*Fundamenta Informaticae 101 (2010) 173–186*
*DOI 10.3233/FI-2010-283*
*IOS Press*



# Finding Patterns In Given Intervals*†

**Maxime Crochemore**‡
*Algorithm Design Group*
*King's College London*
*Strand, London WC2R 2LS, U.K.*
*Maxime.Crochemore@kcl.ac.uk*

**Marcin Kubica**
*Institute of Informatics, Warsaw University*
*Banacha 2, 02-097 Warszawa, Poland*
*kubica@mimuw.edu.pl*

**Tomasz Waleń**
*Institute of Informatics, Warsaw University*
*Banacha 2, 02-097 Warszawa, Poland*
*walen@mimuw.edu.pl*

**Costas S. Iliopoulos**
*Algorithm Design Group*
*King's College London*
*Strand, London WC2R 2LS, U.K.*
*csi@dcs.kcl.ac.uk*

**M. Sohel Rahman**§
*AℓEDA Group, Department of CSE*
*BUET, Dhaka-1000, Bangladesh*
*msrahman@cse.buet.ac.bd*

**Abstract.** In this paper, we study the pattern matching problem in given intervals. Depending on whether the intervals are given a priori for pre-processing, or during the query along with the pattern or, even in both the cases, we develop efficient solutions for different variants of this problem. In particular, we present efficient indexing schemes for each of the above variants of the problem.

**Keywords:** algorithms, data structures, pattern matching, strings.

*Preliminary version appeared in [15].
†Part of this research work was carried out when the authors were visiting McMaster University during StringMasters 2007.
‡Also works: Université Paris-Est, France
§Address for correspondence: AℓEDA Group, Department of CSE, BUET, Dhaka-1000, Bangladesh

# 1. Introduction

The classical pattern matching problem is to find all the occurrences of a given pattern $\mathcal{P} = \mathcal{P}[1..m]$ of length $m$ in a text $\mathcal{T} = \mathcal{T}[1..n]$ of length $n$, both being sequences of characters drawn from a finite character set $\Sigma$. In the indexing version of the problem, the text $\mathcal{T}$ is preprocessed beforehand to construct an index data structure so as to efficiently handle a batch of subsequent queries. The pattern matching problem is interesting as a fundamental computer science problem and is a basic need of many applications, such as text retrieval, music information retrieval, computational biology, data mining, network security, to name a few. Several of these applications require, however, more sophisticated forms of searching. As a result, most recent works in pattern matching have considered '*inexact matching*'. Many types of differences have been defined and studied in the literature, namely, errors (Hamming distance [12], LCS [24,25,29,38], edit distance [12,33]), wild cards or don't cares [12,13,19,28,36,37,39], degenerate strings [20, 26, 40], rotations [3, 7], scaling [4, 5], permutations [11] among others.

Contemporary research on pattern matching has taken many other different and interesting directions ranging from position restricted pattern matching [16, 34] to pattern matching with address error [2] and property matching [6, 27]. In this paper, we are interested in pattern matching in given intervals and focus on building index data structures to handle this problem efficiently. This particular variant of the classic pattern matching problem is motivated by practical applications depending on different settings. For example, in many text search situations one may want to search only a part of the text collection, e.g. restricting the search to a subset of dynamically chosen documents in a document database, restricting the search to only parts of a long DNA sequence, and so on. In these cases, we need to find a pattern in a text interval where the interval is given with the query pattern. On the other hand, in a different setting, the interval or a set thereof may be supplied with the text for preprocessing. For example, in molecular biology, it has long been a practice to consider special genome areas by their structure. Examples are repetitive genomic structures [31] such as tandem repeats, LINEs (Long Interspersed Nuclear Sequences) and SINEs (Short Interspersed Nuclear Sequences) [32]. In this setting, the task may be to find occurrences of a given pattern in a genome, provided it appears in a SINE, or LINE. Finally a combination of these two settings is also of particular interest: find occurrences of a given pattern in a particular part of a genome, provided it appears in a SINE, or LINE.

Note that, if we consider the 'normal' (non-indexing) pattern matching scenario, the pattern matching in given intervals becomes straightforward to solve: we solve the classic pattern matching problem and then output only those that belong to the given intervals. However, the indexing version of the problem seems to be much more complex. Depending on whether the intervals are given a priori for pre-processing (Problem PMGI), or during the query along with the pattern (Problem PMQI) or, even in both the cases (Problem PMI), we develop solutions for different variants of this problem. A slightly different variant of Problem PMGI was studied in [6, 27], whereas Problem PMQI was introduced and handled in [34] and a variant of it was studied in [16] (See Section 2 for details).

The contribution of this paper is as follows. We first handle the more general problem PMI (Section 3) and present an efficient data structure requiring $O(n \log^3 n)$ time and $O(n \log^2 n)$ space with a query time of $O(m + \log \log n + K)$ per query, where $K$ is the size of the output. We then solve Problem PMGI (Section 4) to achieve optimal query time ($O(m + K)$ query time on a data structure with $O(n \log \sigma)$ time and $O(n \log n)$-bit space complexity). Finally, we improve the query time of [34] for Problem PMQI (Section 5) to optimal, i.e., $O(m + K)$ per query. The corresponding data structure, however, requires $O(n^{1.5})$ time and space due to the preprocessing of an intermediate problem, which remains as

the bottleneck in the overall running time. We also show how the query time of Problem PMI can be made optimal, albeit, at the cost of an increased preprocessing time.

| Problems | Preprocessing | Query Time | Reference |
|---|---|---|---|
| PMI | $O(n \log^3 n)$ time<br>$O(n \log^2 n)$ space | $O(m + \log\log n + K)$ | * |
| PMI | $O(n^{1.5})$ time<br>$O(n^{1.5})$ space | Optimal $O(m + K)$ | * |
| PMGI | $O(n \log \sigma)$ time<br>$O(n \log n)$ space | Optimal $O(m + K)$ | * |
| PMQI | $O(n \log^{1+\epsilon} n)$ time<br>$O(n \log^{1+\epsilon} n)$ space | $O(m + \log\log n + K)$ | [34] |
| PMQI | $O(n^{1.5})$ time<br>$O(n^{1.5})$ space | Optimal $O(m + K)$ | * |

Figure 1. Results of the Problems handled in this paper. An '*' in the Reference column indicates our contribution

The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts. The contributions of this paper are presented in Section 3 to 6. Finally, we conclude this paper briefly in Section 7.

## 2. Preliminaries

A *text*, also called a *string*, is a sequence of zero or more symbols from an alphabet $\Sigma$. A text $\mathcal{T}$ of length $n$ is denoted by $\mathcal{T}[1..n] = \mathcal{T}_1\mathcal{T}_2 \ldots \mathcal{T}_n$, where $\mathcal{T}_i \in \Sigma$ for $1 \leq i \leq n$. The *length* of $\mathcal{T}$ is denoted by $|\mathcal{T}| = n$. A string $w$ is a *factor* or *substring* of $\mathcal{T}$ if $\mathcal{T} = uwv$ for $u, v \in \Sigma^*$; in this case, the string $w$ occurs at position $|u| + 1$ in $\mathcal{T}$. The factor $w$ is denoted by $\mathcal{T}[|u| + 1..|u| + |w|]$. A *prefix* (*suffix*) of $\mathcal{T}$ is a factor $\mathcal{T}[x..y]$ such that $x = 1$ ($y = n$), $1 \leq y \leq n$ ($1 \leq x \leq n$).

In traditional pattern matching problem, we want to find the occurrences of a given pattern $\mathcal{P}[1..m]$ in a text $\mathcal{T}[1..n]$. The pattern $\mathcal{P}$ is said to occur at position $i \in [1..n]$ of $\mathcal{T}$ if and only if $\mathcal{P} = \mathcal{T}[i..i+m-1]$. We use $Occ_{\mathcal{T}}^{\mathcal{P}}$ to denote the set of occurrences of $\mathcal{P}$ in $\mathcal{T}$.

The problems we handle in this paper can be defined formally as follows.

**Problem "PMQI" (Pattern Matching in a Query Interval).** Suppose we are given a text $\mathcal{T}$ of length $n$. Preprocess $\mathcal{T}$ to answer following form of queries.

**Query:** Given a pattern $\mathcal{P}$ and a query interval $[\ell..r]$, with $1 \leq \ell \leq r \leq n$, construct the set

$$Occ_{\mathcal{T}[\ell..r]}^{\mathcal{P}} = \{i \mid i \in Occ_{\mathcal{T}}^{\mathcal{P}} \text{ and } i \in [\ell..r]\}.$$

**Problem "PMGI" (Pattern Matching in Given Intervals).** Suppose we are given a text $\mathcal{T}$ of length $n$ and a set of disjoint intervals $\pi = \{[s_1..f_1], [s_2..f_2], \ldots, [s_{|\pi|}..f_{|\pi|}]\}$ such that $s_i, f_i \in [1..n]$ and $s_i \leq f_i$,

for all $1 \leq i \leq |\pi|$. Preprocess $\mathcal{T}$ to answer following form of queries.
**Query:** Given a pattern $\mathcal{P}$ construct the set

$$Occ^{\mathcal{P}}_{\mathcal{T},\pi} = \{i \mid i \in Occ^{\mathcal{P}}_{\mathcal{T}} \text{ and } i \in \varpi \; for \; some \; \varpi \in \pi\}.$$

**Problem "PMI" (Generalized Pattern Matching with Intervals).** Suppose we are given a text $\mathcal{T}$ of length $n$ and a set of intervals $\pi = \{[s_1..f_1], [s_2..f_2], \ldots, [s_{|\pi|}..f_{|\pi|}]\}$ such that $s_i, f_i \in [1..n]$ and $s_i \leq f_i$, for all $1 \leq i \leq |\pi|$. Preprocess $\mathcal{T}$ to answer following form of queries.
**Query:** Given a pattern $\mathcal{P}$ and a query interval $[\ell..r]$ such that $\ell, r \in [1..n]$ and $\ell \leq r$, construct the set

$$Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi} = \{i \mid i \in Occ^{\mathcal{P}}_{\mathcal{T}} \text{ and } i \in [\ell, r] \bigcap \varpi \; for \; some \; \varpi \in \pi\}.$$

Problem PMQI was studied extensively in [34]. The authors in [34] presented a number of algorithms depending on different trade-offs between the time and space complexities. The best query time they achieved was $O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ against a data structure exhibiting $O(n \log^{1+\epsilon} n)$ space and time complexity, where $0 \leq \epsilon \leq 1$. A restricted version of Problem PMQI, where the query interval can only be either a prefix or a suffix, was handled very recently in [16]. For this problem optimal query time was achieved in [16] against a simpler data structure. In particular, the space and time complexity of the data structure presented in [16] is dominated by that of a suffix tree (or array). However, this data structure can not be used to handle the general problem and hence the query time for the general version of Problem PMQI remains non-optimal.

A slightly different variant of Problem PMGI was studied in [6]. In particular, the difference lies in the fact that the problem handled in [6], looks for the occurrences of the given pattern completely confined in the given set of intervals, $\pi$, whereas in Problem PMGI, only the occurrences that start in $\pi$ are of interest. In [6] a data structure requiring $O(n \log \Sigma + n \log\log n)$ time was presented to support optimal $O(m + K)$ time query, where $K$ is the output size. Later in [27] a faster index was devised without loosing the optimal query time. However, neither the data structure presented in [6] nor the one in [27] can be used to solve Problem PMGI.

Problem PMI, as is evident from the definition, is the combination of Problem PMQI and PMGI and hence is a more general problem in this regard. To the best of our knowledge, this is the first attempt to provide a solution for Problem PMI.

In traditional indexing problem one of the basic data structures used is the suffix tree data structure. In our indexing problem, we make use of this suffix tree data structure. A complete description of a suffix tree is beyond the scope of this paper, and can be found in [35, 43] or in any textbook on stringology (e.g. [17, 22]). However, for the sake of completeness, we define the suffix tree data structure as follows. Given a string $\mathcal{T}$ of length $n$ over an alphabet $\Sigma$, the suffix tree $ST_{\mathcal{T}}$ of $\mathcal{T}$ is the compacted trie of all suffixes of $\mathcal{T}\$$, where $\$ \notin \Sigma$. Each leaf in $ST_{\mathcal{T}}$ represents a suffix $\mathcal{T}[i..n]$ of $\mathcal{T}$ and is labeled with the index $i$. We refer to the list (in left-to-right order) of indices of the leaves of the subtree rooted at node $v$ as the leaf-list of $v$; it is denoted by $LL(v)$. Each edge in $ST_{\mathcal{T}}$ is labeled with a nonempty substring of $\mathcal{T}$ such that the path from the root to the leaf labeled with index $i$ spells the suffix $\mathcal{T}[i..n]$. For any node $v$, we let $\ell_v$ denote the string obtained by concatenating the substrings labeling the edges on the path from the root to $v$ in the order they appear. Given the suffix tree $ST_{\mathcal{T}}$ of a text $\mathcal{T}$ we define the "locus" $\mu^{\mathcal{P}}$ of a pattern $\mathcal{P}$ as the node in $ST_{\mathcal{T}}$ such that $\ell_{\mu^{\mathcal{P}}}$ has the prefix $\mathcal{P}$ and $|\ell_{\mu^{\mathcal{P}}}|$ is the smallest of all such nodes. Note that the locus of $\mathcal{P}$ does not exist, if $\mathcal{P}$ is not a substring of $\mathcal{T}$. Therefore, given $\mathcal{P}$, finding $\mu^{\mathcal{P}}$ suffices to determine whether $\mathcal{P}$ occurs in $\mathcal{T}$.

Several algorithms exist that can construct the suffix tree $ST_{\mathcal{T}}$ requiring linear space in $O(n \log \sigma)$ time, where $\sigma = \min(n, |\Sigma|)$ [35, 43]. Notably, this means that, for bounded alphabet the construction time is $O(n)$. Furthermore, in [18], Farach presented an $O(n)$ time algorithm for suffix tree construction, which works even for larger alphabets. In particular, Farach's algorithm works in linear time even if we have $\Sigma = \{1, \ldots n^c\}$, where $c$ is a constant. Given a suffix tree $ST_X$ and a pattern $Y = Y[1..m]$, one can find its locus and hence the fact whether $X$ has an occurrence of $Y$ in $O(m \log |\Sigma|)$ time. In addition to that, all such occurrences can be reported in constant time per occurrence. Finally, we note that the optimal query time of $O(m+K)$ ($K$ is the output size) can be achieved with a suffix tree with $O(n \log n)$ bits of space [8].

## 3. Problem PMI

In this section, we handle Problem PMI. Since this is a more general problem than both PMQI and PMGI, any solution to PMI would also be a solution to both PMQI and PMGI. Our basic idea is to build an index data structure that would solve the problem in two steps. First, it will (implicitly) give us the set $Occ_{\mathcal{T}}^{\mathcal{P}}$. Then, the index would 'select' some of the occurrences to provide us with our desired set $Occ_{\mathcal{T}[\ell..r],\pi}^{\mathcal{P}}$.

We describe now the idea we employ. We first construct a suffix tree $ST_{\mathcal{T}}$. According to the definition of suffix tree, each leaf in $ST_{\mathcal{T}}$ is labeled by the starting location of its suffix. We do some preprocessing on $ST_{\mathcal{T}}$ as follows. We maintain a linked list of all leaves in a left-to-right order. In other words, we realize the list $LL(\mathcal{R})$ in the form of a linked list, where $\mathcal{R}$ is the root of the suffix tree. In addition to that, we set pointers $v.left$ and $v.right$ from each tree node $v$ to its leftmost leaf $v_\ell$ and rightmost leaf $v_r$ (considering the subtree rooted at $v$) in the linked list. It is easy to realize that, with these pointers at our disposal, we can indicate the set of occurrences of a pattern $\mathcal{P}$ by the two leaves $\mu_\ell^{\mathcal{P}}$ and $\mu_r^{\mathcal{P}}$ because all the leaves between and including $\mu_\ell^{\mathcal{P}}$ and $\mu_r^{\mathcal{P}}$ in $LL(\mathcal{R})$ correspond to the occurrences of $\mathcal{P}$ in $\mathcal{T}$. In what follows, we define the terms $\ell_T$ and $r_T$ such that $LL(\mathcal{R})[\ell_T] = \mu_\ell^{\mathcal{P}}$ and $LL(\mathcal{R})[r_T] = \mu_r^{\mathcal{P}}$, where $\mathcal{R}$ is the root of $ST_{\mathcal{T}}$.

Now recall that our data structure has to be able to somehow "select" and report only those occurrences that lie in the intersection of the query interval and one of the given intervals. To solve this we use the following two interesting problems.

**Problem "CRSI" (Colored Range Set Intersection Problem)**. Suppose $V[1..n]$ and $W[1..n]$ are two permutations of $[1..n]$. Also, assume that each $i \in [1..n]$ is assigned a not necessarily distinct color. Preprocess $V$ and $W$ to answer the following form of queries.

**Query:** Find the set of distinct colors of the intersection of the elements of $V[i..j]$ and $W[k..\ell]$, $1 \leq i \leq j \leq n, 1 \leq k \leq \ell \leq n$.

**Problem "CRSG" (Colored Range Search Problem on Grid)**. Suppose $A[1..n]$ is a set of $n$ colored points on the grid $[0..U] \times [0..U]$. Preprocess $A$ to answer the following form of queries.

**Query:** Given a query rectangle $q \equiv (a, b) \times (c, d)$, find the set of distinct colors of points contained in $q$.

Our idea is to first reduce Problem PMI to Problem CRSI and then to the much more studied Problem CRSG. Recall that, we have an array $LL(\mathcal{R})$ and an interval $[\ell_T..r_T]$, which implicitly gives us the set $Occ_{\mathcal{T}}^{\mathcal{P}}$. Recall also that, our goal is to select those $i \in Occ_{\mathcal{T}}^{\mathcal{P}}$ such that $i$ occurs in one of the intervals of $\pi$ and also in $[\ell..r]$. We first construct an array $\mathcal{M} = \mathcal{M}[1..n]$, such that for all $k \in [1..n], \mathcal{M}[k] = k$. Also, we construct a 'color array' $\mathcal{C}$ to assign colors to each $k \in [1..n]$ as follows. For each $k \in [1..n]$

we assign $\mathcal{C}[k] = c_k$, if there exists an $i$ such that $s_i \leq k \leq f_i, [s_i..f_i] \in \pi$; we then say that the color of $k$ is $c_k$. Any other $k \in [1..n]$ is assigned a fixed different color, say $c_{fixed}$. In other words, all the positions of the text $\mathcal{T}$, not covered by any of the intervals of $\pi$ are given a fixed color $c_{fixed}$ and every other position carries a distinct color each. We also realize the inverse relation in the form of the array, $\mathcal{C}^{-1}$, such that $C^{-1}[c_k] = k$, if and only if, $\mathcal{C}[k] = c_k$ and $c_k \neq c_{fixed}$. Note that, there may exist more than one positions having color $c_{fixed}$. We define $\mathcal{C}^{-1}(c_{fixed}) = \infty$.

Now we can reduce our problem to Problem CRSI as follows. We have two arrays $LL(\mathcal{R})$ and $\mathcal{M}$ and, respectively, two intervals $[\ell_T..r_T]$ and $[\ell..r]$. Also we have color array $\mathcal{C}$, which associates a (not necessary distinct) color to each $i \in [1..n]$. Now it is easy to see that, if we can find the distinct colors in the set of intersections of elements of $LL(\mathcal{R})[\ell_T..r_T]$ and $\mathcal{M}[\ell..r]$, then we are (almost) done. The only additional thing we need to take care of is that if we have the color $c_{fixed}$ in our output, we need to discard it. So, the Problem PMI is reduced to Problem CRSI.

On the other hand, we can see that Problem CRSI is just a different formulation of the Problem CRSG. This can be realized as follows. We set $U = n$. Since $V$ and $W$ in Problem CRSI are permutations of $[1..n]$, every number in $[1..n]$ appears precisely once in each of them. We define the coordinates of every number $i \in [1..n]$ to be $(x, y)$, where $V[x] = W[y] = i$. Thus we get the $n$ colored points (courtesy to $\mathcal{C}$) on the grid $[0..n] \times [0..n]$, i.e., the array $A$ of Problem CRSG. The query rectangle $q$ is deduced from the two intervals $[i..j]$ and $[k..\ell]$ as follows: $q \equiv (i, k) \times (j, \ell)$. It is straightforward to verify that the above reduction is correct and hence we can solve Problem CRSI using the solution of Problem CRSG.

To solve Problem CRSG, we are going to use the data structure of Agarwal et al. [1][1]. This data structure can answer the query of Problem CRSG in $O(\log \log U + K)$ time, where $K$ is the number of points contained in the query rectangle $q$. The data structure can be built in $O(n \log n \log^2 U)$ time and requires $O(n \log^2 U)$ space. Algorithm 1 formally states the steps to build our data structure. In the rest of this paper, we refer to this data structure as IDS_PMI. One final remark is that, we can use the suffix array instead of suffix tree as well with some standard modifications in Algorithm 1.

### 3.1. Analysis

Let us now analyze the cost of building the index data structure IDS_PMI. To build IDS_PMI, we first construct a traditional suffix tree requiring $O(n \log |\sigma|)$ time, where $\sigma = \min(n, |\Sigma|)$. The preprocessing on the suffix tree can be done in $O(n)$ time by traversing $ST_{\mathcal{T}}$ using a breadth first or an in order traversal. The color array $\mathcal{C}$ can be setup in $O(n)$ time because $\pi$ is a set of disjoint intervals and it can cover, at most, $n$ points. The construction of the set $A$ of points in the grid $[0..n] \times [0..n]$, on which we will apply the range search, can also be done in $O(n)$ as follows. Assume that $\mathcal{L}$ is the linked list realizing $LL(\mathcal{R})$. Each element in $\mathcal{L}$ is the label of the corresponding leaf in $LL(\mathcal{R})$. We construct $\mathcal{L}^{-1}$ such that $\mathcal{L}^{-1}[\mathcal{L}[i]] = i$. It is easy to see that, with $\mathcal{L}^{-1}$ in our hand, we can easily construct $A$ in $O(n)$. After $A$ is constructed, we build the data structure to solve Problem CRSG, which runs in $O(n \log^3 n)$ time and $O(n \log^2 n)$ space, because, $U = n$. Therefore, the index IDS_PMI can be constructed in $O(n \log^3 n)$ time.

[1]To the best of our knowledge, this is the only data structure that handles the colored range query exploiting the grid property to gain efficiency.

**Algorithm 1** Algorithm to build IDS_PMI

Build a suffix tree $ST_{\mathcal{T}}$ of $\mathcal{T}$. Let the root of $ST_{\mathcal{T}}$ is $\mathcal{R}$.
Label each leaf of $ST_{\mathcal{T}}$ by the starting location of its suffix.
Construct a linked list $\mathcal{L}$ realizing $LL(\mathcal{R})$. Each element in $\mathcal{L}$ is the label of the corresponding leaf in $LL(\mathcal{R})$.
**for** each node $v$ in $ST_{\mathcal{T}}$ **do**
    Store $v.left = i$ and $v.right = j$ such that $\mathcal{L}[i]$ and $\mathcal{L}[j]$ corresponds to, respectively, (leftmost leaf) $v_\ell$ and (rightmost leaf) $v_r$ of $v$.
**end for**
**for** $i = 1$ *to* $n$ **do**
    Set $\mathcal{M}[i] = i$
**end for**
**for** $i = 1$ *to* $n$ **do**
    Set $\mathcal{C}[i] = c_{fixed}$
**end for**
**for** $i = 1$ *to* $|\pi|$ **do**
    **for** $j = s_i$ *to* $f_i$ **do**
        $\mathcal{C}[j] = c_j$
    **end for**
**end for**
**for** $i = 1$ *to* $n$ **do**
    Set $A[i] = \epsilon$
**end for**
**for** $i = 1$ to $n$ **do**
    **if** there exists $(x, y)$ such that $\mathcal{M}[x] = \mathcal{L}[y] = i$ **then**
        Set $A[i] = A[i] \bigcup (x, y)$
    **end if**
**end for**
Preprocess $A$ (and $\mathcal{C}$) for Colored Range Search on a Grid $[0..n] \times [0..n]$.

**Algorithm 2** Algorithm for Query Processing

Find $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$.
Set $i = \mu^{\mathcal{P}}.left$, $j = \mu^{\mathcal{P}}.right$.
Compute the set $B$, where $B$ is the set of distinct colors in the set of points contained in $q \equiv (i, \ell) \times (j, r)$
**return** $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi} = \{\mathcal{C}^{-1}[x] \mid x \in B \text{ and } x \neq c_{fixed}\}$

## 3.2. Query Processing

So far we have concentrated on the construction of IDS_PMI. Now, we discuss the query processing. Suppose, we are given a query pattern $\mathcal{P}$ along with a query interval $[\ell..r]$. We first find the locus $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$. Let, $i = \mu^{\mathcal{P}}.left$ and $j = \mu^{\mathcal{P}}.right$. Then, we perform a colored range search query on $A$ with the rectangle $q \equiv (i, \ell) \times (j, r)$. Let $B$ be the set of those colors as output by the query. Then, it is easy

to verify that, $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi} = \{\mathcal{C}^{-1}[x] \mid x \in B \text{ and } x \neq c_{fixed}\}$. The steps are formally presented in the form of Algorithm 2.

The running time of the query processing is deduced as follows. Finding the locus $\mu^{\mathcal{P}}$ can be done in $O(m)$ time. The corresponding pointers can be found in constant time. The construction of the set $B$ is done by performing the range query in $O(\log\log n + |B|)$ time. Note that $|B|$ is either equal to $|Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}|$ or just one unit more than that. The latter happens when we have $c_{fixed} \in B$. So, in total the query time is $O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}| + 1) = O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}|)$. We state the results of this section in the form of following theorem.

**Theorem 3.1.** For Problem PMI, we can construct the IDS_PMI data structure in $O(n\log^3 n)$ time and $O(n\log^2 n)$ space and we can answer the relevant queries in $O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}|)$ time per query.

# 4. Problem PMGI

In Section 3, we have presented an efficient index data structure, namely IDS_PMI, to solve Problem PMI. In this section, we consider Problem PMGI. Since PMI is a generalized version of PMGI, we can easily use the solution in Section 3 to solve PMGI as follows. We use the same data structure IDS_PMI. During the query, since PMGI doesn't have any query interval, we just need to assume the query interval to be $[1..n]$. So we have the following theorem.

**Theorem 4.1.** For Problem PMGI, we can construct the IDS_PMI data structure in $O(n\log^3 n)$ time and $O(n\log^2 n)$ space and we can answer the relevant queries in $O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T},\pi}|)$ time per query.

However, as it turns out, we can achieve better results for Problem PMGI. And in fact, as we show below, we can solve Problem PMGI optimally. We first discuss how we construct the data structure, namely IDS_PMGI, to solve PMGI. As before, we start by constructing a suffix tree (or suffix array) $ST_{\mathcal{T}}$. Then we do all the preprocessing done on $ST_{\mathcal{T}}$ as we did to construct IDS_PMI. We also construct the color array $\mathcal{C}$. This time however, we do a slightly different encoding as follows. For each $k \in [1..n]$, we assign $\mathcal{C}[k] = -1$, if there exists an $i$ such that $s_i \leq \mathcal{L}[k] \leq f_i, [s_i..f_i] \in \pi$. For all other $k \in [1..n]$ we assign $\mathcal{C}[k] = 0$. In other words, all the positions of the text $\mathcal{T}$, not covered by any of the intervals of $\pi$ gets 0 as their color and every other positions gets the color $-1$. Now we make use of the following interesting problem.

**Problem "RMIN" (Range Minima Query Problem).** We are given an array $A[1..n]$ of numbers. We need to preprocess $A$ to answer following form of queries:
Query: Given an interval $I = [i_s..i_e], 1 \leq i_s \leq i_e \leq n$, the goal is to find the index $k$ (or the value $A[k]$ itself) with minimum (maximum, in the case of Range Maxima Query) value $A[k]$ for $k \in I$. This query is called a *Range Minima Query (RMQ)*.

Problem RMIN has received much attention in the literature. To the best of our knowledge, it all started with Harel and Tarjan [23], who showed how to solve another interesting problem, namely the Lowest Common Ancestor (LCA) problem with linear time preprocessing and constant time queries. Problem LCA has as its input a tree. The query gives two nodes in the tree and requests the lowest common ancestor of the two nodes. It turns out that, solving Problem RMIN is equivalent to solving

the LCA problem [21]. The Harel-Tarjan algorithm was simplified by Schieber and Vishkin [42] and then by Berkman et al. [10], who presented optimal work parallel algorithms for the LCA problem. Finally, Bender and Farach-Colton eliminated the parallelism mechanism and presented a simple serial data structure [9]. Notably, the data structure in [9] requires $O(n \log n)$ bits of space. Recently, Sadakane [41] presented a succinct data structure, which achieves the same time complexity using $O(n)$ bits of space. In all of the above papers, there is an interplay between Problems LCA and RMIN. Very recently, Fischer and Heun [30] presented the first algorithm for Problem RMIN with linear preprocessing time, optimal $2n + o(n)$ bits of additional space, and constant query time that makes no use of Problem LCA.

Now, we preprocess the array $\mathcal{C}$ for Problem RMIN. Note that, in $\mathcal{C}$, we have only two values. So to define a unique value in case of a tie, we consider the index along with the value. More formally, we define $\mathcal{C}[i] \prec \mathcal{C}[j], 1 \leq i \neq j \leq n$ if and only if $C[i] \leq C[j]$ and $i < j$. And we employ RMQ using the relation $\prec$. Finally, for each $\mathcal{C}[i] = -1$, we maintain a pointer to $\mathcal{C}[j] = -1$ such that $j > i$ and $j$ is the smallest index with this property; if there doesn't exist any such $\mathcal{C}[j]$, then $\mathcal{C}[i]$ points to 'NULL'. More formally, we maintain another array $\mathcal{D}[1..n]$ such that for all $i \in [1..n]$ with $\mathcal{C}[i] = -1$, we have $\mathcal{D}[i] = j$, if and only if, $\mathcal{C}[j] = -1$ and $\mathcal{C}[k] = 0, i < k < j$. For all other indices the $\mathcal{D}$ is given a 'NULL' value. This completes the construction of IDS_PMGI. Note that, the overall running time to construct IDG_PMGI remains dominated by the construction of the suffix tree $ST_T$. As a result, the construction time is $O(n)$ for bounded alphabet and $O(n \log \sigma)$ otherwise[2]. Notably, the construction time remains $O(n)$ even for alphabets that are natural numbers from 1 to a polynomial in $n$.

Now we discuss how we perform the query on IDS_PMGI. Suppose we are given a query pattern $\mathcal{P}$. We first find the locus $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$. Let $i = \mu^{\mathcal{P}}.left$ and $j = \mu^{\mathcal{P}}.right$. Now we basically have the set $Occ_{\mathcal{T}}^{\mathcal{P}}$ in $\mathcal{L}[i..j]$. Now we perform a range minima query on $\mathcal{C}$ with the query interval $[i..j]$. This gives us, in constant time, the first index $k \in [i..j]$ such that $C[k] = -1$. Then we follow the linked list realized by $\mathcal{D}$ to report all the indices in the range $[i..j]$ having color $-1$. More formally, we construct the set $B = \{k \mid k \in [i..j] \text{ and } \mathcal{C}[k] = -1\}$. With the help of $\mathcal{D}$ this can be done in $O(|B|)$ time. And it is easy to realize that $Occ_{\mathcal{T},\pi}^{\mathcal{P}} = \{\mathcal{L}[i] \mid i \in B\}$. Therefore we can perform the query in optimal $O(m + |Occ_{\mathcal{T},\pi}^{\mathcal{P}}|)$ time. The following theorem presents the results achieved in this section.

**Theorem 4.2.** For Problem PMGI, we can construct the IDS_PMGI data structure in $O(n \log \sigma)$ time ($\sigma = \min(n, |\Sigma|)$) and $O(n \log n)$ bits of space and the relevant queries can be answered optimally in $O(m + |Occ_{\mathcal{T},\pi}^{\mathcal{P}}|)$ time per query.

For alphabets that are bounded or are natural numbers from 1 to a polynomial in $n$, we have the following result.

**Theorem 4.3.** For Problem PMGI, we can construct the IDS_PMGI data structure in $O(n)$ time and $O(n \log n)$ bits of space and the relevant queries can be answered optimally in $O(m + |Occ_{\mathcal{T},\pi}^{\mathcal{P}}|)$ time per query.

## 4.1. A Simpler Index for Problem PMGI

In this section, we present a simpler indexing scheme to solve Problem PMGI. However, we don't achieve any improvement in the asymptotic behavior of the algorithm. The resulting data structure is referred to as IDS_PMGI*. Below we present the construction of IDS_PMGI*.

[2]Recall that, $\sigma = \min(n, |\Sigma|)$.

As before, we start with the suffix tree data structure. But we do a slightly different preprocessing on the suffix tree than we did before. Recall that, we realize the list $LL(\mathcal{R})$ in the form of a linked list $\mathcal{L}$, where $\mathcal{R}$ is the root of the suffix tree. This time however, we maintain $\mathcal{L}$ as a doubly linked list. Also, recall that, we set pointers $v.left$ and $v.right$ from each tree node $v$ to its leftmost leaf $v_\ell$ and rightmost leaf $v_r$ (considering the subtree rooted at $v$) in the linked list. Now, we modify the pointer $v.left$ so that it points to the immediate left leaf of $v_\ell$ (instead of $v_\ell$ itself). Also, we maintain a list $B_i^{left}, 1 \leq i \leq n$, for each $\mathcal{L}[i]$ as follows: we put $u \in B_i^{left}$, if, and only if, $u.left$ points to $\mathcal{L}[i]$. Similarly we maintain $B_i^{right}, 1 \leq i \leq n$. Now, we traverse $\mathcal{L}$ from left to right and delete all nodes $\mathcal{L}[i]$ such that there exists no $j$ such that $s_j \leq \mathcal{L}[i] \leq f_j, [s_j..f_j] \in \pi$. In other words, we delete those nodes of $\mathcal{L}$ which don't fall within any of the ranges in $\pi$. Note carefully that, while the deletion is done, the forward and back pointers of the underlying doubly linked list structure has to be updated correctly. Furthermore, if $\mathcal{L}[i]$ is deleted and $B_i^{left}(B_i^{right})$ is non-empty, then for all $u \in B_i^{left}(u \in B_i^{right})$, we have to assign the immediate left node of $\mathcal{L}[i]$ to $u.left(u.right)$. Note that, in this way, for every node in the suffix tree we need to change the corresponding pointers at most once, ensuring the spending of no more that $O(n)$ extra cost. This completes the construction of IDS_PMGI*.

The query processing is done as follows. Suppose we are given a query pattern $\mathcal{P}$. We first find the locus $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$. Let $i = \mu^{\mathcal{P}}.left$ and $j = \mu^{\mathcal{P}}.right$. It is easy to realize that, we now directly have the set $Occ^{\mathcal{P}}_{\mathcal{T},\pi}$ in $\mathcal{L}[i+1..j]$. One detail is that we should first check whether $i = j$ because in that case $Occ^{\mathcal{P}}_{\mathcal{T},\pi} = \emptyset$. Therefore, once the locus is located, the set $Occ^{\mathcal{P}}_{\mathcal{T},\pi}$ can be constructed spending optimal $O(1)$ time per occurrences.

It is easy to see that, the asymptotical construction time of IDS_PMGI* is identical to that of IDS_PMGI. Nevertheless, IDS_PMGI* is a much simpler data structure than IDS_PMGI because it doesn't need the RMQ data structure and hence should be preferable among the two. However, IDS_PMGI has an interesting feature that it still retains the capability to answer "normal" pattern queries, because, the original suffix tree is still there in the data structure. Additionally, IDS_PMGI can be combined with other data structures presented in this paper to provide multi-functionality in the context of pattern matching. This unique feature may turn out to be very useful in different settings, where different types of queries could be of interest at the same time.

## 5. Problem PMQI

This section is devoted to Problem PMQI. As is mentioned above, PMQI was studied extensively in [34]. The best query time achieved in [34] was $O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ against a data structure exhibiting $O(n \log^{1+\epsilon} n)$ space and time complexity, where $0 \leq \epsilon \leq 1$. Note that, we can easily use IDS_PMI to solve PMQI by assuming $\pi = \{[1..n]\}$. So, with a slightly worse data structure construction time, we can achieve the same query time of [34] to solve PMQI using our data structure IDS_PMI (that solves a more general problem, namely PMI). However, as pointed out in [34], it would be really interesting to get an optimal query time for this problem. In this section, we attain an optimal query time for PMQI.

The data structure, namely IDS_PMQI, is constructed as follows. As before, we start by constructing a suffix tree (or suffix array) $ST_{\mathcal{T}}$. Then we do all the preprocessing done on $ST_{\mathcal{T}}$ as we did to construct IDS_PMI and IDS_PMGI. Recall that, with $ST_{\mathcal{T}}$ in our hand, preprocessed as above, we can have the set $Occ^{\mathcal{P}}_{\mathcal{T}}$ in the form of $\mathcal{L}[i..j]$ in $O(m)$ time. To achieve the optimal query time we now must 'select'

$k \in \mathcal{L}[i..j]$ such that $k \in [\ell..r]$ without spending more than constant time per selection. To achieve this goal we introduce the following interesting problem.

**Problem "RNV" (Range Next Value Query Problem).** We are given an array $A[1..n]$, which is a permutation of $[1..n]$. We need to preprocess $A$ to answer the following form of queries.

***Query:*** Given an integer $\mathcal{K} \in [1..n]$, and an interval $[i..j], 1 \leq i \leq j \leq n$, the goal is to return the index of the immediate higher (or equal) number ("next value") than $\mathcal{K}$ from $A[i..j]$, if there exists one. More formally, we need to return $\ell$ (or $A[\ell]$ as the value itself) such that $A[\ell] = \min\{A[q] \mid A[q] \geq \mathcal{K}$ and $i \leq q \leq j\}$. If there exists no such $\ell$, then we return $-1$.

We use $RNV_A([\ell..r], \mathcal{K})$ to denote the range next value query on $A[\ell..r]$ for the value $\mathcal{K}$. We first show how we can use it to answer the queries of Problem PMQI efficiently. To complete the construction of IDS_PMQI, we preprocess the array $\mathcal{L}$ for Problem RNV. The query processing is as follows. Recall that, we can have the set $Occ^{\mathcal{P}}_{\mathcal{T}}$ in the form of $\mathcal{L}[i..j]$ in $O(m)$ time. Recall also that, as part of the PMQI query, we are given an interval $[\ell..r]$. Now, we perform an RNV query on $\mathcal{L}$ with the parameters $\ell$ and $[i..j]$. Suppose the query returns the index $q$. It is easy to see that if $\mathcal{L}[q] \leq r$, then $\mathcal{L}[q] \in Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}$. And then, we repeat the RNV query with parameters $\mathcal{L}[q]$ and $[i..j]$. We stop, as soon as a query returns an index $q$ such that $\mathcal{L}[q] > r$. Now, assume that, we can construct a data structure to solve Problem RNV in $f(n)$ time and achieve a query time of $g(n)$ per query. Then, we have the following theorem.

**Theorem 5.1.** For Problem PMQI, we can construct a data structure, namely IDS_PMQI, in $O(n \log \sigma + f(n))$ time and space to answer the relevant queries in optimal $O(m + \Psi(n))$ time per query, where $\Psi(n) = |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}| \times g(n)$.

As is evident from Theorem 5.1, to achieve optimal query time for Problem PMQI, we must get a solution to Problem RNV with optimal query time. Problem RNV was introduced in [15], where a data structure with $O(n^2)$ time and space complexity was presented to achieve the optimal $O(1)$ query time. Very recently, Crochemore et al. [14], presented an improved data structure to solve Problem RNV that supports constant query time and can be built in $O(n^{1.5})$ time and space. Therefore, we get the following result for Problem PMQI.

**Theorem 5.2.** For Problem PMQI, we can construct IDS_PMQI, in $O(n^{1.5})$ time and space to answer the relevant query in optimal $O(m + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ time.

# 6. Optimal Query Time for Problem PMI

In Section 3, we have presented the data structure IDS_PMI to solve Problem PMI. Recall that IDS_PMI can be constructed in $O(n \log^3 n)$ time. The query time is $O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}|)$ per query, which is not optimal. Interestingly, we can achieve optimal query time for Problem PMI albeit at the cost of an increased construction time. Let us refer to this new data structure as IDS_PMI*. The idea is to combine the construction of IDS_PMGI* with IDS_PMQI. First we construct IDS_PMGI*. Then we preprocess $\mathcal{L}$ for Problem RNV, which completes the construction of IDS_PMI*.

Now the query processing is as follows. Again, suppose that we are given a query pattern $\mathcal{P}$. We first find the locus $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$ as before. Let $i = \mu^{\mathcal{P}}.left$ and $j = \mu^{\mathcal{P}}.right$. Recall that, now we directly have the set $Occ^{\mathcal{P}}_{\mathcal{T},\pi}$ in $\mathcal{L}[i+1..j]$. To construct the set $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}$, we just need to perform the RNV queries on $\mathcal{L}$ just as we did for Problem PMQI with IDS_PMQI*. The only difference is that the

parameters of the first query would have to be $\ell$ and $[i+1..j]$ (instead of $[i..j]$). The following theorem summarizes the results we achieve here.

**Theorem 6.1.** For Problem PMI, we can construct the IDS_PMI* data structure in $O(n^{1.5})$ time and space and we can answer the relevant queries in optimal $O(m + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}|)$ time per query.

It is clear that the query time is optimal. However, note that the construction time of IDS_PMI* is higher than that of IDS_PMI. The bottleneck here is the preprocessing time for Problem RNV.

## 7. Conclusion

In this paper, we have considered the problem of pattern matching in given intervals and focused on building index data structures to handle different versions of this problem efficiently. We first handled the more general problem PMI and presented an efficient data structure requiring $O(n \log^3 n)$ time and $O(n \log^2 n)$ space with a query time of $O(m + \log\log n + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r],\pi}|)$ per query. We then solved Problem PMGI optimally ($O(n)$ time and $O(n \log n)$-bits space data structure and $O(m + |Occ^{\mathcal{P}}_{\mathcal{T},\pi}|)$ query time). Finally, we improved the query time of [34] for Problem PMQI to optimal, i.e., $O(m + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ per query, although, at the expense of a more costly data structure requiring $O(n^{1.5})$ time and space. We also showed how to achieve optimal query time for Problem PMI at the cost of an increased preprocessing time. It would be interesting to improve the preprocessing time of both Problem PMI and PMQI and also the query time of the former without increasing the preprocessing cost. Of particular interest is the improvement of the $O(n^{1.5})$ data structure of PMQI without sacrificing the optimal query time. Furthermore, as is shown in [14], Problem RNV is of independent interest and could be investigated further.

## References

[1] Agarwal, P. K., Govindarajan, S., Muthukrishnan, S.: Range Searching in Categorical Data: Colored Range Searching on Grid., *ESA* (R. H. Möhring, R. Raman, Eds.), 2461, Springer, 2002, ISBN 3-540-44180-8.

[2] Amir, A., Aumann, Y., Benson, G., Levy, A., Lipsky, O., Porat, E., Skiena, S., Vishne, U.: Pattern matching with address errors: rearrangement distances., *SODA*, ACM Press, 2006, ISBN 0-89871-605-5.

[3] Amir, A., Butman, A., Crochemore, M., Landau, G. M., Schaps, M.: Two-dimensional pattern matching with rotations., *Theor. Comput. Sci.*, **314**(1-2), 2004, 173–187.

[4] Amir, A., Butman, A., Lewenstein, M.: Real Scaled Matching., *Inf. Process. Lett.*, **70**(4), 1999, 185–190.

[5] Amir, A., Chencinski, E.: Faster Two Dimensional Scaled Matching., *CPM* (M. Lewenstein, G. Valiente, Eds.), 4009, Springer, 2006, ISBN 3-540-35455-7.

[6] Amir, A., Chencinski, E., Iliopoulos, C., Kopelowitz, T., Zhang, H.: Property Matching and Weighted Matching, *CPM*, 2006.

[7] Amir, A., Kapah, O., Tsur, D.: Faster Two Dimensional Pattern Matching with Rotations., *CPM* (S. C. Sahinalp, S. Muthukrishnan, U. Dogrusöz, Eds.), 3109, Springer, 2004, ISBN 3-540-22341-X.

[8] Apostolico, A.: The Myriad Virtues of Subword Trees, *Combinatorial Algorithms on Words*, NATO ISI Series, Springer-verlag, 1985.

[9] Bender, M. A., Farach-Colton, M.: The LCA Problem Revisited., *Latin American Theoretical INformatics (LATIN)*, 2000.

[10] Berkman, O., Breslauer, D., Galil, Z., Schieber, B., Vishkin, U.: Highly Parallelizable Problems (Extended Abstract), *STOC*, ACM, 1989.

[11] Butman, A., Eres, R., Landau, G. M.: Scaled and permuted string matching., *Inf. Process. Lett.*, **92**(6), 2004, 293–297.

[12] Cole, R., Gottlieb, L.-A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares., *STOC* (L. Babai, Ed.), ACM, 2004, ISBN 1-58113-852-0.

[13] Cole, R., Hariharan, R.: Verifying candidate matches in sparse and wildcard matching., *STOC*, 2002.

[14] Crochemore, M., Iliopoulos, C. S., Kubica, M., Rahman, M. S., Walen, T.: Improved Algorithms for the Range Next Value Problem and Applications, *STACS* (S. Albers, P. Weil, C. Rochange, Eds.), 08001, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2008.

[15] Crochemore, M., Iliopoulos, C. S., Rahman, M. S.: Finding Patterns in Given Intervals, *MFCS* (L. Kucera, A. Kucera, Eds.), 4708, Springer, 2007, ISBN 978-3-540-74455-9.

[16] Crochemore, M., Iliopoulos, C. S., Rahman, M. S.: Optimal Prefix and Suffix Queries on Texts., *AofA* (P. Jacquet, Ed.), AH, DMTCS Proceedings, 2007.

[17] Crochemore, M., Rytter, W.: *Jewels of Stringology*, World Scientific, 2002.

[18] Farach, M.: Optimal Suffix Tree Construction with Large Alphabets., *FOCS*, 1997.

[19] Fischer, M., Paterson, M.: String matching and other Products, *in Complexity of Computation, R.M. Karp (editor), SIAM AMS Proceedings*, **7**, 1974, 113–125.

[20] Flouri, T., Iliopoulos, C. S., Rahman, M. S., Vagner, L., Vorácek, M.: Indexing Factors in DNA/RNA sequences., *BIRD08-ALBIO08*, 2008, *Lecture Notes in Bioinformatics*, 436–445.

[21] Gabow, H., Bentley, J., Tarjan, R.: Scaling and Related Techniques for Geometry Problems, *Symposium on the Theory of Computing (STOC)*, 1984.

[22] Gusfield, D.: *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*, Cambridge University Press, 1997, ISBN 0-521-58519-8.

[23] Harel, D., Tarjan, R. E.: Fast Algorithms for Finding Nearest Common Ancestors., *SIAM J. Comput.*, **13**(2), 1984, 338–355.

[24] Hirschberg, D. S.: Algorithms for the Longest Common Subsequence Problem., *J. ACM*, **24**(4), 1977, 664–675.

[25] Hunt, J. W., Szymanski, T. G.: A Fast Algorithm for Computing Longest Subsequences., *Commun. ACM*, **20**(5), 1977, 350–353.

[26] Iliopoulos, C. S., Mouchard, L., Rahman, M. S.: A New Approach to Pattern Matching in Degenerate DNA/RNA Sequences and Distributed Pattern Matching, *Mathematics in Computer Science*, **1**(4), 2008, 557–569.

[27] Iliopoulos, C. S., Rahman, M. S.: Faster index for property matching, *Inf. Process. Lett.*, **105**(6), 2008, 218–223.

[28] Iliopoulos, C. S., Rahman, M. S.: Indexing Factors with Gaps, *Algorithmica*, **55**(1), 2009, 60–70.

[29] Iliopoulos, C. S., Rahman, M. S.: A New Efficient Algorithm for Computing the Longest Common Subsequence, *Theory Comput. Syst.*, **45**(2), 2009, 355–371.

[30] Johannes Fischer, V. H.: A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array., *ESCAPE* (B. Chen, G. Zhang, Eds.), 4614, Springer, 2007.

[31] Jurka, J.: Human repetitive elements., in: *Molecular Biology and Biotechnology.* (R. A. Meyers, Ed.).

[32] Jurka, J.: Origin and evolution of Alu repetitive elements., in: *The impact of short interspersed elements (SINEs) on the host genome.* (R. Maraia, Ed.).

[33] Levenshtein, V.: Binary Codes Capable of Correcting, Deletions, Insertions and Reversals, *Soviet Phys. Dokl.*, **10**, 1966, 707–710.

[34] Mäkinen, V., Navarro, G.: Position-Restricted Substring Searching, *LATIN*, 2006.

[35] McCreight, E. M.: A Space-Economical Suffix Tree Construction Algorithm., *J. ACM*, **23**(2), 1976, 262–272.

[36] Rahman, M. S., Iliopoulos, C., Lee, I., Mohamed, M., Smyth, W.: Finding Patterns with Variable Length Gaps or Don't Cares., *COCOON* (D. Chen, D. Lee, Eds.), 4112, Springer, 2006.

[37] Rahman, M. S., Iliopoulos, C. S.: Indexing Factors with Gaps., *SOFSEM* (J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, F. Plasil, Eds.), 4362, Springer, 2007, ISBN 978-3-540-69506-6.

[38] Rahman, M. S., Iliopoulos, C. S.: A New Efficient Algorithm for Computing the Longest Common Subsequence., *AAIM* (M.-Y. Kao, X.-Y. Li, Eds.), 4508, Springer, 2007, ISBN 978-3-540-72868-9.

[39] Rahman, M. S., Iliopoulos, C. S.: Pattern Matching Algorithms with Don't Cares., *SOFSEM (2)* (J. van Leeuwen, G. F. Italiano, W. van der Hoek, C. Meinel, H. Sack, F. Plasil, M. Bieliková, Eds.), Institute of Computer Science AS CR, Prague, 2007, ISBN 80-903298-9-6.

[40] Rahman, M. S., Iliopoulos, C. S., Mouchard, L.: Pattern Matching in Degenerate DNA/RNA Sequences., *WALCOM*, 2007.

[41] Sadakane, K.: Succinct Data Structures for Flexible Text Retrieval Systems., *Journal of Discrete Algorithms*, **5**(1), 2007, 12–22.

[42] Schieber, B., Vishkin, U.: On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM J. Comput.*, **17**(6), 1988, 1253–1262.

[43] Ukkonen, E.: On-Line Construction of Suffix Trees., *Algorithmica*, **14**(3), 1995, 249–260.