Efficient Dynamic Range Minimum Query

A Heliou, M Léonard, L Mouchard, Mikael Salson

Efficient Dynamic Range Minimum Query

A. Heliou[b,a], M. Léonard[c], L. Mouchard[c,d], M. Salson[e,*]

[a] *Inria Saclay-Île de France, AMIB, Bâtiment Alan Turing, Palaiseau, France*
[b] *Laboratoire d'Informatique de l'École Polytechnique (LIX), CNRS UMR 7161, Palaiseau, France*
[c] *Normandie University, University of Rouen, LITIS EA 4108, TIBS, Rouen, France*
[d] *Centre for Combinatorics on Words & Applications, School of Engineering & Information Technology, Murdoch University, Murdoch WA 6150, Australia*
[e] *CRIStAL (UMR 9189 University of Lille, CNRS), INRIA Lille-Nord Europe, France*

Abstract

The Range Minimum Query problem consists in answering efficiently to a simple question: "what is the minimal element appearing between two specified indices of a given array?". In this paper we present a novel approach that offers a satisfying trade-off between time and space. Moreover we show how the structure can be easily maintained whenever an insertion, modification or deletion modifies the array.

Keywords: Range Minimum Query, Dynamic Structure, Compressed Bit Vector, Longest Common Prefix

1. Introduction

The Range Minimum Query (RMQ) problem consists in finding, for a given array A of comparable elements and two values $i < j$, the index m such that $A[m] = \min_{k \in [i,j]} A[k]$. The Lowest Common Ancestor (LCA) problem consists in finding, for a given rooted tree and two nodes, the common ancestor of the two nodes that is located the farthest from the root. In [1], Gabow et al. show that RMQ and LCA are two equivalent problems (e.g. RMQ can be solved using a LCA on A's Cartesian tree).

LCA algorithms can be used for example to solve the Approximate String Matching (ASM) problem: find the occurrences of all factors in a text T of length n that approximately match a pattern P given a distance D (e.g. Hamming or Levenshtein distances) and a maximal number of errors k. ASM is fundamental to many applications in Computer Science such as search engines, text processing and computational biology to name three of the most demanding domains.

In [2], Landau and Vishkin present a $O(kn)$ algorithm based on a diagonal visit of the dynamic programming matrix and a clever skip-algorithm that

*Corresponding author: Mikael.Salson@univ-lille1.fr

prevents from visiting every position on the diagonals. Other approaches build the Generalized Suffix Tree $GST(T, P)$ of T and P, so all suffixes of T and all suffixes of P have leaves in $GST(T, P)$. $GST(T, P)$ can be preprocessed to perform $O(1)$-time queries of the LCA of two nodes, permitting the search for Longest Common Extension of any suffix of P with a suffix of T in constant time.

This very interesting theoretical result is unfortunately slow in practice, the main reasons being the preprocessing stage for fast LCA queries and the data structure $GST(T, P)$ as mentioned in [3]. In [4], Miranda and Ayala-Rincón propose to adapt the strategy to use a suffix array [5] instead of a suffix tree, and present an algorithm for computing the longest common extension on the extended suffix array, which contains the Longest Common Prefix array in addition to the suffix array itself. In [6], Fischer and Heun present a direct algorithm for the general RMQ-problem with linear preprocessing time and constant query time. Then in [7], Sadakane proposes a compressed suffix tree that can be preprocessed to answer LCA queries theoretically in constant time. Fischer and Heun proposed a constant time solution in space $O(nH_k) + o(n)$ bits where H_k is the k-th order empirical entropy [8]. Finally, in [9], Durocher proposes a solution in $O(1)$ time with $O(n)$ words of space.

When the input array it not accessible anymore at query time (which is not the case of the solution we propose), the RMQ can be solved in constant time using $2n + o(n)$ bits [10, 11].

The dynamic setting of the RMQ problem, when insertions, deletions, substitutions can occur in the input sequence, has been much less studied. There exists a solution providing $\Theta(\log n / \log \log n)$ query time and $O(\log n / \log \log n)$ amortized time for the update in linear space [12, 13].

In this article we propose a new approach for the RMQ-problem that is neither linked to Cartesian trees, to Eulerian tours, to the Four-Russian-Trick neither to splitting the sequence in non-overlapping blocks. Moreover, we show that this structure is flexible enough so it can be easily updated whenever edit operations are modifying the text.

2. Our approach

In what follows, we will consider without loss of generality an array $S[0, n-1]$ of integers. Our goal is, given two integers $0 \leq i_\ell < i_r < n$, to find the minimal value $S[k]$ in $S[i_\ell, i_r]$.

2.1. Basic idea

Finding the minimum in a given range can be naïvely performed by traversing all the values in the range of interest. This can be computed more quickly as soon as potential candidates have been identified beforehand. It is more likely for a value to be the answer to a range minimum query if this value is a minimum among its neighbors. In a numeric sequence, such a value is called a local minimum and therefore we will particularly focus on these values. However in a

range there may have several local minima, and in this case, we do not have the possibility to answer quickly to a range minimum query since we would have to compute which local minima is the right answer. Nevertheless this computation is made on fewer values than originally (if one considers all the values in the requested range) and thus it could be answered more quickly. Moreover, we can apply the same idea recursively on the previously selected local minima. Namely we have to identify, among those local minima, which ones are still local minima. This process can be reiterated until we only have one local minima inside the requested range.

Definition 1. A *k-local minimum* in S, for any $k \geq 1$, is a local minimum among all the $k-1$-local minima. Any value of S is a 0-local minimum in S.

We denote by $S^{[k]}$ the sequence composed of k-local minima in position order, in S. Clearly, $S^{[0]} = S$.

Let consider the sequence $S = \overset{0}{5}\ \overset{1}{4}\ \overset{2}{2}\ \overset{3}{4}\ \overset{4}{3}\ \overset{5}{4}\ \overset{6}{3}\ \overset{7}{4}\ \overset{8}{1}\ \overset{9}{4}\ \overset{10}{3}\ \overset{11}{4}\ \overset{12}{6}\ \overset{13}{2}\ \overset{14}{4}$. Following the previously described idea, a minimum query in the range $[1, 14]$ could be processed this way:

1. Local minima in $S[1, 14]$ are 2 (position 2), 3 (positions 4 and 6), 1 (position 8), 3 (position 10) and 2 (position 13).
2. The subsequence of $S[1, 14]$ only consisting of its local minima is $S[1, 14]^{[1]} = \overset{0}{2}\ \overset{1}{3}\ \overset{2}{3}\ \overset{3}{1}\ \overset{4}{3}\ \overset{5}{2}$. Local minima in this subsequence (or 2-local minima in $S[1, 14]$) are 2 (position 0), 1 (position 3) and 2 (position 5).
3. Again, this corresponds to a shorter subsequence $S[1, 14]^{[2]} = \overset{0}{2}\ \overset{1}{1}\ \overset{2}{2}$ which only contains one local minimum: 1 at position 1. Therefore the minimum in the range [1,14] is 1.

We have explained the main idea of our algorithm but until now, we did not focus on its efficiency. It is clear that computing local minima on demand would not be time-efficient. Since a local minimum will always remain a local minimum whatever the requested range is, we can precompute them all in the input sequence. We will now explain how we preprocess the local minima, what information we need to store and how it can be retrieved efficiently.

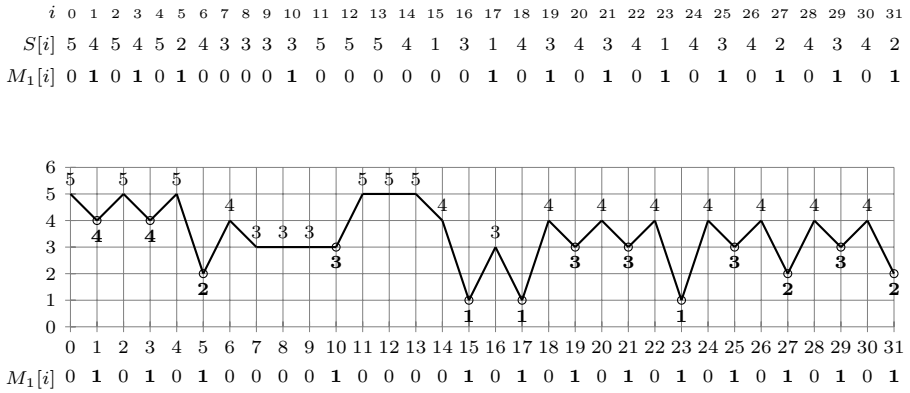
2.2. Storing local minima

The 1-local minimality of each value in S is stored in a bit vector M_1 (*i.e.* $M_1[i] = 1$ iff $S[i]$ is a 1-local minimum):

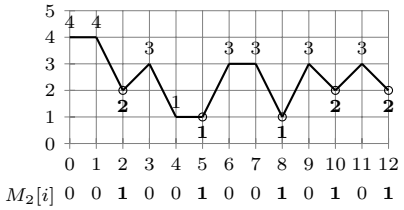
$M_1[0] =$ 1 if $S[0] < S[1]$
0 otherwise.

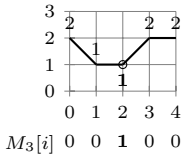
$M_1[n-1] =$ 1 if $S[n-2] > S[n-1]$
0 otherwise.

$M_1[k] =$ 1 if $S[k] < S[k+1]$ and $S[k'] = S[k]$, for all $k' \in [k''\,..\,k]$, and $k'' = 0$ or $S[k''-1] > S[k'']$.
0 otherwise.



The same process is used to identify 2-local minima (in M_2) and 3-local minima (in M_3). Note that in M_2 the number of bits corresponds to the number of 1-local minima since only 1-local minima can be 2-local minima. Generally speaking the number of bits in M_k, $k \geq 1$, is the number of $k-1$-local minima in S. Obviously the number of ones in M_k is the number of k-local minima in S. There is at most one vector M_k without 1-bit. We denote by k_M the total number of bit vectors (*i.e.* from M_1 to M_{k_M}).





We use bit vectors with rank and select capabilities. The $\mathsf{rank}_1(B, i)$ operation consists in computing the number of ones in $B[0\,..\,i]$. Conversely the $\mathsf{select}_1(B, j)$ operation consists in finding the position of the j-th one in B. Bit vectors can be precomputed so that rank and select operations are answered in constant time. Moreover, the bit vector can take a space related to their entropy while still answering the operations in constant time [14, 15]. Some other solutions are designed for more specific inputs: for instance, when having a lot of zeroes in the bit vector, one can use gap encoding and achieve very efficient compression in theory and practice [16, 17, 18].

2.3. Computing the minimum in a given range

Identifying the minimum among local minima can therefore be computed efficiently using these operations. rank operations allow us to determine the number of k-local minima in the requested range as well as determining the indices of the new range of interest for $k+1$-local minima. The process is iterated until the range contains at most one value. The remaining value (if it exists) is the minimum among the local minima.

However this is not necessarily true: computing the minimum among the local minima is not sufficient if one wants to obtain the minimum inside a given range. Indeed, the rightmost and leftmost values of the requested range may be "temporary" local minima, specific to that requested range. This is due to our local minima definition. We say that the first value is a local minimum iff it is less than the second value and the last value is a local minimum iff it is less than the penultimate value. But when considering a specific range, local minima at the beginning and at the end of this range have not been defined this way but using the general definition (for all the other values). Consequently, we will always consider leftmost and rightmost values as if they were local minima. Hence, to determine the minimum at a specific level we need to compare the minimum among the local minima, the first and the last values of the range. More formally, the range minimum query can be solved by:

$$RMQ(S^{[k-1]}, i_\ell, i_r) = \begin{cases} \min(S^{[k-1]}[i_\ell], S^{[k-1]}[i_r]) & \text{if } i_\ell = i_r \\ & \text{or } \mathsf{rank}_1(M_k, i_\ell) = \mathsf{rank}_1(M_k, i_r - 1) \\ \min\left(\begin{array}{l} S^{[k-1]}[i_\ell], S^{[k-1]}[i_r], \\ RMQ\left(\begin{array}{l} S^{[k]}, \mathsf{rank}_1(M_k, i_\ell), \\ \mathsf{rank}_1(M_k, i_r - 1) - 1) \end{array} \right) \end{array} \right) & \text{otherwise} \end{cases}$$

for $0 \leq k < k_M$.

Proof. Let assume that $RMQ(S^{[k-1]}, i_\ell, i_r) = M$ and $\exists i \in [i_\ell, i_r]$ such that $m = S^{[k-1]}[i]$ is the minimum in the range $[i_\ell, i_r]$ and $m < M$.

We first consider the case where $i \neq i_\ell$ and $i \neq i_r$. This case is equivalent to consider that $m < S^{[k-1]}[i_\ell]$ and $m < S^{[k-1]}[i_r]$. Therefore, by definition of local minimality, m should be a k-local minimum. It should even be the local minimum of highest level in this range. Therefore the recursive call on the RMQ function should return m. This recursive call must be made on all the k-local minima within the range $[i_\ell, i_r]$. By definition those k-local minima are identified by a 1 in M_k, and $S^{[k]}$ is only made of the k-local minima. We also note that we are actually only interested in k-local minima in the range $[i_\ell + 1, i_r - 1]$ since the boundaries are treated separately. Let assume that there are b k-local minima until position i_ℓ (included), therefore we must focus on values starting at position b in $S^{[k]}$ (*i.e.* the $b+1$-th local minimum). b can be computed using the rank operation: $b = \mathsf{rank}_1(M_k, i_\ell)$. Similarly, there are e k-local minima until position $i_r - 1$ (included), and the last k-local minima in this range will

be at position $e-1$ in $S^{[k]}$ (as positions start at 0) and $e = \mathsf{rank}_1(M_k, i_r - 1)$. Hence the recursive call on $S^{[k]}$ must be done between positions b and $e-1$, included. By definition this recursive call should return m and since we compute a minimum, m should be returned by $RMQ(S^{[k-1]}, i_\ell, i_r)$ which contradicts our initial hypothesis. Therefore there exists no such $i \in [i_\ell, i_r]$ such that $S^{[k-1]}[i] < M$.

Let us now consider the second case where either $i = i_\ell$ or $i = i_r$. Since $RMQ(S^{[k-1]}, i_\ell, i_r)$ returns the minimum among three values including $S^{[k-1]}[i_\ell]$ and $S^{[k-1]}[i_r]$, it contradicts our initial hypothesis since RMQ should return m. □

Example: Computing $RMQ(S, 2, 16)$ ($i_\ell = 2$, $i_r = 16$)

1. We compute the rank values at positions i_ℓ and $i_r - 1$:
   $\mathsf{rank}_1(M_1, 2) = 1$ and $\mathsf{rank}_1(M_1, 15) = 5$.
   The leftmost and rightmost values are $S[2] = 5$ and $S[16] = 3$.
2. Second recursion level: $i'_\ell = 1$ and $i'_r = 4$
   $\mathsf{rank}_1(M_2, 1) = 0$ and $\mathsf{rank}_1(M_2, 3) = 1$.
   The leftmost and rightmost values are $S^{[1]}[1] = 4$ and $S^{[1]}[4] = 1$.
3. Third recursion level: $i''_\ell = 0$ and $i''_r = 0$
   Return $\min(S^{[2]}[0], S^{[2]}[0]) = 2$.
4. Second recursion level: return $\min(S^{[1]}[1], S^{[1]}[4], 2) = \min(4, 1, 2) = 1$
5. First recursion level: return $\min(S[2], S[16], 1) = \min(5, 3, 1) = 1$

Finally $RMQ(S, 2, 16) = 1$.

Note that in our definition of the RMQ function, the minimum is returned. However the RMQ problem is usually defined as giving the index of the minimum. We returned the value instead of its index for the sake of simplicity. But when computing the minimum, we know from which index it is coming. Therefore we could return the index instead of the value itself.

2.4. Retrieving values from $S^{[k]}$

Our algorithm needs to compare values from $S^{[k]}$ at each recursion level. However, we do not want to explicitly store those values. That would be too space consuming. Hence we show two alternatives for easily retrieving values from $S^{[k]}$.

The first one consists in not storing every $S^{[k]}$ but a limited number of them, and possibly just one (*i.e.* $S^{[0]}$, the original sequence). Suppose that one wants to know the value of $S^{[k]}[i]$, for $1 \leq k \leq k_M - 1$, $0 \leq i < |S^{[k]}|$, and $S^{[k]}$ is not explicitly stored. We know that $S^{[k]}[i]$ is necessarily in $S^{[k-1]}$. Hence we need to determine at which position $S^{[k]}[i]$ is in $S^{[k-1]}$. By definition, $S^{[k]}[i]$ is a $k-1$-local minimum and therefore it is stored as a 1 in M_{k-1}. Moreover it has to be the $i+1$-th 1-bit in M_k. Its position can be easily computed using the select operation: $\mathsf{select}_1(M_k, i+1) = j$. Now we know that the value corresponding to $S^{[k]}[i]$ in $S^{[k-1]}$ is $S^{[k-1]}[j]$. Either $S^{[k-1]}$ is stored explicitly and we can retrieve

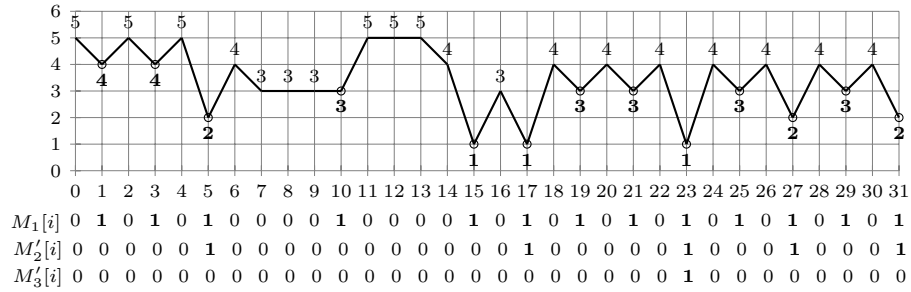


Figure 1: Using bit vectors of the same length as S to avoid the storage of additional $S^{[k]}$ sequences.

the value, or it is not and we recursively apply the same process until we have a sequence explicitly stored. In what follows, we call this solution the "*sampling solution*".

The second solution consists in just storing explicitly S. But the bit vectors M_k, with $1 \leq k \leq k_M$, are defined in a slightly different way so that we can efficiently retrieve any value $S^{[k]}[i]$. Let us denote by M'_k the new way of defining the bit vectors M_k. Every M'_k has the same length: $|S|$, and $M'_k[i] = 1$ iff $S[i]$ is a k-local minimum (see Figure 1). Using such bit vectors, from a position in any M'_k we can directly access the corresponding value in S. This solution is called the "*sparse solution*" because bit vectors are sparse.

2.5. Complexities

So far we have proposed two different solutions to solve the range minimum query problem and, now we focus on their respective time and space complexities.

First, we remark that if we have m k-local minima, we can have at most $\lceil m/2 \rceil$ $k+1$-local minima: two consecutive values cannot both be local minima. Therefore the maximal number of levels (number of bit vectors M_k) $k_M \leq \lceil \log_2 n \rceil$[1].

When answering a query, at each level, we need to retrieve and compare two values, we also need to perform two rank operations. rank operations can be computed in constant time. We denote by t_r the time needed to retrieve a value in S. Hence, at a given level, we need $O(t_r)$ time and overall, for answering the query $O(t_r \log_2 n)$ time. In fact, the depth of recursion depends on the size of the requested range and not on the size of the total sequence. Let us denote by q this size. More precisely, the query complexity time is $O(t_r \log_2 q)$.

The value of t_r depends on the choice made among the two proposed solutions (sample and sparse solution).

[1] In the following, for the sake of simplicity, we will denote $\lceil \log_2 n \rceil$ by $\log n$

	Space (bits)	Time
Sample	$2nH_0 + o(n)$	$O(\log q \log^{1+\varepsilon} \log q)$
Sparse	$3.16n + o(n)$	$O(\log q)$

Table 1: Space (in bits) and time (for a queried range of size q) complexities for the sample and sparse solutions.

Sample If we sample every $\log^{1+\varepsilon} \log n$ sequence $S^{[k]}$, with $\varepsilon > 0$, we would need to perform at most $\log^{1+\varepsilon} \log n$ select operations in time $O(\log^{1+\varepsilon} \log n) = t_r$.

Sparse All bit vectors have the same length, the value can be retrieved in constant time since we directly know its position in S.

For the space consumption we need to account for the bit vectors stored such that they can answer rank and select queries in constant time[2]. In the space complexity we do not account for the space needed for storing S.

Sample We have at most $\log n$ bit vectors whose sizes are, in the worst case, $n, n/2, n/4, \ldots, 2$ bits. In total, that represents $2n - 2$ bits. These bit vectors can be encoded in $2nH_0 + o(n)$ bits for supporting rank and select operations [14]. Above that we store some $S^{[k]}$ sequences (one out of $\log \log^{1+\varepsilon} n$), that means we have $\log n / \log^{1+\varepsilon} \log n$ sequences to store. The total length of the sequences to store is

$$\sum_{i=1}^{\frac{\log n}{\log^{1+\varepsilon} \log n}} \frac{n}{2^{i \log^{1+\varepsilon} \log n}} = o\left(\frac{n}{\log n}\right)$$

Since integers can be stored in $\log n$ bits, the sampled $S^{[k]}$ need $o(n)$ bits of space. Overall the space complexity of the structure is $2nH_0 + o(n)$ bits.

Sparse We have $\log n$ bit vectors of length n, we can encode each of them using $nH_0 + o(n)$ bits. Since most of them are sparse (bit vector M_k has at most $n/2^k$ 1-bit), a compressed encoding will be very efficient on them. The sum of the empirical order entropy is:

$$H_0(M_1) + \cdots + H_0(M_{k_M}) = \sum_{i=1}^{\log n} \frac{i}{2^i} + \left(1 - \frac{1}{2^i}\right) \log\left(\frac{1}{1 - \frac{1}{2^i}}\right) < 3.16$$

Hence, the final space complexity for the sparse solution is at most $3.16n + o(n)$ bits.

[2]Note that select query is not necessary for the sparse solution.

Space and time complexities are summed up in Table 1. It is worth mentioning that both time and space complexities are worst-case complexities. Depending on the input, the space complexity can be much lower. The best case being a monotonically increasing (or decreasing) sequence. In that case, we would have only one local minimum, hence only one bit vector with $n-1$ zeroes and a single one. Such a bit vector can be stored in $o(n)$ bits. Our approach benefits from the complexity of the input sequence. The worst case scenario is reached when at each level half of the values are local minima.

3. Updating the M_k bit vectors

The new approach we introduced for RMQ computation also allows us to consider the dynamic RMQ problem. With our method, adding or removing values from the input sequence just corresponds to adding some bits in bit vectors and computing a few local minimality where values changed.

We consider the insertion of a single value in an existing sequence. Inserting several values consecutively presents no significant difference, from the theoretical viewpoint. Depending on the value to be inserted, several different cases arise. We will present all the different cases and we will examine the changes in the local minimality for values in the neighborhood of the insertion position.

For reasons that will become clearer later, we will need to introduce another bit vector.

Definition 2. A *plateau* consists of at least two identical consecutive values in S. A plateau is such that it cannot be extended either to the right or to the left. In other words the values before or after the plateau differ (if they exist) from the values in the plateau.

We add a new bit vector P of length n delimiting the start and end of the plateaux. P is formally defined in the following way:

$$P[i] = 1 \iff \begin{cases} S[i] = S[i+1] & \text{if } i = 0 \\ S[i] = S[i-1] & \text{if } i = n-1 \\ S[i] \neq S[i-1] \text{ and } S[i] = S[i+1] & \\ S[i] = S[i-1] \text{ and } S[i] \neq S[i+1] & \text{otherwise} \end{cases}$$

Now let assume that position i is inside a plateau, not at the end. Going to the start of the plateau can be done using $\mathsf{select}_1(P, \mathsf{rank}_1(P, i))$. Similarly, now assuming that position i is inside a plateau, not at the start, we can go to the end of the plateau using $\mathsf{select}_1(P, \mathsf{rank}_1(P, i-1)+1)$.

3.1. Analysis of the different cases

3.1.1. Inserting a value lower than the previous one

In Figure 2 we present the cases where the inserted value is lower than the left value. The two first sub-cases (Figure 2a and 2b) present no difficulty. In both cases the inserted value cannot be a local minimum and local minimality of

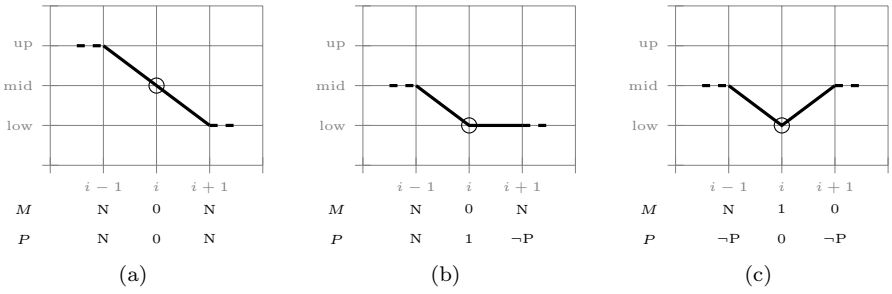


Figure 2: Insertion of a value lower than its left value. Below the coordinates, N stands for "Not changed" meaning that the value at that position does not change in the corresponding bit vector; 0 or 1 means that we will put the corresponding value at that position, whatever the previous value was. ¬P means that we will change the bit by its opposite value.

left and right values remain unchanged. Regarding the plateaux, the second case may create or extend a plateau. Therefore if the value at position $P[i+1]$ was a 1, it was the start of the plateau and it must become a 0 now. On the contrary if it was a 0, that means the value at position $i+2$ is different from the value at position $i+1$. The plateau should now end at position $i+1$. Hence $P[i+1] = 1$. For the third case (Figure 2c), the inserted value is a local minimum. The next value, at position $i+1$ cannot be anymore a local minimum. Hence we need to put a 0 in M. But more generally, from position $i+1$ we need to go at the end of the plateau as described previously. Let denote by e this position, then we set $M[e] = 0$. Note that we may have $e = i+1$.

Regarding the plateau, the inserted value breaks a plateau. We therefore need to change the bits at position $i-1$ and $i+1$ for the same reasons as for the case 2b.

3.1.2. Inserting a value equal to the previous one

Cases in Figure 3a and 3b just consist in inserting a 0 in the M bit vector at the insertion position. In case 3a we just extend the plateau by the middle. We need to insert a 0 in P at position i since we are not at one end of a plateau. Concerning case 3b, we now necessarily end a plateau at position i. The previous bit must be changed for similar reasons as in 2b and 2c.

The last case is slightly more complicated (Figure 3c). By adding a new value that is equal to the left value but lower than the right value, we extend the plateau by one position. Therefore the local minimality of the left value is "transferred" to the inserted value: if the left value was a local minimum, the inserted value becomes a local minimum, otherwise it is not. Finally the left value, in all cases, is not a local minimum anymore.

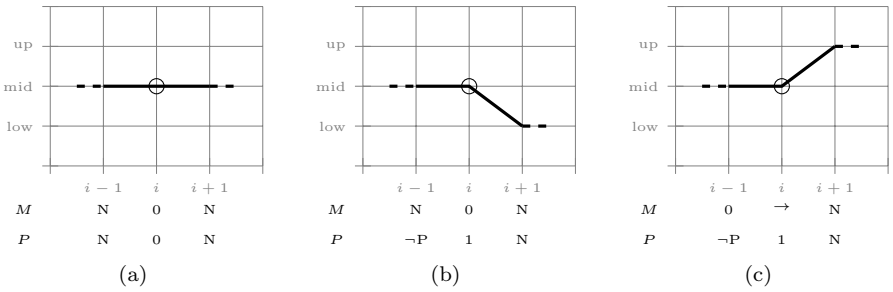


Figure 3: Insertion of a value equal to its left value. The legend remains the same as in Figure 2. $\rightarrow$ means that the inserted value is copied from the former value at the previous position in the bit vector.

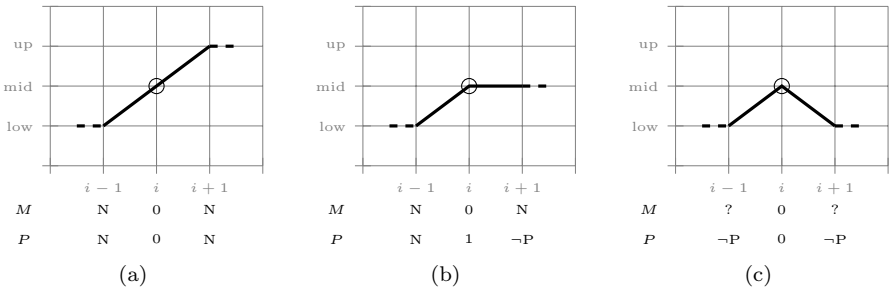


Figure 4: Insertion of a value greater than its left value. The values marked ? need extra computation detailed in the text.

3.1.3. Inserting a value greater than the previous one

When the inserted value is greater than the left value (Figure 4), we still have two easy cases (Figure 4a and 4b) and one (Figure 4c) which is more complicated. Let us focus on this latter case. Inserting a value that is greater than the left value may create a new local minimum. However that also depends on the slope before position $i-1$. We have to reach the start of the plateau, to determine that slope, which can be done in constant time using P. If the slope was decreasing, the value at position $i-1$ is now a local minimum. On the contrary if the slope was increasing we may add a new local minimum after position i. This local minimum would be located at the end of the plateau, after position i and will be added iff the slope is increasing on the left. Therefore is the slope on the left is increasing, we need to go at the end of the plateau on the right and add a local minimum at that position, by setting a 1 in M.

3.1.4. Generalization

Deletions are handled in a similar way. All the cases explored for insertion can be considered for deletion where the value at position i would be deleted. We need to act conversely for the deletions.

The analysis made here is valid for any M_k, $1 \leq k \leq k_M$, since we also proceed by recursion.

3.2. Time and space complexities

Allowing the insertion or deletion of values necessitates to support insertion and deletions in bit vectors. There exist such solutions that also allow to compress the bit vector (*e.g.* Navarro and Nekrich [19]). With such an implementation, rank and select operations are performed in $O(\log n / \log\log n)$ time.

While the majority of cases just requires comparing the left and right values, in some cases we have to traverse the entire plateau, this can be done in constant time using P. The time complexity for updating our dynamic *RMQ* structure when a value is inserted or removed is therefore bounded by $O(\log^2 n / \log\log n)$.

The query time complexities have a $O(\log n / \log\log n)$ penalty compared to the static version due to the dynamic bit vectors used.

In this dynamic setting we also store an additional bit vector P that can be stored in $n + o(n)$ bits. Hence the space complexities from the static case are augmented by $n + o(n)$ bits.

4. Conclusions

We have introduced a new way of computing the range minimum query by considering local minima. This approach sheds a new light on how the range minimum query problem can be viewed. We believe that the approach is interesting by itself from a theoretical viewpoint.

Moreover the structure we presented has space and time complexities which are directly linked to the stored input. This is not reflected by worst-case space and time complexities: as far as we know there exists no such measure as entropy

to reflect the number of local minima a sequence will have and what the value of k_M will be. This approach is the first one to benefit from the composition of the input sequence and which will take less space on less variable sequences.

We also showed how our structure can be adapted to deal with modifications making it the only dynamic structure with a space complexity which is entropy-related.

It would also be interesting to see how the structure behaves in practice compared to other alternatives: in the best case, in the worst case, on real-case data. We also let open the possibility that our approach could be adapted to problems related to the range minimum query, such as the range median query.

References

[1] H. N. Gabow, J. L. Bentley, R. E. Tarjan, Scaling and related techniques for geometry problems, in: Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing, STOC '84, ACM, New York, NY, USA, 1984, pp. 135–143. doi:10.1145/800057.808675.

[2] G. M. Landau, U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in: Proc. of the ACM Symposium on Theory Of Computing (STOC), 1986, pp. 220–230.

[3] G. Navarro, A guided tour to approximate string matching, ACM Computing Surveys 33 (1) (2001) 31–88.

[4] R. C. Miranda, M. Ayala-Rincón, A modification of the Landau-Vishkin algorithm computing longest common extensions via suffix arrays, in: Proc. of Brazilian Symposium on Bioinformatics (BSB), 2005, pp. 210–213.

[5] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, in: Proc. of Symposium on Discrete Algorithms (SODA), 1990, pp. 319–327.

[6] J. Fischer, V. Heun, Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE, in: Proc. of Combinatorial Pattern Matching (CPM), 2006, pp. 36–48.

[7] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.

[8] J. Fischer, V. Heun, Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays, SIAM Journal on Computing 40 (2) (2011) 465–492. doi:10.1137/090779759.

[9] S. Durocher, A Simple Linear-Space Data Structure for Constant-Time Range Minimum Query, in: A. Brodnik, A. Lpez-Ortiz, V. Raman, A. Viola (Eds.), Space-Efficient Data Structures, Streams, and Algorithms, no. 8066 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 48–60.

[10] J. Fischer, Optimal succinctness for range minimum queries, in: Proc. 9th Latin American Theoretical Informatics Symposium, 2010, pp. 158–169.

[11] P. Davoodi, R. Raman, S. R. Satti, Succinct Representations of Binary Trees for Range Minimum Queries, in: J. Gudmundsson, J. Mestre, T. Viglas (Eds.), Computing and Combinatorics, no. 7434 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 396–407.

[12] G. S. Brodal, P. Davoodi, S. S. Rao, Path minima queries in dynamic weighted trees, in: Algorithms and Data Structures, Springer, 2011, pp. 290–301.

[13] P. Davoodi, Data Structures: Range Queries and Space Efficiency, Ph.D. thesis, Aarhus University (2011).

[14] R. Raman, V. Raman, S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees and multisets, in: Proc. of Symposium on Discrete Algorithms (SODA), 2002, pp. 233–242.

[15] G. Navarro, E. Providel, Fast, Small, Simple Rank/Select on Bitmaps, in: R. Klasing (Ed.), Experimental Algorithms, no. 7276 in Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 295–306.

[16] A. Gupta, W.-K. Hon, R. Shah, J. S. Vitter., Compressed data structures: Dictionaries and data-aware measures, Theor. Comput. Sci. 387 (3) (2007) 313–331.

[17] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proc. of the Workshop on Algorithm Engineering and Experiments (ALENEX), 2007.

[18] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theor. Comput. Sci. 387 (3) (2007) 332–347.

[19] G. Navarro, Y. Nekrich, Optimal dynamic sequence representations, SIAM J. Comput.