HAL open science

Efficient dynamic range minimum query

Alice Héliou, Martine Léonard, Laurent Mouchard, Mikael Salson

To cite this version:

Alice Héliou, Martine Léonard, Laurent Mouchard, Mikael Salson. Efficient dynamic range minimum query. Theoretical Computer Science, 2016, 656, pp.108-117. 10.1016/j.tcs.2016.07.002 . hal-01255499v2

HAL Id: hal-01255499

https://hal.science/hal-01255499v2

Submitted on 15 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Efficient dynamic range minimum query

A. Heliou [a,b], M. Léonard [c], L. Mouchard [c,d,b], 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 Univ, UNIROUEN, LITIS, 76000 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

ARTICLE INFO

Article history:
Received 14 August 2015
Received in revised form 16 June 2016
Accepted 4 July 2016
Available online xxxx

Keywords:
Range minimum query
Dynamic structure
Compressed bit vector
Longest common prefix

ABSTRACT

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

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

In [1], Valiant presents a worst-case analysis of the problem of finding the maximum of n elements of a list using k processors. In [2], Shiloach and Vishkin present a model for synchronized parallel computation that finds the maximum of n elements. Ten years after, in [3], Berkman and Vishkin describe a similar problem: the Range-Minima Problem (RMQ for Range Minimum Queries in what follows): "given an array $A = (a_1, a_2, \cdots, a_n)$ of n real numbers, preprocess the array so that for any interval $[a_i, a_{i+1}, \cdots, a_j]$ the minimum over the interval can be retrieved in constant time using a single processor." The solutions to range minimum problems are usually split in two categories: *indexing* solutions, which rely on the input array A at query time, and *encoding* solutions, which do not make use of the input array A when querying. The solutions we propose here are indexing solutions.

Since then, this problem has been intensively studied as it has many important applications, such as pattern matching, text indexing and compression, information retrieval, computational biology (see [4] for more examples).

Regarding the indexing solutions, in [5], Fischer and Heun present an algorithm for the range minimum problem with linear preprocessing time and constant query time. Then in [6], Sadakane proposes a data structure that occupies $2n + o(n)$ bits and answers RMQ 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 [4]. Finally, in [7], Durocher proposes a solution in $O(1)$ time with $O(n)$ words of space.

* Corresponding author.
E-mail address: Mikael.Salson@univ-lille1.fr (M. Salson).

http://dx.doi.org/10.1016/j.tcs.2016.07.002
0304-3975/© 2016 Elsevier B.V. All rights reserved.

Regarding encoding solutions to the range minimum problem (which is not the case of the solution we propose), the result is provided as the index of the minimum rather than its value. In such a case the RMQ can be solved in constant time using $2n + o(n)$ bits [8,4,9].

The dynamic setting of the RMQ problem, where insertions, deletions, substitutions can occur in the original input array, 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 [10,11]. In 2013, Arge et al. [12] sketch possible solutions for the dynamic RMQ in linear space, mainly in the context of external memory queries.

In this article we propose a new approach for the (dynamic) RMQ-problem that is not based on existing ideas for solving the RMQ problem (*i.e.* Cartesian trees, Eulerian tours, Four-Russian-Trick, 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.

In section 2, we present our approach and analyze its complexity. In section 3, we explain how our approach can be extended to handle edit operations such as insertions and deletions. Finally, in section 4, we conclude and draw perspectives.

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 $0 \le i_\ell \le k \le i_r < n$, 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 in time linear to the query range size by traversing all the values in the range of interest. We base our approach on the fact that this can be computed faster 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 two direct 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 be several local minima, and in this case, we do not have the possibility to answer in constant time to a range minimum query since we 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 need to know what values are local minima among the local minima, and so on. This process can be reiterated until we only have one local minimum inside the requested range.

Definition 1. A sequence S of length $n > 1$ has a local minimum at a position:

- $0 < i < n-1$ if: $S[i] < S[i+1]$ and:
  - there is $0 < p \le i$ such that $S[p-1] > S[p]$ and for all $j \in [p..i]$, $S[j] = S[i]$,
  - or, for all $j \in [0..i]$, $S[j] = S[i]$;
- $i = 0$ if $S[0] < S[1]$;
- $i = n-1$ if $S[n-2] > S[n-1]$.

A sequence S of length $n \le 1$ has no local minima.

Definition 2. A *k-local minimum* in S, for any $k \ge 1$, is a local minimum among all the $(k-1)$-local minima. Any value of S is a 0-local minimum in S. We define $S^{[0]} = S$. We denote by $S^{[k]}$ the sequence composed of k-local minima in position order, in S.

Example 1. Let us consider the sequence $S = \overset{0}{5}\,\overset{1}{5}\,\overset{2}{4}\,\overset{3}{2}\,\overset{4}{2}\,\overset{5}{4}\,\overset{6}{5}\,\overset{7}{4}\,\overset{8}{5}\,\overset{9}{3}\,\overset{10}{3}\,\overset{11}{1}\,\overset{12}{4}\,\overset{13}{3}\,\overset{14}{4}\,\overset{15}{6}\,\overset{16}{2}\,\overset{17}{4}$ as a running example, see Fig. 1. Following the previously described idea, a minimum query in the range [1..14] could be processed in this way:

1. Local minima in $S[1..14]$ are 2 (position 4), 4 (position 7), 1 (position 11) and 3 (position 13).
2. The subsequence of $S[1..14]$ consisting only of its local minima is:

   $S^{[1]}[1..14] = \overset{0}{2}\,\overset{1}{4}\,\overset{2}{1}\,\overset{3}{3}$. Local minima in this subsequence (or 2-local minima in $S[1..14]$) are 2 (position 0) and 1 (position 2).
3. The subsequence of $S[1..14]$ consisting only of its 2-local minima is:

   $S^{[2]}[1..14] = \overset{0}{2}\,\overset{1}{1}$ 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 mini-

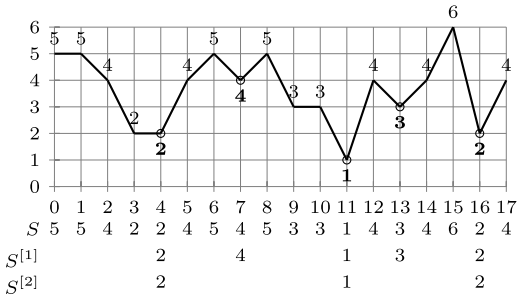

Fig. 1. Example of the 1-local minima and the 2-local minima of the sequence S.

mum 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

Definition 3. The k-local minimality of each value in S is stored in a bit vector M_k (*i.e.* $M_k[i] = 1$ iff $S[i]$ is a k-local minimum).

Generally speaking, the number of bits in M_k, $k \geq 1$, is the number of $k-1$-local minima in S.

We denote by k_M the total number of non-empty local minima sequences (from $S^{[0]}$ to $S^{[k_M-1]}$). There is one vector, M_{k_M}, without a 1-bit.

We enhance the bit vectors M_k 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. Once we have precomputed rank and select data structures on bit vectors M_k, rank and select operations are answered in constant time. Moreover, several solutions allow to efficiently compress the bit vector while still answering the operations in constant time [13–17].

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 query range for the $(k+1)$-local minima. This is iterated until the range contains at most one value. The remaining value (if it exists) is the minimum among the local minima.

However, 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 local minima, *specific* to that requested range.

Example 2. Let us consider the minimum query in the range [5..9] of S.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$S[i]$	5	5	4	2	2	4	5	4	5	3	3	1	4	3	4	6	2	4
$M_1[i]$	0	0	0	0	**1**	0	0	**1**	0	0	0	**1**	0	**1**	0	0	**1**	0

$S[5..9] = \overset{5}{4}\overset{6}{5}\overset{7}{4}\overset{8}{5}\overset{9}{3}$ so the minimum query in the range [5..9] retrieves $S[9] = 3$, but the position 9 is not a 1-local minima of S.

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.

Lemma 2.1. *The range minimum query can be solved by:*

$$RMQ(S^{[k]}, i_\ell, i_r) = \begin{cases} \min(S^{[k]}[i_\ell], S^{[k]}[i_r]), & \text{if } \mathsf{rank}_1(M_{k+1}, i_\ell) \geq \mathsf{rank}_1(M_{k+1}, \max(i_r - 1, 0)) \\ \min\begin{pmatrix} S^{[k]}[i_\ell], S^{[k]}[i_r], \\ RMQ\begin{pmatrix} S^{[k+1]}, \mathsf{rank}_1(M_{k+1}, i_\ell), \\ \mathsf{rank}_1(M_{k+1}, i_r - 1) - 1 \end{pmatrix} \end{pmatrix}, & \text{otherwise} \end{cases}$$

for $0 \leq k < k_M$.

Proof. We will prove this formula by induction over k.

We do the initialization step with $k = k_M - 1$. Let $i_\ell \le i_r$ be indices over $S^{[k_M-1]}$. By definition of k_M, $S^{[k_M-1]}$ does not contain any local minima, thus the minimal element of $S^{[k_M-1]}$ between the indices i_ℓ and i_r is $\min(S^{[k_M-1]}[i_\ell], S^{[k_M-1]}[i_r])$. As $S^{[k_M-1]}$ does not contain any local minima, M_{k_M} does not contain any one, so $\mathsf{rank}_1(M_{k_M}, i_\ell) = \mathsf{rank}_1(M_{k_M}, \max(i_r - 1, 0))$. Thus the formula gives $RMQ(S^{[k_M-1]}, i_\ell, i_r) = \min(S^{[k_M-1]}[i_\ell], S^{[k_M-1]}[i_r])$.

For the induction step we suppose that the formula is correct for any k strictly below some $K < k_M$, and we prove that the formula is correct for $k = K$. Let $i_\ell \le i_r$ be two indices over the sequence $S^{[K]}$, $RMQ(S^{[K]}, i_\ell, i_r)$ has to retrieve the minimal element of $S^{[K]}$ between the indices i_ℓ and i_r.

First case, if there is no local minima in the interval (i_ℓ, i_r) of $S^{[K]}$, then we have $\mathsf{rank}_1(M_{K+1}, i_\ell) = \mathsf{rank}_1(M_{K+1}, \max(i_r - 1, 0)) + \mathbb{1}_{i_\ell = i_r} \mathbb{1}_{M_{K+1}[i_\ell]}$. Consequently $\mathsf{rank}_1(M_{K+1}, i_\ell) \ge \mathsf{rank}_1(M_{K+1}, \max(i_r - 1, 0))$ so the formula gives us $RMQ(S^{[K]}, i_\ell, i_r) = \min(S^{[K]}[i_\ell], S^{[K]}[i_r])$.

Second case, if there is at least one local minimum in the interval (i_ℓ, i_r) of $S^{[K]}$, then the minimum among these local minima can be obtained by using the formula for $k = K + 1$. $\mathsf{rank}_1(M_{K+1}, i_\ell)$ retrieves the number of ones in $S^{[K]}[0..i_\ell]$, so it gives us the index in $S^{[K+1]}$ of the local minimum of $S^{[K]}$ that is just after i_ℓ. $\mathsf{rank}_1(M_{K+1}, i_r - 1)$ retrieves the number of ones in $S^{[K]}[0..i_r - 1]$, so it gives us the index in $S^{[K+1]}$ of the local minimum of $S^{[K]}$ that is just after $i_r - 1$. Consequently, $\mathsf{rank}_1(M_{K+1}, i_r - 1) - 1$ gives us the index in $S^{[K+1]}$ of the local minimum of $S^{[K]}$ that is just before i_r. Thus, the minimum among the local minima inside the interval (i_ℓ, i_r) of $S^{[K]}$ is retrieved by $RMQ(S^{[K+1]}, \mathsf{rank}_1(M_{K+1}, i_\ell), \mathsf{rank}_1(M_{k+1}, i_r - 1) - 1)$. To obtain the minimal element of $S^{[K]}$ between i_ℓ and i_r we need to compute the minimum between the two extremities $S^{[K]}[i_\ell]$, $S^{[K]}[i_r]$ and the minimum among the included local minima. Therefore the formula is correct for $k = K$,

$$RMQ(S^{[K]}, i_\ell, i_r) = \min \begin{pmatrix} S^{[K]}[i_\ell], S^{[K]}[i_r], \\ RMQ \begin{pmatrix} S^{[K+1]}, \mathsf{rank}_1(M_{K+1}, i_\ell), \\ \mathsf{rank}_1(M_{K+1}, i_r - 1) - 1 \end{pmatrix} \end{pmatrix}$$

Consequently, we have proven by induction that the formula of Lemma 2.1 holds for any $0 \le k < k_M$. □

Example 3. Computing $RMQ(S, 4, 15)$ $(i_\ell = 4,\ i_r = 15)$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$S[i]$	5	5	4	2	2	4	5	4	5	3	3	1	4	3	4	6	2	4
$M_1[i]$	0	0	0	0	**1**	0	0	**1**	0	0	0	**1**	0	**1**	0	0	**1**	0

i	0	1	2	3	4
$S^{[1]}[i]$	2	4	1	3	2
$M_2[i]$	**1**	0	**1**	0	**1**

i	0	1	2
$S^{[2]}[i]$	2	1	2
$M_3[i]$	0	**1**	0

1. We compute the rank values at positions $i_\ell = 4$ and $i_r - 1 = 14$:
   $\mathsf{rank}_1(M_1, 4) = 1$ and $\mathsf{rank}_1(M_1, 14) = 4$.
   The leftmost and rightmost values are $S[4] = 2$ and $S[15] = 6$.
2. Second recursion level:
   $i'_\ell = \mathsf{rank}_1(M_1, i_\ell) = 1$ and $i'_r = \mathsf{rank}_1(M_1, i_r - 1) - 1 = 3$;
   $\mathsf{rank}_1(M_2, 1) = 1$ and $\mathsf{rank}_1(M_2, 2) = 2$.
   The leftmost and rightmost values are $S^{[1]}[1] = 4$ and $S^{[1]}[3] = 3$.
3. Third recursion level: $i''_\ell = 1$ and $i''_r = 1$, return $\min(S^{[2]}[1], S^{[2]}[1]) = 1$.
4. Second recursion level: return $\min(S^{[1]}[1], S^{[1]}[3], 1) = \min(4, 3, 1) = 1$.
5. First recursion level: return $\min(S[4], S[15], 1) = \min(2, 6, 1) = 1$.

Finally $RMQ(S, 4, 15) = 1$.

Note that in our formula for the RMQ computation, 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. While 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. 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

S	5	5	4	2	2	4	5	4	5	3	3	1	4	3	4	6	2	4
$M_1[i]$	0	0	0	0	**1**	0	0	**1**	0	0	0	**1**	0	**1**	0	0	**1**	0
$M'_2[i]$	0	0	0	0	**1**	0	0	0	0	0	0	**1**	0	0	0	0	**1**	0
$M'_3[i]$	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0

Fig. 2. Using bit vectors of the same length as S to avoid the storage of additional $S^{[k]}$ sequences.

maximal number of levels (number of bit vectors M_k) k_M is at most $\lceil \log_2 n \rceil + 1$. For the sake of simplicity, we will denote $\log_2 n$ by $\log n$ in what follows. Therefore storing explicitly all the $S^{[k]}$ would require $O(n \log n)$ space in the worst case. We want to access $S^{[k]}$ values in sublinear additional space (with $0 < k$), 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 $1 \le k \le k_M - 1$, $0 \le i < |S^{[k]}|$, 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: $\text{select}_1(M_k, i+1) = j$. 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 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 $1 \le k \le k_M$, 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 Fig. 2). 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 for $k > 1$ the bit vectors M'_k are sparse.

2.5. Complexities

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

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 any $S^{[k]}$, with $0 < k < k_M$. Hence, at a given level, we need t_r time to access a given value and overall, for each level, we need $O(t_r \log n)$ time to answer the query. 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 $O(t_r \log q)$.

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

Sample If we sample every $\log^{1+\varepsilon} \log n$ sequence $S^{[k]}$, with $\varepsilon > 0$ and $0 < k < k_M$, 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, a k-local minimum is identified by a one at a position i in M'_k, with $0 \le k < k_M$. The value of this k-local minimum can be retrieved in constant time in $S[i]$, therefore $t_r = O(1)$.

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.[1] In the space complexity we do not account for the space needed for storing S.

Sample We have at most $k_M = O(\log n)$ bit vectors whose sizes are, in the worst case, $n, n/2, n/4, \ldots, 2$ bits. In total, we can represent these with $2n - 2$ bits. These bit vectors can be encoded in $2nH_0(M) + o(n)$ bits for supporting rank and select operations [13], where $H_0(M)$ is the empirical zero-th order entropy of all the concatenated M_k bit vectors. Above that we store some $S^{[k]}$ sequences (one out of $\log^{1+\varepsilon} \log n$), 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.

[1] Note that select queries are not necessary for the sparse solution.

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

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

Sparse We have at most $k_M = O(\log n)$ 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 $n/2^k$ ones), a compressed encoding will be very efficient on them. The sum of the empirical order entropies is:

$$H_0(M'_1) + \cdots + H_0(M'_{k_M}) \leq \sum_{i=1}^{\lceil \log n \rceil + 1} \left(\frac{1}{2^i} \log 2^i + \left(1 - \frac{1}{2^i}\right) \log \left(\frac{1}{1 - \frac{1}{2^i}} \right) \right)$$
$$\leq \sum_{i=1}^{\lceil \log n \rceil + 1} \left(\frac{i}{2^i} + \left(1 - \frac{1}{2^i}\right) \log \left(\frac{1}{1 - \frac{1}{2^i}} \right) \right) < 3.16$$

The equation can be solved using a computer algebra system (such as the online WolframAlpha). Hence, the final space complexity for the sparse solution is at most $3.16n + o(n)$ bits.

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 is sensitive to the distribution of the input sequence. The worst case scenario is reached when at each level half of the values are local minima.

Regarding construction time, the bit vectors must be built before any query is performed. This can be done in time $O(n)$ or $O(n \log n)$ in the sample or sparse cases respectively.

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 involves adding or removing some bits in bit vectors and computing extra values to determine whether there are new local minima 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 algorithmic 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 need to introduce another bit vector.

Definition 4. A *plateau* is a maximal interval of equal values in the input sequence S.

For each bit vector M_k, we add a new bit vector P_k of length $|M_k|$ delimiting the start and the end of the plateaux. P_k is formally defined in the following way:

$$P_k[i] = 1 \iff \begin{cases} S^{[k-1]}[i] = S^{[k-1]}[i+1] & \text{if } i = 0 \\ S^{[k-1]}[i] = S^{[k-1]}[i-1] & \text{if } i = n-1 \\ (S^{[k-1]}[i] = S^{[k-1]}[i+1] \neq S^{[k-1]}[i-1]) & \text{otherwise} \\ \text{or } (S^{[k-1]}[i] = S^{[k-1]}[i-1] \neq S^{[k-1]}[i+1]) & \end{cases}$$

Now assume that position i is inside a plateau, but not on the right end of the plateau. Jumping to the start of the plateau can be done using $\mathsf{select}_1(P_k, \mathsf{rank}_1(P_k, i))$. Similarly, assume that position i is inside a plateau, not at the start. We can go to the end of the plateau using $\mathsf{select}_1(P_k, \mathsf{rank}_1(P_k, i-1) + 1)$.

Note that while M'_k bit vectors are the sparse versions of M_k bit vectors, we denote by P'_k the sparse versions of P_k bit vectors. We still have $|P'_k| = |M'_k|$.

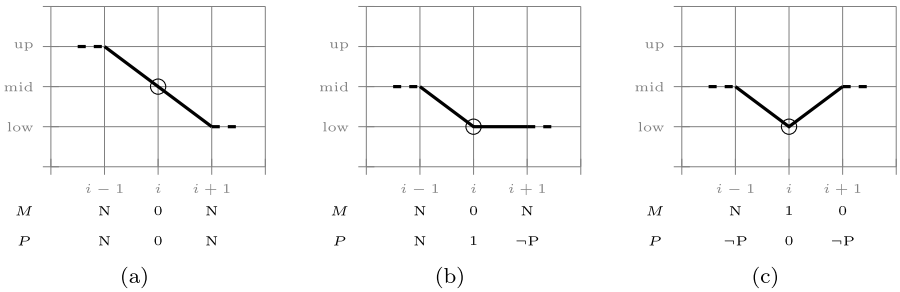

Fig. 3. Insertion of a value at position i which is lower than its left value. Below the coordinates, N stands for "Not changed" meaning that the value at that position does not change in the corresponding bit vector; 0 or 1 means that we will put the corresponding value at that position, whatever the previous value was. ¬P means that we will change the bit by its opposite value.

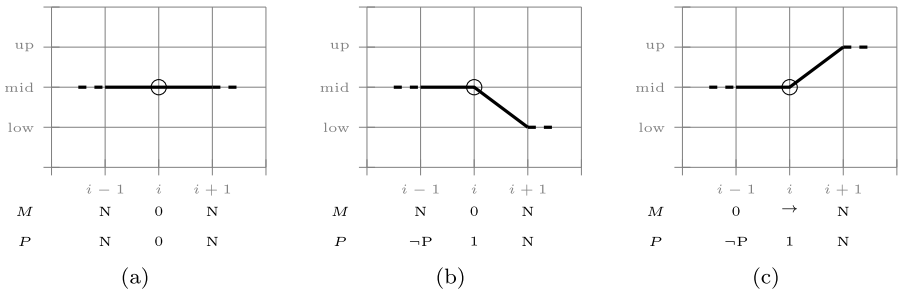

Fig. 4. Insertion of a value at position i equal to its left value. The legend remains the same as in Fig. 3. → means that the inserted value is copied from the value at position $i-1$ in the bit vector before the insertion.

3.1. Analysis of the different cases

As our algorithm is recursive, we will only focus on the operations needed for updating the bit vectors at one given level, k. For the sake of clarity we will denote by M and P the bit vectors respectively storing the local minima, M_k, and the plateaux, P_k.

3.1.1. Inserting a value lower than the previous one

In Fig. 3 we present the cases where the inserted value at position i is lower than its left value, at position $i-1$. The two first sub-cases (Fig. 3a and 3b) present no difficulty. In both cases the inserted value cannot be a local minimum, and the local minimality of both 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 one, it was the start of the plateau and it must become a zero now. On the contrary if it was a zero, 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 (Fig. 3c), 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 zero in M at that position. But more generally, from position $i+1$ we need to go at the end of the plateau as described previously. Let e denote this position. 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 in Fig. 3b.

3.1.2. Inserting a value equal to the previous one

Now we consider the case where the inserted value at position i is equal to its left value, at position $i-1$. Cases in Fig. 4a and 4b just consist in inserting a zero in the bit vector M at the insertion position. In Fig. 4a, the insertion of a value increases the length of the plateau, hence the insertion of a zero at position i in P. We need to insert a zero in P at position i since we are not at the end of a plateau. Concerning case in Fig. 4b, we now necessarily end a plateau at position i. The previous bit in P must be changed for similar reasons as in Fig. 3b and 3c.

The last case is slightly more involving (Fig. 4c). 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 does not. Finally the left value, in all cases, is not a local minimum anymore. Bit vector P is modified as in Fig. 4b.

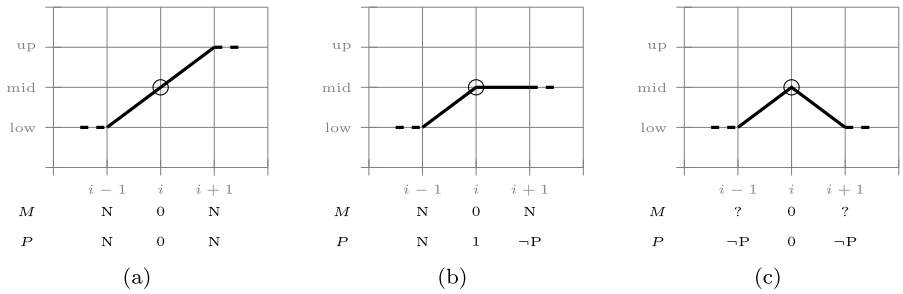

Fig. 5. Insertion of a value at position i greater than its left value. The values marked ? need extra computations detailed in the text.

3.1.3. *Inserting a value greater than the previous one*

When the inserted value at position i is greater than the left value at position $i-1$ (Fig. 5), we still have two easy cases (Fig. 5a is the symmetric of Fig. 3a and Fig. 5b is the symmetric of Fig. 4b) and one (Fig. 5c) which is more complicated. Let us focus on the last case. Inserting a value that is greater than the left value may create a new local minimum depending on the value preceding the start of the current plateau. We have the position s of the start of the plateau directly using P, as explained previously. If the value at position $s-1$ is greater than the value at position $i-1$ or if $s=0$, then the value at position $i-1$ is now a local minimum. On the contrary if the value at position s is lower than the value at position $i-1$, we must add a new local minimum after position i. This local minimum would be located at the position of the right end of the right plateau, after position i. This will be done by setting a one in M at that position.

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 is deleted. For deletions, we need to act in reverse order compared to what was done for insertions.

The analysis made here used a bit vector M that stores positions of local minima. This bit vector M represents any M_k, $1 \le k \le k_M$, since we proceed by recursion. Similarly the bit vector P works in connection with the bit vector M, therefore we have k_M bit vector P_k.

3.2. *Time and space complexities*

Allowing the insertion or deletion of values necessitates to support insertions and deletions in M_k and P_k bit vectors. There exist such solutions that also allow to compress the bit vector (*e.g.* Navarro and Nekrich [18]). With such an implementation, updating the bit vector as well as rank and select operations are performed in $O(\log n/\log\log n)$ amortized 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 $O(\log n/\log\log n)$ amortized time using Navarro and Nekrich's solution to store P_k.

When inserting or deleting a value, we may have to modify up to k_M bit vectors M_k. Each of those modifications require a constant number of modifications and accesses in M_k and P_k which are done in $O(\log n/\log\log n)$ amortized time. We also need a constant number access to $S^{[k]}$, for $1 \le k \le k_M$. Querying also requires a constant number of rank and select operations and a constant number of access to $S^{[k]}$. Therefore time complexities for update and queries will be the same. The structure we present here is built on top of S. Thus updating S is not in the scope of our approach. A simple and naive solution would be to store S in a balanced tree. As time complexities will depend on the time required for accessing a value in S (for example in case Fig. 5c), we will denote it by t_a. We also recall that there are $k_M = O(\log n)$ levels to be considered.

Then depending on the paradigm used for storing the bit vectors, the complexities would differ.

Sample We need to update the sampled $S^{[k]}$ sequences. Those sequences could be stored using the data structure of Navarro and Nekrich [18].

- **Space** The M_k bit vectors can still be stored in space $2nH_0(M)+o(n)$ bits, where M is the concatenation of the M_k bit vectors, to which we add the P_k bit vectors in space $2nH_0(P)+o(n)$ bits, where P is the concatenation of the P_k bit vectors. As in the static case, there are $o\left(\frac{n}{\log n}\right)$ integers to store for the sampled $S^{[k]}$. Storing them in a balanced binary tree would need $\Theta(\log n)$ bits per integer, thus we would need $o(n)$ bits to store all the samples.
- **Update and query** Access to any sampled value is done in $O(\log n \log^{1+\varepsilon} \log n/\log\log n) = O(\log n \log^{\varepsilon} \log n)$ (or in t_a for S) in the worst case and there are at most $O(\log n)$ levels to be updated or queried. Therefore the query time would be $O(\log^2 n \log^{\varepsilon} \log n)$, by assuming for simplicity that $t_a = O(\log n \log^{\varepsilon} \log n)$ (which is well beyond the time complexity for updating a balanced tree for instance).

Table 2
Space and time (update and query) complexities for our solutions and for the solution presented in [10,11]. We do not show complexities from Arge et al. [12] as they focus on an external memory solution, which has different constraints.

	Space (bits)	Time
Sample	$2n(H_0(M) + H_0(P)) + o(n)$	$O(log^2 n \log^{\varepsilon}(\log n))$
Sparse	$6.32n + o(n)$	$O(\log^2 n/\log\log n + t_a \log n)$
[10,11]	$O(n)$	$O(\log n/\log\log n)$

Sparse

Space As in the static case, M'_k bit vectors are stored using a $nH_0 + o(n)$ bit solution [18]. Therefore the space used is still bounded by $3.16n + o(n)$ bits for the M'_k bit vectors. However we have additional P'_k bit vectors, in which the number of ones is at least divided by two at each level (as for M'_k bit vectors). Hence P'_k bit vectors can be stored in $3.16n + o(n)$ bits. Therefore the total space is at most $6.32n + o(n)$ bits.

Update and query The time complexities have a $O(\log n/\log\log n)$ penalty compared to the query in the static version due to the dynamic bit vectors used leading to a $O(\log^2 n/\log\log n + t_a \log n)$ amortized time complexity.

Those time complexities are worse than the one presented in [10,11] but the space complexity is better, as can be seen in Table 2. We therefore introduce a new trade-off for the dynamic range minimum problem. As for the static case, we recall that our complexities are worst case complexities which could improve depending on the actual sequence to query.

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 solved. 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. For instance with a monotonically increasing sequence, our approach needs $o(n)$ bits both in the static and dynamic cases and the queries will be answered in constant time in the static case. On the contrary, space consumption of other approaches will only depend on the length of the sequence and not on its composition.

We also showed how our structure can be adapted to deal with modifications making it the only dynamic RMQ 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 the possibility open that our approach could be adapted to problems related to the range minimum query, such as the range median query.

References

[1] L.G. Valiant, Parallelism in comparison problems, SIAM J. Comput. 4 (3) (1975) 348–355.
[2] Y. Shiloach, U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, J. Algorithms 2 (1) (1981) 88–102.
[3] O. Berkman, U. Vishkin, Recursive star-tree parallel data-structure, March 1990.
[4] J. Fischer, V. Heun, Space-efficient preprocessing schemes for range minimum queries on static arrays, SIAM J. Comput. 40 (2) (2011) 465–492, http://dx.doi.org/10.1137/090779759.
[5] 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.
[6] K. Sadakane, Compressed suffix trees with full functionality, Theory Comput. Syst. 41 (4) (2007) 589–607.
[7] S. Durocher, A simple linear-space data structure for constant-time range minimum query, in: A. Brodnik, A. López-Ortiz, V. Raman, A. Viola (Eds.), Space-Efficient Data Structures, Streams, and Algorithms, in: Lecture Notes in Comput. Sci., vol. 8066, Springer Berlin Heidelberg, 2013, pp. 48–60.
[8] J. Fischer, Optimal succinctness for range minimum queries, in: Proc. 9th Latin American Theoretical Informatics Symposium, 2010, pp. 158–169.
[9] 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, in: Lecture Notes in Comput. Sci., vol. 7434, Springer Berlin Heidelberg, 2012, pp. 396–407.
[10] 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.
[11] P. Davoodi, Data structures: range queries and space efficiency, Ph.D. thesis, Aarhus University, 2011.
[12] L. Arge, J. Fischer, P. Sanders, N. Sitchinava, On (dynamic) range minimum queries in external memory, in: Algorithms and Data Structures: 13th International Symposium, Proceedings, WADS 2013, London, ON, Canada, August 12–14, 2013, 2013, pp. 37–48.
[13] 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.
[14] G. Navarro, Simple rank/select on bitmaps, in: R. Klasing (Ed.), Experimental Algorithms, in: Lecture Notes in Comput. Sci., vol. 7276, Springer Berlin Heidelberg, 2012, pp. 295–306.

[15] A. Gupta, W.-K. Hon, R. Shah, J.S. Vitter, Compressed data structures: dictionaries and data-aware measures, Theoret. Comput. Sci. 387 (3) (2007) 313–331.
[16] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: Proc. of the Workshop on Algorithm Engineering and Experiments, ALENEX, 2007.
[17] V. Mäkinen, G. Navarro, Rank and select revisited and extended, Theoret. Comput. Sci. 387 (3) (2007) 332–347.
[18] G. Navarro, Y. Nekrich, Optimal dynamic sequence representations, SIAM J. Comput. 43 (5) (2014) 1781–1806.

Please cite this article in press as: A. Heliou et al., Efficient dynamic range minimum query, Theoret. Comput. Sci. (2016), http://dx.doi.org/10.1016/j.tcs.2016.07.002