Output-Sensitive Pattern Extraction in Sequences

Roberto Grossi[1], Giulia Menconi[1], Nadia Pisanti[1], Roberto Trani[1], and Søren Vind*[2]

1 **Università di Pisa, Dipartimento di Informatica**
grossi@di.unipi.it, menconigiulia@gmail.com, pisanti@di.unipi.it, tranir@cli.di.unipi.it

2 **Technical University of Denmark, DTU Compute**
sovi@dtu.dk

Abstract

Genomic Analysis, Plagiarism Detection, Data Mining, Intrusion Detection, Spam Fighting and Time Series Analysis are just some examples of applications where extraction of recurring patterns in sequences of objects is one of the main computational challenges. Several notions of patterns exist, and many share the common idea of strictly specifying some parts of the pattern and to *don't care* about the remaining parts. Since the number of patterns can be exponential in the length of the sequences, *pattern extraction* focuses on statistically relevant patterns, where any attempt to further refine or extend them causes a loss of significant information (where the number of occurrences changes). Output-sensitive algorithms have been proposed to enumerate and list these patterns, taking polynomial time $O(n^c)$ per pattern for constant $c > 1$, which is impractical for massive sequences of very large length n.

We address the problem of extracting maximal patterns with at most k don't care symbols and at least q occurrences. Our contribution is to give the first algorithm that attains a *stronger* notion of output-sensitivity, borrowed from the analysis of data structures: the cost is proportional to the *actual* number of occurrences of each pattern, which is at most n and practically much smaller than n in real applications, thus avoiding the aforementioned cost of $O(n^c)$ per pattern.

1998 ACM Subject Classification E.1 Data Structures

Keywords and phrases Pattern Extraction, Motif Detection, Pattern Discovery, Motif Trie

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2014.303

1 Introduction

In *pattern extraction*, 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 {A, C, G, 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 extraction 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

* Supported by a grant from the Danish National Advanced Technology Foundation.

© Roberto Grossi, Giulia Menconi, Nadia Pisanti, Roberto Trani, and Søren Vind;
licensed under Creative Commons License CC-BY

34th Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2014).
Editors: Venkatesh Raman and S. P. Suresh; pp. 303–314

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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, $\texttt{TA} \star \texttt{C} \star \texttt{ACA} \star \texttt{GTG}$ 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 $\mathcal{M}$ [16].

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 in 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 $O\Big(n(k + \log \Sigma) + (k+1)^3 \times \sum_{m \in \mathcal{M}} \text{occ}(m)\Big)$ 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 $O(n \log \Sigma + \sum_{m \in \mathcal{M}} \text{occ}(m))$ 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 $\text{occ}(m) \leq n$ compares favorably with $O(n^c)$ per motif m, and $\text{occ}(m)$ 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

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

$\sum_p |\Sigma|^p \times (\sum_{i=1}^{\min\{k,p\}} \binom{p}{i}) = O(|\Sigma|^n n^k)$. 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.)

Related Work. Although the literature on pattern extraction 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. [15] 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.

Applications. Although the pattern extraction problem has found immediate applications in stringology and biological sequences, 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 survey of some significant applications, but others are no doubt left out due to the difference in terminology used (see [1] for further references). In computational biology, motif discovery in biological sequences identifies areas of interest[19, 21, 11, 1]. 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 extraction to detect and block SPAM [18]. In the data mining community pattern extraction is used extensively [13] as a core method in web page content extraction [7] and time series analysis [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].

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.

[2] The set intersection problem (SIP) in appendix A of [15] 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 example.

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

Reading Guide. Our solution has two natural parts. 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 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 and 5 we give 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.

2 Preliminaries

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 $x, y \in \Sigma^*$.

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].

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.

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 $\text{dc}(m) \leq k$, and occurs frequently $\text{occ}(m) \geq q$. A *maximal motif* $m \in M_{qk}$ cannot be extended or specialized into another motif $m' \in M_{qk}$ such that $\text{occ}(m') = \text{occ}(m)$. 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.

▶ **Fact 1.** *If motif $m \in M_{qk}$ is right-maximal then it is a suffix of a maximal motif.*

3 Motif Tries and Pattern Extraction

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 $\mathcal{M}$ to obtain a strong output-sensitive cost. Due to space constraints, all proofs have been omitted in the present version.

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 *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 $\leq k+1$ solid blocks and $\leq k$ don't cares, where the first and last solid block must be different from ε. 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 $\Pi \subseteq [1, \ldots, 2n]$, and so each motif $m \in M_{qk}$ is actually represented as a sequence of $\leq k+1$ integers from 1 to $|\Pi| = O(n)$. Note that the order on the integers in Π shares the following grouping property with the strings over Σ.

▶ **Lemma 2.** *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.

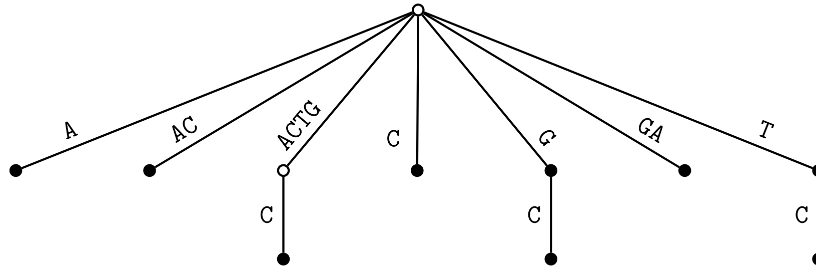

Figure 2 Motif trie for Example 1. The black nodes are maximal motifs (with their occurrence lists shown in Fig. 1(b)).

3.2 Motif Tries

We now exploit 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 1 to $|\Pi|$. 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 3** (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 $O(|T|) = O((k + 1)^2 \cdot |\mathcal{M}|)$, 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 $\Sigma \cup \{\star\}$. Each node $u \in T$ stores a list L_u of occurrences of motif u in S, i. e. u occurs at p in S for $p \in L_u$.

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).

3.3 Reporting Maximal Motifs using Motif Tries

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 $O(|T|) = O((k + 1)^2 \cdot |\mathcal{M}|) = O((k + 1)^2 \cdot \sum_{m \in \mathcal{M}} \mathrm{occ}(m))$ time as follows.

[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.

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 $|L_u| = |L_v|$. 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 1, $|R| = O((k+1) \cdot |\mathcal{M}|)$.

Next we perform a radix sort on the set of pairs $\langle |L_u|, \text{reverse}(u)\rangle$, where $u \in R$ and 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 2, 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': 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.

4 Building Motif Tries

The goal of this section is to show how to efficiently build the motif trie T discussed in Section 3.2. Suppose without loss of generality that enough new symbols are prepended and appended to the sequence S to avoid border cases. We want to store the maximal motifs of S in T as strings of length $\leq k+1$ over Π. Some difficulties arise as we do not know in advance which are the maximal motifs. Actually, we plan to find them *during* the output-sensitive construction of T, which means that we would like to obtain a construction bound close to the term $\sum_{m \in \mathcal{M}} \text{occ}(m)$ stated in Theorem 1.

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 a slow way to build a suffix tree, however the oracle allows us to amortize the total cost to construct the trie. In particular, we can bound the total cost by the total number of occurrences of the maximal motifs stored in the motif 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 $\mathcal{M}$.

In Section 5 we will show how to implement Generate(u) and prove:

▶ **Lemma 4.** *Algorithm* Generate(u) *produces the children of* u *and can be implemented in time* $O(\text{sort}(L_u) + (k+1) \cdot |L_u| + \sum_{i=1}^{d} |L_{u_i}|)$.

By summing the cost to execute procedure 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 4, each node exists once in the first two terms and once in the third term, so the

latter can be ignored when summing over T (as it is dominated by the other terms)

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

We bound

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

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 the cost of its $\leq k$ ancestors, so

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

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:

$$(k+1) \times \sum_{\text{leaf } u \in T} |L_u| = O\left((k+1)^2 \times \sum_{m \in \mathcal{M}} \mathrm{occ}(m)\right) \quad .$$

Adding the $O(n \log \Sigma)$ cost for the suffix tree and the LCA ancestor data structure of Section 3.1, we obtain:

▶ **Theorem 5.** *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 $\mathrm{occ}(m)$ in S for $m \in \mathcal{M}$ can be built in time and space $O\Big(n(k + \log \Sigma) + (k+1)^3 \times \sum_{m \in \mathcal{M}} \mathrm{occ}(m)\Big)$.*

5 Implementing Generate(u)

We now show how to implement Generate(u) within the time bounds stated by Lemma 4. The idea is as follows. 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 of u and a set of maximal intervals in E_u. By exploiting the properties of these intervals, we are able to find them efficiently through a number of scans of E_u. The bijection implies that we thus efficiently obtain the new children of u.

5.1 Nodes of the Motif Trie as Maximal Intervals

The key point in the efficient implementation of the oracle Generate(u) is to relate each node u and its future children $u_1, \ldots, u_d$ labeled by solid blocks $b_1, \ldots, b_d$, respectively, to some suitable intervals that represent their occurrence lists $L_u, L_{u_1}, \ldots, L_{u_d}$. 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 $b_1, \ldots, b_d$ can be prefixes of each other and one of the intervals can be the empty string ε. To select them carefully, we need some definitions and properties.

Extensions. For a position $p \in L_u$, 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 symbol (for the don't care). We may write $\text{ext}(p)$, omitting the motif u if it is clear from the context. We also say that the *skipped characters* $\text{skip}(p)$ at position $p \in L_u$ are the $d = \text{dc}(u) + 2$ characters in S that specialize u into its occurrence p: formally, $\text{skip}(p) = \langle c_0, c_1, \ldots, c_{d-1}\rangle$ 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.

We denote by E_u the list L_u sorted using as keys the integers for $\text{ext}(p)$ where $p \in L_u$. (We recall from Section 3.1 that the suffixes are represented in the alphabet Π, and thus $\text{ext}(p)$ can be seen as an integer in Π.) By Lemma 2 consecutive positions in E_u share common prefixes of their extensions. Lemma 6 below states that these prefixes are the candidates for being correct edge labels for expanding u in the trie.

▶ **Lemma 6.** *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 $\text{ext}(p, u)$.*

Intervals. Lemma 6 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. We write $I = [i, j]$ if I covers the range of indices from i to j in E_u. The *longest common prefix* of an interval is defined as $\text{LCP}(I) = \min_{p_1, p_2 \in I} \mathit{lcp}(\text{ext}(p_1), \text{ext}(p_2))$, which is a solid block in Π as discussed at the end of Section 3.1. By Lemma 2, $\text{LCP}(I) = \mathit{lcp}(\text{ext}(E_u[i]), \text{ext}(E_u[j]))$ can be computed in $O(1)$ time, where $E_u[i]$ is the first and $E_u[j]$ the last element in $I = [i, j]$.

Maximal Intervals. An interval $I \subseteq E_u$ is *maximal* if *(1)* there are at least q positions in I (i. e. $|I| \ge q$), *(2)* motif u cannot be specialized with the skipped characters in $\text{skip}(p)$ where $p \in I$, and *(3)* any other interval $I' \subseteq E_u$ that strictly contains I has a shorter common prefix (i. e. $|\text{LCP}(I')| < |\text{LCP}(I)|$ for $I' \supset I$) [5]. We denote by $\mathcal{I}_u$ the *set of all maximal intervals* of E_u, and show that $\mathcal{I}_u$ 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 7.** *Let $I_1, I_2 \in \mathcal{I}_u$ be two maximal intervals, where $I_1 \ne I_2$ and $|I_1| \le |I_2|$. Then either I_1 is contained in I_2 with a longer common prefix (i. e. $I_1 \subset I_2$ and $|\text{LCP}(I_1)| > |\text{LCP}(I_2)|$) or the intervals are disjoint (i. e. $I_1 \cap I_2 = \emptyset$).*

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

[5] In the full version we show that condition *(2)* is needed to avoid the enumeration of either motifs from $M_{qk} \setminus \mathcal{M}$ or duplicates from $\mathcal{M}$.

5.2 Exploiting the Properties of Maximal Intervals

We now use the properties shown above to implement the oracle GENERATE(u), resulting in Lemma 4. Observe that the task of GENERATE(u) can be equivalently seen by Lemma 8 as the task of finding 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. We describe three main steps, where the first takes $O(\text{sort}(L_u) + (k+1) \cdot |L_u|)$ time, and the others require $O(\sum_{i=1}^{d} |L_{u_i}|)$ time. The interval $I = E_u$ corresponding to the solid block ε is trivial to find, so we focus on the rest. We assume $\text{dc}(u) < k$, as otherwise we are already done with u.

Step 1. Sort occurrences and find maximal runs of skipped characters. We perform a radix-sort of L_u using the extensions as keys, seen as integers from Π, thus obtaining array E_u. To facilitate the task of checking condition *(2)* for the maximality of intervals, we compute for each index $i \in E_u$ the smallest index $R(i) > i$ in E_u such that motif u cannot be specialized with the skipped characters in $\text{skip}(E_u[j])$ where $j \in [i, R(i)]$. That is, there are at least two different characters from Σ hidden by each of the skipped characters in the interval. (If $R(i)$ does not exist, we do not create $[i, R(i)]$.) We define $|P_I|$ as the minimum number of different characters covered by each skipped character in interval I, and note that $|P_{[i,R(i)]}| \geq 2$ by definition.

To do so we first find, for each skipped character position, all indices where a maximal run of equal characters end: $R(i)$ is the maximum indices 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. We denote by $\mathcal{R}$ the set of intervals $[i, R(i)]$ for $i \in E_u$.

▶ **Lemma 9.** *For any maximal interval $I \equiv [i, j] \in \mathcal{I}_u$, there exists $R(i) \leq j$, and thus $[i, R(i)]$ is an initial portion of I.*

Step 2. Find maximal intervals with handles. We want to find all maximal intervals covering each position of E_u. To this end, we introduce *handles.* For each $p \in E_u$, its *interval domain* $D(p)$ is the set of intervals $I' \subset E_u$ such that $p \in I'$ and $|P_{I'}| \geq 2$. We let ℓ_p be the length of the longest shared solid block prefix b_i over $D(p)$, namely, $\ell_p = \max_{I' \in D(p)} |\text{LCP}(I')|$. For a maximal interval $I \subseteq \mathcal{I}_u$, if $|\text{LCP}(I)| = \ell_p$ for some $p \in I$ we call p a *handle* on I. Handles are relevant for the following reason.

▶ **Lemma 10.** *For each maximal interval $I \subseteq \mathcal{I}_u$, either there is a handle $p \in E_u$ on I, or I is fully covered by ≥ 2 adjacent maximal intervals with handles.*

Let $\mathcal{H}_u$ denote the set of maximal intervals with handles. We now 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 $\text{LCP}(I')$ value among all intervals containing p.

From the definition, a handle on a maximal interval I' requires $|P_{I'}| \geq 2$, which is exactly what the intervals in $\mathcal{R}$ satisfy. As the LCP value can only drop when extending an interval, these are the only candidates for maximal intervals with handles. Note that from Lemma 9, $\mathcal{R}$ contains a prefix for all of the (not expanded) maximal intervals because it has all intervals from left to right obeying the conditions on length and skipped character conditions. Furthermore, $|\mathcal{R}| = O(|E_u|)$, since only one $R(i)$ is calculated for each starting position. Among the intervals $[i, R(i)] \in \mathcal{R}$, we will now show how to find those with maximum LCP (i. e. where the LCP value equals ℓ_p) for all p.

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 $I' = [i, R(i)] \in \mathcal{R}$ 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 $\mathcal{C}$. Thus, if considering the intervals left to right in $\mathcal{C}$, we consider intervals with larger LCP values from left to right in S before moving to smaller LCP values.

Consider an interval $I_p = [i, R(i)] \in \mathcal{C}$. The idea is that we determine if I_p has already been added to $\mathcal{H}_u$ by some previously processed handled maximal interval. If not, we expand the interval (making it maximal) and add it to $\mathcal{H}_u$, otherwise I_p is discarded. When $\mathcal{C}$ is fully processed, all occurrences in E_u are covered by maximal intervals with handles.

First, since maximal intervals must be fully contained in each other (from Lemma 7), we determine if $I_p = [i, R(i)] \in \mathcal{C}$ is already fully covered by previously expanded intervals (with larger LCP values) – if not, we must expand I_p. Clearly, if either i or $R(i)$ is not included in any previous expansions, we must expand I_p. Otherwise, if both i and $R(i)$ is part of a single previous expansion $I_q \in \mathcal{C}$, I_p should not be expanded. If i and $R(i)$ is part of two different expansions I_q and I_r we compare their extent values: I_p must be expanded if some $p \in I_p$ is not covered by either I_q or I_r. To enable these checks we mark each $j \in E_u$ with the longest processed interval that contains it (during the expansion procedure below).

Secondly, to expand I_p maximally to the left and right, we use pairwise *lcp* queries on the border of the interval. Let $a \in I_p$ be a border occurrence and $b \notin I_p$ be its neighboring occurrence in E_u (if any, otherwise it is trivial). When $|\,lcp(a, b)| < |\operatorname{LCP}(I_p)|$, the interval cannot be expanded to span b. When the expansion is completed, I_p is a maximal interval and added to $\mathcal{H}_u$. As previously stated, all elements in I_p are marked as being part of their longest covering handled maximal interval by writing I_p on each of its occurrences.

Step 3. Find composite maximal intervals covered by maximal intervals with handles. From Lemma 10, the only remaining type of intervals are composed of maximal intervals with handles from the set $\mathcal{H}_u$. A *composite maximal interval* must be the union of a sequence of adjacent maximal intervals with handles. We find these as follows. We order $\mathcal{H}_u$ according to the starting position and process it from left to right in a greedy fashion, letting $I_h \in \mathcal{H}_u$ be one of the previously found maximal intervals with handles. Each interval I_h is responsible for generating exactly the composite maximal intervals where the sequence of covering intervals starts with I_h (and which contains a number of adjacent intervals on the right). Let $I'_h \in \mathcal{H}_u$ be the interval adjacent on the right to I_h, and create the composed interval $I_c = I_h + I'_h$ (where $+$ indicates the concatenation of consecutive intervals). To ensure that a composite interval is new, we check as in Step 2 that there is no previously generated maximal interval I with $|\operatorname{LCP}(I)| = |\operatorname{LCP}(I_c)|$ such that $I_c \subseteq I$. This is correct since if there is such an interval, it has already been fully expanded by a previous expansion (of composite intervals or a handled interval). Furthermore, if there is such an interval, all intervals containing I_c with shorter longest common prefixes have been taken care of, since from Lemma 7 maximal intervals cannot straddle each other. If I_c is new, we know that we have a new maximal composite interval and can continue expanding it with adjacent intervals. If the length of the longest common prefix of the expanded interval changes, we must perform the previous check again (and add the previously expanded composite interval to $\mathcal{I}_u$).

By analyzing the algorithm described, one can prove the following two lemmas showing that the motif trie is generated correctly. While Lemma 11 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 11** (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).*

▶ **Lemma 12** (Completeness). *If $m \in \mathcal{M}$, T stores m and its suffixes.*

References

1. Mohamed Ibrahim Abouelhoda and Moustafa Ghanem. String mining in bioinformatics. In *Scientific Data Mining and Knowledge Discovery*, pages 207–247. Springer, 2010.
2. Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *JDA*, 2(1):53–86, 2004.
3. Alberto Apostolico, Matteo Comin, and Laxmi Parida. Bridging lossy and lossless compression by motif pattern discovery. In *General Theory of Information Transfer and Combinatorics*, pages 793–813. Springer, 2006.
4. Hiroki Arimura and Takeaki Uno. An efficient polynomial space and polynomial delay algorithm for enumeration of maximal motifs in a sequence. *JCO*, 2007.
5. Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Proc. 2nd WCRE*, pages 86–95, 1995.
6. Sergey Brin, James Davis, and Héctor García-Molina. Copy detection mechanisms for digital documents. *SIGMOD Rec.*, 24(2):398–409, May 1995.
7. Chia-Hui Chang, Chun-Nan Hsu, and Shao-Cheng Lui. Automatic information extraction from semi-structured web pages by pattern discovery. *Decis Support Syst*, 34(1):129–147, 2003.
8. Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE Trans Inf Theory*, 50(7):1545–1551, 2004.
9. Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.
10. Maria Federico and Nadia Pisanti. Suffix tree characterization of maximal motifs in biological sequences. *Theor. Comput. Sci.*, 410(43):4391–4401, 2009.
11. Roberto Grossi, Andrea Pietracaprina, Nadia Pisanti, Geppino Pucci, Eli Upfal, and Fabio Vandin. MADMX: A strategy for maximal dense motif extraction. *J. Comp. Biol.*, 18(4):535–545, 2011.
12. D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.
13. Nizar R. Mabroukeh and Christie I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM CSUR*, 43(1):3, 2010.
14. Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.
15. L. Parida, I. Rigoutsos, and D. E. Platt. An output-sensitive flexible pattern discovery algorithm. In *Proc. 12th CPM*, pages 131–142, 2001.
16. Laxmi Parida, Isidore Rigoutsos, Aris Floratos, Daniel E. Platt, and Yuan 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.
17. Lukáš Pichl, Takuya Yamano, and Taisei Kaizoji. On the symbolic analysis of market indicators with the dynamic programming approach. In *Advances in Neural Networks-ISNN*, pages 432–441. Springer, 2006.
18. Isidore Rigoutsos and Tien Huynh. Chung-Kwei: a Pattern-discovery-based System for the Automatic Identification of Unsolicited E-mail Messages. In *CEAS*, 2004.
19. Marie-France Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *Proc. 3rd LATIN*, pages 374–390. Springer, 1998.
20. Reza Sherkat and Davood Rafiei. Efficiently evaluating order preserving similarity queries over historical market-basket data. In *Proc. 22nd ICDE*, pages 19–19, 2006.
21. Esko Ukkonen. Maximal and minimal representations of gapped and non-gapped motifs of a string. *Theor. Comput. Sci.*, 410(43):4341–4349, 2009.