Optimal Enumeration of Regular Pattern Matches

PAWEŁ GAWRYCHOWSKI*, Institute of Computer Science, University of Wrocław, Poland
FLORIN MANEA*, Department of Informatics, University of Göttingen, Germany
PAUL SARNIGHAUSEN-CAHN*, Department of Informatics, University of Göttingen, Germany
STEFAN SIEMER*, Department of Informatics, University of Göttingen, Germany

Enumerating all matches of a regular expression with capture variables (variable regex, for short) to a document is a fundamental task in textual information extraction, e.g., in connection to document spanners. State-of-the-art results [Amarilli et al., ACM TODS 2021] show how the matches of a variable regex, representable by a sequential variable-set automaton, to a document can be enumerated with delay constant in the size of the document and polynomial in the size of the query, after a preprocessing requiring time linear in the size of the document and polynomial in the size of the query. However, the polynomial dependency on the size of the query in both the delay and the preprocessing is actually quite high: at least quadratic.

We consider here the matching problem for regular expressions with capture variables which can be represented as regular patterns: $w_0x_0w_1x_1\cdots w_kx_kw_{k+1}$, where, for $i \in \{0,\ldots,k+1\}$, w_i is a string of terminal letters and, for $i \in \{0,\ldots,k\}$, x_i is a variable. This problem naturally lies in the intersection of information extraction and stringology, as it can also be seen as computing all the ways in which $k+1$ given strings $w_0,\ldots,w_{k+1}$ occur, in this order and without overlaps, in a given text T. We provide an optimal enumeration algorithm for this problem. After a preprocessing requiring linear time in the size of the document T and the size of the query pattern α, we can enumerate all matches with worst-case constant delay (in combined complexity, i.e., both in the document- and the query-size).

CCS Concepts: • **Theory of computation → Design and analysis of algorithms**.

Additional Key Words and Phrases: Pattern matching; Enumeration algorithms; Variable regex

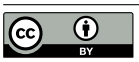
ACM Reference Format:
Paweł Gawrychowski, Florin Manea, Paul Sarnighausen-Cahn, and Stefan Siemer. 2026. Optimal Enumeration of Regular Pattern Matches. *Proc. ACM Manag. Data* 4, 2 (PODS), Article 112 (May 2026), 25 pages. https://doi.org/10.1145/3801908

1 Introduction

A *pattern* (with variables) is a string which consists of *terminal letters* (e. g., $\mathsf{a}, \mathsf{b}, \mathsf{c}$), treated as constants, and *variables* (e. g., x_1, x_2); formally, a pattern is a string $\alpha = (\prod_{i=0}^{k} w_i x_i) w_{k+1}$ where, for $i \in \{0,\ldots,k+1\}$, w_i is a string over the alphabet of terminal letters and, for $i \in \{0,\ldots,k\}$, x_i is a variable. A pattern is mapped to a string by substituting its variables by strings of terminals. For example, $x_1\mathsf{b}x_1\mathsf{bab}x_2\mathsf{a}x_2$ can be mapped to $\mathsf{aabaababbbab}$ by the substitution $(x_1 \to \mathsf{aa}, x_2 \to \mathsf{b})$. If a pattern α can be mapped to a string of terminals T (also called text), we say that α matches T, and a substitution that maps α to T is called *valid* for α and T. The problem of deciding whether

*All authors contributed equally to this research.

Authors' Contact Information: Paweł Gawrychowski, gawry@cs.uni.wroc.pl, Institute of Computer Science, University of Wrocław, Wrocław, Poland; Florin Manea, florin.manea@cs.uni-goettingen.de, Department of Informatics, University of Göttingen, Göttingen, Germany; Paul Sarnighausen-Cahn, paul.cahn@cs.uni-goettingen.de, Department of Informatics, University of Göttingen, Göttingen, Germany; Stefan Siemer, stefan.siemer@cs.uni-goettingen.de, Department of Informatics, University of Göttingen, Göttingen, Germany.



ACM 2836-6573/2026/5-ART112
https://doi.org/10.1145/3801908

there exists a substitution mapping a given pattern α to a given text T (i.e., a valid substitution for α and T) is called the *matching problem.*

PATTERN MATCHING (Match)
Input: Pattern α and text T.
Output: Decide whether there is a valid substitution for α and T.

The matching problem Match for patterns with variables can be canonically seen as a task which requires compiling structured information from an input text document, i.e., a textual *information extraction* task. Indeed, a match between a pattern α (seen as query) and a text T (seen as document) maps every variable occurring in the pattern to a substring of the text, or, in other words, extracts and stores in each variable a substring (also known as a span) of the text. Putting this in context, a pattern can be seen as a regular expression with capture variables (*variable regex*), and the matching problem becomes, as such, a particular case of the problem requiring computing a match between such a variable regex and a document. This is a problem that was heavily investigated (see, e.g., [2, 19, 22] and the references therein) in relation to the study of document spanners [13], an information extraction framework whose main idea is (paraphrasing [19]) to use regular languages to locate the data to be extracted, and variables to store this data. Even more generally, as already pointed in [19], such matching problems can be seen as MSO queries on strings (where capture variables are encoded by pairs of first-order variables); see also [6, 11, 27, 33].

While the variable regex matching problem as well as its particular case, the pattern matching problem, are generally intractable [17, 18], restricted classes of variable regexes for which the matching problem (or, alternatively seen, information extraction problem) becomes tractable were considered in the literature (see, e.g., [17–19, 22, 23, 30] and the references therein). Of particular interest for our work is a class of such variable regexes defined via variable-set automata (or VAs). Intuitively, VAs are nondeterministic finite automata which have transitions labelled with (opening and closing) variable-markers. These markers are used to extract, in a run of the VA on an input text (also called an evaluation of the VA over the input text), substrings of the respective text, and store them in the variables indicated by the markers. A VA is called sequential if each variable marker (both opening and closing) occurs at most once on every accepting run on an input text, and, for each variable, if an opening marker occurs at some point, then the corresponding closing marker also occurs. For more details on such automata see, e.g., [2, 19, 22]. Checking whether a string is accepted by a sequential VA (which can be canonically seen as the matching problem for the variable regexes corresponding to that VA) is clearly tractable (via, e.g., state-set simulation), the key observation being that a matching algorithm does not need to memorize the assignment of each variable. Imposing this restriction on patterns for variables yields the well-studied class of *regular patterns*, which is also in the main focus of this paper. These are patterns (or, alternatively, variable regexes) of the form $\alpha = (\prod_{i=0}^{k} w_i x_i) w_{k+1}$ where, for $i \in \{0, \ldots, k+1\}$, w_i is a string over the alphabet of terminal letters, and, for $i \in \{0, \ldots, k\}$, x_i is a variable, and $x_i \neq x_j$ for all $i \neq j$; they are called *regular* as the languages described by these regexes are clearly regular, which is not the case for languages described by arbitrary patterns, which allow repeated variables.

The corresponding matching problem is the following:

PATTERN MATCHING FOR REGULAR PATTERNS ($\mathtt{Match}_{\mathtt{Reg}}$)
Input: Regular pattern $\alpha = (\prod_{i=0}^{k} w_i x_i) w_{k+1}$ and text T.
Output: Decide whether there is a valid substitution for α and T.

This matching problem can be solved in linear time $O(|T| + |\alpha|)$ by a greedy strategy [16].

A natural task in the context of textual information extraction, studied in [2, 19, 22] for sequential VAs, is the computation of all the evaluations of the VA over the input text, and their efficient enumeration. Initially, in [22] it is shown that one can enumerate the mappings of the variables with delay linear in the input text and quadratic in the size of the VA. Further, in [19] it is shown that, if the VA is fixed (or, in other words, if we consider the *data complexity* perspective), after a processing linear in the size of the input document, one can enumerate the mappings of the variables with constant delay. However, both the preprocessing time and the delay depend exponentially on the size of the VA. Finally, in [2] it is shown that the respective enumeration problem can be solved efficiently in *combined complexity*: after a processing linear in the size of the input document and polynomial in the size of the VA, one can enumerate the mappings of the variables with delay constant in the size of the input text and polynomial in the size of the VA. Worth noting, it was already shown in [6, 11, 27, 33] that the enumeration of MSO Queries on strings (and as such also the computation of all matches of a variable regex to an input text) can be done with constant delay in *data complexity*, but such algorithms were not efficient in *combined complexity*; see the discussion from [19]. We stress that, while the algorithm of [2] achieves delay independent on the size of the input text, its dependency on the size of the VA is actually quite high: at least quadratic. We comment that even checking if a word is accepted by a given nondeterministic automaton [5], or whether it belongs to a language described by a regular expression (even if the regular expression is somewhat restricted) [8], requires quadratic time. Thus, it is natural to seek a restricted (but still nontrivial) class of patterns for which a better time bound can be attained.

As a teaser, we will show that enumerating all the mappings under which a given regular pattern α matches an input document T can be done with constant delay in *combined complexity*, i.e., w.r.t. both $|T|$ (the size of T) and $|\alpha|$ (the size of α), after a linear time preprocessing (again, both w.r.t. $|T|$ and $|\alpha|$). Before going more into details, let us describe the setting in which we obtain these results.

Enumeration framework for $\texttt{Match}_{\texttt{Reg}}$. A match (solution of the matching problem), for an input pattern $\alpha = (\prod_{i=0}^{k} w_i x_i) w_{k+1}$ and a text T of length $|T| = n$, is defined by a *substitution* of the variables x_i by substrings $T[b_i : e_i]$ which is *valid*, in the sense that $w_0 T[b_0 : e_0] w_1 T[b_1 : e_1] \cdots w_k T[b_k : e_k] w_{k+1} = T$. Such a *valid substitution* (or match of α to T) is represented as a k-tuple $(j_1, \ldots, j_k)$, called *valid embedding* of α in T, if, for all $i \in \{0, \ldots, k\}$, x_i is mapped to the substring $T[j_i + |w_i| : j_{i+1} - 1]$, with $j_0 = 1$ and $j_{k+1} = n - |w_{k+1}| + 1$; more precisely, j_i is the position of T where the match of the string w_i (or, in other words, its *embedding* in T) starts in a correct matching of α to T. Note that two valid substitutions (i.e., matches) are different, meaning that at least one variable is mapped to two different substrings of w in the respective substitutions, if and only if the corresponding embeddings are different.

Our goal is to design an algorithm which takes α and T as input, and reports all the matches of α to T, without repetitions. Each such match is given as a valid embedding, which in turn is described by k numbers. Thus, each valid embedding would need $O(k)$ time to be explicitly written down. As standard in many other papers in the area of enumeration algorithms (see the handbook [28], the survey on combinatorial Gray codes [31], as well as any recent paper in this area, for example [3]), we avoid explicitly writing down a full description of each solution (i.e., outputting the string to which each variable is mapped). Instead, we maintain an array `Sol` in which the current valid embedding is stored. Then, during the execution of the algorithm a procedure `report` is called, signalling that the array `Sol` currently contains a valid embedding, which was not enumerated yet; we say that, with this call of `report`, the match corresponding to the respective valid embedding is reported.[1]

[1]Note the similarities with the usage of the `Python` function `yield` in the context of Generators.

The motivation behind such a setting is mainly laid out in [28]. From a theoretical perspective, it allows separating the actual process of generating the embeddings, and, as such, the traversal of the solution space of our problem, from their output. From a practical perspective, it allows for pipelining the enumeration and some further processing: paraphrasing Knuth [28], we are interested in having the solutions to our combinatorial problem momentarily present in a data structure, allowing other programs to examine them; as another example, we might want to evaluate each produced solution and report only the most desirable ones, and it seems plausible that the score of the current solution can be maintained effectively after each change to the `Sol` array; alternatively, as yet another example, we might want to only output the differences between two consecutive solutions. All such cases can be easily formalized in this setting.

Enumeration of Regular Patterns Matches ($\texttt{EnumMatch}_{\texttt{Reg}}$)
Input: Regular pattern $\alpha = (\prod_{i=0}^{k} w_i x_i) w_{k+1}$ and text T.
Output: Report all valid substitutions (as valid embeddings) for α and T.

To analyse the complexity of such an algorithm, we split its computation in two phases. The first phase, called *preprocessing*, consists in the computation done before the first call of `report`; the *preprocessing time* is the time needed, in the worst case, by the respective computation, expressed as a function in k and n. The rest of the computation is the actual *enumeration.* As defined in [25, 26, 28], the *delay* is the time elapsed, in the worst case, between two calls of the `report` function, again expressed as a function in k and n; note that, as already mentioned above, we focus here on the time needed to compute the current embedding from the previous one (that is, the time elapsed between two consecutive calls of `report`), and do not include the time needed to actually output that embedding (or some representation of this embedding). The algorithm is said to have *output-linear delay* if the delay is $O(k)$ and it is said to have *constant delay* if the delay is $O(1)$. Such a constant-delay algorithm has an interesting combinatorial interpretation: enumerating all valid embeddings in such a way that two consecutive valid embeddings differ only on a constant number of positions can be seen as *a combinatorial Gray code* for the class of embeddings of α in T: this is a listing that contains each object from the class exactly once such, that any two consecutive objects in the list differ only by a small change [31]. In this case, one could also explicitly output in $O(1)$ time, when `report` is called, an edit script which summarizes the changes done in the current valid embedding w.r.t. the previous one.

Before moving on to describing our contribution, let us note that, although [2, 19, 22] (as well as [6, 11, 27, 33]) do not explicitly consider this enumeration framework, the complexity aspects of their algorithms hold as described above in this framework as well (most importantly, the delays of the respective algorithms remain as stated, and do not decrease by the fact that the time needed to output the solutions is not counted in the delay).

Our results. Note that in the results reported below, which concern enumeration algorithms, both the preprocessing time and the delay are given in *combined complexity*, meaning that there is no hidden dependency on the size of the input pattern (which is, thus, not considered to be constant).

Florenzano et al. [19] mention that an algorithm running in total time $O(f(n) + |S|)$ would be an ideal solution to the problem of enumerating all matches of a variable regex to a text T, where $|T| = n$, f is a function not depending on the size of the output, and $|S|$ is the size of the output, suggesting that the output is obtained such that the time between generating two of its consecutive elements is $O(1)$ on average. Thus, our first result, shows that all matches of pattern α to text T can be reported in overall $O(n + |S|)$ time (we assume, w.l.o.g., that $|\alpha| \in O(n)$), where S is the set of valid embeddings.

Theorem 1.1. *We can enumerate the set S of all valid embeddings of α in T with constant delay, after a preprocessing running in $O(n + |S|)$ time. The used space is $O(n + \min\{|S|, nk\})$.*

Further, we present an algorithm that still reports all matches of α to T in $O(n + |S|)$ time, but has the additional properties that the preprocessing time is $O(n)$ and that for all $s \leq |S|$, the first s embeddings are reported in $O(n + s)$ time; in other words, this algorithm works with amortized $O(1)$ delay.

Theorem 1.2. *We can enumerate the set S of all valid embeddings of α in T with amortized constant delay, after a preprocessing running in $O(n)$ time. The used space is $O(n + \min\{|S|, nk\})$.*

Finally, we give an algorithm for the considered problem that works with preprocessing time $O(n)$ and constant delay. It is worth noting that employing the algorithms from [2, 19] to the restricted class of regular patterns that we consider would lead to enumeration algorithms with a delay that depends (polynomially) on the length $|\alpha|$ of the input pattern.

Theorem 1.3. *We can enumerate the set S of all valid embeddings of α in T with worst-case constant delay, after a preprocessing running in $O(n)$ time. The used space is $O(n \min\{|S|, nk\})$.*

While Theorem 1.3 improves both previous results, we prefer stating our results in this incremental fashion, which allows us to introduce our tools and their functionality gradually.

The algorithms presented here heavily take advantage, on the one hand, that the strings separating the variables are fixed (not described by regular expressions), which allows us to use tools from stringology and efficient data structures, and, on the other hand, by a deeper analysis and usage of enumeration techniques. This allowed us to obtain (for the first time, as far as we are aware) optimal algorithms in *combined complexity* for a non-trivial subcase of the problem of enumerating all matches of a regular expression with capture variables. While our techniques are tailored towards the special case of regular patterns and unlikely to be directly applicable to the more general cases of [2, 19], we hope that they will spark further interest in transferring tools from stringology to enumerating all matches of a regular expression, for more general classes of regular expressions. Further, we believe that some of our insights, for example Theorem 3.6, might find further applications.

As a first step towards future work, we consider a dynamic version of the problem, which requires building an index for T, that allows us to enumerate matches of regular patterns. More precisely, we are given a text T and then, one by one, regular patterns $\alpha_1, \alpha_2, \ldots$ and we want to report, with constant delay, all matches of α_j to T, after α_j is read. Ideally, we would like to avoid a linear time processing of T for each pattern α_j (as it would be the case if we would apply our previously mentioned algorithms every time). We obtain the following result.

Theorem 1.4. *We can preprocess the text T in $O(n \log n)$ time, so that when given a pattern α with k variables, after an additional processing running in $O(|\alpha| + k \log n)$ time, the set S of all valid embeddings of α in T can be enumerated with constant delay.*

Overview of our algorithms. Our four algorithms are based on a common idea. We can identify, greedily, a left-canonical valid embedding $(\ell_1, \ldots, \ell_k)$ and a right-canonical valid embedding $(r_1, \ldots, r_k)$, such that for every other valid embedding $(j_1, \ldots, j_k)$ we have that $\ell_i \leq j_i \leq r_i$, for all $i \in [k]$. Intuitively, to compute the left-canonical embedding, after checking that the prefix w_0 of the pattern coincides with a prefix of T, we let ℓ_1 to be the leftmost position where w_1 occurs in T after the prefix of length $|w_0|$, ℓ_2 to be the leftmost position where w_2 occurs in T after position $\ell_1 + |w_1| - 1$, and, in general, ℓ_{i+1} is the leftmost position where w_{i+1} occurs in T after position $\ell_i + |w_i| - 1$; we need to also check that the suffix w_{k+1} of α to the suffix of T of length $|w_k|$ and this suffix starts after $\ell_k + |w_k| - 1$. The left-canonical embedding is computed in linear time,

by traversing the text T left to right, and employing standard string matching algorithms. The right-canonical embedding can be computed similarly, but scanning T from right to left. The main observation is now that, for $i \in [k]$, every position j_i, with $\ell_i \leq j_i \leq r_i$, where w_i occurs is part of at least one valid embedding. So, it makes sense to focus on the sets O_i storing all the positions where w_i occurs in T, in the range $T[\ell_i : r_i + |w_i| - 1]$, and a first important observation is that $\sum_{i=1}^{k} |O_i| = O(k + |S|)$, where S is the set of valid embeddings of α in T.

Now, in our first algorithm, from Theorem 1.1, we develop a method to enumerate the elements of all the sets O_i with worst-case constant delay, after a linear time preprocessing (in which we construct a series of indexing data structures for strings, allowing us to retrieve efficiently the occurrences of the strings $w_1, \ldots, w_k$ in T). Then we sort these sets (in linear time w.r.t. their total size, using radix sort), and show that this is enough in order to enumerate the set of valid embeddings of α in T with worst-case constant delay, using a recursive procedure. This leads to an algorithm which enumerates the elements of S in $O(n + |S|)$ time, overall. However, before the enumeration of S starts in this algorithm, we actually use $O(n + \sum_{i=1}^{k} |O_i|) = O(n + |S|)$ time in the preprocessing.

To reduce the preprocessing time, while keeping the enumeration efficient, in Theorem 1.2, we use the first $O(\sum_{i=1}^{k} |O_i|)$ enumeration steps to report, with $O(1)$ delay, embeddings of the form $(\ell_1, \ldots, \ell_{i-1}, j, r_{i+1}, \ldots, r_k))$ (called simple) and also construct simultaneously the sets O_i. Then, we can afford to sort the lists O_i again, in linear time $O(n + \sum_{i=1}^{k} |O_i|)$, as this amortizes with the time spent to enumerate the simple embeddings. Then, we continue the algorithm as in the initial approach, only taking care that we do not output the simple embeddings again. This algorithm does not have constant worst-case delay, but it has the property that, after a preprocessing running in $O(n)$ time, it enumerates the valid embedding with amortized constant delay: the first t embeddings are reported in $O(t)$ time.

The next algorithm, from Theorem 1.3, is loosely based on the idea of the previous algorithm, to retrieve the elements of O_i on the fly, during the enumeration, but avoids computing and sorting these sets in order to keep the delay constant in the worst-case. Instead, it organizes very carefully the calls to the enumeration procedure, making sure that in every call, when we set the i^{th} component of an embedding in order to produce a new valid embedding, we have access to an element of O_i that can be used to define the respective embedding-component.

Finally, the algorithm from Theorem 1.4, shows that a longer preprocessing time, of $O(n \log n)$, allows us to implement the initial approach with worst-case constant delay, and, moreover, to also be able to process and answer efficiently multiple queries without needing to reprocess the entire text. In this case, the idea is to obtain a succinct representation of the sets O_i, allowing us to access their elements in an order that can still be managed efficiently by our initial enumeration algorithm.

Connections to other areas. As mentioned, our results exhibit a combinatorial Gray code for the class of embeddings of a pattern in a text. The study of combinatorial Gray codes is a well developed and intensely studied area of combinatorial algorithms, rooted in the study of Gray codes and with applications in various other areas of theory, see [31].

Matching a regular pattern to a text can be seen as a generalization of the problem of matching a subsequence in a text (which is connected to complex event processing and recognition, see, e.g., [24] and the references therein); indeed, testing whether $u = u[1] \cdots u[m]$ is a subsequence of T is equivalent to testing whether the pattern $x_1 u_1 \cdots x_k u_k x_{k+1}$ matches T. Also, it is easy to observe that the matching problem for a text T and a variable regex built using only concatenation (over terminal letters and variables), where, additionally, each variable occurs only once and either captures a substring of the input regex or is not constrained at all, can be reduced to matching a regular pattern to the respective text T, by simply removing every capture variable in the matching

step, and then computing the substrings they store according to their scope and the values of the unconstrained variables. Without going more into details (and using the formalism of [19]) we give an example: the expression $x\{\mathtt{ab}z\{y\mathtt{bc}\}\}\mathtt{ab}v\{\mathtt{abab}u\}$ can be reduced to the pattern $\mathtt{ab}y\mathtt{bcababab}u$, and for each match of the pattern to the text T (which defines the substrings substituting the unconstrained variables y and u) we can canonically compute the string extracted by every capture variable x, z, v.

While deciding whether a regular pattern α matches a text T is straightforward, enumerating efficiently all variable-assignments under which α matches T can be interpreted as a natural, far less obvious, stringology problem: enumerate all the ways in which the tuple of strings $(w_0, \ldots, w_k)$ occur in the text T, in this order, without overlaps, such that w_0 is a prefix and w_k a suffix of T. Moreover, regular patterns are a well investigated class of pattern languages, especially in the context of algorithmic learning theory, in particular related to the computation of descriptive patterns describing a set of strings [4, 15, 34].

Computational model. We assume the standard unit-cost word RAM with logarithmic word-size ω (so each memory word consists of $\omega \geq \log n$ bits). Standard arithmetical and bitwise operations (and indirect addressing) on such words are assumed to take $O(1)$ time [20]. To avoid clutter, we present our enumeration algorithms as recursive procedures, and we assume tail calls to a recursive function are implemented without adding a new stack frame to the call stack. They can be rephrased as iterative procedures with the same complexity.

2 Preliminaries

Notation. Let $\mathbb{N}$ denote the set of natural numbers. For $m, n \in \mathbb{N}$, we define the intervals $[m : n] = \{m, m+1, \ldots, n\}$, $[m : n) = \{m, m+1, \ldots, n-1\}$ and $[n] = [1 : n]$. Let Σ be a finite alphabet of *terminal symbols*. Let Σ^* denote the set of all strings over Σ, including the empty string ε, and $\Sigma^+ \coloneqq \Sigma^* \setminus \{\varepsilon\}$. For strings $w_1, w_2, \ldots, w_k \in \Sigma^*$, their concatenation is denoted by $\prod_{i=1}^{k} w_i$. For a string $T \in \Sigma^*$, we write $|T| = n$ to denote its *length*, i.e., the number of symbols in T. The symbol occurring on position $i \in [n]$ of T is denoted by $T[i]$. For $1 \leq i \leq j \leq n$, we denote by $T[i : j]$ the string $T[i]T[i+1]\cdots T[j]$; we set $T[i : j] = \varepsilon$ if $i > j$. For indices $i, j \in [n]$ we say $T[1 : i]$ is a prefix, $T[j : n]$ a suffix, and $T[i : j]$ a substring of T.

Recall that the problem discussed here is $\mathtt{EnumMatch}_{\mathtt{Reg}}$. Consider an input instance of this problem: a pattern $\alpha = (\prod_{i=0}^{k} w_i x_i) w_{k+1}$ and a string T, over an alphabet of terminal letters Σ. We note that we can assume without loss of generality that $w_0 = w_{k+1} = \varepsilon$. Otherwise, we write $T = w_0 T' w_{k+1}$ (if this is not possible then there are no matches), and $w = w_0 w' w_{k+1}$. Then, the matches of $\alpha' = x_0(\prod_{i=1}^{k} w_i x_i)$ to w' correspond bijectively to the matches of α and w. We also assume for the rest of this paper that $|\alpha| = m$ and $|T| = n$. Clearly, we can assume that $|w_1 \cdots w_k| = \sum_{i=1}^{k} |w_i| \leq n$ because otherwise there is no solution. Let us also note that the case when there exists some $i \in [k]$ such that $w_i = \varepsilon$ is superfluous, both for the $\mathtt{Match}_{\mathtt{Reg}}$ and $\mathtt{EnumMatch}_{\mathtt{Reg}}$. In the case of $\mathtt{Match}_{\mathtt{Reg}}$ we could simply replace $x_{i-1}x_i$ by a new variable x'_i and solve the problem for the transformed pattern. A similar approach works for $\mathtt{EnumMatch}_{\mathtt{Reg}}$, with the observation that for each valid embedding of the transformed pattern we have to also enumerate the solutions considering all possibilities to split the string assigned to x'_i into strings assigned to the variables x_{i-1} and x_i (or, in other words, to consider all possibilities to embed the empty string w_i between the embedding of w_{i-1} and w_{i+1}). This would only clutter our presentation, so we assume that $w_i \neq \varepsilon$ for all $i \in [k]$. Thus, we assume that the total size of the input is $O(n)$. Finally, as usual in stringology, we assume that the input alphabet is polynomial, that is, $\Sigma = \{1, \ldots, \mathrm{poly}(n)\}$.

Substring search. The basic problem considered in the area of string algorithm is locating an occurrence of a pattern (without variables, i.e., a string) in a text. It is often called the *pattern matching problem*, but, to avoid ambiguity, we call it the substring search problem.

SUBSTRING SEARCH (`substr`)
Input: String w with $|w| = m$ and text T with $|T| = n$.
Output: Find the minimal index $p \in [n - m + 1]$ such that $T[p : p + m - 1] = w$.

It is well known that `substr` can be solved efficiently:

LEMMA 2.1 ([29]). *Let $w, T \in \Sigma^*$ with $|w| = m$ and $|T| = n$. After $O(m)$ time preprocessing, the minimal i such that $T[i - m + 1 : i] = w$, can be found in $O(i + m)$ time. Further, all i such that $T[i - m + 1 : i] = w$ can be enumerated, in increasing order, in $O(n)$ time.*

Radix sort. For a sequence of n keys $(x_1, \ldots, x_n)$, each key being a k-tuple over $[1 : m]$, radix sort is a non-comparison-based sorting algorithm that sorts the keys lexicographically by processing their components from least significant to most significant. Each pass applies counting sort to the corresponding component of the keys. It is known that radix sort can sort n keys of length k in $O((n + m)k)$ time [1]. If m is polynomial n (i.e., upper bounded by $n^{O(1)}$), then radix sort can be implemented in $O(nk)$ time. For a sequence of n variable-length keys $x_1, \ldots, x_n$ (i.e., each key is a tuple, but not all keys have the same size), whose components are from $[1 : m]$, radix sort works in $O(m + \sum_{i=1}^{n} |x_i|)$ time [1]; again, if m is bounded by $n^{O(1)}$, then radix sort works in $O(n + \sum_{i=1}^{n} |x_i|)$ time.

Tries and compacted tries. A trie is a rooted tree in which the edges outgoing from a node are ordered and labelled with distinct symbols from the alphabet. Each node naturally corresponds to a string obtained by concatenating the labels of the edges on its path from the root. For a set of strings $S = \{w_1, \ldots, w_n\}$, with $w_i \in \Sigma^*$ for $i \in [n]$, such that no string is a prefix of another one, there exists a unique trie $\mathcal{T}$ such that there is a one-to-one correspondence between the leaves of $\mathcal{T}$ and the strings. Then, a compacted trie $\mathcal{T}'$ is obtained from a trie $\mathcal{T}$ by replacing every maximal path consisting of nodes $v_0, v_1, \ldots, v_k$ such that v_{i+1} is the only child of v_i, for every $i \in \{0, \ldots, k - 1\}$, with a single edge labelled with the concatenations of the labels on the path. For a trie $\mathcal{T}$ corresponding to a prefix-free set of strings S, its compacted trie $\mathcal{T}'$ consists of $O(n)$ edges, and the label of each of its edges is a substring of some string w_i. Thus, we can store a description of such a compacted trie in $O(n)$ space, by representing the label of each edge as a substring $w_i[b : e]$. The nodes of $\mathcal{T}'$ are called *explicit*, while the nodes of $\mathcal{T}$ that have been dissolved when creating $\mathcal{T}'$ are called *implicit*. The edges outgoing from each node of $\mathcal{T}'$ are arranged in the sorted order of their first symbols, which also corresponds to the lexicographical order of their labels.

Suffix trees and arrays. For a string $T[1 : n]$, its suffix tree ST is the compacted trie corresponding to the non-empty suffixes of $T\$$, where $\$$ is a special symbol not occurring in T and lexicographically smaller than all letters of Σ. Thus, ST has $n + 1$ leaves, each corresponding to a string of the form $T[i : n]\$$. It is known that, under the assumption of polynomial alphabet, the suffix tree of $T[1 : n]$ can be constructed in $O(n)$ time [14]. Next, the suffix array SA is a permutation of $[n]$ corresponding to the lexicographical ordering of the suffixes of $T[1 : n]$. This naturally corresponds to writing down the suffixes corresponding to the leaves of ST (except for the suffix $\$$) arranged in the order corresponding to the pre-order traversal of the whole ST. More generally, for any node v of ST, the pre-order traversal of the subtree rooted at v results in visiting the leaves corresponding to a range $SA[b : e]$. We can compute and store this range $[b : e]$ for every node v with a bottom-up traversal in $O(n)$ time. Finally, if s is the label of the path from the root to v and $SA[b : e]$ is its

corresponding range, then the suffixes of $T[1:n]\$$ that have s as a prefix are exactly the suffixes $T[i:n]\$$ such that i occurs in $SA[b:e]$. One can show the following lemma.

Lemma 2.2. *Let $T[1:n]$ be a text and let $w_1, \ldots, w_k$ be strings with $\sum_{i=1}^{k} |w_i| \leq n$. Then, in total $O(n)$ time, we can compute, for every $i \in [k]$, the range $SA[a_{w_i} : b_{w_i}]$ containing exactly the positions j such that $T[j : j + |w_i| - 1] = w_i$.*

Proof. We build the suffix tree and preprocess its node to store the corresponding ranges of the suffix array. Then, we lexicographically sort all strings $w_1, \ldots, w_k$ with radix sort for variable length keys in $O(n + \sum_{i=1}^{k} |w_i|) = O(n)$ time. Next, for every $i \in [k]$, we want to find the node of the suffix tree corresponding to w_i (the node might be explicit or implicit) and return its corresponding range (if a node is implicit, we return the range of its nearest explicit descendant). To find the node corresponding to w_i, we first go up from the already known node corresponding to w_{i-1} to reach the node corresponding to the longest common prefix of w_{i-1} and w_i, and then we navigate down with the remaining suffix of w_i. Whenever we are at a node v and need to find its outgoing edge with label starting form c, we simply scan the outgoing edges arranged in the lexicographical order. However, when we have to again find the outgoing edge from the same node v, we start the scan from the last previously considered edge (instead of the very first one). Because the strings are considered in the sorted order, this preserves the correctness. Further, each outgoing edge is skipped over at most once, so the total time is as claimed. □

Range minimum/maximum and reporting. For an array $A[1:n]$, a range minimum query (RMQ) is defined as $\texttt{RMinQ}_A(a, b) = \arg\min A[a : b]$ with $1 \leq a \leq b \leq n$. Similarly, a range maximum query is defined as $\texttt{RMaxQ}_A(a, b) = \arg\max A[a : b]$. The following is known.

Lemma 2.3 ([7]). *An array $A[1 : n]$ can be preprocessed in $O(n)$ time to answer range minimum/maximum queries in $O(1)$ time.*

Lemma 2.4. *A permutation of $[n]$ stored in an array $A[1 : n]$ can be preprocessed in $O(n)$ time for the following queries: given integers $a, b, x \in [n]$ with $a \leq b$, enumerate all elements of the set $\{i \in [a : b] \mid A[i] \geq x\}$ with worst-case constant delay.*

Proof. The preprocessing amounts to constructing the RMQ structure from Theorem 2.3 in $O(n)$ time. After receiving the integers $a, b, x \in [n]$ with $a \leq b$, we proceed as follows. We maintain a stack storing intervals $[a_i, b_i]$ such that $\texttt{RMaxQ}_A(a_i, b_i) \geq x$. We first check if $\texttt{RMaxQ}_A(a, b) \geq x$ and if so push the interval $[a, b]$ onto the stack. Then, as long as the stack is non-empty we pop its topmost element $[a_t, b_t]$. We compute $p := \texttt{RMaxQ}_A(a_t, b_t)$ and report p as the next element of the set. Then, if $a_t < p$ and $\texttt{RMaxQ}_A(a_t, p - 1) \geq x$ we push $[a_t, p - 1]$ onto the stack. Similarly, if $p < b_t$ and $\texttt{RMaxQ}_A(p + 1, b_t) \geq x$ we push $[p + 1, b_t]$ onto the stack. It is straightforward to verify that each element of the set is pushed and popped exactly once, and the delay is $O(1)$. □

Observation 2.1. *By combining Lemmas 2.2 and 2.4 we obtain the following primitive: Let $T[1 : n]$ be a text and let $w_1, \ldots, w_k$ be strings with $\sum_{i=1}^{k} |w_i| \leq n$. After $O(n)$ preprocessing, given an index i and a position x, we can enumerate all occurrences of w_i starting at position x or later with $O(1)$ worst-case delay.*

Figure 1 illustrates Observation 2.1.

3 Warm-up: Computing All Valid Embeddings

We recall that a *match* of α to T is represented as a k-tuple $(j_1, \ldots, j_k)$ called *valid embedding*, where j_i is the position in T where w_i is *embedded*.

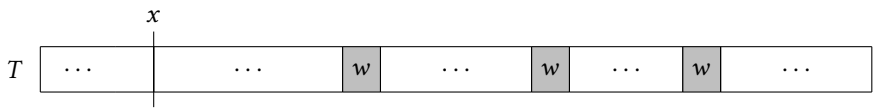


Fig. 1. All occurrences of w in T starting at position x or later.

Definition 3.1. A *left-canonical embedding* $(\ell_1, \ldots, \ell_i, \ldots, \ell_k)$ is a valid embedding where each ℓ_i is the minimal possible among all valid embeddings. Analogously, $(r_1, \ldots, r_i, \ldots, r_k)$ is a *right-canonical embedding* if each r_i is the maximal possible among all valid embeddings.

Lemma 3.2. *We can compute the* left-canonical *and* right-canonical embedding *or decide that they do not exist in $O(n)$ time.*

Proof. We describe the case of computing a *left-canonical embedding*; the right-canonical case can be solved analogously. We proceed greedily from left to right. Using Lemma 2.1, we first find the leftmost occurrence of w_1 in T, starting at position 1. Next, assume that $\ell_1, \ldots, \ell_i$ is already computed, we then search for the leftmost occurrence of w_{i+1} in the remaining suffix $T[\ell_i + |w_i| : n]$ with the same algorithm. If at any step such an occurrence does not exist, we return that no valid embedding exists. If all $w_1, \ldots, w_k$ have been found, then $(\ell_1, \ldots, \ell_k)$ is a *left-canonical embedding*. The entire procedure runs in $O(n)$ time, since the i-th call to the algorithm from Lemma 2.1 takes $O(|w_i| + \ell_i - \ell_{i-1} + |w_i|)$ time, which telescopes to $O(n)$. The *right-canonical embedding* can be obtained analogously by performing the same greedy search on the reversals (and reversed order) of all strings and the text and then translating the indices to the non-reversed text. □

Note that the existence of a left-canonical embedding implies the existence of a right-canonical embedding (they might be, however, identical). From now on, we assume that a valid embedding exists, and so the left- and the right-canonical embeddings also exist. We state their simple properties.

Observation 3.1. *$\ell_i + |w_i| \leq \ell_{i+1}$ and $r_i + |w_i| \leq r_{i+1}$, for all $i \in [k-1]$, so, in particular, $\ell_1 < \ell_2 < \ldots < \ell_k$ and $r_1 < r_2 < \ldots < r_k$. Further, $\ell_i \leq r_i$ for all $i \in [k]$.*

Lemma 3.3. *For all valid embeddings $(j_1, \ldots, j_k)$, we have that $j_i \in [\ell_i : r_i]$, for all $i \in [k]$. Moreover, every occurrence of w_i starting in $T[\ell_i : r_i]$ belongs to a valid embedding.*

Proof. Let us consider the contraposition. If $j_i < \ell_i$, then $j_1, \ldots, j_i$ can not be matched in the prefix $T[1 : \ell_i - 1]$. Analogously, when $j_i > r_i$, then $j_i, \ldots, j_k$ can not be matched in the suffix $T[r_i + 1 : n]$. Therefore, it holds that $j_i \in [\ell_i : r_i]$.

For the second part of our claim, consider such an occurrence starting at position j_i. From the first part of this lemma, we know that $j_i \in [\ell_i : r_i]$. Therefore, $(\ell_1, \ldots, \ell_{i-1}, j_i, r_{i+1}, \ldots, r_k)$ is a valid embedding by Observation 3.1. □

Let $O_i \coloneqq \{p \in [\ell_i : r_i] \mid T[p : p + |w_i| - 1] = w_i\}$ denote the set of valid occurrences of w_i, i.e., starting positions of w_i in T, that belong to at least one valid embedding. Further, let S be the set of all valid embeddings. Note that for $j_i \in O_i \setminus \{\ell_i\}$ every embedding $(\ell_1, \ldots, \ell_{i-1}, j_i, r_{i+1}, \ldots, r_k)$ is valid, and every two such embeddings are distinct. Hence, the following result follows immediately.

Lemma 3.4. $\sum_{i=1}^{k} (|O_i| - 1) \leq \min\{kn, |S|\}$.

Proof. It is immediate that $\sum_{i=1}^{k} (|O_i| - 1) \leq kn$, as we trivially have that $|O_i| \leq n$. So, we only show that $\sum_{i=1}^{k} (|O_i| - 1) \leq |S|$.

For $i \in [k]$, we construct valid embeddings $(j_1, \ldots, j_{i-1}, j_i, j_{i+1}, \ldots, j_k)$ as follows. We fix $j_1 = \ell_1, ..., j_{i-1} = \ell_{i-1}$ and $j_{i+1} = r_{i+1}, ..., j_k = r_k$. Then, choosing any $j_i \in O_i \setminus \{\ell_i\}$ results in a valid embedding (by Theorem 3.3), and the valid embeddings obtained in this way, for some i, are pairwise distinct because they differ on the i-th coordinate. Further, two embeddings constructed as above for different values of i, i_1 and i_2, differ on position $\min\{i_1, i_2\}$, so are also distinct. Thus, we have constructed $\sum_{i=1}^{k}(|O_i| - 1)$ distinct valid embeddings. The result follows. □

Next, we design an algorithm that outputs efficiently the set S of all valid embeddings. To set a baseline, we first note that, given the sorted sets O_i, we can output all valid embeddings by a straightforward recursive approach: after setting $\texttt{Sol}[i]$, we set $\texttt{Sol}[i+1]$ to a value from O_{i+1}, greater or equal to $\texttt{Sol}[i]$, and recurse.

Lemma 3.5. *Given sorted sets $O_1, \ldots, O_k$, all valid embeddings can be enumerated in $O(|S|k)$ time.*

Proof. We compute the set of valid embeddings recursively. We maintain an array Sol, such that before a call to report we have $\texttt{Sol}[i] = j_i$, for all $i \in [k]$, i.e., the array stores the current embedding. Assume that $j_1, \ldots, j_{i-1}$ have already been fixed. For the current index i, we iterate in decreasing order over the sorted set $O_i^x := \{j_i \mid j_i \in O_i, j_i \geq x\}$ where $x = j_{i-1} + |w_{i-1}|$ is passed as an argument in the recursive call from j_{i-1}. For each choice of j_i, we set $\texttt{Sol}[i] = j_i$ and recurse. Whenever $i = k$ after updating $\texttt{Sol}[k]$, we report the current array Sol. This clearly computes all valid embeddings one by one and takes $O(k)$ between two calls to report. □

We will now strengthen the above lemma in two directions. First, we will replace the requirement that the lists $O_1, \ldots, O_k$ are given sorted by two requirements (which trivially hold for sorted lists). Second, we will improve the delay between the enumeration of two embeddings to $O(1)$, and obtain a total $O(|S|)$ time to enumerate S.

Lemma 3.6. *Provided data structures allowing us to:*

- *enumerate, for a given i and a number x, with constant delay, all elements of O_i greater or equal to x, starting with r_i,*
- *retrieve the largest two elements in O_i in $O(1)$ time,*

all valid embeddings can be enumerated, after an additional preprocessing running in $O(k)$ time, with constant delay. The space used, additionally to that allocated for the provided data structures, is $O(k)$.

Proof. We define a new recursive function enum where, as opposed to the previous approach, each recursive call will report a valid embedding. The function has two parameters: an integer i and a boolean b. When the function is called, the prefix $\texttt{Sol}[1], \ldots, \texttt{Sol}[\texttt{i} - 1]$ has been already fixed, and additionally $\texttt{Sol}[\texttt{i}] = r_i, \ldots, \texttt{Sol}[\texttt{k}] = r_k$ holds. The goal of the function is to report all valid embeddings with the fixed prefix, except for the valid embedding currently stored in the Sol array. We require that $\ell_i < r_i$, so at least one valid embedding will be indeed reported in the recursive call. Further, b stores the parity of the depth of the recursive call (this will be possibly different than simply the parity of i, and will be explained later).

To be able to implement the recursive function efficiently, we first define an auxiliary function $\mathsf{jump} : [k] \to [k+1]$ which basically guides our recursive approach more efficiently. The function is defined as $\mathsf{jump}(k) = k + 1$ and for each $i \in [k-1]$:

$$\mathsf{jump}(i) = \begin{cases} i+1, & \text{if } |\{x \in O_{i+1} \mid x \geq r_i + |w_i|\}| \geq 2, \\ \mathsf{jump}(i+1), & \text{otherwise.} \end{cases}$$

This function computes, for some i, the smallest $j > i$ such that there are two occurrences of w_j, which can still lead to a valid embedding, i.e., to the right of $r_{j-1} + |w_{j-1}| - 1$. In other words, it allows us to jump, in our recursion, over the positions of that embedding that can only be assigned

as in the right-canonical embedding. The function jump is computed in $O(k)$ time during the preprocessing as follows. We proceed with i from $k-1$ down to 1, and when computing $\mathsf{jump}(i)$, we simply retrieve the largest two elements in O_{i+1} and, if both are greater or equal to $r_i + |w_i|$, we set $\mathsf{jump}(i) = i+1$; otherwise, we set $\mathsf{jump}(i) = \mathsf{jump}(i+1)$.

After that, still in the preprocessing phase, we first set the elements of `Sol` to be $(r_1, \ldots, r_k)$, defining a right-canonical embedding. We `report` the current embedding, and if $\mathsf{jump}(1) \neq k+1$ (otherwise we are already done) then we call $\mathsf{enum}(\mathsf{jump}(1), \mathtt{false})$.

In the recursive function, we enumerate with constant delay (as allowed by this lemma's prerequisites), all elements of O_i greater or equal to $j_{i-1} + |w_{i-1}|$ (for convenience, we assume $j_0 = 1$ and $w_0 = \varepsilon$). We initially disregard the current element if it is equal to r_i, and note that this does not increase the delay as it can happen only once and $|O_i| \geq 2$. Let the currently considered element of O_i be $x \neq r_i$. We set $\mathtt{Sol}[i] = x$ and if $b = \mathtt{false}$ then we `report` the current embedding. Further, we check if the largest two elements of O_{i+1} are greater or equal to $x + |w_i|$, and if so we call $\mathsf{enum}(i+1, \mathtt{not}\ \ b)$. Otherwise, if $\mathsf{jump}(i+1) \neq k+1$ we call $\mathsf{enum}(\mathsf{jump}(i+1), \mathtt{not}\ \ b)$. Finally, if $b = \mathtt{true}$ then we `report` the current embedding. Once we have enumerated all elements of $O_s \setminus \{r_s\}$, we set $\mathtt{Sol}[i] = r_i$, and if $\mathsf{jump}(i) \neq k+1$ then we call $\mathsf{enum}(\mathsf{jump}(i), \mathtt{not}\ \ b)$. This very last recursive call is implemented with tail recursion, so that we don't need to account for the time to return from the recursive call. The pseudocode of this procedure is given below.

Algorithm 1: The main call and the recursive procedure `enum`.

```
for i ∈ [k] do
    Sol[i] ← r_i;
report;
if jump(1) ≠ k + 1 then
    enum(jump(1), false);
function enum(i, b):
    for x ∈ O_i \ {r_i} s.t. x ≥ j_{i−1} + |w_{i−1}| do
        Sol[i] ← x;
        if not b then
            report;
        if the largest two elements of O_{i+1} are ≥ x + |w_i| then
            enum(i + 1, not b);
        else
            if jump(i) ≠ k + 1 then
                enum(jump(i), not b);
        if b then
            report;
    Sol[i] ← r_i;
    if jump(i) = k + 1 then
        return;
    else
        enum(jump(i), b);      ▷ tail recursion
```

Let us first analyse the complexity of this algorithm. The preprocessing takes, clearly, $O(k)$ time. Analysing the delay is more complicated.

We first observe that after entering a recursive call we will `report` a valid embedding after $O(1)$ time. This is because $\ell_i < r_i$, so there will be at least one x considered in the for loop. If $b = \texttt{false}$ then we immediately `report` the current embedding. If $b = \texttt{true}$ then either we call $\texttt{enum}(i+1, \texttt{false})$ or $\texttt{enum}(\texttt{jump}(i), \texttt{false})$, where we again consider at least one x in the for loop and then immediately `report` the current embedding (as now $b = \texttt{true}$) there, or do not perform any recursive call and `report` the current embedding before finishing the iteration of the for loop.

Now we can analyse the delay between two calls to `report`. Consider a call to `report` inside the for loop. Then, if there is another iteration of the for loop, then either $b = \texttt{false}$ and we immediately `report` the current embedding in its beginning, or $b = \texttt{true}$ and we either perform a recursive call, in which we `report` a valid embedding after $O(1)$ time as argued above, or we `report` the current embedding before finishing the iteration. The remaining problematic case is that of `report` in the very last iteration of the for loop. If $\texttt{jump}(i) \neq k+1$ then we will immediately have another recursive call (implemented with tail recursion, but this is irrelevant at this point), so as argued above we will `report` a valid embedding after $O(1)$ time. Otherwise, we return from the recursive call. Due to tail recursion, we might actually immediately return to any of the predecessor recursive calls $\texttt{enum}(i', b')$, where $b' = \texttt{not}\ b$ (we note that this is why the very last call is $\texttt{enum}(\texttt{jump}(i), b)$ and not $\texttt{enum}(\texttt{jump}(i), \texttt{not}\ b)$) and continue in line 16. If $b' = \texttt{true}$ then we immediately `report` a valid embedding. Otherwise, if there is another iteration of the for loop we will `report` a valid embedding after $O(1)$ time, similarly if we call $\texttt{enum}(\texttt{jump}(i'), b')$. The remaining case is that $b' = \texttt{false}$ and we return from the recursive call to some predecessor recursive call $\texttt{enum}(i'', b'')$, where $b'' = \texttt{not}\ b'$, so $b'' = \texttt{true}$. But then repeating the argument one more time for i'' and b'' shows that we will `report` a valid embedding after $O(1)$ time.

As far as the space complexity of this approach is concerned, on top of the space needed for the provided data structures, we use $O(k)$ space to store the jump function, the `Sol` array, and (implicitly) the recursion stack.

Now we will explain why the algorithm correctly enumerates all valid embeddings. The main idea is that for a prefix $(j_1, \ldots, j_{i-1})$, we consider one by one all choices of j_i and, for each of them, every possible continuation: either we detect that only a right canonical embedding is possible for the remaining positions, or we detect the next position where we choose something else than the right canonical embedding and continue from there.

So, for each choice $j_i = x$, $x \neq r_i$, we report the embedding $(j_1, \ldots, j_{i-1}, x, r_{i+1}, \ldots, r_k)$ in the current iteration of the for loop. In the corresponding recursive call, we report other embeddings that extend the prefix $(j_1, \ldots, j_{i-1}, x)$. Once all these embeddings are enumerated (each exactly once), due to the fact that our computation considers the elements of each set O_t, with $t \in [k]$, such that r_t is always the last to be processed, we end up with the embedding $(j_1, \ldots, j_{i-1}, x, r_{i+1}, \ldots, r_k)$ again, but this is not reported in during the recursive calls (otherwise we would have repetitions in our enumeration, which is not desirable). Then we proceed to the next x, if there is any.

Now, for the case when $j_i = r_i$, if $\texttt{jump}(s) = k+1$, we are done: the current embedding is $(j_1, \ldots, j_{i-1}, r_i, \ldots, r_k)$ has been already reported earlier in the recursion, and there are no other embeddings with the prefix $(j_1, \ldots, j_{i-1}, r_i)$. Otherwise, we proceed as before, and enumerate all embeddings that extend $(j_1, \ldots, j_{i-1}, r_i)$, other than the one where all coordinates to the right of s are set to their value from the right-canonical embedding, i.e., $(j_1, \ldots, j_{i-1}, r_i, r_{i+1}, \ldots, r_k)$. Once we are done with all these embeddings, the current valid embedding is $(j_1, \ldots, j_{i-1}, r_i, r_{i+1}, \ldots, r_k)$ which is not reported (as explained above, it has been already reported). Instead, the last call of `enum` made during the enumeration of these embeddings returns (due to the elimination of

tail recursion) to a call of enum for some parameter $i' < i$, such that the current embedding is $(j_1, \ldots, j_{i'}, r_{i'+1}, \ldots, r_k)$ with $j_{i'} \neq r_{i'}$.

In conclusion, we report all embeddings extending $(j_1, \ldots, j_{s-1}, x)$, for all $x \in O_s$, and each is reported exactly once. □

For the rest of the section, we show that we can actually obtain the sorted sets $O_1, O_2, \ldots O_k$, which would allow us to apply the previous lemma. A simple (but slow) approach is as follows. Assume that we have computed the left-canonical embedding $(\ell_1, \ldots, \ell_k)$ and a right-canonical embedding $(r_1, \ldots, r_k)$. We compute each O_i naively by repeatedly invoking Lemma 2.1 on the relevant substring of the text $T[\ell_i : r_i + |w_i|)$ to extract the occurrences of w_i one by one in the sorted order. This takes $\sum_{i=0}^{k} O(r_i - \ell_i + |w_i|) = O(nk)$ time altogether for all sets. Combining this with Theorem 3.5 yields total $O(k(n + |S|))$ time to output all valid embeddings. We will next show how to obtain the sorted lists $O_1, O_2, \ldots O_k$ in $O(|S|)$ time with a more efficient approach.

Definition 3.7 (Epochs). Consider the sequence of intervals $[\ell_1 : r_1 + |w_1|), \ldots, [\ell_k : r_k + |w_k|)$ defined by the left- and right-canonical embeddings. We define another sequence of intervals $e_1, e_2, \ldots, e_v$, called *epochs*, as follows:

- $e_0 = \emptyset$. For $g \geq 1$, assume that $[\ell_1 : r_1 + |w_1|), \ldots, [\ell_{a-1} : r_{a-1} + |w_{a-1}|)$ are contained in epochs $e_1, \ldots, e_{g-1}$ and $[\ell_a : r_a + |w_a|)$ is not contained in any of these epochs. Then the next epoch e_g is defined as the interval $[\ell_a : r_b + |w_b|)$, such that $\ell_b < r_a + |w_a|$ and $\ell_{b+1} \geq r_a + |w_a|$ (or $b = k$). Thus, e_g contains $[\ell_a : r_a + |w_a|), \ldots, [\ell_b : r_b + |w_b|)$.
- For $g \geq 1$, let $\mathsf{left}_g = \min e_g$ and $\mathsf{right}_g = \max e_g$, and set $c_g = r_a + |w_a|$ to be the *center* of the epoch e_g. We call, respectively, $[\mathsf{left}_g : c_g - 1]$ the left and $[c_g + 1 : \mathsf{right}_g]$ the right part of e_g.

We note that $v \leq k$ and, for every $g \in [v]$, the union of the epochs $e_1, \ldots, e_g$ covers a prefix of the sequence of intervals $[\ell_1 : r_1 + |w_1|), \ldots, [\ell_k : r_k + |w_k|)$, and, in fact, that every position of T is covered by at most two epochs. The following lemma holds.

Lemma 3.8. $\sum_{g=1}^{v} (\mathsf{right}_g - \mathsf{left}_g) = O(n)$.

Proof. For two consecutive epochs e_g and e_{g+1}, we have by, Theorem 3.7, that $c_g < \mathsf{left}_{g+1}$ and $\mathsf{right}_g < c_{g+1}$. Hence, $[\mathsf{left}_g : c_g] \cap [\mathsf{left}_{g+1} : \mathsf{right}_{g+1}] = \emptyset$ and $[c_{g+1} : \mathsf{right}_{g+1}] \cap [\mathsf{left}_g : \mathsf{right}_g] = \emptyset$. Therefore, if $[\mathsf{left}_g : \mathsf{right}_g] \cap [\mathsf{left}_{g+1} : \mathsf{right}_{g+1}] \neq \emptyset$, then the common positions must lie in $[c_g + 1 : \mathsf{right}_g] \cap [\mathsf{left}_{g+1} : c_{g+1}]$. These parts of the epochs occur strictly one after

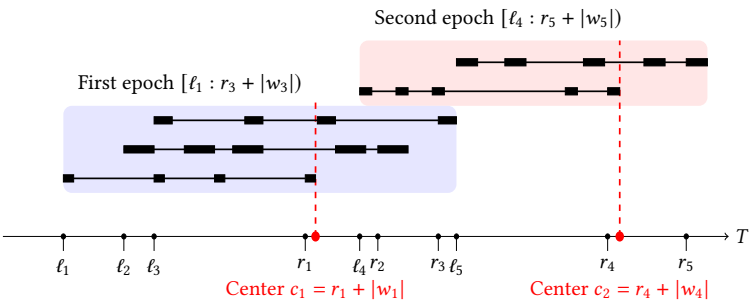


Fig. 2. Visualization of two epochs with labeled intervals $[\ell_i, r_i]$, occurrences (black rectangles) of the strings w_i, with $i \in [4]$, and epoch centers indicated in red.

another for g from 1 to v. Consequently, the sum of all intersections of epochs can be bounded as $\sum_{g=1}^{v-1} |[c_g + 1 : \mathsf{right}_g] \cap [\mathsf{left}_{g+1} : c_{g+1}]| \leq 2n$. Thus, also the sum of the lengths of all epochs is $\sum_{g=1}^{v} (\mathsf{right}_g - \mathsf{left}_g) = O(n)$. □

Definition 3.9. For an epoch e_g that includes $[\ell_i : r_i]$, the set of occurrences of w_i fully included in left part of an epoch is denoted by L_i and the set of the occurrences fully in the right part by R_i. The set of occurrences fully in $T[c_g - |w_i| + 1 : c_g + |w_i|)$ is denoted by M_i,

As every occurrence of w_i from an epoch e_g is in exactly one of the sets L_i, M_i, or R_i, we have $O_i = L_i \cup M_i \cup R_i$. Further, in Theorem 3.10, we state that L_i, R_i, and M_i can be computed efficiently: the part regarding L_i and R_i follows from applying Observation 2.1 for each left part (read left to right) and, respectively, right part (read right to left) of an epoch, while the part on M_i follows from Theorem 2.1.

Lemma 3.10. *The epochs can be computed in $O(k)$ time and can be preprocessed in $O(n)$ time for the following queries:*

(1) *given an index $i \in [k]$, enumerate the elements of L_i and R_i with constant delay.*
(2) *given an index $i \in [k]$, enumerate the elements of M_i with constant delay.*

Proof. We first note that a procedure computing the epochs in linear time follows directly from the definition of the epochs. If e_g contains $[\ell_a : r_a + |w_a|), \ldots, [\ell_b : r_b + |w_b|)$ (but not $[\ell_{a-1} : r_{a-1} + |w_{a-1}|)$ nor $[\ell_b : r_b + |w_b|)$), we can store, for each $i \in [a : b]$, e_g as the epoch which contains $[\ell_i : r_i + |w_i|)$.

For both queries, we need the following $O(n)$ time preprocessing. Recall that each interval $[\ell_i : r_i + |w_i|)$ is contained in some epoch. We will separately preprocess each epoch e_g and all i such that the corresponding interval is contained in e_g in $O(|e_g|)$ time. This will sum up to $O(n)$ over all the epochs. When preprocessing a single epoch, we need that the alphabet of each relevant w_i is polynomial in $|e_g|$, and not only polynomial in n (which might be much larger). This is done by renaming the symbols in each fragment $T[\mathsf{left}_g : \mathsf{right}_g)$ and each relevant w_i so that the alphabet is $[|e_g| + 1]$ (recall that each w_i occurs in the fragment). To this end, for each position in either $T[\mathsf{left}_g : \mathsf{right}_g)$ or any relevant w_i we add a pair consisting of the index g and the symbol at the corresponding position to a list. The pair is decorated with the information about its source. Then we use radix sort to sort the list of pairs in $O(n)$ time. This allows us to consistently rename the symbols in each fragment as required by scanning the sorted list and identifying symbols that are equal.

Let us now show to preprocess the epochs in $O()$ time for the following queries: given an index $i \in [k]$, enumerate the elements of L_i and R_i with constant delay. We consider a single epoch e_g and all i such that the corresponding interval is contained in e_g. We comment that the total length of the corresponding patterns w_i is at most $|e_g|$. After the renaming described above, we can assume that the alphabet of $T[\mathsf{left}_g : \mathsf{right}_g)$ and each such w_i is $[|e_g| + 1]$, allowing us to build a suffix tree/array for $T[\mathsf{left}_g : \mathsf{right}_g)$ in linear time. To obtain L_i, we define the **left part** $LP_g = T[\mathsf{left}_g : c_g - 1]$. Then, we apply Observation 2.1 to the left part and each relevant w_i. The preprocessing takes $O(|e_g|)$ time and allows us to enumerate all occurrences of w_i starting at position ℓ_i or later in the left part with $O(1)$ delay. To obtain R_i, we define the **right part** $RP_g = T[c_g + 1 : \mathsf{right}_g]$. We proceed analogously but we reverse RP_g and all the relevant w_i before applying Observation 2.1.

Now we move to preprocessing the epochs in $O(n)$ time for the following queries: given an index $i \in [k]$, enumerate the elements of M_i with constant delay. For each epoch e_g and every string w_i contained in e_g, define the **middle part** $MP_i = T[c_g - |w_i| + 1 : c_g + |w_i|)$. By Theorem 2.1, we can enumerate all occurrences of w_i in MP_i from left to right in $O(|MP_i|) = O(|w_i|)$ time. We explicitly

store these occurrences in a sorted list M_i. Since $\sum_{i=1}^{k} 2|w_i| = O(n)$, the whole preprocessing takes $O(n)$ time. Then, the elements of any M_i can be enumerated by directly accessing the corresponding sorted list. □

We define, for $i \in [k]$, the list O_i as the union of L_i, R_i, and M_i, a The previous lemmas allow us to obtain the following result.

Lemma 3.11. *We can compute all sorted $O_1, \ldots, O_k$ in $O(|S| + n)$ time and $O(n + \sum_{i=1}^{k} |O_i|)$ space.*

Proof. We apply Theorem 3.10 to preprocess T in $O(n)$ time. Then we compute and store (as the list O_i) the elements of L_i, R_i, and M_i, in $O(|L_i| + |R_i| + |M_i|)$ time, for all $i \in [k]$. We can sort the concatenation of all lists O_i in total $O(|S| + n)$ time using radix sort (while also keeping track of the original list O_i from which each element in this concatenation came from), due to the fact that all positions are integers bounded by $|T| = n$ and $k \leq n$. Then, using this sorted list of the concatenation of the lists O_i, and based on the fact that we know the original list of each element of this sorted list, we can obtain the sorted $O_1, \ldots, O_k$ in $O(\sum_{i=1}^{k} |O_i|) \in O(|S|)$.

The space required for this computation is $O(n)$ for T and the additional data structures constructed for it and $O(n + \sum_{i=1}^{k} |O_i|)$ for computing, sorting, and storing the respective lists. □

After using Theorem 3.11, we can apply Theorem 3.6 to the sorted lists $O_1, \ldots, O_k$ (as the two requirements of Theorem 3.6 trivially hold for the respective sorted lists) and directly obtain Theorem 1.1.

Theorem 1.1. *We can enumerate the set S of all valid embeddings of α in T with constant delay, after a preprocessing running in $O(n + |S|)$ time. The used space is $O(n + \min\{|S|, nk\})$.*

4 Amortized Constant Time Enumeration

Theorem 1.1 shows an enumeration algorithm with *average* constant delay, i.e., the total running time of the algorithm is $O(n + |S|)$, where S is the set of valid embeddings of α in T. We show next that we can actually achieve linear time preprocessing and amortized constant delay. To decrease the preprocessing time to $O(n)$ we move some of the work done in the preprocessing to the the actual enumeration phase, although this might affect the worst-case delay between two consecutive `reports`. More precisely, we need to avoid precomputing the lists O_i in the preprocessing phase of the algorithm, as their total length might significantly exceed $O(n)$.

Our algorithm maintains, in the array `Sol`, valid embeddings $(j_1, \ldots, j_k)$. We call such a valid embedding $(j_1, \ldots, j_k)$ *simple* when, for some $i \in [k]$, we have (i) $j_1 = \ell_1, \ldots, j_{i-1} = \ell_{i-1}$, (ii) $\ell_i < j_i \leq r_i$, and (iii) $j_{i+1} = r_{i+1}, \ldots, j_k = r_k$. By Lemma 3.4, the number of simple embedding is $\sum_{i=1}^{k}(|O_i| - 1)$.

The simple embeddings can be enumerated efficiently. In fact, it suffices to run the preprocessing from Theorem 3.10, and then we can enumerate, for i from 1 to k, such that $\ell_i \neq r_i$, the elements of $O_i = L_i \cup R_i \cup M_i$, other than ℓ_i. We start with the embedding $(\ell_1, \ldots, \ell_k)$. Then, for some i, the valid embedding maintained by our algorithm has the prefix $(\ell_1, \ldots, \ell_{i-1})$, the suffix $(r_{i+1}, \ldots, r_k)$, and we just keep changing the i^{th} component of the embedding, according to the current element of $O_i \setminus \{\ell_i\}$, and reporting it. When moving to $i + 1$, we just have to set the i^{th} component to ℓ_i, and start the enumeration for $O_{i+1} \setminus \{\ell_i\}$. The only subtlety is that, in order to keep the delay constant, we only need to consider in our enumeration those i for which $\ell_i \neq r_i$ (as there are no simple embeddings with prefix $(\ell_1, \ldots, \ell_{i-1})$ and suffix $(r_{i+1}, \ldots, r_k)$ if $\ell_i = r_i$). Moreover, we also compute, this way, the lists O_i. This is formalized in the following lemma.

Lemma 4.1. *After a preprocessing running in $O(n)$ time, all simple embeddings can be enumerated with constant delay. After all simple embeddings have been enumerated, we have the unsorted list O_i, for every $i \in [k]$.*

Proof. We use the preprocessing from Theorem 3.10 in $O(n)$ time. Then, we construct a sorted list I of all indices i such that $\ell_i < r_i$, i.e there are at least two valid occurrences of w_i. We also set $j_1 = r_1, \ldots, j_k = r_k$. This takes $O(k) = O(n)$ time, which is accounted for in the preprocessing. Then, we iterate over all indices $i \in I$, in increasing order. For each such i, we enumerate all valid occurrences of w_i in $L_i \cup R_i \cup M_i$ of w_i with constant delay, using Theorem 3.10. Let the currently considered occurrence of w_i be at position j_i; note that $\ell_i \leq j_i \leq r_i$. We first add j_i to the unsorted list O_i and, if $\ell_i < j_i$, then we `report` the current (simple) embedding $(j_1, \ldots, j_k)$. The delay between reporting two simple embeddings is constant because, for every i, we consider at least two positions j_i and, for every two consecutively considered positions j_i at least one results in reporting a simple embedding. After we have considered all valid occurrences of w_i, starting on positions between ℓ_i and r_i, we set $j_i = \ell_i$, such that after processing i the current embedding is of the form $(\ell_1, \ldots, \ell_i, r_{i+1}, \ldots r_k)$. We then proceed to the next i from the list I. □

We now proceed as follows. In the preprocessing phase we run the algorithm from Lemma 4.1, and then continue running it to enumerate up to n simple embeddings, however we don't actually report them. If in fact there are no more simple embeddings than we have already obtained, for every i, the unsorted list O_i. We can sort all of those lists together with radix sort in additional $O(n)$ time to obtain, for every i, the sorted list O_i. This concludes the preprocessing phase, and then we can proceed as in the previous section. The remaining case is that there are more than n simple embeddings. Then we again run the algorithm from Lemma 4.1, but now we do report the obtained simple embeddings. After having reported all the $\Omega(n)$ simple embeddings we have the unsorted list O_i, and we can sort all of them together with radix sort in additional $O(n + \sum_i |O_i|)$ time to obtain, for every i, the sorted list O_i. We observe that $O(n + \sum_i |O_i|)$ is only amortized constant time per every valid embedding reported so far. Having the sorted lists O_i, we now proceed as in the previous section, but we need to be careful as to not report the simple embeddings again. To this end, we can keep track of whether the currently considered valid embedding is simple with the following lemma.

Lemma 4.2. *After $O(k)$ preprocessing, we can keep track of whether the currently considered valid embedding is simple in $O(1)$ time after changing any j_i.*

Proof. We observe that the current valid embedding $(j_1, \ldots, j_k)$ is simple if and only if (i) if $j_i = r_i$ then $j_{i+1} = r_{i+1}$ holds for every $i \in [k-1]$, (ii) if $j_{i+1} = \ell_{i+1}$ then $j_i = \ell_i$ holds for every $i \in [k-1]$, and (ii) if $\ell_i < j_i < r_i$ then $j_{i-1} = \ell_{i-1}$ (or $i = 1$) and $j_{i+1} = r_{i+1}$ (or $i = k$). We maintain three counters: the first corresponds to the number of indices i such that $j_i = r_i$ but $j_{i+1} \neq r_{i+1}$, the second corresponds to the number of indices i such that $j_{i+1} = \ell_{i+1}$ but $j_i \neq \ell_i$, and the third corresponds to the number of indices i such that $\ell_i < j_i < r_i$ but it does not hold that $j_{i-1} = \ell_{i-1}$ (or $i = 1$) and $j_{i+1} = r_{i+1}$ (or $i = k$). The initial values of all counters can be computed in $O(k)$ time. Then, the conditions for every i depend only on j_{i-1}, j_i and j_{i+1}, so after changing any j_i we only need to re-evaluate the conditions for $i-1$, i and $i+1$, which allows us to maintain every counter in constant time. Then, the current valid embedding is simple if and only if all three counters are zero. □

By proceeding as described as above, and then running the enumeration algorithm from Theorem 1.1 together with the procedure from Theorem 4.2 (with its preprocessing run in the beginning)

to disregard the already reported simple embeddings we obtain Theorem 1.2. The space consumption of our algorithm is just as in the case of Theorem 1.1, as no additional data structures are required.

THEOREM 1.2. *We can enumerate the set S of all valid embeddings of α in T with amortized constant delay, after a preprocessing running in $O(n)$ time. The used space is $O(n + \min\{|S|, nk\})$.*

5 Enumeration with Worst-Case Constant Delay

We now describe how to further improve the solution described in the previous section so that every valid embedding is reported in $O(1)$ worst-case time. Compared to the previous section, now we cannot afford to "pause" reporting to sort the lists O_i, and further we are not necessarily able to afford to consider some valid embedding twice (as we might have multiple such embeddings in a row, which would lead to a non-constant delay). Instead, we avoid generating the sorted lists O_i altogether.

Recall that each list O_i consists of the list L_i (the occurrences of w_i fully contained in the left part of the epoch), M_i (the occurrences of w_i straddling both parts of the epoch), and R_i (the occurrences of w_i fully contained in the right part of the epoch). We need to be able to enumerate all elements of O_i greater or equal to x, starting from the largest, with constant delay, and retrieve the largest two elements in O_i. It is enough to implement both operations separately for L_i, M_i, and R_i in such complexities. Then, to enumerate the elements of O_i greater or equal to x we will first enumerate such elements of L_i, then M_i, and finally R_i. To retrieve the largest two elements in O_i we will retrieve such elements in L_i, M_i, and R_i (so at most 6 in total), and then return the two largest.

We observe that applying the second part of Theorem 3.10 actually results in constructing all sorted lists M_i in $O(n)$ time. This clearly allows us to enumerate the elements of M_i greater or equal to x, starting from the largest, with constant delay, and provides constant time access to the largest two elements in M_i. Further, we recall that the approach used to obtain the first part of Theorem 3.10 to enumerate the elements of L_i and R_i is based on applying Theorem 2.4. We need to look more closely at the underlying procedure before describing how to provide the required access to L_i and R_i.

Revisiting Theorem 2.4. The iterative procedure described in the proof can be seen as traversing an implicitly defined heap. Each node of the heap corresponds to a range of positions $[a' : b']$ such that $A[i] \geq x$, where $i = \mathtt{RMaxQ}_A(a', b')$, and stores the position i. The root corresponds to the whole $[a : b]$. Then, for a node v corresponding to $[a' : b']$, where $i = \mathtt{RMaxQ}_A(a', b')$, its left child corresponds to $[a' : i - 1]$ (if the largest element in the range is at least x, otherwise there is no left child), and symmetrically its right child corresponds to $[i + 1 : b']$. Then, the procedure traverses the nodes of the heap and enumerates their stored elements in such a way that at any point in time if the element stored at a node u has been already enumerated then so has the element stored at the parent of u (if any). We modify the procedure so that it not only enumerates the elements, but actually maintains the already discovered nodes of the heap. Thus, after the procedure terminates, the whole heap is available. Further, we modify the procedure so that as soon as a node u is discovered we also have access to both of its children (by issuing the corresponding two $\mathtt{RMaxQ}_A$ queries, which takes additional constant time).

Providing access to L_i. To enumerate the elements of L_i greater or equal to x with constant delay, it is enough to directly apply Theorem 2.4 as in the original proof. To retrieve the largest two elements in L_i we observe that the largest element corresponds to the root of the implicitly defined heap. Then, the second largest element corresponds to one of the children of the root of the heap. Thus, we can run the traversal procedure to obtain the first element, which is in fact the largest

element. Then, as explained above, as we have already discovered the root we can inspect its children, and choose the second largest element in constant time.

Providing access to R_i. This is more complicated, because we need to enumerate the elements of R_i greater or equal to x, but Theorem 2.4 needs to be applied on the reversed fragment of the text, and as such would only be able to enumerate the elements of R_i smaller or equal to x. To overcome this difficulty, we will observe that there is a certain monotonicity regarding the parameter x in subsequent calls to the enumeration procedure. In particular, we always first enumerate all elements of R_i, i.e., with x being the center of the corresponding epoch, which can be done with Theorem 2.4. Then, if we now need to enumerate elements of R_i greater or equal to x then earlier during the execution of the algorithm we have enumerated elements of R_i greater or equal to some $x' < x$. This allows us to reuse the results of the previous enumeration.

In a recursive call $\mathtt{enum}(i, b)$, we want to enumerate the occurrences of w_i starting at positions in $[j_{i-1}+|w_{i-1}| : r_i]$. For each such occurrence at position x, we then call $\mathtt{enum}(i, \mathtt{not}\ \ b)$, where $i' = i+1$ or $i' = \mathsf{jump}(i+1)$, to enumerate the occurrences of $w_{i'}$ starting at positions $[j_{i'-1} + |w_{i'-1}| : r_{i'}]$, and so on. In every such recursive call, we will first enumerate the occurrences in L_i (without any assumptions about their order). Then, we will enumerate the precomputed occurrences in M_i (in the sorted order). Finally, we will enumerate the occurrences in R_i by running Theorem 2.4. We conceptually arrange all occurrences of w_i in $M_i \cup R_i$ to form a heap H_i. More precisely, we first construct a path formed by listing the elements of M_i in the sorted order. Then, we attach as the only child of the last node of the path the root of the heap obtained by running Theorem 2.4 to enumerate all elements of R_i. We stress that initially only the path is constructed, and the remaining part of H_i is explored on demand. More precisely, if we have already reached a node of H_i then in constant time we can reach its children. Further, for every position x in the middle part MP_i we store a pointer to the node of H_i that corresponds to the successor of x in $M_i \cup R_i$, which is always a node of the path or the only child of its last node. The preprocessing takes only $O(n)$ time and space, as the total length of all middle parts is $O(n)$.

We maintain the following invariant inside a recursive call $\mathtt{enum}(i, b)$: if the currently considered occurrence of w_i is from $M_i \cup R_i$, then we already have the list of all relevant occurrences of $w_{i'}$ in $M_{r'} \cup R_{i'}$ (where i' might be in a different epoch than i). We think that every node of H_i stores a list of nodes of $H_{i'}$. However, these lists are generated during the recursion to make sure that if we are considering an occurrence corresponding to some node of H_i then its list is already prepared.

The base case is that we need to prepare the list for the root of H_i. We observe that the very first recursive call concerning some string w_i, that is not the last in its epoch, starts with an occurrence in L_i. Thus, when considering an occurrence x in L_i we can prepare for the first occurrence in $M_i \cup R_i$. More precisely, the subsequent recursive call first enumerates the relevant occurrences in $L_{i'}$ using Theorem 2.4, and then enumerates the relevant occurrences in $M_{i'} \cup R_{i'}$. The latter is done by traversing $H_{i'}$ starting either at the root or (if $x + |w_i|$ falls within $MP_{i'}$) at the node corresponding to the preprocessed successor of $x + |w_i| - 1$ in $M_{i'} \cup R_{i'}$, which can be accessed in constant time. During the traversal, we create a list of nodes that should be stored for the root of H_i. For a string w_i that is the last in its epoch, L_i might be empty, so we start with an occurrence in M_i. The subsequent recursive call enumerates all occurrences of $w_{i'}$ that is the first in its epoch (observe that this is possibly after skipping multiple epochs that consist of single patterns, that is, i' could be larger than $i+1$), and thus has an empty $M_{i'} \cup R_{i'}$, so there is nothing to store for the root of H_i.

In the general case, when considering an occurrence of w_i at position x corresponding to a node u of H_i, we prepare for considering the at most two subsequent occurrences of w_i at positions x' and x'', with $x < x', x''$, corresponding to the children of u in H_i. More precisely, when enumerating

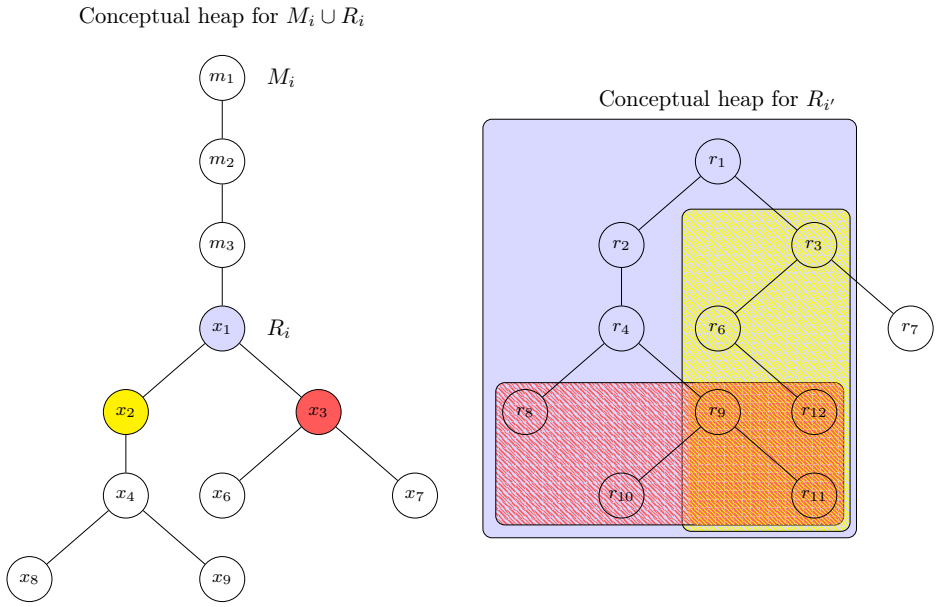


Fig. 3. The heap H_i associated to $M_i \cup R_i$ (left) and the part of the heap $H_{i'}$ associated to $M_{i'} \cup R_{i'}$ corresponding to $R_{i'}$ (right). The nodes x_t correspond to the occurrences of w_i in $M_i \cup R_i$, arranged in a heap. The nodes r_t correspond to the occurrences of $w_{i'}$ in $R_{i'}$. Every node x_t keeps a list of the occurrences of $w_{i'}$ relevant for the occurrence of w_i corresponding to x_t. In this figure, for a coloured node x_t from the left heap, the box of the same colour on the right side denotes the list of x_t. Note that any occurrence relevant for a child of x_t is also relevant for x_t.

the relevant occurrences of $w_{i'}$ in $M_{i'} \cup R_{i'}$ we check if they will be relevant for x' and x'', and if so add them to the corresponding list. This is illustrated in Figure 3.

Conclusion. By providing access to the elements of O_i as described above, we can run again our enumeration procedure, and obtain Theorem 1.3. As far as the space consumption is concerned, we note that, in this case, we will end up, as in the previous cases, storing, for every $i \in [k]$, the lists O_i completely and, additionally, for each element of each set $M_i \cup R_i$ a pointer to a list of elements (from the conceptual heap) of $M_{i'} \cup R_{i'}$ (for some $i' > i$). Overall, adding this to the space used by the data structures constructed in the preprocessing of all our algorithms, this leads to a total space consumption of $O(n + \sum_{i=1}^{k} |O_i| + n(\sum_{i=1}^{k} |O_i|)) = O(n \min\{|S|, nk\})$.

Theorem 1.3. *We can enumerate the set S of all valid embeddings of α in T with worst-case constant delay, after a preprocessing running in $O(n)$ time. The used space is $O(n \min\{|S|, nk\})$.*

6 An Index for the Enumeration of Regular Pattern Matches

The problem we approach in this section is the following:

INDEX FOR THE ENUMERATION OF REGULAR PATTERNS MATCHES ($\mathtt{EnumMatch}_{\mathtt{Reg}}$)
Input: Text $T[1:n]$.
Task: Preprocess T for the following queries: given a regular pattern $\alpha = (\prod_{i=0}^{k} w_i x_i) w_{k+1}$, report all valid substitutions (as valid embeddings) for α and T.

Preprocessing and data structures. As before, we construct for T its suffix tree ST and suffix array SA, in $O(n)$ time. Now, we build an additional data structure $\mathcal{W}$ which has the following properties:

- $\mathcal{W}$ is a balanced, binary, ordered tree. Each node v of this tree is associated with a range $[a:b]$ of the suffix array SA. The node v stores the associated range $[a:b]$ as well as an (increasingly) ordered doubly-linked list containing the elements of $SA[a:b]$.
- If v is a leaf of $\mathcal{W}$, then its range is $[i:i]$, for some $i \in [n]$, corresponding to position i of SA.
- If v is an internal node of $\mathcal{W}$, associated to the range $[a:b]$, with children v_1 and v_2, then $a < b$ and the range of v_1 is $[a:c]$ and the range of v_2 is $[c+1:b]$, for some $c \in [a:b-1]$. Moreover, for each element $r = S[i]$ occurring in the ordered list stored in v (i.e., $a \le i \le b$), we store pointers to the predecessor and the successor of r in the ordered lists of v_1 and v_2.

The tree $\mathcal{W}$ is strongly related to both segment trees [12] and wavelet trees [32]. Its construction can be performed in $O(n \log n)$ time and space, as explained in [9].

LEMMA 6.1 ([9]). *$\mathcal{W}$ can be constructed in $O(n \log n)$ time and space.*

PROOF. We briefly sketch the way $\mathcal{W}$ is constructed.
$\mathcal{W}$ is constructed in rounds, bottom-up. In each round, we have a list $L = (u_1, \ldots, u_\ell)$ of nodes, sorted increasingly by the left border of the range they represent. The ranges of every two nodes are disjoint, and their union is $[1:n]$. Initially, we have the list of leaves (ordered by the position of SA they represent). In each round, we define a new node v_i which has as children the nodes u_{2i-1} and u_{2i}, on positions $2i-1$ and $2i$ of the list L as children (for i from 1 to $\lfloor \ell/2 \rfloor$). The range of v_i is the union of the ranges of u_{2i-1} and u_{2i}, and the ordered list stored in v_i is obtained by merging the ordered lists of u_{2i-1} and u_{2i}. Note that, during the merging process, we can trivially compute for every element of the ordered list stored in v_i the predecessor and the successor of that element in the ordered lists of u_{2i-1} and u_{2i}. All the new nodes and, in the case when ℓ was odd, the last node of L are considered as the input list for the next round. Each round can be implemented in $O(n)$ time (proportional to the total size of the merged lists), and there are $O(\log n)$ rounds until we obtain a list with a single node. The conclusion follows. □

For a range $[a:b]$, with $a, b \in [n]$, we say that a node v of $\mathcal{W}$ is a canonical node for $[a:b]$ if v is associated to a range $[c:d]$ with $a \le c \le d \le b$ (i.e., the range $[c:d]$ is included in the range $[a:b]$) and the range of the parent node of v is not included in $[a:b]$. It can be shown that for each range $[a:b]$ there are at most $O(\log n)$ canonical nodes in $\mathcal{W}$ (see, e.g., [32] and the references therein). One can show, in fact, the next lemmas:

LEMMA 6.2. *The following queries can be answered in $O(\log n)$ time each using $\mathcal{W}$:*

- $\mathtt{cNodes}(a, b)$*: returns the list of the canonical nodes of the range $[a:b]$.*
- $\mathtt{cNodesPred}(a, b, r)$*: for a value $r \in [n]$, returns a list of pointers to the predecessors of r in the ordered lists of the canonical nodes of $[a:b]$; the returned list is decreasingly sorted w.r.t. the values pointed by its elements (i.e., the predecessors of r).*
- $\mathtt{cNodesSucc}(a, b, \ell)$*: for a value $\ell \in [n]$, returns a list of pointers to the successors of ℓ in the ordered lists of the canonical nodes of $[a:b]$; the returned list is increasingly sorted w.r.t. the values pointed by its elements (i.e., the successors of ℓ).*

Proof. The answer to $\texttt{cNodes}(a, b)$ is computed as follows: We traverse $\mathcal{W}$ recursively. Let v be the node currently considered in the traversal. If the range $[c : d]$ associated to v is included in $[a : b]$, then we output v and do not recurse in the subtree of v. If the intersection of the range $[c : d]$ and $[a : b]$ is empty, we return without producing an output and do not recurse in the subtree of v. Otherwise, we recursively consider the children of v. Given that the number of canonical nodes for $[a : b]$ is $O(\log n)$, the claim follows.

To answer $\texttt{cNodesPred}(a, b, r)$ we proceed similarly, with the single difference being that we now also have the current predecessor of r as a parameter p. We also compute a list L. We start our recursive traversal in the root of $\mathcal{W}$ with $p = r$. We traverse $\mathcal{W}$ recursively. Let v be the node currently considered in the traversal, and p be the predecessor-parameter. If the range $[c : d]$ associated to v is included in $[a : b]$, then we put p in the output list L and do not recurse in the subtree of v. If the intersection of the range $[c : d]$ and $[a : b]$ is empty, we return without adding anything to L and do not recurse in the subtree of v. Otherwise, we recursively consider the children v_1 and v_2 of v, with the predecessor-parameters being, respectively, p_1 the predecessor of p in the ordered list stored in v_1 (corresponding to the range of the suffix array covered by v_1), and p_2, the predecessor of p in the ordered list stored in v_2. Note that these predecessors are precomputed. Given that the number of canonical nodes for $[a : b]$ is $O(\log n)$, this process ends in $O(\log n)$ time and produces an unsorted list. Now, we can sort this list in $O(\log n)$ time, using an algorithm based on atomic heaps, as described in [21].

The queries $\texttt{cNodesSucc}(a, b, \ell)$ are answered analogously. □

Lemma 6.3. *After running* $\texttt{cNodesPred}(a, b, r)$*, we can:*

- *for any given $\ell \in [n]$, enumerate with constant delay the elements $S[i]$ of $S[a : b]$ such that $\ell \leq S[i] \leq r$, starting with the largest one.*
- *return the largest and second largest elements of $S[a : b]$, which are not larger than r.*

Lemma 6.4 ([10]). *Provided the suffix tree ST (respectively, the suffix array SA) of a text T, for a given string w, of length $m \leq n$, we can find the range $SA[a_w : b_w]$ containing exactly the positions j such that $T[j : j + |w| - 1] = w$ in $O(m + \log n)$ time.*

Processing a query. Assume that we are now given a query $\alpha = x_0(\prod_{i=0}^{k} w_i x_i)$, and we want to enumerate all the embeddings of α in T. We show how we can produce all the data structures needed to perform the enumeration algorithms from the previous sections.

We proceed as follows, for each $i \in [k]$. Firstly, we use Theorem 6.4 to compute for each w_i the range $SA[a_i : b_i]$ containing exactly the positions j such that $T[j : j + |w_i| - 1] = w_i$. This takes, in total, $O(|\alpha| + k \log n)$ time.

Now, we can find the right-canonical (respectively, left-canonical) embedding $(r_1, \ldots, r_k)$ (respectively, $(\ell_1, \ldots, \ell_k)$) for α and T. We define r_k to be the largest element of $S[a_k : b_k]$ (can be obtained in $O(\log n)$, using Theorem 6.3). Then, for i from $k - 1$ to 1, we execute the query $\texttt{cNodesPred}(a_i, b_i, r_{i+1})$ as in Theorem 6.2 and obtain the list $\mathcal{I}_i$ of pointers to the predecessors of r_{i+1} in the ordered lists of the canonical nodes of $[a_i : b_i]$; the returned list is decreasingly sorted w.r.t. the values pointed by its elements; we define r_k as being the element pointed by the first element in the list returned by $\texttt{cNodesPred}(a_i, b_i, r_{i+1})$. In total, computing $r_1, \ldots, r_k$ takes $O(k \log n)$ time. The left-canonical embedding $(\ell_1, \ldots, \ell_k)$ is computed by a symmetrical procedure.

Further, we can also compute the largest two elements in $S[a_i : b_i]$ smaller or equal to r_i, for all $i \in [k]$, by Theorem 6.3. This takes, again, $O(k \log n)$ time.

At this point, recall Theorem 6.3 and note that the list $\mathcal{I}_i$ offers us a way to enumerate one by one the elements of the set O_i (defined as in Section 3, for α and T), which are greater or equal to a given number ℓ, with constant delay. We consider the current element of the list $\mathcal{I}_i$ (initially,

the current element is the first pointer of this list). This points to an element t in the ordered list of some canonical node v. If $t \geq \ell$, then we return, one by one, with constant delay, by simply traversing the ordered list of v, all elements smaller or equal to t and greater or equal to ℓ. Once there are no more such elements in the list of v, we advance in the list $\mathcal{I}_i$ and repeat this procedure for the next element of that list (unless it points to an element smaller than ℓ, when we simply stop).

Enumeration. We can now directly use Theorem 3.6 to enumerate the valid embeddings of α in T, as its requirements hold. First, we are able to enumerate, one by one, in $O(1)$ time per element, the elements of O_i which are greater or equal to the $(i-1)^{\text{th}}$ component of the embedding, starting with r_i. This is possible, as described above, once the list $\mathcal{I}_i$ is computed. Second, we are able to retrieve the largest two elements of the set O_i. As noted in Section 4, in Theorem 3.6, this leads to an enumeration algorithm with constant delay for all the valid embeddings for α and T. In conclusion, we obtain Theorem 1.4. The space needed is $O(n \log n)$ for the initial preprocessing, and, then, $O(|\alpha| + nk)$ for the preprocessing of α and the enumeration of its valid embeddings.

Theorem 1.4. *We can preprocess the text T in $O(n \log n)$ time, so that when given a pattern α with k variables, after an additional processing running in $O(|\alpha| + k \log n)$ time, the set S of all valid embeddings of α in T can be enumerated with constant delay.*

7 Conclusion

We have considered the matching problem for regular patterns. This problem naturally lies in the intersection of information extraction and stringology. We have designed an enumeration algorithm that preprocesses the document and the query pattern in optimal linear time, and then enumerates the matches with worst-case constant delay. While our techniques are perhaps specific to the considered class of regular expressions, we hope that our work will trigger further interest in transferring methods from stringology and data structures to information extraction.

Acknowledgments

Paweł Gawrychowski is partially supported by the Polish National Science Centre grant number 2023/51/B/ST6/01505.
Florin Manea, Paul Sarnighausen-Cahn and Stefan Siemer are supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) through the Heisenberg project 466789228 and the research project 562495345.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.
[2] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2021. Constant-Delay Enumeration for Nondeterministic Document Spanners. *ACM Trans. Database Syst.* 46, 1 (2021), 2:1–2:30. doi:10.1145/3436487
[3] Antoine Amarilli and Mikaël Monet. 2023. Enumerating Regular Languages with Bounded Delay. In *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023, Hamburg, Germany, March 7-9, 2023 (LIPIcs, Vol. 254)*, Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 8:1–8:18. doi:10.4230/LIPICS.STACS.2023.8
[4] Dana Angluin. 1980. Finding Patterns Common to a Set of Strings. *J. Comput. Syst. Sci.* 21, 1 (1980), 46–62. doi:10.1016/0022-0000(80)90041-0
[5] Arturs Backurs and Piotr Indyk. 2016. Which Regular Expression Patterns Are Hard to Match?. In *FOCS*. IEEE Computer Society, 457–466.
[6] Guillaume Bagan. 2006. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary,*

September 25-29, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4207), Zoltán Ésik (Ed.). Springer, 167–181. doi:10.1007/11874683_11

[7] Michael A. Bender and Martin Farach-Colton. 2000. The LCA Problem Revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1776)*, Gaston H. Gonnet, Daniel Panario, and Alfredo Viola (Eds.). Springer, 88–94. doi:10.1007/10719839_9

[8] Karl Bringmann, Allan Grønlund, Marvin Künnemann, and Kasper Green Larsen. 2024. The NFA Acceptance Hypothesis: Non-Combinatorial and Dynamic Lower Bounds. *TheoretiCS* 3 (2024).

[9] Bernard Chazelle. 1988. A Functional Approach to Data Structures and Its Use in Multidimensional Searching. *SIAM J. Comput.* 17, 3 (1988), 427–462. doi:10.1137/0217026

[10] Richard Cole and Moshe Lewenstein. 2003. Multidimensional matching and fast search in suffix trees. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*. ACM/SIAM, 851–852. http://dl.acm.org/citation.cfm?id=644108.644251

[11] Bruno Courcelle. 2009. Linear delay enumeration and monadic second-order logic. *Discret. Appl. Math.* 157, 12 (2009), 2675–2700. doi:10.1016/J.DAM.2008.08.021

[12] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. 2008. *Computational geometry: algorithms and applications, 3rd Edition*. Springer. doi:10.1007/978-3-540-77974-2

[13] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12:1–12:51. doi:10.1145/2699442

[14] M. Farach. 1997. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 137–143. doi:10.1109/SFCS.1997.646102

[15] Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. 2018. Revisiting Shinohara's algorithm for computing descriptive patterns. *Theor. Comput. Sci.* 733 (2018), 44–54. doi:10.1016/J.TCS.2018.04.035

[16] Henning Fernau, Florin Manea, Robert Mercas, and Markus L. Schmid. 2020. Pattern Matching with Variables: Efficient Algorithms and Complexity Results. *ACM Trans. Comput. Theory* 12, 1 (2020), 6:1–6:37. doi:10.1145/3369935

[17] Henning Fernau and Markus L. Schmid. 2015. Pattern matching with variables: A multivariate complexity analysis. *Inf. Comput.* 242 (2015), 287–305. doi:10.1016/J.IC.2015.03.006

[18] Henning Fernau, Markus L. Schmid, and Yngve Villanger. 2016. On the Parameterised Complexity of String Morphism Problems. *Theory Comput. Syst.* 59, 1 (2016), 24–51. doi:10.1007/S00224-015-9635-3

[19] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2018. Constant Delay Algorithms for Regular Document Spanners. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*. ACM, 165–177. doi:10.1145/3196959.3196987

[20] Michael L. Fredman and Dan E. Willard. 1990. BLASTING through the Information Theoretic Barrier with FUSION-TREES. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, Harriet Ortiz (Ed.). ACM, 1–7. doi:10.1145/100216.100217

[21] Michael L. Fredman and Dan E. Willard. 1994. Trans-Dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths. *J. Comput. Syst. Sci.* 48, 3 (1994), 533–551. doi:10.1016/S0022-0000(05)80064-9

[22] Dominik D. Freydenberger, Benny Kimelfeld, and Liat Peterfreund. 2018. Joining Extractions of Regular Expressions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*. ACM, 137–149. doi:10.1145/3196959.3196967

[23] Dominik D. Freydenberger and Markus L. Schmid. 2017. Deterministic Regular Expressions with Back-References. In *34th Symposium on Theoretical Aspects of Computer Science, STACS 2017, March 8-11, 2017, Hannover, Germany (LIPIcs, Vol. 66)*, Heribert Vollmer and Brigitte Vallée (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 33:1–33:14. doi:10.4230/LIPICS.STACS.2017.33

[24] André Frochaux and Sarah Kleest-Meißner. 2023. Puzzling over Subsequence-Query Extensions: Disjunction and Generalised Gaps. In *Proceedings of the 15th Alberto Mendelzon International Workshop on Foundations of Data Management (AMW 2023), Santiago de Chile, Chile, May 22-26, 2023 (CEUR Workshop Proceedings, Vol. 3409)*. CEUR-WS.org. https://ceur-ws.org/Vol-3409/paper3.pdf

[25] Leslie Ann Goldberg. 1991. *Efficient algorithms for listing combinatorial structures*. Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/10917

[26] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. 1988. On Generating All Maximal Independent Sets. *Inf. Process. Lett.* 27, 3 (1988), 119–123. doi:10.1016/0020-0190(88)90065-8

[27] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of monadic second-order queries on trees. *ACM Trans. Comput. Log.* 14, 4 (2013), 25:1–25:12. doi:10.1145/2528928

[28] Donald Ervin Knuth. 2014. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley.

[29] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. 1977. Fast Pattern Matching in Strings. *SIAM J. Comput.* 6, 2 (1977), 323–350. doi:10.1137/0206024

[30] Florin Manea and Markus L. Schmid. 2019. Matching Patterns with Variables. In *Combinatorics on Words - 12th International Conference, WORDS 2019, Loughborough, UK, September 9-13, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11682)*. Springer, 1–27. doi:10.1007/978-3-030-28796-2_1

[31] Torsten Mütze. 2022. Combinatorial Gray codes-an updated survey. *CoRR* abs/2202.01280 (2022). arXiv:2202.01280 https://arxiv.org/abs/2202.01280

[32] Gonzalo Navarro. 2012. Wavelet Trees for All. In *Combinatorial Pattern Matching - 23rd Annual Symposium, CPM 2012, Helsinki, Finland, July 3-5, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7354)*, Juha Kärkkäinen and Jens Stoye (Eds.). Springer, 2–26. doi:10.1007/978-3-642-31265-6_2

[33] Matthias Niewerth and Luc Segoufin. 2018. Enumeration of MSO Queries on Strings with Constant Delay and Logarithmic Updates. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, 179–191. doi:10.1145/3196959.3196961

[34] Takeshi Shinohara. 1982. Polynomial Time Inference of Extended Regular Pattern Languages. In *RIMS Symposium on Software Science and Engineering, Kyoto, Japan, 1982, Proceedings (Lecture Notes in Computer Science, Vol. 147)*. Springer, 115–127. doi:10.1007/3-540-11980-9_19

Received December 2025; accepted February 2026