Symposium on Theoretical Aspects of Computer Science 2008 (Bordeaux), pp. 205-216
www.stacs-conf.org

IMPROVED ALGORITHMS FOR THE RANGE NEXT VALUE PROBLEM AND APPLICATIONS

MAXIME CROCHEMORE [2,1], COSTAS S. ILIOPOULOS [2], MARCIN KUBICA [3], M. SOHEL RAHMAN [2], AND TOMASZ WALEŃ [3]

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, France

[2] Algorithm Design Group, Department of Computer Science
Kings College London, Strand, London WC2R 2LS, England
E-mail address: Maxime.Crochemore@kcl.ac.uk,{csi,sohel}@dcs.kcl.ac.uk
URL: http://www.dcs.kcl.ac.uk/adg

[3] Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warszawa, Poland
E-mail address: {kubica,walen}@mimuw.edu.pl

Abstract. The Range Next Value problem (Problem RNV) is a recent interesting variant of the range search problems, where the query is for the immediate next (or equal) value of a given number within a given interval of an array. Problem RNV was introduced and studied very recently by Crochemore et. al [Finding Patterns In Given Intervals, MFCS 2007]. In this paper, we present improved algorithms for Problem RNV. We also show how this problem can be used to achieve optimal query time for a number of interesting variants of the classic pattern matching problems.

1. Introduction

We study the Range Next Value (RNV) problem, which is defined as follows:

Problem 1.1. Range Next Value (Problem RNV). We are given an array $A[1..n]$, which is a permutation of $[1..n]$. We need to preprocess A to answer queries of the following form:
Query: Given an integer $\mathcal{K} \in [1..n]$, and an interval $[\ell..r], 1 \leq \ell \leq r \leq n$, the goal is to return the value $A[k]$ of the immediate higher or equal number ('next value') than $\mathcal{K}$

Key words and phrases: Algorithms, Data structures.

Part of this research work was done when the authors were visiting McMaster University, Canada to attend StringMasters @ McMaster (2007). C.S. Iliopoulos is partially supported by the EPSRC and Royal Society grants. M. Kubica and T. Waleń are supported by the grant of the Polish Ministry of Science and Higher Education N206 004 32/0806. M.S. Rahman is supported by the Commonwealth Scholarship Commission in the UK and is on leave from the Department of CSE, BUET, Dhaka-1000, Bangladesh.

from $A[\ell..r]$ if there exists one. More formally, we need to return such $A[k]$, that $A[k] = \min\{A[q] \mid A[q] \geq \mathcal{K} \text{ and } \ell \leq q \leq r\}$. If there is no such k, then we return -1.

We use $RNV_A([\ell..r], \mathcal{K})$ to denote the range next value query on array $A[\ell..r]$ for the value $\mathcal{K}$. Problem RNV was introduced, very recently, in [4], to solve an interesting variant of the classic pattern matching problem, namely Pattern Matching in a Query Interval (Problem PMQI) [8]. In problem PMQI, we are given a text, which we can preprocess for subsequent queries and each query has a query interval in addition to a pattern to search for. The goal is to find only those occurrences of the pattern in the text that start in the given query interval. This problem is interesting, because, in many text search situations, one may want to search only in a part of the text, e.g. restricting the search to only parts of a long DNA sequence. To achieve an optimal query time, in [4], Problem PMQI was reduced to Problem RNV and the latter was solved with a constant query time against a data structure requiring $O(n^2)$ preprocessing time and space. It was left as an open problem to devise a better data structure without losing the constant time query capability. The goal of this paper is to present such a data structure. Notably, Problem RNV turns out to be useful in a number of other problems as well. As we will show in Section 5, Problem RNV can be used to get optimal query times for a number interesting problems studied in [7] and related to *string statistics* problem [3, 1].

It is worth-mentioning here that, despite extensive results on various range searching problems, we are not aware of any result from the literature that directly addresses Problem RNV. It seems to be possible to get a query time of $O(\log \log n)$ by using an efficient data structure for the much studied "3-sided Query" problem along with a 'persistent' data structure to 'select' the appropriate answer from the answer set of a "3-sided Query" [9]. However, our goal is to facilitate constant time query capability with a data structure requiring $o(n^2)$ time and space. In the rest of this paper, we follow the following convention adopted from [2]: if an algorithm has preprocessing time $f(n)$ and query time $g(n)$, we will say that the algorithm has complexity $\langle f(n), g(n)\rangle$.

The rest of the paper is organized as follows. In Section 2, we review the $\langle O(n^2), O(1)\rangle$ algorithm presented in [4]. In Sections 3 and 4, we present two different algorithms to solve Problem RNV with complexity $\langle O(n^{1.5}), O(1)\rangle$ and $\langle O(n^{1+\epsilon}), O(1)\rangle$ respectively. In Section 5, we discuss their possible applications.

2. The $\langle O(n^2), O(1)\rangle$ Algorithm

In this section, we briefly review the algorithm for Problem RNV (referred to as *Algorithm CIR* henceforth) presented in [4]. First, we formally define the much studied Range Minimum Query Problem, which is used by the CIR algorithm.

Problem 2.1. Range Minimum Query (Problem RMQ). We are given an array $A[1..n]$ of numbers. We need to preprocess A to answer the following form of queries:
Query: Given an interval $[\ell..r], 1 \leq \ell \leq r \leq n$, the goal is to find the minimum (maximum, in the case of Range Maximum Query) value $A[k]$ for $\ell \leq k \leq r$.

We use $RMQ_A([\ell..r])$ to denote the range minimum query on array A for the interval $[\ell..r]$. Problem RMQ has received much attention in the literature and Bender and Farach-Colton presented an algorithm with complexity $\langle O(n), O(1)\rangle$, using $O(n \log n)$-bits

of space [2][1]. Recently, Sadakane [10] presented a succinct data structure, which achieves the same time complexity using $O(n)$ bits of space. Very recently, Fischer and Heun [5] presented an algorithm with the same time complexity requiring optimal $2n + o(n)$ bits of additional space.

2.1. Algorithm CIR

Algorithm CIR maintains n arrays $B_i, 1 \leq i \leq n$. Each array B_i has n elements. So, B could be viewed as a two dimensional array. Algorithm CIR fills each array B_i depending on A as follows. For each $1 \leq i \leq n$ it stores 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 $1 \leq j \leq n$, algorithm CIR sets $B_i[j] = A[j] - i$, if $A[j] \geq i$; otherwise it sets $B_i[j] = \infty$. Then, each $B_i, 1 \leq i \leq n$, is preprocessed for the RMQ problem. This completes the construction of the data structure. It is clear that, Algorithm CIR requires $O(n^2)$ preprocessing time. The query processing is as follows. Consider the query $RNV_A([\ell..r], \mathcal{K})$. Then, we simply need to apply range minimum query in $B_{\mathcal{K}}$ for the interval $[\ell..r]$, i.e., we need to execute the query: $RMQ_{B_{\mathcal{K}}}([\ell..r])$. This gives us the following theorem.

Theorem 2.2. [4]. *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.*

3. An Improved Algorithm with Complexity $\langle O(n^{1.5}), O(1)\rangle$

In this section, we present an algorithm that improves on Algorithm CIR. In what follows, we use the following notations. Given an array $A[1..n]$, we denote by $\widehat{A}$, the underlying set comprising of all the (distinct) elements of A. In other words, $\widehat{A} = \{A[i] \mid 1 \leq i \leq n\}$. We define $\min(A) = A[i]$, such that $A[i] \leq A[j]$ for all j in $[1..n]$. Given a sub-array $A[\ell..r], 1 \leq \ell \leq r \leq n$, of the array A, we further define $left(A[\ell..r]) = \ell$ and $right(A[\ell..r]) = r$. We say that, a range $[\ell..r]$ is *nonexistent*, if $\ell > r$; otherwise, $[\ell..r]$ is said to be *existent*. Furthermore, given a range $[\ell..r], 1 \leq \ell \leq r \leq n$, and a sub-array $A[i..j], 1 \leq i \leq j \leq n$ of an array $A[1..n]$, we say that the range $[\ell..r]$ is *confined* in the sub-array $A[i..j]$, if, and only if, we have $i \leq \ell \leq r \leq j$. Now, recall that, our goal is to construct a data structure requiring $o(n^2)$ time and space without losing the constant time query capability. Below we present the idea we employ.

In this section, we will assume that, we are looking for the immediate higher value (instead of 'equal or higher') than the given value $\mathcal{K}$ in Problem RNV. It is easy to realize that, it doesn't really create any problem for the actual case.

In the first phase, we divide the array $A[1..n]$ into $\lceil n/\wp \rceil = q$ sub-arrays $D_j, 1 \leq j \leq q$. Now, we add the number 0 to the beginning of each $D_j, 1 \leq j \leq q$. It is easy to realize that, each D_j has exactly $\wp + 1$ elements except possibly the last one, which may have less. Now, we apply a slight variation of Algorithm CIR on each $D_j, 1 \leq j \leq q$ as follows. For each D_j, we maintain $|D_j|$ arrays, $B_j^{\ell}[1..|D_j|], \ell \in D_j$. Notably, the naming convention followed

[1]The same result was achieved in [6], albeit with a more complex data structure.

for the B_j^ℓ arrays are for better exposition. For example, if $D_j = \langle 0, 1, 9, 2, 6\rangle$, then we have $B_j^0, B_j^1, B_j^9, B_j^2$ and B_j^6. Now, we fill each such $B_j^\ell[1..|D_j|]$ as follows:

$$B_j^\ell[i] = \begin{cases} D_j[i] & \text{If } D_j[i] > \ell \\ \infty & \text{Otherwise} \end{cases} \tag{3.1}$$

In the second phase, we construct q arrays $E_i[0..n], 1 \le i \le q$. E_i is filled up as follows:

$$E_i[j] = \begin{cases} j & \text{If } j \in D_i \\ E_i[j-1] & \text{Otherwise}^a \end{cases} \tag{3.2}$$

[a]Recall that $0 \in D_i$ for all $1 \le i \le q$.

In the third phase, we construct n arrays $F_k[1..q], 1 \le k \le n$, where we fill:

$$F_k[i] = \min\{D_i[j] : D_i[j] > k \text{ and } 1 \le j \le |D_i|\} = RNV_{D_i}([1..|D_i|], k)$$

Please note, that all the F_k arrays can be computed in $O(nq)$ time. Finally, we preprocess each F_k $(1 \le k \le n)$ and all B_j^ℓ arrays for the RMQ problem. This completes the construction of our data structure. In what follows, we use RNV_DS1 to refer to this data structure.

Algorithm 1 Function $RNV\_Query(A[\ell..r], \mathcal{K}))$

```
1: let ℓ' = (i1 − 1) · ℘ < ℓ ≤ i1 · ℘
2: let r' = (i2 − 1) · ℘ < r ≤ i2 · ℘
3: if i1 = i2 then
4:     {ℓ and r are in the same block}
5:     Set j = i1, u = E_{i1}[K]
6:     return RMQ_{B_j^u}([(ℓ − ℓ')..(r − r')])
7: else
8:     Set val1 = val2 = val3 = ∞.
9:     Set u1 = E_{i1}[K], u2 = E_{i2}[K].
10:    Set val1 = RMQ_{B_{i1}^{u1}}([(ℓ − ℓ')..|Du|]){Executing RNV_{D_{i1}}([(ℓ − ℓ')..|D_{i1}|], K}
11:    Set val3 = RMQ_{B_{i2}^{u2}}([1..(r − r')]){Executing RNV_{D_{i2}}([1..(r − r')], K}
12:    if i2 − i1 > 1 then
13:        Set val2 = RMQ_{F_K}([(i1 + 1)..(i2 − 1)])
14:    end if
15:    return min{val1, val2, val3}
16: end if
```

3.1. Query Processing

In this section, we discuss the query processing. Suppose, we are considering the following query: $RNV_A([\ell..r], \mathcal{K})$. We compute, ℓ', r', i_1 and i_2, such that, $\ell' = (i_1-1)\cdot\wp < \ell \le i_1\cdot\wp$ and $r' = (i_2 - 1) \cdot \wp < r \le i_2 \cdot \wp$. Then, we can divide the range $[\ell..r]$ into 3 consecutive ranges, namely $[\ell..i_1 \times \wp], [i_1 \times \wp + 1..(i_2 - 1) \times \wp]$ and $[(i_2 - 1) \times \wp + 1..r]$ (See Figure 1). Now, we proceed with the query processing as follows. We have the following cases.

Case 1: $i_1 = i_2$**:** In this case, the range $[\ell..r]$ is in the same sub-array D_{i_1}. So, we only perform the following RMQ query, the answer of which is returned as the desired result: $RNV_{D_{i_1}}([(\ell - \ell')..(r - r')], \mathcal{K}) = RMQ_{B_{i_1}^{E_{i_1}[\mathcal{K}]}}([(\ell - \ell')..(r - r')])$.

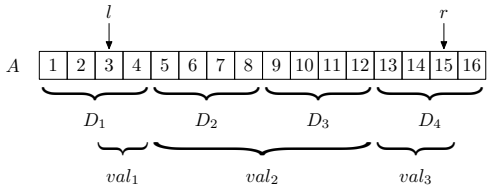

Figure 1: The situation of an RNV query

Case 2: $i_2 > i_1$**:** In this case, we first initialize val_1, val_2 and val_3 to ∞ and then we proceed with the query processing as follows. We first perform the following RMQ queries:

$$val_1 = RNV_{D_{i_1}}([(\ell - \ell')..|D_{i_1}|], \mathcal{K}) = RMQ_{B_{i_1}^{E_{i_1}[\mathcal{K}]}}([(\ell - \ell')..|D_{i_1}|]) \tag{3.3}$$

$$val_3 = RNV_{D_{i_2}}([1..(r - r')], \mathcal{K}) = RMQ_{B_{i_1}^{E_{i_2}[\mathcal{K}]}}([1..(r - r')]) \tag{3.4}$$

Then, if we have $i_2 - i_1 > 1$, then we perform the following RMQ query:

$$val_2 = RMQ_{F_{\mathcal{K}}}([(i_1 + 1)..(i_2 - 1)]) \tag{3.5}$$

Finally, we return the minimum of val_1, val_3 and val_2 as the final result.

3.2. Correctness and Running Time

In this section, we discuss the correctness of the above algorithm and its running time.

Lemma 3.1. *With the data structure RNV_DS1, we can correctly answer any queries of the form* $RNV_{D_i}([\ell..r], \mathcal{K})$, $1 \leq \ell \leq r \leq |D_i|$, $1 \leq \mathcal{K} \leq n$, $1 \leq i \leq q$.

Proof. Recall that, $RNV_{D_i}([\ell..r], \mathcal{K})$ is executed by calculating $RNV_{D_i}([\ell..r], E_i[\mathcal{K}])$, which in turn, is executed by performing the query $RMQ_{B_i^{E_i[\mathcal{K}]}}([\ell..r])$. From the correctness of Algorithm CIR, it is clear that, $RNV_{D_i}([\ell..r], E_i[\mathcal{K}]) \equiv RMQ_{B_i^{E_i[\mathcal{K}]}}([\ell..r])$. So it remains to show that $RNV_{D_i}([\ell..r], \mathcal{K}) \equiv RNV_{D_i}([\ell..r], E_i[\mathcal{K}])$. This is shown as follows. Recall that, by definition, if $\mathcal{K} \in D_i$ then $E_i[\mathcal{K}] = \mathcal{K}$. So, if $\mathcal{K} \in D_i$, we are done. Therefore, assume otherwise. Now, in this case, $E_i[\mathcal{K}]$ is the nearest smaller value of $\mathcal{K}$ in D_i. For the values $v < \min(\widehat{D_i} \setminus 0)$, this is ensured by the addition of 0 in each D_i. Therefore, it is easy to realize that, $RNV_{D_i}([\ell..r], E_i[\mathcal{K}])$ would return the same value as returned by $RNV_{D_i}([\ell..r], \mathcal{K})$ and hence, the lemma follows. ∎

Theorem 3.2 (Correctness)**.** *With the data structure RNV_DS1, we can correctly answer any query of the form* $RNV_A([\ell..r], \mathcal{K}), 1 \leq \ell \leq r \leq n, 1 \leq \mathcal{K} \leq n$.

Proof. Recall that, the range $[\ell..r]$ is transformed into (up to) 3 consecutive ranges, namely $r_1 \equiv [\ell..i_1 \times \wp], r_2 \equiv [i_1 \times \wp + 1..(i_2 - 1) \times \wp]$ and $r_3 \equiv [(i_2 - 1) \times \wp + 1..r]$. Now, the range r_1 (resp. r_3) is confined within the sub-array D_{i_1} (resp. D_{i_2}). On the other hand, if r_2 is existent, then it can span over one or more sub-arrays, $D_{i_1+1}, \ldots, D_{i_2-1}$ and, in that case, it completely contains those sub-arrays, i.e. $i_1 \times \wp + 1 = left(D_{i_1+1})$ and $(i_2 - 1) \times \wp = right(D_{i_2-1})$. It is clear that, the minimum of the results of the corresponding RNV queries in the three ranges, namely, r_1, r_2 and r_3, is the final result. Now, recall that, we have the following two cases.

case 1: $i_1 = i_2$**:** It is easy to verify that, this case arises when the range $[\ell..r]$ is confined in the sub-array D_{i_1}. Therefore, it is easy to verify that, we have $RNV_A([\ell..r], \mathcal{K}) \equiv RNV_{D_{i_1}}([(\ell - \ell')..(r - r')], \mathcal{K})$, and by Lemma 3.1, we get the correct result.

case 2: $i_2 > i_1$**:** It is easy to see that, if we have $i_2 - i_1 > 1$, then all 3 intervals are existent; otherwise, r_2 is non-existent. Now, recall that, we initialize val_1, val_3 and val_2 to ∞. Since, both the ranges r_1 and r_3 are confined in the sub-arrays D_{i_1} and D_{i_2} respectively, by Lemma 3.1, the two corresponding queries, namely, Queries 3.3 and 3.4 are correctly executed and the results are stored in val_1 and val_3.

Now, assume that the range r_3 is existent and that, $v_i = RNV_{D_i}([1..|D_i|], \mathcal{K}), i \in [i_1 + 1..i_2 - 1]$. Then, it is easy to verify that:

$$RNV_A([\ell..r], \mathcal{K}) = \min(val_1, val_3, \min_{i \in [i_1+1..i_2-1]}(v_i)).$$

Now, we return $\min(val_1, val_3, val_2)$ as the answer. Hence, it suffices to show that $val_2 = \min_{i \in [i_1+1..i_2-1]}(v_i)$. Recall that, val_2 is evaluated according to Equation 3.5. By definition, each entry of $F_{\mathcal{K}}$ correctly (Lemma 3.1) stores the result of the RNV query for the value $\mathcal{K}$ and for the whole range for the corresponding sub-array. Therefore, the range minimum query does provide us with the desired value.

Finally, if r_2 is nonexistent, then val_2 remains assigned to ∞. Therefore, the result returned, i.e., the minimum of val_1, val_3 and val_2, is correct.

■

Theorem 3.3. *The data structure RNV_DS1 can be constructed in $O(n\wp + n^2/\wp)$ time.*

Proof. We deduce the construction time of RNV_DS1 phase by phase as follows.

Phase 1: Each sub-array $D_j, 1 \le j \le q = \lceil n/\wp \rceil$ has at most $\wp+1$ elements. It is easy to see that, the application of the (slight variation of) Algorithm CIR requires $O(\wp^2)$ time per sub array. Therefore, in total, time required by Phase 1 is $O(\wp^2) \times q = O(\wp^2) \times \lceil n/\wp \rceil = O(n\wp)$ in the worst case.

Phase 2: Initializing and filling up the arrays $E_i[0..n], 1 \le i \le q$ requires $O(n) \times q = O(n^2/\wp)$ time.

Phase 3: In this phase, we construct the arrays $F_k[1..q]$, for $1 \le k \le n$. This can easily be done in $O(nq) = O(n^2/\wp)$ time. We also preprocess arrays F_k and B_j^l for the RMQ queries, what requires also $O(nq) = O(n^2/\wp)$ time.

Therefore, in total, the time required for the construction of RNV_DS1 is $O(n\wp)+O(n^2/\wp)+O(n^2/\wp) = O(n\wp + n^2/\wp)$. ■

Corollary 3.4. *The data structure RNV_DS1 can be constructed in $O(n^{1.5})$ time.*

Proof. This can be achieved if we assume that $\wp = \sqrt{n}$. ■

Theorem 3.5. *Given the data structure RNV_DS1, we can answer the RNV queries in $O(1)$ time per query.*

Proof. It is clear that, given RNV_DS1, an RNV query is answered by executing up to 2 RNV queries on the sub-arrays and possibly 1 RMQ queries on the appropriate F array. Each of these queries requires $O(1)$ time. Therefore, the theorem follows. ■

4. An Improved Algorithm with Complexity $\langle O(n^{1+\epsilon}), O(1)\rangle$

In this section, we present a different algorithm for problem RNV by taking a slightly different approach. We start with a slightly different $\langle O(n^2), O(1)\rangle$ algorithm and present a new algorithm built on top it. This algorithm follows a similar strategy as algorithm CIR and is referred to as the *base algorithm* henceforth.

4.1. The Base Algorithm:

We define arrays $B_j, 1 \le j \le n$ as follows:

$$B_j[i] = \begin{cases} A[i] & \text{if } A[i] \ge j \\ \infty & \text{if } A[i] < j \end{cases}$$

Now, the preprocessing is done as follows.

1: **for** $j = 1, \ldots, n$ **do**
2: Preprocess sequence B_j for Problem RMQ
3: **end for**

After the above data structure is constructed, we can perform the queries as follows. Similar to what was done in algorithm CIR, given the query $RNV_A([\ell..r], \mathcal{K})$, we just return $RMQ_{B_\mathcal{K}}([l..r])$. It is easy to see that the base algorithm is correct and its running time is $\langle O(n^2), O(1)\rangle$. In the rest of this section, we present an improved algorithm based on the base algorithm.

4.2. Improved algorithm

In this section we describe a method for improving preprocessing time of any RNV algorithm, the cost paid for the improvement is slight (namely $O(1)$) increase of the RNV query time. Suppose, we are given the array A of length n, the parameter $\wp$, and an algorithm RNVALG for Problem RNV with complexity $\langle f(n), g(n)\rangle$. We will show how to improve the preprocessing time of RNVALG.

In the first phase we divide possible values of parameter $\mathcal{K}$, into $\lceil n/\wp \rceil = q$ interval sets K_j, where $K_j = \{i : (j-1)\cdot\wp < i \le j\cdot\wp\}$. For each j $(1 \le j \le q)$ we compute following arrays:

- array B'_j $(|B'_j| = n)$ — containing information about elements of array A strictly larger than $(j-1)\cdot\wp$

$$B'_j[i] = \begin{cases} A[i] & \text{if } A[i] > (j-1)\cdot\wp \\ \infty & \text{otherwise} \end{cases}$$

- set $\mathcal{C}_j = \{i : A[i] \in K_j\}$ — containing indices of the elements of array A with values from the range K_j; by C_j we will denote the array consisting of elements of $\mathcal{C}_j$ sorted in the ascending order, $|C_j| \le \wp$,
- array D_j $(|D_j| \le \wp)$ — contains the elements of array A from the range K_j, in the order as they appear in A; each element is decreased by $(j-1)\cdot\wp$, to ensure that the array D_j is a permutation of $\{1..|D_j|\}$:

$$D_j[i] = A[C_j[i]] - (j-1)\cdot\wp, \quad \text{for } 1 \le i \le |C_j|$$

- array E_j ($|E_j| = n+1$) — containing indices used for translating queries from array A_j to array D_j; $E_j[i]$ denotes the number of elements from $A[1..i]$ from the range K_j:

$$E_j[i] = \begin{cases} E_j[i-1]+1 & \text{if } A[i] \in K_j \text{ and } i > 0 \\ E_j[i-1] & \text{if } A[i] \notin K_j \text{ and } i > 0 \\ 0 & \text{if } i = 0 \end{cases}$$

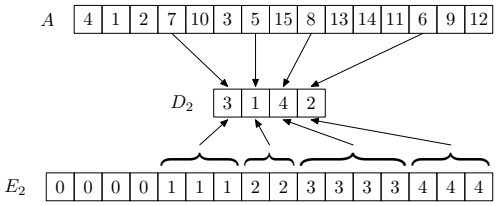

Figure 2: Example of computing arrays D_j and E_j, for $n = 15$, $\wp = 4$, $j = 2$, $\mathcal{K}_j = [5..8]$

Then the algorithm preprocesses each array B'_j for range minimum queries, and each array D_j for range next value queries (using RNVALG).

Algorithm 2 Construction of RNV_DS2($\wp$,RNVALG)

```
1: for j = 1, ..., ⌈n/℘⌉ do
2:     compute arrays B'_j, C_j, D_j, E_j,
3:     preprocess sequence B'_j for RMQ (Range Minimum Queries)
4:     preprocess sequence D_j for RNV (Range Next Value Queries) using RNVALG
5: end for
```

The B'_j arrays will be used for answering the range next value queries if the answer is outside of the range K_j. The D_j will be used if the answer is within the range K_j. Since we do not know in the advance which case is valid, the algorithm tries both cases, and then chooses the smaller result.

Algorithm 3 Query Processing of RNV_DS2($\wp$,RNVALG)

```
1: Set a_1 = a_2 = ∞
2: Set j, such that: x = (j − 1) · ℘ < 𝒦 ≤ j · ℘
3: if j < q then
4:     a_1 = RMQ_{B'_{j+1}}([ℓ..r])
5: end if
6: Set ℓ' = E_j[ℓ − 1] + 1; r' = E_j[r]
7: if ℓ' ≤ r' then
8:     a_2 = RNV_{D_j}([ℓ'..r']], 𝒦 − x) + x {using algorithm RNVALG}
9: end if
10: return min(a_1, a_2)
```

Theorem 4.1. *If we are given the $\langle f(n), g(n)\rangle$ RNV algorithm, then using the RNV_DS2, we can construct $\langle O((n^2 + nf(\wp))/\wp),\ g(\wp) + O(1)\rangle$ algorithm for RNV.*

Proof. The preprocessing of the RNV_DS2 requires:

- computing $n/\wp$ arrays B'_j (each of length n), this step requires $O(n^2/\wp)$ time,

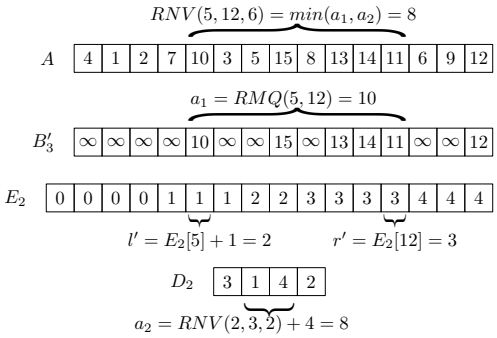

Figure 3: RNV_DS2 processing query $RNV(5, 12, 6)$ (assuming $\wp = 4$)

- preprocessing $n/\wp$ arrays B'_j for the Range Minimum Queries, this step also requires $O(n^2/\wp)$ time,
- computing $n/\wp$ arrays D_j (each of length $\wp$), clearly this step requires $O(n)$ time,
- preprocessing $n/\wp$ arrays D_j for the Range Next Value Queries using $\langle f(n), g(n)\rangle$ algorithm, this step requires $O(f(\wp) \cdot n/\wp)$ time,
- computing $n/\wp$ arrays E_j (each of length $n + 1$), this step requires $O(n^2/\wp)$ time.

The total preprocessing time is $O((n^2 + nf(\wp))/\wp)$.

Answering the Range Next Value queries requires:

- one range minimum query on the B'_j array, what can be done in $O(1)$ time,
- one recursive call of the range next value query for D_j array using $\langle f(n), g(n)\rangle$ RNV algorithm, requiring $g(\wp)$ time,
- constant number of additional operations (i.e. accessing arrays E_j)

Clearly the total query time is $g(\wp) + O(1)$. ∎

Corollary 4.2. *RNV_DS2 can be constructed in $O(n^{1.5})$ running time and space.*

Proof. This can be achieved if we use the RNV_DS2 construction method, with $\wp = \sqrt{n}$, and using as RNVALG, the base algorithm (with complexity $\langle O(n^2), O(1)\rangle$). ∎

We can obtain even more efficient algorithm, carefully iterating RNV_DS2 construction.

Theorem 4.3. *For any given positive constant $\epsilon > 0$, we can construct $\langle O(n^{1+\epsilon},\ O(1)\rangle$ algorithm form RNV using the RNV_DS2.*

Proof. Let RNV_DS2(0) denote the base algorithm for RNV (with the complexity $\langle O(n^2), O(1)\rangle$). For any $i > 0$, let RNV_DS2(i) denote the algorithm obtained using RNV_DS2 with RNVALG = RNV_DS2($i-1$) and $\wp = n^{\frac{i}{i+1}}$. From the theorem 4.1 the RNV_DS2(1) has the complexity $\langle O(n^{1.5}), O(1)\rangle$, the RNV_DS2(2) has the complexity $\langle O(n^{1+\frac{1}{3}}), O(1)\rangle$. By simple induction, one can easily prove, that the RNV_DS2(i) has the complexity $\langle O(n^{1+\frac{1}{i}}), O(i)\rangle$. ∎

5. Applications

In this section, we discuss possible applications of Problem RNV. As has already been mentioned in Section 1, the study of the RNV problem in [4] was motivated by Problem

PMQI, a variant of the classic pattern matching problem. Problem PMQI is formally defined as follows (We use $Occ_{\mathcal{T}}^{\mathcal{P}}$ to denote the occurrence set for the classic pattern matching problem):

Problem 5.1. Pattern Matching in a Query Interval (Problem PMQI). Suppose we are given a text $\mathcal{T}$ of length n. Preprocess $\mathcal{T}$ to answer queries of the following form.
Query: We are given a pattern $\mathcal{P}$ of length m and a query interval $[\ell..r]$, with $1 \leq \ell \leq r \leq n$. Let us denote by $Occ_{\mathcal{T}}^{\mathcal{P}}$ the set of all occurrences of $\mathcal{P}$ in $\mathcal{T}$. We are to construct the set:

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

Using the reduction of [4] from Problem PMQI to Problem RNV, we obtain the following theorem.

Theorem 5.2. *We can construct a data structure for Problem PMQI in* $O(\max(n^{1+\epsilon}, n \log \sigma))$ *time and* $O(n^{1+\epsilon})$ *space, and we can answer the relevant queries in the optimal* $O(m + |Occ_{\mathcal{T}[\ell..r]}^{\mathcal{P}}|)$ *time per query.*

A more general problem called PMI was also handled in [4].

Problem 5.3. Generalized Pattern Matching with Intervals (Problem PMI). 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 queries of the following form.
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

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

To solve Problem PMI, a data structure with $O(n \log^3 n)$ time, $O(n \log^2 n)$ space was constructed in [4]; the query time achieved was $O(m + \log \log n + |Occ_{\mathcal{T}[\ell..r],\pi}^{\mathcal{P}}|)$. It was left as an open problem to achieve the optimal query time for Problem PMI [4]. Interestingly, using Problem RNV, we can get the optimal query time for Problem PMI as well. The details are left for the journal version; but we report the new result in the following theorem.

Theorem 5.4. *For problem PMI, we can construct a data structure in* $O(\max(n^{1+\epsilon}, n \log \sigma))$ *time and* $O(n^{1+\epsilon})$ *space, and we can answer the relevant queries in the optimal* $O(m + |Occ_{\mathcal{T}[\ell..r],\pi}^{\mathcal{P}}|)$ *time per query.*

In the rest of this section, we consider three recent variants of the classic pattern matching problem, which we define below after defining some related concepts. Given two occurrences $i, j \in [1..n - m + 1], j > i$ of a pattern $\mathcal{P}[1..m]$ in a text $\mathcal{T}[1..n]$, we say that j is *minimal* with respect to i, if, and only if, there exists no occurrence of $\mathcal{P}$ in $\mathcal{T}$ in the range $[i + 1..j - 1]$. And, two occurrences $i, j \in [1..n - m + 1]$ of $\mathcal{P}$ in $\mathcal{T}$ are said to be *non-overlapping*, if, and only if, $|j - i| \geq m$. Otherwise, they are said to be overlapping.

Problem 5.5. Suppose we are given a text $\mathcal{T}$ of length n. Preprocess $\mathcal{T}$ to answer the following form of queries:
Query: Given a pattern $\mathcal{P}$ of length m, and an index i, we want to find out an occurrence $i' \geq i$ of $\mathcal{P}$ in $\mathcal{T}$, such that i' is minimal with respect to i.

Problem 5.6. Suppose we are given a text $\mathcal{T}$ of length n. Preprocess $\mathcal{T}$ to answer the following form of queries:

Query: Given a pattern $\mathcal{P}$ of length m, and a list of indices $\mathcal{U} = \langle i_1, \ldots, i_\ell \rangle$, our goal is to construct the list $\mathcal{V} = \langle j_1, \ldots, j_\ell \rangle$, such that, for all $k \in [1..\ell]$, j_k is an occurrence of $\mathcal{P}$ in $\mathcal{T}$ and $j_k \in \mathcal{V}$ is, either minimal with respect to $i_k \in \mathcal{U}$ or equal to $Nill$. The latter case means that there doesn't exist any occurrence to the right of i_k.

Problem 5.7. Suppose we are given a text $\mathcal{T}$ of length n. Preprocess $\mathcal{T}$ to answer the following form of queries:
Query: Given a pattern $\mathcal{P}$ of length m, and an interval $[i..j]$, we want to find an ascending sequence $\mathcal{U} = \langle i_1, \ldots, i_\ell \rangle$ of non-overlapping occurrences of $\mathcal{P}$ in $\mathcal{T}$, such that $i \leq i_1 \leq i_\ell \leq j$ and ℓ is maximal.

Problems 5.5 to 5.7 were handled very recently in [7]. The corresponding data structures presented in [7] for the above problems requires $O(n \log n)$ storage and $O(n \log n \log \log n)$ expected preprocessing time each. The query time achieved in [7], for Problem 5.6 and 5.7 is $O(m + \ell \log \log n)$ and for Problem 5.5 is $O(m + \log \log n)$. Notably, none of the query times achieved in [7] are optimal. In the rest of this section, we briefly show, how Problem RNV can be used to achieve optimal query times for the above problems. We remark however that, we omit many of the details for space constraint and left them for the journal version.

5.1. Problems 5.5 and 5.6

It is clear that, Problem 5.5 is a simpler version of the Problem 5.6. Interestingly, we can use Problem RNV to solve both the problems efficiently. We first consider Problem 5.5. Following the techniques of [4], we construct a suffix tree and do some preprocessing on it to get $Occ_T^{\mathcal{P}}$ implicitly in the form of an array $\mathcal{L}$ and an interval $[a..b]$. More specifically, using the techniques of [4], after the preprocessing, we can implicitly have $Occ_T^{\mathcal{P}}$ in $\mathcal{L}[a..b]$ in $O(m)$ time. Now, it is easy to see that to solve the query of problem 5.5, we simply need to get the answer of the following query:

$$RNV_{\mathcal{L}}([a..b], i) \tag{5.1}$$

Therefore, we have the following result.

Theorem 5.8. *For Problem 5.5, we can construct a data structure in $O(\max(n^{1+\epsilon}, n \log \sigma))$ time and $O(n^{1+\epsilon})$ space, and we can answer the relevant queries in the optimal $O(m)$ time per query.*

Proof. For the preprocessing, we first construct the suffix tree and do the preprocessing of [4], requiring $O(n \log \sigma)$ time, where $\sigma = \min(n, |\Sigma|)$. Then we preprocess $\mathcal{L}$ for Problem RNV. Total construction time and space complexity is, $O(\max(n^{1+\epsilon}, n \log \sigma))$ and $O(n^{1+\epsilon})$ respectively. As for the query, we require $O(m)$ time to get $Occ_T^{\mathcal{P}}$ implicitly [4]. Then, we just need to perform the Query 5.1 requiring constant time. Hence, the result follows. ∎

We can easily extend the above result for Problem 5.6, simply by executing RNV queries, $RNV_{\mathcal{L}}([a..b], i)$ for all $i \in \mathcal{U}$. Therefore, we get the following theorem.

Theorem 5.9. *For Problem 5.6, we can construct a data structure in $O(\max(n^{1+\epsilon}, n \log \sigma))$ time and $O(n^{1+\epsilon})$ space, and we can answer the relevant queries in the optimal $O(m + \ell)$ time per query.*

5.2. Problem 5.7

To solve Problem 5.7, we follow the greedy strategy of [7] as follows. Suppose, we have the set $Occ_{\mathcal{T}}^{\mathcal{P}}$ in the list $\mathcal{W} = \langle i_1, \ldots, i_{|Occ_{\mathcal{T}}^{\mathcal{P}}|} \rangle$ in ascending order. Now, we construct another list $\mathcal{Y}$ as follows. We first put i_1 in $\mathcal{Y}$. We use $last(\mathcal{Y})$ to denote the most recently put index in $\mathcal{Y}$. Now we scan the list $\mathcal{W}$ from left to right and put $i_k \in \mathcal{W}$ in $\mathcal{Y}$, only if i_k and $last(\mathcal{Y})$ are non-overlapping. It was proved in [7] that, $|\mathcal{Y}|$ is maximal. Therefore, we have the following theorem.

Theorem 5.10. *For Problem 5.7, we can construct a data structure in* $O(\max(n^{1+\epsilon}, n \log \sigma))$ *time and* $O(n^{1+\epsilon})$ *space, and we can answer the relevant queries in the optimal* $O(m + \ell)$ *time per query.*

Proof. We do the same preprocessing as we did for Problems 5.5 and 5.6 and hence achieve the same preprocessing time and space complexity. Now, we consider the query. We start with the query $RNV_{\mathcal{L}}([a..b], i + 1)$. Now suppose, the query returns q. Now, if $q \leq j$, then we put q in $\mathcal{U}$ and perform the query $RNV_{\mathcal{L}}([a..b], q + m)$ and continue as before. We stop when we get a query result q' such that $q' > j$. It is easy to verify that this would correctly construct a maximal list $\mathcal{U}$. Finally, since each of the queries require constant time, the result follows. ∎

References

[1] A. Apostolico and F. P. Preparata. Data structures and algorithms for the string statistics problem. *Algorithmica*, 15(5):481–494, 1996.

[2] M. A. Bender and M. Farach-Colton. The lca problem revisited. In G. H. Gonnet, D. Panario, and A. Viola, editors, *Latin American Theoretical INformatics (LATIN)*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[3] G. S. Brodal, R. B. Lyngsø, A. Östlin, and C. N. S. Pedersen. Solving the string statistics problem in time $O(n \log n)$. In P. Widmayer, F. T. Ruiz, R. M. Bueno, M. Hennessy, S. Eidenbenz, and R. Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 728–739. Springer, 2002.

[4] M. Crochemore, C. S. Iliopoulos, and M. S. Rahman. Finding patterns in given intervals. In A. Kucera and L. Kucera, editors, *MFCS*, volume 4708 of *Lecture Notes in Computer Science*, pages 645–656. Springer, 2007.

[5] J. Fischer and V. Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In B. Chen, M. Paterson, and G. Zhang, editors, *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007.

[6] 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.

[7] O. Keller, T. Kopelowitz, and M. Lewenstein. Range non-overlapping indexing and successive list indexing. In F. K. H. A. Dehne, J.-R. Sack, and N. Zeh, editors, *WADS*, volume 4619 of *Lecture Notes in Computer Science*, pages 625–636. Springer, 2007.

[8] V. Mäkinen and G. Navarro. Position-restricted substring searching. In J. R. Correa, A. Hevia, and M. A. Kiwi, editors, *LATIN*, volume 3887 of *Lecture Notes in Computer Science*, pages 703–714. Springer, 2006.

[9] E. Porat. Private communication.

[10] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.