, and F. Vayani. Circular sequence comparison: Algorithms and applications. *Algorithms for Molecular Biology*, 11(1):12, 2016.

[39] A. Dahiya, and D. Garg. Maximal pattern matching with flexible wildcard gaps and one-off constraint. In *Proc. 3rd ICACCI*, pages 1107–1112, 2014.

[40] M. Federico, P. Peterlongo, N. Pisanti, and M.-F. Sagot. Rime: Repeat identification. *Discrete Applied Mathematics*, 163: 275–286, 2014.