Practical Range Minimum Queries Revisited

Niklas Baumstark[1], Simon Gog[2], Tobias Heuer[3], and Julian Labeit[4]

1 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`niklas.baumstark@student.kit.edu`
2 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`simon.gog@kit.edu`
3 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`tobias.heuer@student.kit.edu`
4 **Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany**
`julian.labeit@student.kit.edu`

Abstract

Finding the position of the minimal element in a subarray $A[i..j]$ of an array A of size n is a fundamental operation in many applications. In 2011, Fischer and Heun presented the first index of size $2n + o(n)$ bits which answers the operation in constant time for any subarray. The index can be computed in linear time and queries can be answered without consulting the original array. The most recent and currently fastest practical index is due to Ferrada and Navarro (DCC'16). It reduces the range minimum query (RMQ) to more fundamental and well studied queries on binary vectors, namely rank and select, and a RMQ query on an array of sublinear size derived from A. A *range min-max tree* is employed to solve this recursive RMQ call. In this paper, we review their practical design and suggest a series of changes which result in consistently faster query times. Specifically, we provide a customized select implementation, switch to two levels of recursion, and use the *sparse table* solution for the recursion base case instead of a range min-max tree.

We provide an extensive empirical evaluation of our new implementation and also compare it to the state of the art. Our experimental study shows that our proposal significantly outperforms the previous solutions on established benchmarks (up to a factor of three) and furthermore accelerates real world applications such as traversing a succinct tree or listing all distinct elements in an interval of an array.

1998 ACM Subject Classification E.1 Data Structures, E.4 Coding and Information Theory

Keywords and phrases Succinct Data Structures, Range Minimum Queries, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2017.12

1 Introduction

Index data structures are computed once for a given input – for instance a document collection or a set of points – and can then be used to answer queries efficiently, without scanning the whole data set again. Hence, a query is reduced to operations whose running time is sublinear in the size of the original input. In the era of Big Data it is not surprising that index structures form the backbone of many search application, such as pattern matching in strings or range queries on point sets. The drawback of traditional index structures is that they are usually larger than the original data and have to reside in main memory to facilitate fast queries. An example are pointer-based search trees. This motivates the development of space-efficient index structures which often are not only substantially smaller than traditional

indexes but can also answer queries without access to the original data. In a seminal paper, Jacobson showed in 1989 how an arbitrary tree of n nodes can be represented in $2n + o(n)$ bits while traversal operations such as finding the parent and children of a node are still supported in constant or logarithmic time [14]. The tree is represented as a vector over $\{0, 1\}$ and navigation is reduced to two operations: $\text{RANK}_1(i, B)$ counts the number of bits set in the prefix of size i of the binary vector B and $\text{SELECT}_1(j, B)$ returns the position of the j-th set bit in B. Today, there exist many other space-efficient counterparts of classic structures which are based on these two fundamental operations; compressed text indexes such as the FM-index are probably the most prominent example [8]. More examples can be found in Navarro's textbook [16].

Improving the performance of basic data structures in this area is of utmost importance as an improvement will often directly translate to faster text indexes or other complex structures [11]. In this paper, we develop a space-efficient data structure to solve range minimum queries, which occur as sub-problems in many different applications. To give a concrete example, they arise in information retrieval problems such as top-k completion [13] and general top-k document retrieval [16].

▶ **Definition 1.** Given an array $A[1..n]$ of n numbers. The *range minimum query* (RMQ) problem is to find an index structure that returns for any range $A[i..j]$ the position of the leftmost minimum. More formally, for any pair of positions $1 \leq i \leq j \leq n$,

$$\text{RMQ}_A(i, j) = \arg\min_{i \leq k \leq j} \langle A[k], k \rangle .$$

Two tuples $\langle a, b \rangle$ and $\langle c, d \rangle$ are compared lexicographically, i.e. $\langle a, b \rangle < \langle c, d \rangle \iff a < c \vee (a = c \wedge b < d)$.

Most solutions of the RMQ problem are based on the reduction to a restricted version of the problem. The restriction is that adjacent entries of the input array $A[1..n]$ only differ by ± 1. Such arrays can be represented as a bit vector B, where $B[1] = 1$ and $B[i] = 1$ if $A[i] - A[i-1] = +1$ and $B[i] = 0$ otherwise. An entry $A[i]$ can be reconstructed by $A[i] = \text{RANK}_1(i, B) - \text{RANK}_0(i, B) = 2 \cdot \text{RANK}_1(i, B) - i$. RANK_0 computes the number of zero bits in a prefix of a vector, analogously to RANK_1. This formula is also called $\text{EXCESS}(i, B)$ and we use it to define the restricted problem:

▶ **Definition 2.** Given a bit vector $B[1..n]$, the ± 1RMQ problem is to find an index structure that returns for any range $B[i..j]$ the position of the leftmost and rightmost excess minimum. More formally, for any pair of positions $1 \leq i \leq j \leq n$,

$$\text{RMQ}^{\pm}_B(i, j) = \arg\min_{i \leq k \leq j} \langle \text{EXCESS}(k, B), k \rangle ,$$

$$\text{RRMQ}^{\pm}_B(i, j) = \arg\min_{i \leq k \leq j} \langle \text{EXCESS}(k, B), -k \rangle .$$

The RMQ problem is well studied and various solutions have been proposed. The first optimal space index requires only $2n + o(n)$ bits and was invented by Fischer and Heun [9]. It can be built in linear time and answers each query in constant time without accessing the original array. Several authors implemented variants of the index [12, 11, 5]. Here, we briefly describe the general idea behind these solutions in order to highlight the contributions of this paper. Any range minimum query can be translated to a *lowest common ancestor query* (LCA) on the *Cartesian tree* of A. The tree can be stored in a succinct representation which uses a bitvector B of length $2n$. Mapping an array element of A to a node in the tree is

reduced to a SELECT operation. The LCA operation can be translated back into a ±1RMQ query on the succinct tree representation. There are several options to solve the ±1RMQ problem, and most practical implementations use the *range min-max tree* to do so [17, 1]. In a last step the result of the ±1RMQ is mapped back to the corresponding position on A, via a RANK operation. Recently, Ferrada and Navarro [5] showed that with a specific tree representation no more than three basic operation calls ($2 \times$ SELECT, $1 \times$ RANK) are required on top of the ±1RMQ call to answer a query[1]. Previous implementations in the SDSL [11] and SUCCINCT [12] library require the execution of another relatively expensive basic operation for an ancestor test. To decrease space usage, Ferrada and Navarro replaced the constant time index for SELECT by a binary search over the RANK index [6] or a combination of select dictionary and scanning [5]. While this negatively affects the overall query time, they show that their solution is still the fastest on a wide range of benchmarks. However, they also noted that the library implementations are still faster on real-world applications such as suffix tree traversal.

In this paper we suggest a series of improvements to Ferrada and Navarro's index. Specifically,

- We show that the knowledge about the height of the Cartesian tree can be used to accelerate SELECT. For trees with logarithmic height we get constant query time without using any extra space. This is a major improvement compared to other previous implementations.
- Navarro & Ferrada proposed two succinct tree representations (rightmost and leftmost path-mapped general tree). One can choose between the two options so as to minimize the height of the tree, but they require different basic structures (RANK_1/RANK_0 and rightmost/leftmost ±1RMQ). We use a single mapping and simulate the other by reversing the input.
- We introduce an effective optimization for small query ranges. Replacing the second SELECT for the right border of the range by a local scan on the parentheses sequence significantly improves the performance on real world applications.
- We replace the traditionally used range min-max tree by a recursive call to our optimized solution and resort to the sparse table approach after a constant number of recursions.

Combining these optimizations we obtain an index that outperforms the existing implementations not only on established benchmarks but also on real-world applications. The remainder of the paper is organized as follows: In Section 2 we review the previous work and present the simplified framework of Ferrada and Navarro. In Section 3 we present our optimizations in detail and subsequently provide experimental evidence of their effectiveness in Section 4.

2 Previous Work

A straightforward constant query time solution to the RMQ problem is to precompute every possible query and store each answer into a lookup table of size $\mathcal{O}\left(n^2\right)$. However the memory requirement of this approach is prohibitively high. Bender & Farach-Colton [2] presented an elegant $\mathcal{O}\left(n \log n\right)$-space solution, which uses a sparse version of the naive lookup table. They precompute a table $M[1..n][1..\log n]$ with $M[i][j] = \text{RMQ}_A(i, i + 2^j - 1)$; i.e. for all queries with an interval size that is a power of two, the answers are stored. An arbitrary query

[1] We note that this simplified approach was also described by Davoodi et al. [4] in CACOON 2012.

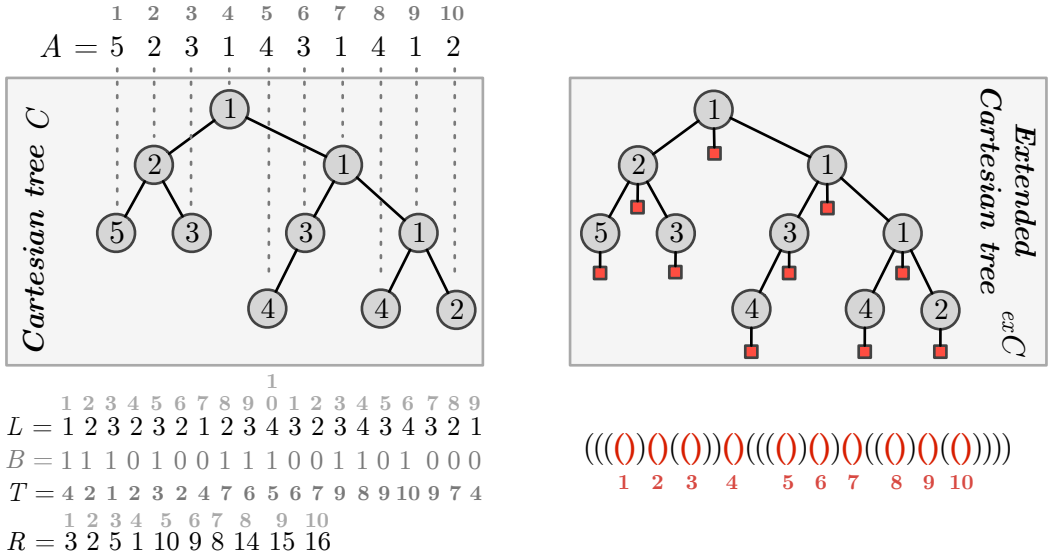

Figure 1 Example of a Cartesian and extended Cartesian tree. Note that we use the leftmost tie-breaking policy.

RMQ(i, j) can be answered by first determining the maximal k with $2^k \leq j - i + 1$. Then $A[M[i, k]]$ and $A[M[j - 2^k + 1, k]]$ are compared to determine the result. The method requires just four memory accesses (two to A and two to M), a comparison, and a *log*-calculation on integers which corresponds to counting leading zero bits[2]. We refer to this solution as SparseTable. More space-efficient RMQ solutions are based on the Cartesian tree introduced in [19].

▶ **Definition 3.** Given an array $A[1..n]$ of numbers. The *Cartesian tree* $\mathcal{C}$ is a binary tree which is recursively defined as follows: (1) If A is empty, then $\mathcal{C}(A)$ is the empty tree. (2) Otherwise, let $p = \arg\min_{1 \leq i \leq |A|} \langle A[i], i \rangle$ be the position of the leftmost minima. The root of $\mathcal{C}(A)$ is element $A[p]$. Its left subtree is $\mathcal{C}(A[1..p-1])$ if $p > 1$ and its right subtree $\mathcal{C}(A[p+1..|A|])$ if $p < |A|$.

We denote as $\mathcal{C}_L/\mathcal{C}_R$ the Cartesian tree where leftmost/rightmost minima are selected in the case of ties, respectively.

Figure 1 shows the Cartesian tree for an example array. Note that one can map between array entries and nodes. The in-order index Inorder(v) of a node v corresponds to the nodes index in A. Conversely, we define the mapping from an index to its node with Innode[3]. Gabow et al. observed that an RMQ query in A can be reduced to calculating the lowest common ancestor (LCA) in $\mathcal{C}(A)$ [10].

$$\text{RMQ}_A(i, j) = \text{Inorder}(\text{LCA}_{\mathcal{C}(A)}(\text{Innode}(i), \text{Innode}(j))) .$$

Berkman and Vishkin noted, that the problem of calculating the LCA can again be reduced to an RMQ query on an array L of the depths of the nodes in the *Euler tour* T through $\mathcal{C}(A)$ [3]. In Figure 1 arrays $T[1..2n]$ and $L[1..2n]$ contain the nodes (identified by their

[2] This operation is part of the instruction set of most modern CPUs.
[3] Note that we omit the mapping in cases where we can directly identify nodes by their in-order index.

in-order index) and their corresponding depth in the Euler tour. Note that L can be replaced by a bit vector B as $L[i] = 2 \cdot \text{RANK}_1(i, B) - i$). With an additional array $R[1..n]$ which contains the first occurrence of each node in the Euler tour LCA queries can be answered as follows:

$$\text{LCA}_{\mathcal{C}(A)}(\text{INNODE}(i), \text{INNODE}(j)) = \text{INNODE}(T[\text{RMQ}^{\pm}_B(R[i], R[j])]) \,.$$

Bender & Farach-Colton [2] solve the ±1RMQ problem by partitioning B into blocks of size $s = \frac{1}{2}\log n$ and creating a SPARSETABLE structure over the $\frac{n}{s}$ block minima. As there can be at most $2^s = \sqrt{n}$ different block types, one can afford to store a lookup table for all $\mathcal{O}\left(2^s \cdot s^2\right)$ in-block RMQ queries. For an arbitrary range $[i..j]$, the query is divided into three sub-queries, including at most two in-block queries in the case where i and j are not block aligned and a SPARSETABLE query for the blocks with indexes in the interval $[\lceil \frac{i+1}{s} \rceil, \lfloor \frac{j-1}{s} \rfloor]$. The EXCESS values of the three positions are compared and the position of the leftmost minimum is returned. This solution requires a linear number of words and the space is dominated by the SPARSETABLE on the sequence of $\frac{2n}{\log n}$ block minima.

Sadakane [18] showed that the ±1RMQ problem can be solved with just sublinearly many bits in addition to B by first dividing B into blocks of size $\log^3 n$ (SPARSETABLE requires $\mathcal{O}\left(\frac{n}{\log n}\right) \in o(n)$ bits), subdividing those into in sub-blocks of size $\frac{1}{2}\log n$ (SPARSETABLE in total again in $o(n)$ bits) and handling the inner blocks again with tabulation as above. He also observed that B can be transformed into a succinct representation – the *balanced parentheses sequence* (BP) of $\mathcal{C}$ – by replacing 1/0 with opening/closing parentheses and appending a closing parenthesis at the end.

BPs were originally introduced by Munro & Raman [15]. The $2n$-bits BP is defined by a depth-first traversal where an opening parenthesis is appended when arriving at a node v and a closing parenthesis is appended after processing v's subtree. Nodes in BP are either in DFS-preorder (if they are identified with their corresponding opening parenthesis) or in DFS-postorder. To use the presented RMQ framework, which is based on in-order, Sadakane introduced the *extended Cartesian tree* $ex\mathcal{C}$, which adds one pseudo-leaf per node (see Figure 1) [18]. These n leaves serve as in-order markers of the original nodes. Now INNODE and INORDER can be realized by SELECT and RANK on the parentheses pattern "`()`" (or "`10`" if interpreted as bits) and the query can be expressed as follows:

$$\text{RMQ}_A(i, j) = \text{RANK}_{\texttt{10}}(\text{RMQ}^{\pm}_{BP(ex\mathcal{C})}(\text{SELECT}_{\texttt{10}}(i), \text{SELECT}_{\texttt{10}}(j))) \,. \tag{1}$$

In 2016, Ferrada and Navarro [5] showed how the balanced parentheses representation can be directly applied to the Cartesian tree. They transform the binary Cartesian tree $\mathcal{C}$ into a leftmost path-mapped general Cartesian tree $\mathcal{C}^L$ (see Figure 2) using a known mapping [15]. A new pseudo-root is introduced and the node from the leftmost path in the binary tree (from the leaf to root) are attached to the new root. This process is applied recursively on the subtrees of the former leftmost path. In $\mathcal{C}^L$ the node with pre-order index $i+1$ corresponds to the node with in-order index i in $\mathcal{C}$ (the $+1$ is due to the added pseudo-root). INNODE and INORDER can by directly realized by SELECT and RANK on the opening parentheses and the RMQ expressed as follows:

$$\text{RMQ}_A(i, j) = \text{RANK}_{\texttt{1}}(\text{RRMQ}^{\pm}_{BP(\mathcal{C}^L)}(\text{SELECT}_{\texttt{1}}(i+1) - 1, \text{SELECT}_{\texttt{1}}(j+1))) \,. \tag{2}$$

Note that the ±1RMQ has to return the rightmost minimum. Alternatively, they provide the formula for the rightmost path mapping ($\mathcal{C}^R$); see Figure 2. In this case pre-order is

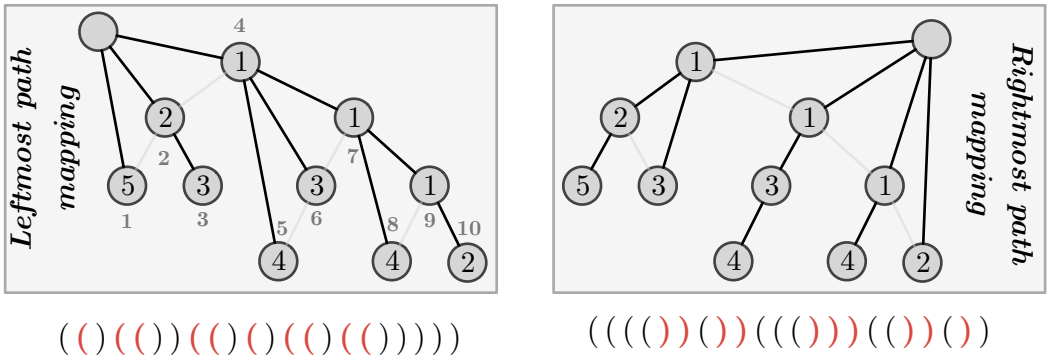

Figure 2 Transformation of a binary Cartesian tree into a general tree. We show the leftmost path mapping ($\mathcal{C}^L$) and rightmost path mapping ($\mathcal{C}^R$) for the example of Figure 1.

replaced by post-order, $\text{SELECT}_1/\text{RANK}_1$ by $\text{SELECT}_0/\text{RANK}_0$ and the ± 1RMQ has to return the *leftmost* minimum.

$$\text{RMQ}_A(i,j) = \text{RANK}_0(\text{RMQ}^{\pm}_{BP(\mathcal{C}^R)}(\text{SELECT}_0(i), \text{SELECT}_0(j))) \,. \tag{3}$$

Ferrada and Navarro already noted the possibility of constructing indexes based on both mappings and choosing the more attractive one, e.g. the one which minimizes the height of the tree. In Figure 2 $\mathcal{C}^L$ has a depth of five, while $\mathcal{C}^R$ has a depth of four. This observation acts as a starting point of our work.

3 An Optimized Recursive Solution

Our first optimization considers the SELECT_1 operation, which is used to find the position of node v with $\text{PREORDER}(v) = i$ when answering range minimum queries using Equation 2. The result of SELECT can be directly determined if $\text{DEPTH}(v)$ is known:

▶ **Lemma 4.** *Let T be a rooted ordered tree and BP be the balanced parentheses sequence of T. The position of the representing opening parenthesis in B of a node $v \in T$ with $\text{PREORDER}(v) = i$ is $\text{SELECT}_1(i, BP) = 2i - \text{DEPTH}(v) - 1$, where $\text{DEPTH}(v) \geq 0$ denotes the distance of v to the root node.*

Proof. By construction the opening parenthesis of v is appended to BP after processing the $i-1$ nodes with smaller pre-order IDs. The $i - 1 - \text{DEPTH}(v)$ nodes which are not ancestors of v were fully processed and we have written a parentheses pair for each of them. The $\text{DEPTH}(v)$ ancestors are not yet fully processed as v is a node in their subtree. So we have written only one parenthesis per ancestor. In total we output v's opening parenthesis at position $2 \cdot (i - 1 - \text{DEPTH}(v)) + \text{DEPTH}(v) + 1 = 2i - \text{DEPTH}(v) - 1$ in an 1-index based sequence. ◀

The answer of every SELECT_1 query can therefore be estimated using the depth of T or the maximal excess in BP More precisely the answer must be within the interval $BP[2i - \text{MAX\_EXCESS}(BP) - 1..2i - 1]$, where $\text{MAX\_EXCESS}(BP) := \max_{0 \leq i \leq |BP|} \text{EXCESS}(i, BP)$. We note that for trees of logarithmic depth we can directly calculate the correct 64-bit word containing the answer without using any extra space. To guarantee constant running time for larger tree sizes, we can employ a sampling scheme similar which uses $o(n)$ space and is very similar to the traditional solution for SELECT.

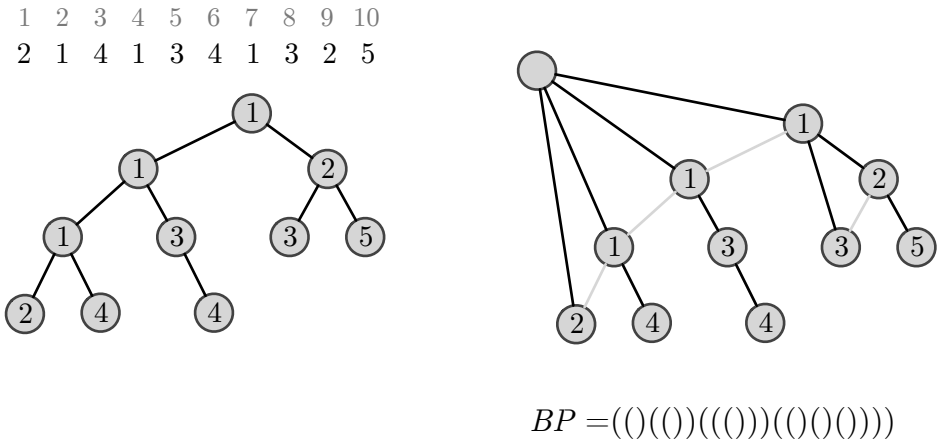

Figure 3 Left: Cartesian tree $\mathcal{C}_R(A[n..1])$ built over $A[n..1]$ (the reverse array of the example in Figure 1; the rightmost tie-breaking policy is used. Right: Leftmost-path general tree $\mathcal{C}_R^L(A[n..1])$ of $\mathcal{C}_R(A[n..1])$. Note that this is the mirrored tree of $\mathcal{C}_L^R(A[1..n])$ in Figure 2 (right).

Next we show that it is not actually necessary to implement the rightmost mapping and its underlying primitives as it can be simulated using the leftmost mapping on the reversed input and a changed tie-breaking policy.

▶ **Theorem 5.** *Let $A[1..n]$ be an array of integers and $\overleftarrow{A} = A[n..1]$ the array in reverse order. Then $\mathcal{C}_L^R(A)$ is isomorphic to $\mathcal{C}_R^L(\overleftarrow{A})$.*

Proof. First we show that reversing the input and changing the tie braking policy yields the same Cartesian tree, hence $\mathcal{C}_L(A)$ is isomorphic to $\mathcal{C}_R(\overleftarrow{A})$. The statement is trivially true for $n = 1$. For $n > 1$ the root of $\mathcal{C}_L(A)$ is the leftmost minimum $m = A[p]$. As $\overleftarrow{A}$ contains the same values as A, m also has to be the minimum in $\overleftarrow{A}$. The mirrored position $p' = n + 1 - p$ contains m and has to be the rightmost minimum in $\overleftarrow{A}$. Otherwise, there would be a $p'' > p'$ with $\overleftarrow{A}[p''] = m$ which is mapped to a position $q = n + 1 - (n + 1 - p) < p$ with $A[q] = m$. This contradicts the assumption that $A[p]$ is the leftmost minimum. The left child of the root of $\mathcal{C}_L(A)$ is $\mathcal{C}_L(A[1..p-1])$ and the right child of the root of $\mathcal{C}_R(\overleftarrow{A})$ is $\mathcal{C}_R(\overleftarrow{A}[p'+1..n])$. By definition $\overleftarrow{A}[p'+1..n]$ is the reverse array of $A[1..p-1]$. By induction $\mathcal{C}_L(A[1..p-1])$ is isomorphic to $\mathcal{C}_R(\overleftarrow{A}[p'+1..n])$. The same argument can be apply to $\overleftarrow{A}[1..p'-1]$ and $A[p+1..n]$. Thus $\mathcal{C}_L(A)$ is isomorphic to $\mathcal{C}_R(\overleftarrow{A})$.

Further we observe that $\mathcal{C}_L(A)$ is the mirrored version of $\mathcal{C}_R(\overleftarrow{A})$. I.e. the rightmost path in $\mathcal{C}_L(A)$ corresponds to the leftmost path in $\mathcal{C}_R(\overleftarrow{A})$. Therefore, the leftmost path mapped tree $\mathcal{C}_R^L(\overleftarrow{A})$ of $\mathcal{C}_R(\overleftarrow{A})$ is the mirrored version of the rightmost path mapped tree $\mathcal{C}_L^R(A)$ of $\mathcal{C}_L(A)$. Hence $\mathcal{C}_L^R(A)$ is isomorphic to $\mathcal{C}_R^L(\overleftarrow{A})$. ◀

The right tree in Figure 2 and the right tree in Figure 3 show both trees, $\mathcal{C}_L^R(A)$ and $\mathcal{C}_R^L(\overleftarrow{A})$, for our running example. A query $\mathrm{RMQ}_A(i,j)$ can be answered with $\mathcal{C}_R^L(\overleftarrow{A})$ as follows. The query range $[i,j]$ is mirrored $[\mu(j),\mu(i)]$, with $\mu(x) = n + 1 - x$, and we get the position p of the rightmost minimum in $\overleftarrow{A}[\mu(j),\mu(i)]$ as $\mathcal{C}_R^L$ breaks ties with rightmost policy. The mirrored position $\mu(p)$ of p in turn is the leftmost minimum in $A[i,j]$. This observation helps to simplify the query algorithm, as it does not need to support both rightmost and the leftmost mapping.

The technique of reversing the input sequence and adjusting the query range can also be used to implement ±1RRMQ and we therefore get a recursive algorithm. Remember that Bender &Farach-Colton divided $BP(\mathcal{C})$ into blocks of fixed size s and built the structure again over the sequence E of block minima. The recursive structure built for E again profits from our proposed optimizations in Lemma 4 and of Theorem 5. The recursion is terminated when the length of E is in $o(\frac{n}{\log^2 n})$ and the SparseTable structure can be applied.

Algorithm 1 summarizes the construction of our RMQ-index. The recursion base case in Line 3 constructs SparseTable. For the remaining Λ levels, first the leftmost-path mapped Cartesian tree with leftmost tie breaking policy $\mathcal{C}_L^L$ over A and the leftmost-path mapped Cartesian tree with rightmost tie breaking policy $\mathcal{C}_R^L$ over the reverse of A is built. The tree of minimal depth is selected and its balanced parentheses sequence stored in BP_λ along with a flag indicating whether A was reversed; see Lines 5–8. Next, BP is partitioned into blocks of size s_λ and two new arrays of size $\frac{n}{s_\lambda}$ are generated in linear time by iterating over BP: Array I_λ and I_λ contain for each block of BP the position respectively the excess-value of the rightmost element with minimal excess. In the pseudo-code, the entries in I_λ are considered as absolute positions in A_λ. In practice these values are stored relative to the start index of each block. So only $\log s_\lambda$ bits per element are required. The second array E_λ contains the excess-value for each entry in I_λ. The entries in E_λ are stored as absolute values, each taking $\log \text{max\_excess}(BP)$ bits. In Line 11 we recursively index E. Note that we index the *reverse* $\overleftarrow{E}$ of E in the recursive call. We will see that this approach results in a very simple query algorithm.

The time complexity of Algorithm 1 is linear in the original input size n for an appropriate choice of Λ and s. For $\Lambda = 3$ and $s = [\log n, \log n, \log n]$ the SparseTable structure in Line 3 is built in the fourth recursive level for an array of size $n''' = \frac{n}{\log^3 n}$; i.e. as SparseTable is constructed in $\mathcal{O}(n''' \log n''')$ this step takes $o(n)$ time. It is easy to see the all remaining steps up to and including the third recursion level take linear time.

The space complexity for this choice is $2n + o(n)$ bits if E_λ is represented implicitly[4]. The $2n$-bit term is due to BP_1 and storing the relative values of I_1 we get additional $\mathcal{O}\left(\frac{n \log\log n}{\log n}\right) = o(n)$ bits of space for the first level. For deeper recursion levels the input is sublinear in n and we can therefore store both BP_λ and I_λ in sublinear space. Finally, SparseTable is in $o(n)$ as the length of the input was reduced to $n''' = \frac{n}{\log^3 n}$ and SparseTable takes $\mathcal{O}\left(n''' \log^2 n'''\right)$ bits.

We have split the query implementation in two parts. The general leftmost RMQ query (see Algorithm 2) and the rightmost ±1RMQ query (see Algorithm 3). We start by explaining Algorithm 2. The recusion base case in Line 3 is easily solved by the precomputed SparseTable. Otherwise we follow Equation 2 by using select to map the query interval $[i, j]$ to position in BP_λ (see Line 6 and 9), solving the ±1RRMQ and mapping the corresponding position back to the original array via rank (see Line 10). Note that handling the tree height minimization only required minor adjustments: First, we introduce a function $\mu(x) = x + rev_\lambda \cdot (n + 1 - x)$ which maps a position x in the original array to the corresponding position in the preprocessed array. Second, we swap the left and right bound of the given query range for cases where the array was reversed during preprocessing (see Line 4 and 5). This ensures that $\mu(i + 1) \leq \mu(j + 1)$ and the range $[\ell, r]$ for the recursive call has a positive size.

[4] This can be achieve by a rank structure over BP_λ.

Algorithm 1 Recursive construction of a Λ-level RMQ-index with leftmost tie-breaking policy for an array A with block sizes s. The λ parameter corresponds to the current recursion level. On level λ the data structure stores the parentheses sequence BP_λ, a flag rev_λ which indicates whether the sequence was reversed, and two arrays $(I_\lambda,\ E_\lambda)$ which contain for each block of BP_λ the position respectively the EXCESS-value of its rightmost element with minimal EXCESS.

```
1:  procedure PREPROCESSING(A[1..n], λ, Λ, s = [s_1, ..., s_Λ])
2:      if λ > Λ then
3:          construct SPARSETABLE over A
4:      else
5:          if depth(C_L^L(A[1..n])) ≤ depth(C_R^L(A[n..1])) then
6:              ⟨BP_λ, rev_λ⟩ ← ⟨C_L^L(A[1..n]), 0⟩
7:          else
8:              ⟨BP_λ, rev_λ⟩ ← ⟨C_R^L(A[n..1]), 1⟩
9:          I_λ ← [±1RRMQ_{BP_λ}((i − 1)s_λ + 1, i · s_λ)  |  1 ≤ i ≤ (2n+2)/s_λ]
10:         E_λ ← [EXCESS(i, BP_λ)  |  i ∈ I_λ]
11:         PREPROCESSING(E_λ[n..1], λ + 1, Λ, s)          ▷ Recursive construction
```

In the experimental part we will observe that most of the query time is spent on the two SELECT operations in Line 6 and 9). This motivates the optimization in Line 7. For small ranges ($j - i < \log n$) we scan a constant number of words for the index and excess of the rightmost minimum. On success ($\neq \perp$), i.e. we reached the j-th opening parenthesis during the scan, we output the result and avoid the second select call in Line 9.

In Algorithm 3 we finally outline our ±1RRMQ implementation. The basic concept follows the idea of Bender & Farach-Colton (see Section 2). In Line 2 we determine the range of blocks $[\ell', r']$ which intersects the query interval $[\ell, r]$. In the next line, we recursively determine the position of the rightmost minimum excess value in the blocks, which are fully contained inside the query interval $[\ell, r]$. We mirror the range bounds as well as the result, since we built the RMQ over the reversed array $\overleftarrow{E}$ of E (see Line 11 in Algorithm 1). Note that the recursive call returns the leftmost minimum of the reversed array, which corresponds to the desired rightmost minimum in the non-reversed array. Next, we obtain the position and EXCESS-value of the rightmost minimum in the two fringe blocks ℓ' and r' by accessing the corresponding entries of I and E. If an obtained position is outside the query range $[\ell, r]$ and the corresponding EXCESS-value smaller than the EXCESS-value obtained from the recursive call, we scan the fringe block to identify the position and EXCESS-value of the rightmost minimum inside the query range. We note, that such a strategy to avoid local scans was similarly suggested by Ferrada and Navarro [5] in the context of range min-max trees.

It is easy to see that the time complexity of a RMQ query is constant. There are a constant number of recursive calls and all basic operations, except SCAN, require only constant time. Note that SCAN can be implemented in constant time for blocks of size $\frac{1}{2}\log n$ by employing lookup tables of sublinear size. In the next section, we will see that scanning a block of a reasonable size, e.g. a constant number of cache lines, will not dominate the practical query time.

Algorithm 2 Recursive query implementation on a Λ-level RMQ-index with leftmost tie-breaking policy. We use two helper functions in the code: An array position x is mapped to the corresponding position in the reversed or non-reversed array by $\mu(x) = x + rev_\lambda \cdot (n+1-2x)$. Method SCAN($BP$) returns a (EXCESS-value,position)-pair of the rightmost element with minimal EXCESS at or to the left of the j-th opening parentheses. In case the j-th opening parentheis is not located in the block $\perp$ is returned.

```
1:  procedure RMQ(i, j, λ, Λ)
2:      if λ > Λ then
3:          return SparseTable(i, j)
4:      if rev_λ = 1 then                ▷ If C_R^L was built on the reversed sequence on level λ
5:          ⟨i, j⟩ ← ⟨j, i⟩                                          ▷ swap left and right bound
6:      ℓ ← select_1(μ(i + 1), BP_λ) − 1
7:      if j − i < log n and (⟨e, p⟩ ← Scan(BP_λ[ℓ..ℓ + 2 log n])) ≠ ⊥ then
8:          return μ(rank_1(p, BP_λ))
9:      r ← select_1(μ(j + 1), BP_λ)
10:     return μ(rank_1(±1RRMQ(l, r, λ, Λ), BP_λ))                    ▷ Cf. Equation 2
```

4 Experimental Evaluation

We created a generic C++ implementation of the proposed RMQ-index. We refer to it as NEWRMQ[5] and compare it to the follows baselines: the two library implementations SDSL[6] [11] and SUCCINCT[7] [12] and the code of Ferrada and Navarro [5] (F&N'16[8]).

The experiments were executed on a single core of a machine equipped with four Intel Xeon E5-4640 processors, with a combined number of 32 cores and 64 hyper-threads, and 512 GiB of memory. All programs were compiled using GCC 4.8.4 with optimizations turned on.

Following the methodology in [5] we generated three variants of artificial inputs. (1) Random inputs RANDOM: Values were drawn uniformly at random from the range $[1, n]$. (2) Pseudo-increasing inputs INC-δ: For a given δ, entry $A[i]$ was chosen at random in $[i - \delta, i + delta]$. (3) Pseudo-decreasing inputs DEC-δ: For a given δ, entry $A[i]$ was chosen at random in $[n - i - \delta, n - i + \delta]$. We varied input lengths ($n = 10^x, x \in \mathbb{N}_+$) and for each variant and generated 10^4 random queries $[i, j]$ for each range size $j - i + 1 \in \{10^1, .., 10^{x-1}\}$.

In an initial experiment we explored the effect of varying the block size s_λ and number of recursion levels Λ in NEWRMQ on RANDOM. While we tested a large range of block sizes and recursion levels we restrict our presentation to the most promising parameters. Specifically, $s \in \{1024, 2048, 4096\}$, which corresponds to 2, 4, and 8 cache lines, and $\Lambda \in \{1, 2, 3\}$. Figure 4 depicts query times and Table 2 index space. In the following we excluded all version with $s \in \{2048, 4069\}$ since the query time for median sized intervals was much worse than for $s = 1024$. We also excluded the $\Lambda = 1$ variants due to their larger memory overhead

[5] Our code is available at `https://github.com/kittobi1992/rmq-experiments`.
[6] Available at `https://github.com/simongog/sdsl-lite` (Accessed at 20.12.2016).
[7] Available at `https://github.com/ot/succinct` (Accessed at 20.12.2016).
[8] Available at `https://github.com/hferrada/rmq` (Accessed at 20.12.2016).

Algorithm 3 Rightmost ±1RMQ query implementation on a Λ-levelRMQ-index.

```
1: procedure ±1RRMQ(ℓ, r, λ, Λ)
2:     ⟨ℓ′, r′⟩ ← ⟨⌈(ℓ+1)/s_λ⌉ − 1, ⌊r/s_λ⌋ + 1⟩          ▷ Leftmost/rightmost covered blocks
3:     p′ ← |E_λ|+1−RMQ(|E_λ|+1−(r′−1), |E_λ|+1−(ℓ′+1), λ+1, Λ)          ▷ Recurse
4:     ⟨⟨p_ℓ′, e_ℓ′⟩, ⟨p_r′, e_r′⟩⟩ ← ⟨⟨I_λ[ℓ′], E_λ[ℓ′]⟩, ⟨I_λ[r′], E_λ[r′]⟩⟩
5:     if e_ℓ′ < E_λ[p′] ∧ p_ℓ′ < ℓ then          ▷ Try to avoid Scan of leftmost block.
6:         ⟨p_ℓ′, e_ℓ′⟩ ← Scan(BP_λ[ℓ..(ℓ′ + 1)s_λ])
7:     if e_r′ < E_λ[p′] ∧ p_r′ > r then          ▷ Try to avoid Scan of rightmost block.
8:         ⟨p_r′, e_r′⟩ ← Scan(BP_λ[(r′ − 1)s_λ..r])
9:     ⟨e, −p⟩ ← min{⟨e_ℓ′, −p_ℓ′⟩, ⟨E_λ[p′], −p′⟩, ⟨e_r′, −p_r′⟩}
10:    return p
```

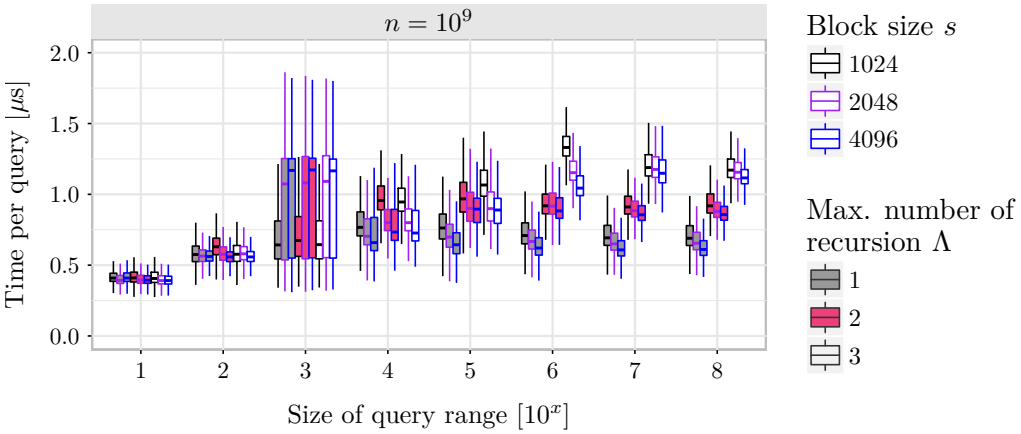

Figure 4 Query time distribution for the recursive RMQ-index on input RANDOM.

and the $\Lambda = 3$ variants due to their slow performance for large intervals. In the remaining experiments NewRMQ is therefore parametrized with by $s = 1024$ and $\Lambda = 2$. Figure 5 shows how much time is spent on the basic operations. Most time is spent on SELECT_1 and ACCESS of E and I. These operations consist mainly of memory accesses and get therefore more expensive for larger inputs due to address translation. Note that SELECT_1 is cheaper for smaller ranges due to caching effects and also the time spent in Scan is notable for small query ranges. For larger query ranges Scan is not triggered, as we can exclude the results from the fringe bocks by the optimizations presented in Algorithm 3.

Next, we compare NewRMQ to the other implementations on RANDOM. The results in Figure 6 for the three competitors are consistent with the outcome of Navarro & Ferrada's study [5]. The experiment shows that the optimization for large query ranges, which avoids scanning the fringe blocks, is much more effective for NewRMQ than for F&N'16, where it is only applied to the range min-max tree. We found that the range min-max tree is not necessarily the cause for many cache misses but for many cache references; see Figure 7 for detailed numbers. Applying SparseTable and the tailored SELECT in NewRMQ reduced the number of cache references significantly.

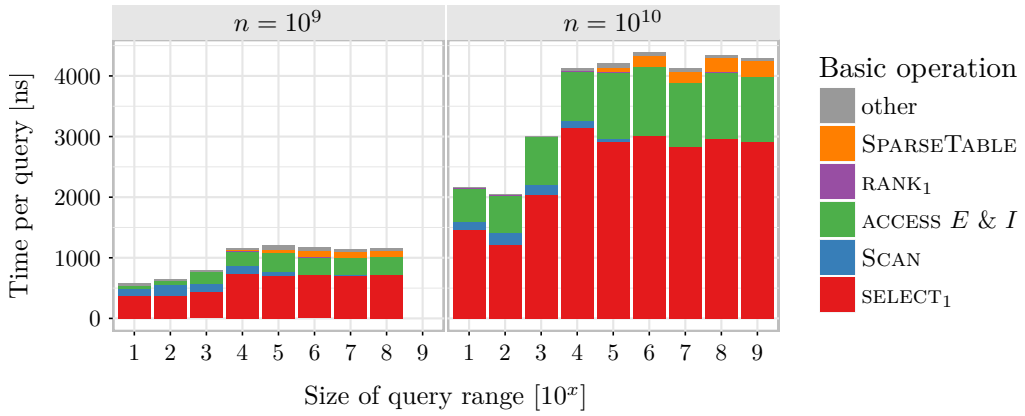

Figure 5 Query time breakdown for NewRMQ obtained by measuring time spent in each basic operation.

Table 1 Statistics for LCP arrays of the full *Pizza&Chilli* texts.

	dblp	dna	english	sources
Depth of $\mathcal{C}^L_L(A)$	543	371	664	**765**
Depth of $\mathcal{C}^L_R(\overleftarrow{A})$	**54**	**120**	**132**	3232
Depth of suffix tree	124	305	148	3238
Ratio of avoided second select calls (by optimization in Line 7 of Algo. 2)	88.32%	91.41%	87.31%	88.33%

Next, we explore the performance on pseudo-increasing and -decreasing inputs (inc-δ and dec-δ). Figure 8 and Figure 9 show the results. As expected the performance of NewRMQ is very similar on both inputs, since we minimize the height of the tree in both cases. For these inputs, it is very likely that the result of a query is located in the left or right fringe block. Therefore the optimization that avoid the scan of fringe blocks for large query ranges is not as effective as in the previous experiment.

Finally, we also consider the performance on a real-world application, namely the traversal of a suffix tree. Here we build our structure over the LCP array and use RMQs to implement the child operation for a node. A stack is used to maintain the ancestors of the currently traversed node. Query ranges – in non-degenerate suffix trees – are typically small and therefore the optimization in Line 7 in Algorithm 2 takes effect. Figure 10 depicts the timing results while Table 1 quantifies the saved operations and also reports the heights of the two Cartesian tree variants.

5 Conclusion

In this work we concerned ourselves with a practical solution for the range minimum query problem. In order to develop a fast solution that is also space efficient, we build upon previous theory and implementation ideas. We propose a new implementation that incorporates novel optimizations that improve the practical performance even further. Compared to existing solutions we replace the range min-max tree with a simpler recursive approach terminated by a sparse table.

Our experimental results show that our new implementation is up to three times faster than previous implementations, while retaining low space usage only slightly above the

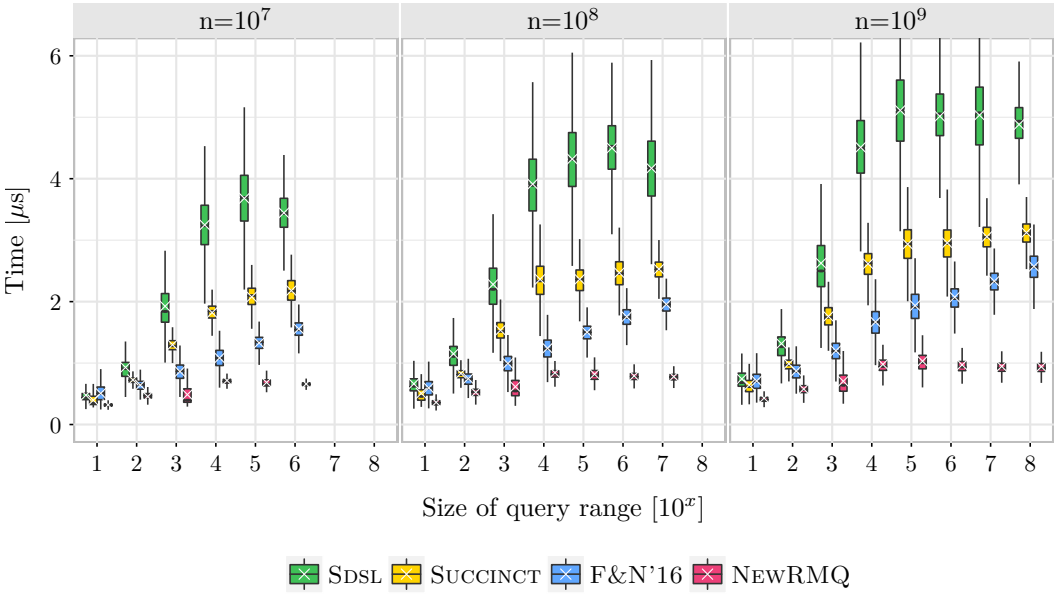

Figure 6 Query time distribution for all implementations on input RANDOM.

theoretical lower bound of 2 bits per input element. For all tested inputs, including both artificial uniform and pseudo-increasing/decreasing random sequences as well as a selected real-world application, we consistently outperform previous implementations.

References

1 D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX*, pages 84–97. SIAM, 2010.

2 M. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN*, pages 88–94. Springer, 2000.

3 O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM Journal on Computing*, 22(2):221–242, 1993.

4 Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. Succinct representations of binary trees for range minimum queries. In *Proc. CACOON*, pages 396–407, 2012.

5 H. Ferrada and G. Navarro. Improved range minimum queries. *J. Discrete Alg.*, 2016. To appear.

6 H. Ferrada and G. Navarro. Improved range minimum queries. In *Proc. DCC*, pages 516–525, 2016.

7 P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *J. Exp. Algorithmics*, 13:12:1.12–12:1.31, February 2009. doi:10.1145/1412228.1455268.

8 P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. FOCS*, pages 390–398, 2000.

9 J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.

10 H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143. ACM, 1984.

11 S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. SEA*, pages 326–337, 2014.

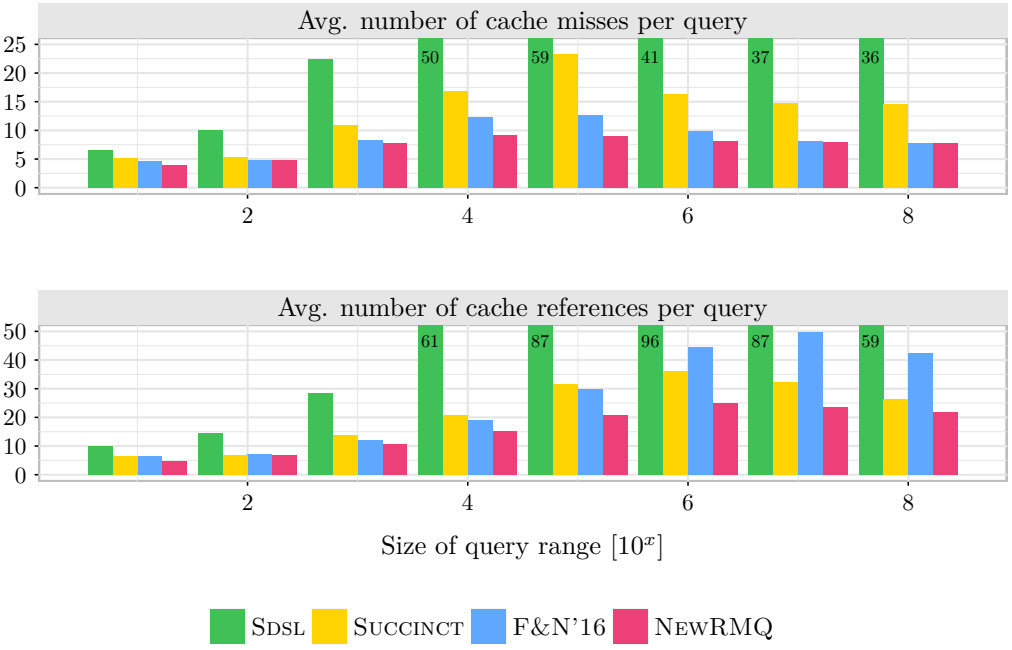

Figure 7 Cache access statistics for all implementations on input RANDOM of size $n = 10^9$.

12 R. Grossi and G. Ottaviano. Design of practical succinct data structures for large data collections. In *Proc. SEA*, pages 5–17, 2013.

13 B. Hsu and G. Ottaviano. Space-efficient data structures for top-k completion. In *Proc. WWW*, pages 583–594. ACM, 2013.

14 G. Jacobson. Space-efficient static trees and graphs. In *Proc. FOCS*, pages 549–554, 1989.

15 J. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

16 G. Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

17 G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms*, 10(3):16:1–16:39, May 2014. doi:10.1145/2601073.

18 K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

19 J. Vuillemin. A unifying look at data structures. *Communications of the ACM*, 23(4):229–239, 1980.

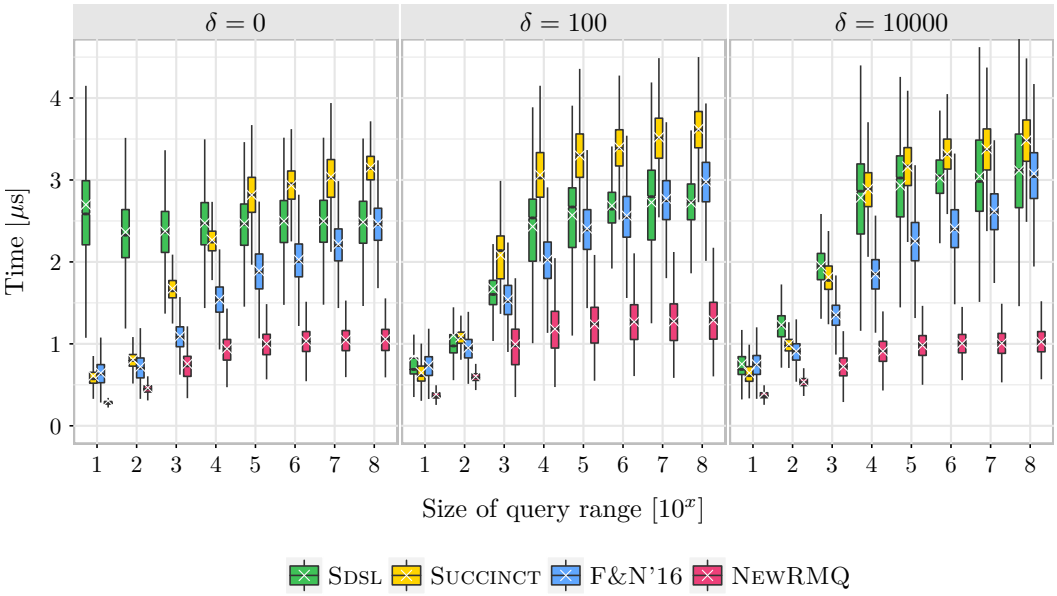

Figure 8 Query time distribution for pseudo-increasing input arrays of size $n = 10^9$.

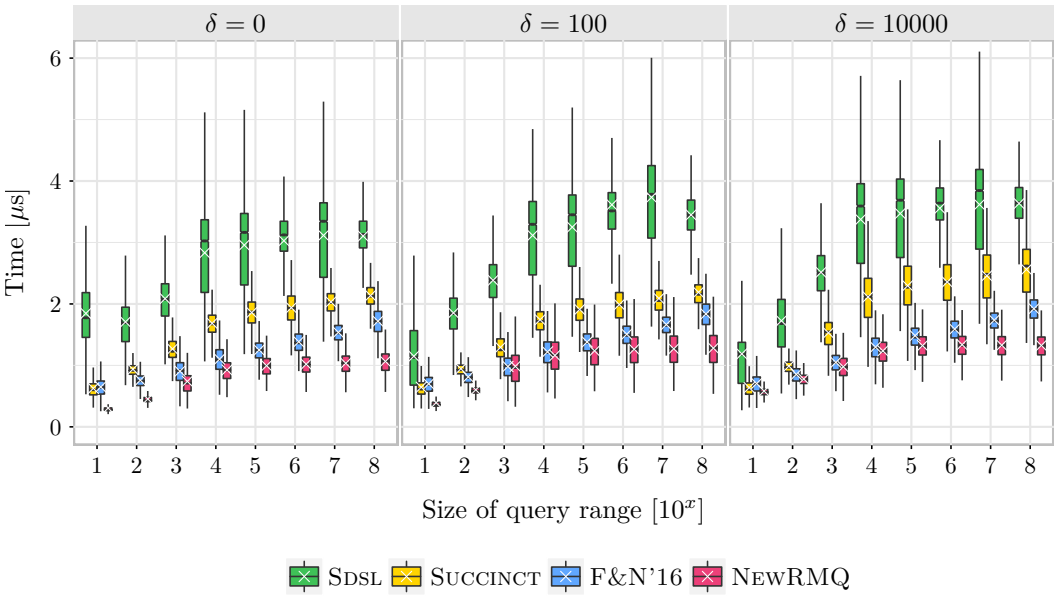

Figure 9 Query time distribution for pseudo-decreasing input arrays of size $n = 10^9$.

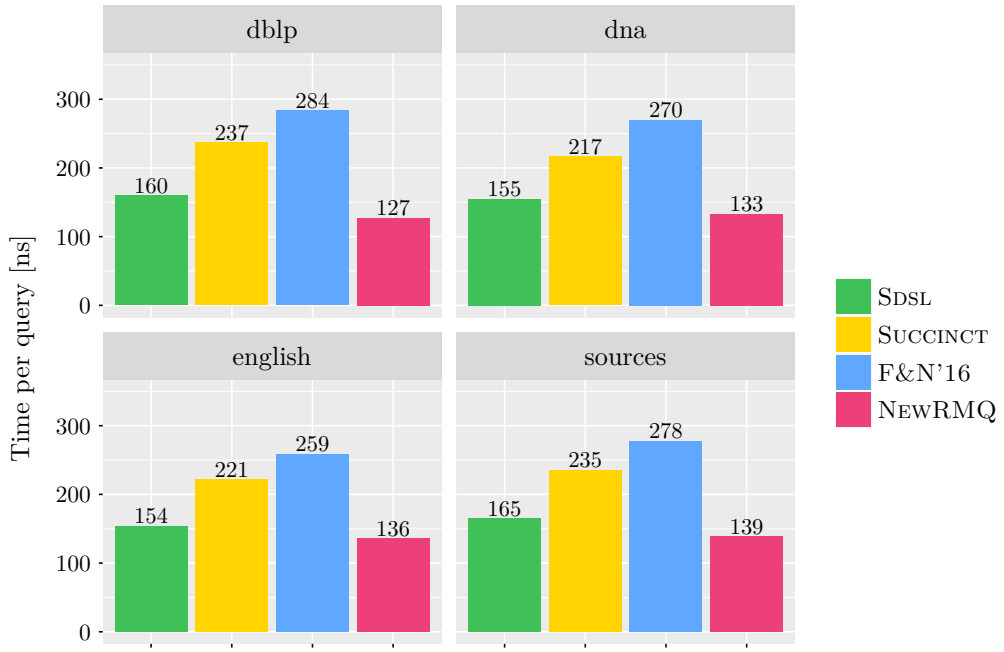

Figure 10 Application benchmark: DFS traversal of a suffix tree. RMQs over the LCP-array are used to calculate the children of a node. We used different texts of the *Pizza&Chilli* corpus[7].

Table 2 Memory consumption dependent on experiment, input size, and implementation. For Figure 5 and Figure 8 we show the data for strictly increasing respectively decreasing sequences.

Implementation	Space in bits per element with varying n					
	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^7$	$n = 10^8$	$n = 10^9$
Data of Fig. 4						
$s = 1024,\ \Lambda = 1$	2.41	2.33	2.37	2.44	2.54	2.65
$s = 1024,\ \Lambda = 2$	2.41	2.18	2.16	2.16	2.16	2.17
$s = 1024,\ \Lambda = 3$	2.41	2.18	2.16	2.16	2.16	2.16
$s = 2048,\ \Lambda = 1$	2.32	2.23	2.24	2.27	2.32	2.37
$s = 2048,\ \Lambda = 2$	2.32	2.16	2.14	2.14	2.14	2.15
$s = 2048,\ \Lambda = 3$	2.32	2.16	2.14	2.14	2.14	2.14
$s = 4096,\ \Lambda = 1$	2.27	2.18	2.18	2.19	2.21	2.24
$s = 4096,\ \Lambda = 2$	2.27	2.18	2.14	2.14	2.14	2.14
$s = 4096,\ \Lambda = 3$	2.27	2.18	2.14	2.13	2.13	2.13
Data of Fig. 6						
SDSL	2.64	3.26	2.61	2.55	2.54	2.54
SUCCINCT	2.80	2.71	2.70	2.71	2.71	2.70
F&N'16	4.52	2.31	2.10	2.09	2.10	2.10
NEWRMQ	2.41	2.18	2.16	2.16	2.16	2.17
Data of Fig. 8						
SDSL	2.62	3.26	2.60	2.54	2.53	2.53
SUCCINCT	2.80	2.71	2.70	2.71	2.71	2.70
F&N'16	4.63	2.43	2.24	2.26	2.28	2.31
NEWRMQ	2.41	2.17	2.16	2.15	2.15	2.16
Data of Fig. 9						
SDSL	2.64	3.27	2.62	2.55	2.54	2.54
SUCCINCT	2.80	2.71	2.70	2.71	2.71	2.70
F&N'16	4.51	2.29	2.07	2.05	2.06	2.06
NEWRMQ	2.41	2.17	2.16	2.15	2.15	2.16