Finding Patterns In Given Intervals

Maxime Crochemore[1,2,4], Costas S. Iliopoulos[1,3,⋆], and M. Sohel Rahman[1,3,⋆⋆,⋆⋆⋆]

[1] Algorithm Design Group
Department of Computer Science
King's College London
Strand, London WC2R 2LS, England
http://www.dcs.kcl.ac.uk/adg
[2] Institut Gaspard-Monge
University of Marne-la-Vallée, France
[3] {csi, sohel}@dcs.kcl.ac.uk
[4] Maxime.Crochemore@kcl.ac.uk

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 cases, we develop solutions for different variants of this problem. In particular, we present efficient indexing schemes for each of the above variants of the problem.

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 Σ. This problem is interesting as a fundamental computer science problem and is a basic need of many applications, such as text retrieval, music 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 has considered *'inexact matching'*. Many types of differences have been defined and studied in the literature, namely, errors (Hamming distance, LCS [10, 17], edit distance [10, 20]), wild cards or don't cares [10, 11, 14, 23], rotations [3, 7], scaling [4, 5], permutations [9] among others.

Contemporary research on pattern matching has taken many other different and interesting directions ranging from position restricted pattern matching [21] to pattern matching with address error [2] and property matching [6]. In this paper, we are interested in pattern matching in given intervals and focus on

⋆ Supported by EPSRC and Royal Society grants.
⋆⋆ Supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP).
⋆⋆⋆ On Leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.

building an index data structure 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 intervals are 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 [18] such as tandem repeats, LINEs (Long Interspersed Nuclear Sequences) and SINEs (Short Interspersed Nuclear Sequences) [19]. 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 become 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], whereas Problem PMQI was introduced and handled in [21] (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) optimally ($O(m+K)$ query time on a data structure with $O(n)$ time and $O(n \log n)$-bit space complexity). Finally, we improve the query time of [21] for Problem PMQI (Section 5) to optimal i.e. $O(m+K)$ per query. The corresponding data structure, however, requires $O(n^2)$ time due to a costly preprocessing of an intermediate problem, which remains as the bottleneck in the overall running time.

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 5. We conclude briefly in Section 6.

2 Preliminaries

A *text*, also called a *string*, is a sequence of zero or more symbols from an alphabet Σ. 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*

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

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*

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

Problem PMQI was studied extensively in [21]. The authors in [21] presented a number of algorithms depending on different trade-offs between the time and space complexities. The best query time they achieved was $O(m + \log\log n + |Occ_{\mathcal{T}[\ell..r]}^{\mathcal{P}}|)$ against a data structure exhibiting $O(n \log^{1+\epsilon} n)$ space and time complexity, where $0 \leq \epsilon \leq 1$. A slightly different version of Problem PMGI was studied in [6][5]. 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, π, whereas in Problem PMGI, only the occurreces that starts in π are of interest. 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.

[5] In [6] a data structure requiring $O(n \log \Sigma + n \log\log n)$ time was presented to support $O(m + K)$ time query, where K is the output size.

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 [22, 25] or in any textbook on stringology (e.g. [12, 16]). 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 Σ, 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 ℓ_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. Several algorithms exist that can construct the suffix tree $ST_{\mathcal{T}}$ in $O(n \log \Sigma)$ time[6] [22, 25, 13]. The space requirement of suffix tree is $O(n \log n)$ bits. 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 $\mathcal{T}$. Given a suffix tree of a text $\mathcal{T}$, a pattern $\mathcal{P}$, one can find its locus and hence the fact whether $\mathcal{T}$ has an occurrence of $\mathcal{P}$ in optimal $O(|\mathcal{P}|)$ time.In addition to that, all such occurrences can be reported in constant time per occurrence.

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_ℓ and rightmost leaf v_r (considering the subtree rooted at v) in the linked list. It is easy to realize that, with these set of 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

[6] For bounded alphabet the running time remains linear, i.e. $O(n)$.

terms ℓ_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 lies 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 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 π 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 π 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.

Algorithm 1 Algorithm to build IDS_PMI

1: Build a suffix tree $ST_{\mathcal{T}}$ of $\mathcal{T}$. Let the root of $ST_{\mathcal{T}}$ is $\mathcal{R}$.
2: Label each leaf of $ST_{\mathcal{T}}$ by the starting location of its suffix.
3: Construct a linked list $\mathcal{L}$ realizing $LL(\mathcal{R})$. Each element in $\mathcal{L}$ is the label of the corresponding leaf in $LL(\mathcal{R})$.
4: **for** each node v in $ST_{\mathcal{T}}$ **do**
5: Store $v.left = i$ and $v.right = j$ such that $\mathcal{L}[i]$ and $\mathcal{L}[j]$ corresponds to, respectively, (leftmost leaf) v_ℓ and (rightmost leaf) v_r of v.
6: **end for**
7: **for** $i = 1$ *to* n **do**
8: Set $\mathcal{M}[i] = i$
9: **end for**
10: **for** $i = 1$ *to* n **do**
11: Set $\mathcal{C}[i] = c_{fixed}$
12: **end for**
13: **for** $i = 1$ *to* $|\pi|$ **do**
14: **for** $j = s_i$ *to* f_i **do**
15: $\mathcal{C}[j] = c_j$
16: **end for**
17: **end for**
18: **for** $i = 1$ *to* n **do**
19: Set $A[i] = \epsilon$
20: **end for**
21: **for** $i = 1$ to n **do**
22: **if** there exists (x, y) such that $\mathcal{M}[x] = \mathcal{L}[y] = i$ **then**
23: Set $A[i] = A[i] \bigcup (x, y)$
24: **end if**
25: **end for**
26: Preprocess A (and $\mathcal{C}$) for Colored Range Search on a Grid $[0..n] \times [0..n]$.

To solve Problem CRSG, we are going to use the data structure of Agarwal et al. [1][7]. 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.

[7] 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.

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. The preprocessing on the suffix tree can be done in $O(n)$ by traversing $ST_{\mathcal{T}}$ using a breadth first or in order traversal. The color array $\mathcal{C}$ can be setup in $O(n)$ because π 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 requires $O(n \log^3 n)$ time and $O(n \log^2 n)$ space because $U = n$. Since, we can assume $\Sigma \leq n$, the index IDS_PMI can be constructed in $O(n \log^3 n)$ time.

Algorithm 2 Algorithm for Query Processing

1: Find $\mu^{\mathcal{P}}$ in $ST_{\mathcal{T}}$.
2: Set $i = \mu^{\mathcal{P}}.left$, $j = \mu^{\mathcal{P}}.right$.
3: Compute the set B, where B is the set of distinct colors in the set of points contained in $q \equiv (i, \ell) \times (j, r)$
4: **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 is 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 $\mu^{\mathcal{P}}$ requires $O(m)$ time. The corresponding pointers can be found in constant time. The construction of the set B is done by performing the range query and hence requires $O(\log \log n + |B|)$ 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 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. 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 2. *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 $s_i \leq k \leq f_i, [s_i..f_i] \in \pi$. 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 π 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 $k \in I$.

Problem RMIN has received much attention in the literature and Bender and Farach-Colton showed that we can build a data structure in $O(n)$ time using $O(n \log n)$-bit space and can answer subsequent queries in $O(1)$ time per query [8][8]. Recently, Sadakane [24] presented a succinct data structure which achieves the same time complexity using $O(n)$ bits of space.

Now, we preprocess the array $\mathcal{C}$ to answer the range minima queries (RMQ). 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

[8] The same result was achieved in [15], albeit with a more complex data structure.

RMQ using the relation $\prec$. This can be easily done in $O(n)$ slightly modifying the preprocessing used in [8][9]. 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 index 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.

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 $Occ^{\mathcal{P}}_{\mathcal{T}}$ 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 [8], 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 $Occ^{\mathcal{P}}_{\mathcal{T},\pi} = \{\mathcal{L}[i] \mid i \in B\}$. Therefore we can perform the query in optimal $O(m+|Occ^{\mathcal{P}}_{\mathcal{T},\pi}|)$ time. The following theorem present the results achieved in this section.

Theorem 3. *For Problem PMGI, we can construct the IDS_PMGI data structure in $O(n \log \Sigma)$ time and $O(n \log n)$ bits of space and the relevant queries can be answered optimally in $O(m + |Occ^{\mathcal{P}}_{\mathcal{T},\pi}|)$ time per query.*

For bounded alphabets, we have the following result.

Theorem 4. *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 $O(m + |Occ^{\mathcal{P}}_{\mathcal{T},\pi}|)$ time per query.*

5 Problem PMQI

This section is devoted to Problem PMQI. As is mentioned above, PMQI was studied extensively in [21]. The best query time achieved in [21] 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 [21] to solve PMQI using our data structure IDS_PMI to solve a more general problem, namely PMI. However, as pointed out in [21], it would be really interesting to get an optimal query time for this problem. In this section, we attain an optimal query time

[9] In particular, the only modification needed is in the construction of the Cartesian tree.

for PMQI. However, the optimal query time is achieved against a $O(n^2)$ preprocessing time.

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 $Occ_{\mathcal{T}}^{\mathcal{P}}$ 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 an permutation of $[1..n]$. We need to preprocess A to answer the following form of queries.*
Query: Given an integer $k \in [1..n]$, 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 k from $A[i..j]$ if there exists one. More formally, we need to return ℓ (or $A[\ell]$ as the value itself) such that $i \leq \ell \leq j$ and $A[\ell] = \min\{A[q] \mid A[q] \geq k \text{ and } i \leq q \leq j\}$

Despite extensive results on various range searching problems we are not aware of any result that directly addresses this problem. Recall that our goal now is to answer the RNV queries in $O(1)$ time per query. We below give a solution where we can preprocess A in $O(n^2)$ time and then can answer the subsequent queries in $O(1)$ time per query. The idea is is as follows. We maintain n arrays $B_i, 1 \leq i \leq n$. Each array, B_i has n elements. So we could view B as a two dimensional array as well. We fill each array B_i depending on A as follows. For each $1 \leq i \leq n$ we store in B_i the difference between i and the corresponding element of A and then replace all negative entries of B_i with ∞. More formally, for each $1 \leq i \leq n$ and for each $1 \leq j \leq n$ we set $B_i[j] = A[j] - i$ if $A[j] \geq i$; otherwise we set $B_i[j] = \infty$. Then we preprocess each $B_i, 1 \leq i \leq n$ for range minima query [8]. This completes the construction of the data structure. It is clear that it will require $O(n^2)$ time. The query processing is as follows. Suppose the query parameters are k and $[i..j]$. Then we simply apply range minima query in B_k for the interval $[i..j]$. So we have the following theorem.

Theorem 5. *For Problem RNV, we can construct a data structure in $O(n^2)$ time and space to answer the relevant queries in $O(1)$ time per query.*

Now we show how we can use the result of Theorem 5 to answer the queries of Problem PMQI optimally. 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 $Occ_{\mathcal{T}}^{\mathcal{P}}$ 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 ℓ and $[i..j]$. Suppose the query returns the index q. It is easy to see that if $\mathcal{L}[q] \leq r$, then $\mathcal{L}[q] \in Occ_{\mathcal{T}[\ell..r]}^{\mathcal{P}}$. 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 $\mathcal{L}[q] > r$. So, in this way, given $\mathcal{L}[i..j]$, we can get the set $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}$ in $O(|Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ time. So we have the following theorem.

Theorem 6. *For Problem PMQI, we can construct a data structure, namely IDS_PMQI, in $O(n^2)$ time and space to answer the relevant query in optimal $O(m + |Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}|)$ time.*

It is clear that the bottleneck in the construction time lies in the preprocessing of Problem RNV. So any improvement on Theorem 5 would improve Theorem 6 as well. One interesting fact is that, the occurrences in $Occ^{\mathcal{P}}_{\mathcal{T}[\ell..r]}$ as output by our algorithm will always remain sorted according to their position in $\mathcal{T}$, which in many applications may turn out to be useful.

6 Conclusion

In this paper, we have considered the problem of pattern matching in given intervals and focused on building index data structure 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 [21] 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 $O(n^2)$ time and space. It would be interesting to improve the preprocessing time of both Problem PMI and PMQI and also the query time of the former. Of particular interest is the improvement of the $O(n^2)$ data structure of PMQI without sacrificing the optimal query time. Furthermore, we believe that Problem RNV is of independent interest and could be investigated further.

References

1. P. K. Agarwal, S. Govindarajan, and S. Muthukrishnan. Range searching in categorical data: Colored range searching on grid. In R. H. Möhring and R. Raman, editors, *ESA*, volume 2461 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 2002.
2. A. Amir, Y. Aumann, G. Benson, A. Levy, O. Lipsky, E. Porat, S. Skiena, and U. Vishne. Pattern matching with address errors: rearrangement distances. In *SODA*, pages 1221–1229. ACM Press, 2006.
3. A. Amir, A. Butman, M. Crochemore, G. M. Landau, and M. Schaps. Two-dimensional pattern matching with rotations. *Theor. Comput. Sci.*, 314(1-2):173–187, 2004.
4. A. Amir, A. Butman, and M. Lewenstein. Real scaled matching. *Inf. Process. Lett.*, 70(4):185–190, 1999.

5. A. Amir and E. Chencinski. Faster two dimensional scaled matching. In M. Lewenstein and G. Valiente, editors, *CPM*, volume 4009 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2006.
6. A. Amir, E. Chencinski, C. Iliopoulos, T. Kopelowitz, and H. Zhang. Property matching and weighted matching. In *CPM*, pages 01–15, 2006.
7. A. Amir, O. Kapah, and D. Tsur. Faster two dimensional pattern matching with rotations. In S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, editors, *CPM*, volume 3109 of *Lecture Notes in Computer Science*, pages 409–419. Springer, 2004.
8. M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Latin American Theoretical INformatics (LATIN)*, pages 88–94, 2000.
9. A. Butman, R. Eres, and G. M. Landau. Scaled and permuted string matching. *Inf. Process. Lett.*, 92(6):293–297, 2004.
10. R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In L. Babai, editor, *STOC*, pages 91–100. ACM, 2004.
11. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *STOC*, pages 592–601, 2002.
12. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002.
13. M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS*, pages 137–143, 1997.
14. M. Fischer and M. Paterson. String matching and other products. *in Complexity of Computation, R.M. Karp (editor), SIAM AMS Proceedings*, 7:113–125, 1974.
15. H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Symposium on the Theory of Computing (STOC)*, pages 135–143, 1984.
16. D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.
17. D. S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.
18. J. Jurka. Human repetitive elements. In R. A. Meyers, editor, *Molecular Biology and Biotechnology.*
19. J. Jurka. Origin and evolution of alu repetitive elements. In R. Maraia, editor, *The impact of short interspersed elements (SINEs) on the host genome.*
20. V. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
21. V. Mäkinen and G. Navarro. Position-restricted substring searching. In *LATIN*, pages 01–12, 2006.
22. E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.
23. M. S. Rahman, C. Iliopoulos, I. Lee, M. Mohamed, and W. Smyth. Finding patterns with variable length gaps or don't cares. In D. Chen and D. Lee, editors, *COCOON*, volume 4112 of *Lecture Notes in Computer Science*, pages 146–155. Springer, 2006.
24. K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.
25. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.