Finding Range Minima in the Middle: Approximations and Applications

Johannes Fischer and Volker Heun

Abstract. A *Range Minimum Query* asks for the position of a minimal element between two specified array-indices. We consider a natural extension of this, where our further constraint is that if the minimum in a query interval is not unique, then the query should return an approximation of the *median* position among all positions that attain this minimum. We present a succinct preprocessing scheme using $Dn+o(n)$ bits in addition to the static input array (small constant D), such that subsequent "range median of minima queries" can be answered in constant time. This data structure can be built in linear time, with little extra space needed at construction time. We introduce several new combinatorial concepts such as Super-Cartesian Trees and Super-Ballot Numbers. We give applications of our preprocessing scheme in text indexes such as (compressed) suffix arrays and trees.

Mathematics Subject Classification (2000). Primary 68W05; Secondary 68W25; Tertiary 68R05.

Keywords. Range Minimum Query, Succinct Data Structure, Cartesian Tree, Ballot Number, Text Indexing, Suffix Tree, Suffix Array.

1. Introduction

The Range Minimum Query (RMQ) problem is to preprocess an array A of n numbers (or other objects from a totally ordered universe) such that queries $\text{RMQ}_A(\ell, r)$ asking for the position of the minimum element in $A[\ell, r]$ can be answered in constant time; more formally, $\text{RMQ}_A(\ell, r) = \text{argmin}_{\ell \leq i \leq r} A[i]$. This problem is of fundamental algorithmic importance due to its connection to many other problems, such as computing lowest common ancestors in trees [4]. Several authors have given algorithms to solve this problem with a linear-time preprocessing [2–4, 8, 17, 25],

Work on this article was partially funded by the German Research Foundation (DFG, Bioinformatics Initiative).

leading to the currently best solution using only $2n + o(n)$ bits in addition to the input array [9].

In the case when the minimum in the query interval is not unique, RMQs can return an arbitrary minimum (although it is easy to adapt all of the above algorithms to return the leftmost or rightmost minimum). However, a natural (and useful, as we shall see!) extension of this is to require that queries return the *median* position of all positions in the query interval that attain the minimum value:

Definition 1.1. For an array $A[0, n-1]$ of n objects from a totally ordered universe, for a given interval $[\ell, r]$ with $0 \leq \ell \leq r < n$, let $M_{[\ell,r]} = \{i \in [\ell, r] : A[i] = \min_{\ell \leq j \leq r} A[j]\}$ denote the set of positions where the sub-array $A[\ell, r]$ attains the minimum. Then the query $\text{RMQ}_A^{\text{med}}(\ell, r)$ asks for an index $m \in M_{[\ell,r]}$ whose rank in $M_{[\ell,r]}$ is between $C\mu$ and $(1 - C)\mu$ among all $\mu = |M_{[\ell,r]}|$ indexes in $M_{[\ell,r]}$ (for some constant $0 < C \leq 1/2$). We call such queries *range median of minima queries.*

The main contribution of this article (Sect. 3) is to give a linear-time preprocessing scheme such that range median of minima queries can be answered in constant time, summarized as follows:

Theorem 1.2. *For a static array A of n elements from a totally ordered universe, there is a data structure with space-occupancy of* $\log(3+2\sqrt{2})\, n+o(n) \approx 2.54311\, n+ o(n)$ *bits*[1] *that allows to answer approximate range median of minima queries (with $C = 1/16$ in Def. 1.1) in $O(1)$ time. This data structure can be built in $O(n)$ time, using only $o(n)$ additional bits at construction time.*

For other tradeoffs between space and approximation ratio, see Thm. 1.3.

1.1. Overview of Applications

One natural application of range median of minima queries comes from text indexing. Basically, such queries are needed for finding quickly (in $O(\log |\Sigma|)$ time) the correct outgoing edge of a node in a suffix tree (Σ being the underlying alphabet), when the suffix tree is only represented implicitly via the suffix- and LCP-array [1], or by a parentheses sequence, as in Sadakane's Compressed Suffix Tree (CST) [24]. We will see that parts of our preprocessing scheme are necessary for the quick construction of one component of the CST. We emphasize that the original proposal [24] *does* make use of range median of minima queries, but *does not* explain how to construct the corresponding structures in linear time — a gap that we close here. A different but related application of RMQ^{med} is an $O(m \log |\Sigma|)$-time algorithm to locate a length-m-pattern in a text over the alphabet Σ, with the help of full-text- or word-suffix arrays [1, 6].

[1]Throughout this article, log denotes the binary logarithm.

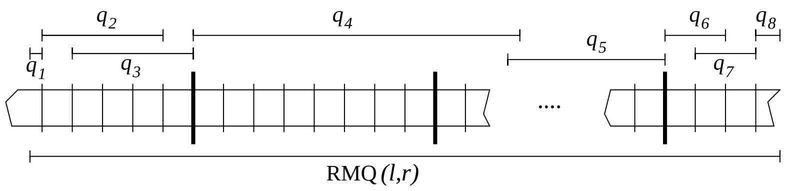


FIGURE 1. The internal decomposition of RMQ$^{\text{med}}(\ell, r)$ into 8 sub-queries $q_1, \dots, q_8$. Thin lines denote block-boundaries, thick lines superblock-boundaries.

1.2. Outline and Overview of Techniques

In the sequel, we briefly sketch how to preprocess an array $A[0, n-1]$ in linear time for RMQ$^{\text{med}}$. Our general idea is similar to other RMQ-algorithms [2–4,8,9,17,25]: decompose a query into several sub-queries, each of which has been precomputed; then the overall minimum inside of the query range can be obtained by taking the minimum over all precomputed intervals. There are two major deviations from previous approaches. First, instead of storing the leftmost minimum, each precomputed query stores the *perfect* median among all its minima. And second, in addition to returning the position of the minimum, each sub-query must also return the number of minima in the query-interval.

The array A to be preprocessed is (conceptually) divided into superblocks $B'_0, \dots, B'_{n/s'-1}$ of size $s' := \log^{2+\epsilon} n$, where B'_i spans from $A[is']$ to $A[(i+1)s'-1]$. Here, $\epsilon > 0$ is an arbitrary constant. Likewise, each superblock is divided into (conceptual) blocks of size $s := \log n/(2 \log \rho)$ (constant ρ to be defined later). The choice of s will become evident after Eq. (3.1). Call the resulting blocks $B_0, \dots, B_{n/s-1}$. The reason for introducing the block-division is that a query RMQ$^{\text{med}}(\ell, r)$ can be divided into at most five sub-queries: one *superblock-query* that spans several superblocks, two *block-queries* that span the blocks to the left and right of the superblock-query, and two *in-block-queries* to the left and right of the block-queries.

We will preprocess the in-block-queries by the "Four-Russians-Trick"; i.e., precomputation of all answers for a sufficiently small number of possible instances (Sect. 3.1). The (super-)block-queries are handled in Sect. 3.2 by a standard two-level storage scheme [20]. It suffices to precompute the (super-)block-queries only for lengths being a power of two, as every query can be decomposed into two (possibly overlapping) sub-queries of length $2^{r-\ell+1}$, the largest power of two that fits into the query range. Thus, a general range minimum query is internally answered by decomposing it in fact into 8 (usually overlapping) sub-queries (see also Fig. 1): 2 in-block-queries q_1 and q_8 at the very ends of the query-interval, 4 block queries q_2, q_3, q_6, q_7, and 2 superblock-queries q_4 and q_5.

A key technique for answering in-block-queries is a generalization of Cartesian Trees [29] to *Super-Cartesian Trees* (Sect. 3.1.1), including algorithms for their construction (Sect. 3.1.2) and enumeration (Sect. 3.1.3). Super-Cartesian Trees will be used for a unique representation of different blocks having the same results for all possible range median of minima queries within such a block. To avoid an explicit space-consuming construction of Super-Cartesian Trees, we present a sophisticated enumeration of these Super-Cartesian Trees using newly introduced Super-Ballot numbers (Sect. 3.1.4). We believe that in particular these new concepts will turn out to have more interesting applications in the future, as Super-Cartesian Trees can be used to describe arbitrary properties about the minima in an interval.

Finally, we have to find the pseudo-median among the perfect medians of the eight sub-queries. Assume now that for each sub-query q_i $(1 \leq i \leq 8)$ we know that at position p_i there is the perfect median among the μ_i minima in the respective query interval. Let k be the minimum value in the complete query-interval ($k := \min_{i \in [1,8]} A[p_i]$), and $I \subseteq [1,8]$ be the set indicating which sub-query-intervals contain the minimum, $I := \{i \in [1,8] : A[p_i] = k\}$. Further, let j be an interval that contains most of these minima, i.e., $j := \operatorname{argmax}_{i \in I} \mu_i$. The algorithm then returns the value p_j as the final answer to RMQ^{med}; this guarantees that the returned position has rank between $\frac{1}{16}\mu$ and $\frac{15}{16}\mu$ among all $\mu \leq \sum_{i \in I} \mu_i \leq 8\mu_j$ positions that attain the minimum value, as p_j has rank $\frac{\mu_j/2}{\mu}\mu \geq \frac{\mu_j/2}{8\mu_j}\mu = \frac{1}{16}\mu$ (and rank $\leq \frac{15}{16}\mu$ by a similar calculation).

We thus conclude that if a query interval contains μ minima, we have an algorithm which returns a pseudo-median with rank between $\frac{1}{16}\mu$ and $\frac{15}{16}\mu$, provided that the precomputed queries return the true median of minima. Section 3 explains how this goal is met.

1.3. Approximation Scheme

In Sect. 4 we show that it is possible to approximate the median of the minima more precisely than in Thm. 1.2, if one is willing to use more space. In particular, for any constant $\alpha > 0$ it is possible to approximate the median of μ minima with an element having a rank in $[(\frac{1}{2} - \alpha)\mu, (\frac{1}{2} + \alpha)\mu]$ if we store for each block the positions of the r-quantiles instead of the perfect median position for an appropriately chosen r (depending on α). As long as r is a constant, it can be easily verified that our algorithm presented in Sect. 3 can be modified to determine for each block the positions of the r-quantiles (as well as the number of minima between each pair of consecutive r-quantiles) instead of the perfect median position.

Theorem 1.3. *For a static array A of n elements from a totally ordered universe, and any constant $0 < \alpha < 1/2$, there is a data structure with space-occupancy of $\log(3 + 2\sqrt{2})\, n + o(\frac{n}{\alpha}) \approx 2.54311\, n + o(\frac{n}{\alpha})$ bits that allows to answer approximate range median of minima queries (with $C = \frac{1}{2} - \alpha$ in Def. 1.1) in $O(\frac{1}{\alpha})$ time. This data structure can be built in $O(\frac{n}{\alpha})$ time.*

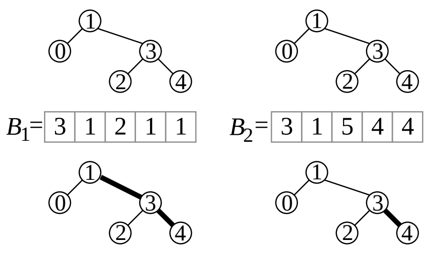


FIGURE 2. Two blocks B_1 and B_2 with equal Cartesian Trees (top), but a different layout of minima. The Super-Cartesian Trees (bottom) reflect the positions of minima by red (i.e., thick) edges.

2. Preliminaries

We use the standard word-RAM model of computation, where fundamental arithmetic operations on words consisting of $\Theta(\log n)$ consecutive bits can be computed in constant time. For an array A of n elements, we write $A[\ell, r]$ to denote A's subarray from ℓ to r. One important definition [29] that plays a central role in almost every algorithm for answering RMQs is as follows:

Definition 2.1. The *Cartesian Tree* of an array $A[\ell, r]$ is a binary tree $\mathcal{C}(A)$ whose root is a minimum element of A (ties are broken to the left), labeled with the position i of this minimum. The left child of the root is the Cartesian Tree of $A[\ell, i-1]$ if $i > \ell$, otherwise it has no left child. The right child is defined analogously for $A[i+1, r]$.

Cartesian Trees are sometimes called *treaps*, as they exhibit the behavior of both a search tree (here on the array-indices) and a min-heap (here on the array-values).

3. Preprocessing for Range Median of Minima Queries

In this section, we present the details of the preprocessing algorithm sketched in the introduction, thereby proving Thm. 1.2.

3.1. Preprocessing for Short Queries

Let us first consider how to precompute the perfect median of the minima inside the blocks of size s (queries q_1 and q_8 in Fig. 1). We cannot blindly adopt the solutions based on the Cartesian Tree [3, 4, 8, 9, 25], because for normal RMQs blocks with the same Cartesian Tree are regarded as equal, totally ignoring their distribution of minima. As an example, look at the two blocks B_1 and B_2 in Fig. 2, where $\mathcal{C}(B_1) = \mathcal{C}(B_2)$. But for B_1 we want $\text{RMQ}^{\text{med}}(1, 5)$ to return position 4 (the median position of the 3 minima), whereas for B_2 the same RMQ should return position 2, because this is the unique position of the minimum. We overcome this

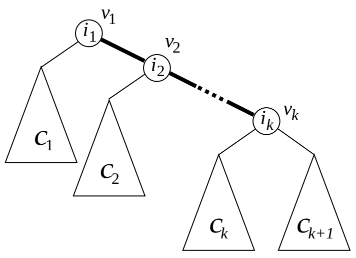


FIGURE 3. Illustration to the definition of Super-Cartesian Trees. The red (i.e., thick) edges can be considered as right edges with a different "label"; this equals the definition of Schröder Trees.

problem by introducing a new kind of Cartesian Tree which is tailored to meet our special needs for this task.

3.1.1. Super-Cartesian Trees. We first define Super-Cartesian Trees.

Definition 3.1. Let $A[\ell, r]$ be an array. The *Super-Cartesian Tree* $\mathcal{C}^{\text{sup}}(A)$ of A is a node-labeled binary tree (with labels from $[\ell, r]$), where the right outgoing edge of a node can be either *red* or *blue*. It is recursively constructed as follows:

- If $\ell > r$, $\mathcal{C}^{\text{sup}}(A)$ is the empty tree.
- Otherwise, let $\ell \leq i_1 \leq \cdots \leq i_k \leq r$ $(k \geq 0)$ be the set of positions where $A[\ell, r]$ attains its minima.
  - Construct $\mathcal{C}_1 := \mathcal{C}^{\text{sup}}(A[\ell, i_1 - 1])$, $\mathcal{C}_2 := \mathcal{C}^{\text{sup}}(A[i_1 + 1, i_2 - 1])$, $\ldots$, $\mathcal{C}_{k+1} := \mathcal{C}^{\text{sup}}(A[i_k + 1, r])$ recursively.
  - Create k nodes $v_1, \ldots, v_k$, where v_j is labeled with i_j.
  - Node v_1 is the root of $\mathcal{C}^{\text{sup}}(A)$.
  - For $i > 1$, v_i is connected as the right child of v_{i-1}, using a *red* edge.
  - For $1 \leq i \leq k$, the left child of v_i is the root of $\mathcal{C}_i$.
  - The right child of v_k is the root of $\mathcal{C}_{k+1}$, connected with a blue edge.

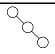
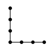
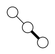
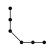
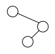
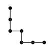
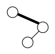
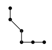
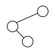
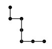
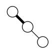
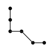
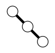
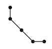
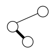
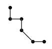
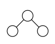
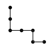
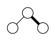
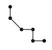
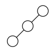
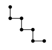
See Fig. 3 for an illustration to the definition of the Super-Cartesian Tree. Here and in the remainder of this article, we draw red right edges with a **thicker** stroke to distinguish them from the blue right edges. Fig. 2 shows $\mathcal{C}^{\text{sup}}$ for our two example blocks, and the column labeled "$\mathcal{C}^{\text{sup}}(A)$" in Tbl. 1 gives further examples. The reason for calling this tree the *Super-Cartesian Tree* will become clear when we analyze the number of such trees (Sect. 3.1.3).

The desired connection between RMQ$^{\text{med}}$ and Super-Cartesian Trees is given by the following lemma, which follows immediately from Def. 3.1.

Lemma 3.2 (Relating RMQs and Super-Cartesian Trees). *Let A and B be two arrays, both of size n. Then the following two statements are equivalent:*

1. *For all $0 \leq i \leq j < n$: the minima in $A[i, j]$ occur at the same positions as the minima in $B[i, j]$*
2. *$\mathcal{C}^{\text{sup}}(A) = \mathcal{C}^{\text{sup}}(B)$.*

Table 1. Example-arrays of length 3, their Super-Cartesian Trees, and their corresponding paths in the graph in Fig. 4. The last column shows how the algorithm in Fig. 5 calculates the index of $\mathcal{C}^{\text{sup}}(A)$ in an enumeration of all Super-Cartesian Trees.

array A	$\mathcal{C}^{\text{sup}}(A)$	path	number in enumeration
123			0
122			$\hat{C}_{03} = 1$
132			$\hat{C}_{03} + \hat{C}_{02} = 2$
121			$\hat{C}_{03} + \hat{C}_{02} + \hat{C}_{02} = 3$
231			$\hat{C}_{03} + \hat{C}_{02} + \hat{C}_{02} + \hat{C}_{01} = 4$
112			$\hat{C}_{13} = 5$
111			$\hat{C}_{13} + \hat{C}_{02} = 6$
221			$\hat{C}_{13} + \hat{C}_{02} + \hat{C}_{01} = 7$
212			$\hat{C}_{13} + \hat{C}_{12} = 8$
211			$\hat{C}_{13} + \hat{C}_{12} + \hat{C}_{02} = 9$
321			$\hat{C}_{13} + \hat{C}_{12} + \hat{C}_{02} + \hat{C}_{01} = 10$

This lemma implies that for answering in-block-queries, we can use a global lookup-table P that stores the answers only for all *possible* Super-Cartesian Trees on s nodes, in contrast to storing the answer for all n/s *occurring* blocks. In order to index into P, each block B_i, $0 \le i \le n/s - 1$, has to be given a "number," such that blocks with the same number have the same Super-Cartesian Tree. The computation of these block numbers will be explained in Sect. 3.1.4. Before that, in Sections 3.1.2 and 3.1.3 we take a closer look at Super-Cartesian Trees.

3.1.2. Construction of Super-Cartesian Trees. We give a linear algorithm for constructing $\mathcal{C}^{\text{sup}}$. This is a straightforward extension of the algorithm for constructing the usual Cartesian Tree [11], treating the "equal"-case in a special manner, as explained next. Let $\mathcal{C}^{\text{sup}}_{i-1}(A)$ be the Super-Cartesian Tree for $A[0, i-1]$. We want to construct $\mathcal{C}^{\text{sup}}_{i}(A)$ by inserting a new node w with label i at the correct position in $\mathcal{C}^{\text{sup}}_{i-1}(A)$. Let $v_1, \ldots, v_k$ be the nodes on the rightmost path in $\mathcal{C}^{\text{sup}}_{i-1}(A)$ with labels $\ell_1, \ldots, \ell_k$, respectively, where v_1 is the root, and v_k is the rightmost leaf (the node labeled $i-1$). Let x be defined such that $A[\ell_x] \le A[i]$ and $A[\ell_{x'}] > A[i]$ for all $x < x' \le k$. We first create a new node w labeled with i. We then remove v_x's right child v_{x+1} and append it as the left child of w. Now, if $A[\ell_x] = A[i]$, connect

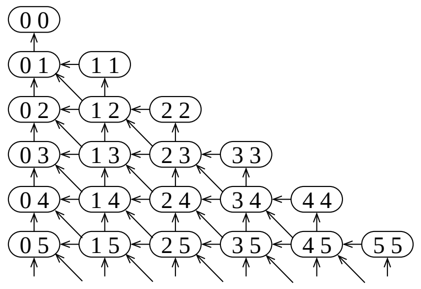


FIGURE 4. The infinite graph whose vertices are (p, q) for all $0 \leq p \leq q$. There is an edge from (p, q) to $(p-1, q)$ if $p > 0$, and to $(p, q-1)$ and $(p-1, q-1)$ if $q > p$.

w with a *red* edge to v_x. Otherwise (i.e., $A[\ell_x] < A[i]$), w becomes the "normal" right child of v_x (i.e., it is connected to v_x with a *blue* edge). An easy amortized argument [11] shows that the overall running time is linear.

3.1.3. The Number of Super-Cartesian Trees. We show that there is a one-to-one correspondence between Super-Cartesian Trees and paths from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in the graph defined in Fig. 4. This bijection is obtained from the above construction algorithm for $\mathcal{C}^{\text{sup}}$ (Sect. 3.1.2), where we first bijectively map the tree to a sequence of numbers $\ell_1, \ldots, \ell_n$, which can in turn be mapped bijectively to a path in the graph, explained as follows.

Consider step i of the above construction algorithm. Let ℓ_i count the number of *blue* edges on $\mathcal{C}^{\text{sup}}_{i-1}$'s rightmost path that are traversed (and therefore removed from the rightmost path) when searching for w's correct insertion point v_x. Note that the ℓ_i's do *not* count the traversed (i.e. removed) red edges on $\mathcal{C}^{\text{sup}}_{i-1}$'s rightmost path. In the graph in Fig. 4, we translate this into a sequence of ℓ_i upwards moves, and then either a *leftwards* move if $A[\ell_x] < A[i]$, or a *diagonal* move if $A[\ell_x] = A[i]$. This gives a one-to-one correspondence between Super-Cartesian Trees and paths from $\boxed{s\ s}$ to $\boxed{0\ x}$ for some x (which is then canonically continued to $\boxed{0\ 0}$), because in step i we constrain the number of upwards moves in the graph in Fig. 4 by the number of strict leftwards moves that have already been made. See again Tbl. 1 for examples.

It is well known [27] that the number of paths from $\boxed{s\ s}$ to $\boxed{0\ 0}$ is given by the s'th *Super Catalan Number* $\hat{C}_s$ (also known under the name *Little Schröder Numbers* due to their connection with Schröder Trees). These numbers are quite well understood, for our purpose it suffices to know that [19, Thm. 3.6]

$$\hat{C}_s = \frac{\rho^s}{\sqrt{\pi s}(2s-1)}(1 + O(s^{-1})) \ , \tag{3.1}$$

with $\rho := 3 + 2\sqrt{2} \approx 5.8284$. In particular, $\hat{C}_s \leq \rho^s$ for large enough s.

As already mentioned, this means that we do not have to precompute the in-block-queries for all n/s *occurring* blocks, but only for $O(\rho^s)$ "sample blocks" of size s with pairwise different Super-Cartesian Trees. (Now our choice for the block-size s becomes clear.) We simply do a naive precomputation of all possible s^2 queries inside of each sample block. Because the blocks are of size s, the resulting table P needs $O(\hat{C}_s \times s^2 \cdot \log s) = O(\sqrt{n} \log^2 n \log\log n) = o(n)$ bits, and the time to compute it is $O(\hat{C}_s \times s^3) = o(n)$. (The additional factor s accounts for finding the median of the minima in each step.) As explained in the introduction, we also have to store the number of minima for each possible query. This information can be stored along with P within the same space bounds.

3.1.4. Computing the Block Types. All that is left now is to assign a *type* to each block that can be used for indexing into table P, i.e., we wish to find a surjection

$$t : \mathcal{A}_s \to \{0, \ldots, \hat{C}_s - 1\}, \text{ and } t(B_i) = t(B_j) \text{ iff } \mathcal{C}^{\text{sup}}(B_i) = \mathcal{C}^{\text{sup}}(B_j) \ , \tag{3.2}$$

where $\mathcal{A}_s$ is the set of arrays of size s.

A simple strategy (not meeting exactly the optimal space) would be as follows. There are six classes of nodes in the Super-Cartesian Tree: (1) leaves, (2) nodes with only a left edge, (3) nodes with only a red right edge, (4) nodes with only a blue right edge, (5) nodes with a left edge and a red right edge, and (6) nodes with a left edge and a blue right edge. These node classes can be encoded as a number of $\lceil \log 6 \rceil = 3$ bits. Writing these 3-bit numbers of the nodes in preorder gives a type-representation of a size-s block using $3s$ bits. Storing the types for all blocks would thus require $\frac{n}{s} \times 3s = 3n$ bits.[2] The rest of this section is devoted to improve on this memory requirement, and to give a more practical algorithm for computing these types, without the need to actually construct the trees.

Our strategy is to simulate the construction algorithm for Super-Cartesian Trees (Sect. 3.1.2), thereby simulating a walk along the corresponding path in the graph in Fig. 4. These paths can be enumerated as follows. First observe that the number of paths from an arbitrary node $\boxed{p\ q}$ to $\boxed{0\ 0}$ in the graph in Fig. 4 is given by the recurrence

$$\hat{C}_{00} = 1, \quad \hat{C}_{pq} = \begin{cases} \hat{C}_{p(q-1)} + \hat{C}_{(p-1)q} + \hat{C}_{(p-1)(q-1)}, & \text{if } 0 \leq p < q \neq 0 \\ \hat{C}_{p(q-1)} & \text{if } p = q \neq 0 \ , \end{cases} \tag{3.3}$$

and $\hat{C}_{pq} = 0$ otherwise. This follows from the fact that the number of paths from node $\boxed{p\ q}$ to $\boxed{0\ 0}$ is given by summing over the number of paths from each of the at most three cells that can be reached from $\boxed{p\ q}$ in a single step. Because the numbers $\hat{C}_{pq}$ generalize the Ballot Numbers in the same way as the Super-Catalan

[2]We are grateful to the anonymous reviewer who pointed out this type-representation. Additional space could be saved by using the 6 prefix-free codewords 00, 01, 100, 101, 110, and 111, and assigning the 2-bit codewords to the two most frequent node classes. Then the overall size would be $\frac{16}{6}n + o(n) \approx 2.666\,n + o(n)$ bits, the $o(n)$ term needed for marking the beginnings of blocks in the resulting bit stream [23].

```
Input: a block $B_j$ of size $s$
Output: the type $t(B_j)$ of $B_j$, as defined by Eq. (3.2)
1  Let $R$ be an array of size $s+1$          // simulates rightmost path of $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$
2  $R[1] \leftarrow -\infty$                  // stopper element on stack $R$
3  $q \leftarrow s, N \leftarrow 0, h \leftarrow 0$ // $h = \#$ horizontal edges on $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$'s rightmost path
4  for $i \leftarrow 1, \ldots, s$ do
5      while $R[q+i+h-s] > B_j[i]$ do
6          $N \leftarrow N + \hat{C}_{(s-i)q} + \hat{C}_{(s-i)(q-1)}$ // accounts for upwards move in Fig. 4
7          while $R[q+i+h-s-1] = R[q+i+h-s]$ do $h--$
8          $q--$
9      if $R[q+i+h-s] = B_j[i]$ then
10         $N \leftarrow N + \hat{C}_{(s-i)q}$      // accounts for diagonal move in Fig. 4
11         $h{+}{+}, q--$
12     $R[q+i+h-s+1] \leftarrow B_j[i]$            // push $B_j[i]$ on $R$
13 return $N$
```

FIGURE 5. An algorithm to compute the type of a block B_j.

Numbers generalize the Catalan Numbers, we call them *Super-Ballot Numbers*[3]. The first few resulting Super-Ballot Numbers, laid out such that they correspond to the nodes in Fig. 4, are

$$\begin{array}{llllll}
1 \\
1 & 1 \\
1 & 3 & 3 \\
1 & 5 & 11 & 11 \\
1 & 7 & 23 & 45 & 45 \\
1 & 9 & 39 & 107 & 197 & 197\,.
\end{array} \tag{3.4}$$

Due to this construction the Super Catalan Numbers appear on the rightmost diagonal of (3.4); in symbols, $\hat{C}_s = \hat{C}_{ss}$. Although we do not have a closed formula for our Super-Ballot Numbers, we can construct the $s \times s$-array at startup, by means of (3.3).

We are now ready to describe the algorithm in Fig. 5 which computes a function satisfying (3.2). It simulates the construction algorithm for Super-Cartesian Trees (Sect. 3.1.2), *without actually constructing them*! The general idea behind this is given by the bijection from Super-Cartesian Trees to paths from $\boxed{s\ s}$ to $\boxed{0\ 0}$ in the graph in Fig. 4, as explained in Sect. 3.1.3. By moving along this path, we count the number of paths that have been "skipped" when making an upwards

[3]A different generalization of the Ballot Numbers [12] also goes under the name "Super Ballot Numbers" and should not be confused with our concept.

(line 6) or diagonal move (line 10). At the beginning of step i of the outer for-loop, the position in the graph is $\boxed{(s-i+1)\ q}$, and h keeps the number of red edges on the rightmost path in $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$. Stack R keeps the nodes on the rightmost path of $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$, with $q+i+h-s$ pointing to R's top (i.e., to $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$'s rightmost leaf). The loop in line 7 simulates the traversal of red edges on $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$'s rightmost path by removing all elements that are equal to the top element on the stack. The if-statement in line 9 accounts for adding a new red edge to $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$, which is translated into a diagonal move in Fig. 4.

In total, if we denote by ℓ_i the number of iterations of the while-loop from line 5 to 8 in step i of the outer for-loop, ℓ_i equals the number of blue edges in $\mathcal{C}^{\text{sup}}_{i-1}(B_j)$ that are removed from the rightmost path. It follows from Sect. 3.1.3 that the algorithm in Fig. 5 correctly computes a function satisfying (3.2) in $O(s)$ time. Again, see Tbl. 1 for examples.

Thus, we store the type of each block (i.e., the number of its Super-Cartesian Tree in the above enumeration) in an array $T[0, n/s-1]$. The size of T is

$$|T| = \frac{n}{s}\lceil \log \hat{C}_s \rceil \le \frac{n}{s}(\log \rho^s + 1) = n \log \rho + \frac{n}{s} \approx 2.54311\, n + o(n) \text{ bits.}$$

3.2. Preprocessing for Long Queries

To complete the proof of Thm. 1.2, we now show how to precompute the perfect median of the minima in all queries that exactly span 2^j blocks or superblocks for all reasonable j (queries q_2–q_7 in Fig. 1). We note that also this problem is harder than in the case of "normal" RMQs [9], as it is now not obvious how to compute the median position of all minima in an interval I from the median positions in I's first and second half, respectively. Because the techniques are similar, we only show how to preprocess the superblocks; the reader can convince himself that the space-bounds for blocks are also within $o(n)$, provided that we store the positions of minima only relative to the beginning their superblock [9].

For each superblock B'_j we define a (temporary) array $X'_j[1, n'_j]$ (where $n'_j \le s'$ denotes the number of minima in block B'_j) such that $X'_j[i]$ holds the position of the i'th minimum in B'_j. The arrays X'_j are not represented explicitly; instead, we set up a *bit-vector* $D'_j[1, s']$ such that $D'_j[i]$ is 1 iff $B'_j[i]$ is a minimum in B_j. We prepare each D'_j for constant time select_1-operations, where $\text{select}_1(D'_j, i)$ returns the position of the i'th 1 in D'_j. Then

$$X'_j[i] = \text{select}_1(D'_j, i)\ .$$

We already note at this point that the D'_j's and their corresponding structures for select_1 are only auxiliary structures and can hence be deleted after the preprocessing for the long queries. The D'_j's take a total of $n/s' \cdot s' = n$ bits.[4] The

[4]As we saw in Sect. 3.1, the final data structure for the short queries uses more than n bits. Therefore if we first construct the data structures for the long queries, the temporary n bits for the D'_j's constitute no extra space at construction time, as this space can later be re-used for storing array T.

additional structures for constant time select_1-operations take additional $o(n)$ bits overall (if the D'_j's are concatenated into a single bit-vector of length n) and can be computed in $O(n)$ time using standard structures for rank/select [5, 14, 20].

We now define a table $M'[0, \frac{n}{s'}][0, \log(n/s')]$, where $M'[i][j]$ stores the *perfect* median position of all minima in the sub-array $A[is', (i+2^j)s'-1]$, i.e., in the sub-array covering all superblocks from B'_i to B'_{i+2^j-1}. These are exactly all possible sub-queries q_4 and q_5 in Fig. 1. Thus, to answer a query q_4 or q_5, we simply look up the corresponding table entry in M'. Because we also need to know the number of minima for each of the sub-queries q_4 and q_5, we also set up a table Y' of similar dimensions as M', such that $Y'[i][j]$ stores the number of minima in $A[is', (i+2^j)s'-1]$.

We wish to fill tables M' and Y' in a top-down manner, i.e., filling $M'[\cdot][j]$ and $Y'[\cdot][j]$ before $M'[\cdot][j+1]$ and $Y'[\cdot][j+1]$. To get started, initialize the 0'th column of Y' in $O(n)$ time by setting $Y'[i][0]$ to the number of minima in superblock B'_i for all $0 \le i \le \frac{n}{s'}$. Then initialize the 0'th column of M' with the position of the true median in the superblocks: $M'[i][0] = X'_j[\lceil Y'[i][0]/2 \rceil]$.

We now show how to fill entry i of tables M' and Y' on level $j > 0$, i.e., how to compute the value $M'[i][j]$ as the perfect median of all minima in $A[is', (i+2^j)s'-1]$, and $Y'[i][j]$ as the number of minima in this sub-array. Because of the order in which M' and Y' are filled, we proceed by splitting the interval into two smaller intervals of size $\xi := 2^{j-1}s'$.

Suppose there are $y_\ell := Y'[i][j-1]$ minima in the left half $A[is', is'+\xi-1]$, and $y_r := Y'[i+2^{j-1}][j-1]$ minima in the right half $A[is'+\xi, (i+2^j)s'-1]$. The easy case is when the overall minimum occurs only in one half, say the left one: then we can safely set $M[i][j]$ to $M[i][j-1]$, and $Y'[i][j]$ to y_ℓ.

The more difficult case is when the minimum occurs in both halves. The value $Y'[i][j]$ is simply set to $y_\ell + y_r$. We know that the true median has rank $r := \lceil (y_\ell + y_r)/2 \rceil$ among all minima in the interval. If $y_\ell \ge y_r$, we know that the true median *must* be in the left half, and that it must have rank r in there. If, on the other hand, $y_\ell < y_r$, the median must be in the right half, and must have rank $r' := r - y_\ell$ in there. In either case, we know in which half we have to look for a new minimum with a certain rank.

We recurse in this manner "upwards," until we reach level 0, where we can select the appropriate minimum from our X'_j-arrays (via D'_j). Due to the "height" of table M', the number of recursive steps is bounded by $O(\log(n/s'))$. In total, filling M' takes $O(\frac{n}{s'}\log(n/s')\log(n/s')) = O(n)$ time (recall $s' = \log^{2+\epsilon} n$).

The size of M' (and also that of Y') is $O(\frac{n}{s'} \times \log(n/s') \cdot \log n) = o(n)$ bits.

4. Approximation Scheme

In this section, we will show that median of minima can be approximated as stated in Thm. 1.3 provided that we have precomputed the r-quantiles ($r = O(1)$) instead

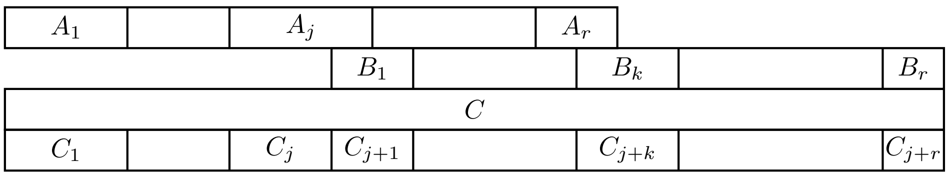


FIGURE 6. Estimating the number of minima of two overlapping subarrays based on their r-quantiles.

of medians (also known as 2-quantiles).[5] Let A be a totally ordered set, then the r-quantiles are a list $(\eta_1, \ldots, \eta_r)$ of r elements from A such that the rank of element η_i is $\lceil \frac{|A|\cdot i}{r} \rceil$. Note that η_r is almost the largest element of A. Furthermore, we have $|\{a \in A : \eta_{i-1} < a \leq \eta_i\}| \in \{\lfloor \frac{|A|}{r} \rfloor, \lceil \frac{|A|}{r} \rceil\}$ for each $i \in [1 : r]$ (here, η_0 is a fictive element smaller than each element in A).

4.1. Approximating the Number of Minima for Overlapping Ranges

First, we try to approximate the number of minima of the subarrays defined by the overlapping queries q_2 and q_3, q_4 and q_5, or q_6 and q_7, respectively, provided that the r-quantiles of the minima and the number of minima between two consecutive r-quantiles are given. Therefore it is sufficient to show how to approximate the number of minima in two overlapping arrays A and B. We assume that the arrays are partitioned into $(A_1, \ldots, A_r)$ as well as $(B_1, \ldots, B_r)$ by the positions of the r-quantiles, respectively (cf. Fig. 6). The number of minima in A_i and B_i is denoted by ν_i and μ_i, respectively. Let $\nu = \sum_{i=1}^{r} \nu_i$ and $\mu = \sum_{i=1}^{r} \mu_i$ the number of minima in A and B, respectively. By the definition of r-quantiles, we know that $\nu_i \in \{\lfloor \frac{\nu}{r} \rfloor, \lceil \frac{\nu}{r} \rceil\}$ and $\mu_i \in \{\lfloor \frac{\mu}{r} \rfloor, \lceil \frac{\mu}{r} \rceil\}$. Furthermore, let C be the array with the index set given by the union of the index sets of subarrays A and B (cf. Fig. 6).

First, we observe that the number of minima in the joined subarray C is exactly known, if the minima in the two subarrays differ. Thus, we assume in the following that the minimum in subarray A is equal to the minimum of subarray B.

Let j be the largest index such that A_j is not a subarray of B and let k be the smallest index such that B_k is not a subarray of A (cf. Figure 6). Further, we assume without loss of generality that $\nu \leq \mu$. Then $x := \sum_{i=1}^{j-1} \nu_i + \mu$ is a lower bound and $\sum_{i=1}^{j} \nu_i + \mu = x + \nu_j$ is an upper bound on the number of minima in the array C. In what follows, we show that x is a $(1 + 2/r)$-approximation of the number of minima in C. Because $x + \nu_j$ is an upper bound on the number of minima, it is sufficient to show that $x + \nu_j \leq (1 + \frac{2}{r})x$, which is equivalent to $\nu_j \leq \frac{2}{r}x$. Since

$$\nu_j \leq \left\lceil \frac{\nu}{r} \right\rceil \leq \left\lceil \frac{\mu}{r} \right\rceil \leq \frac{2}{r}\mu \leq \frac{2}{r}\left(\sum_{i=1}^{j-1} \nu_i + \mu\right) = \frac{2}{r}x \ ,$$

[5]It is easy to verify that the algorithm described in Sect. 3 can be modified to precompute these r-quantiles, and that only the space of the $o(n)$-structures is multiplied by r.

we are done. Note that $\lceil \frac{\mu}{r} \rceil \leq \frac{2}{r}\mu$ could only be wrong if $\mu < r$. But in that case, we know the positions of all minima in the subarray B (and, of course, also in A, since $\nu \leq \mu$). The number of minima in array C can then be easily computed in time $O(r)$, which is constant.

Furthermore, the (conceptual) array C is partitioned into $j + r$ subarrays $C_1, \ldots, C_{j+r}$ (cf. Fig. 6). The subarrays $C_1, \ldots, C_{j-1}$ and $C_{j+1}, \ldots, C_{j+r}$ have the same index set as $A_1, \ldots, A_{j-1}$ and $B_1, \ldots, B_r$, respectively, and subarray C_j is just a prefix of A_j. The last element in each C_i that is a minimum in C is called a *pivot*. Note that the last element in each C_i (except for C_j) is an r-quantile in either A or B. Thus, subarray C_j may not have a pivot. The number of minima in C_i is ν_i for $i \in [1 : j-1]$ and μ_{i-j} for $i \in [j+1 : j+r]$. For the subarray C_j, we know that the number of minima is at least 0 and at most $\lceil \frac{\nu}{r} \rceil \leq \lceil \frac{\mu}{r} \rceil$. Let $\lambda(C_i)$ denote this lower bound on the number of minima in subarray C_i.

4.2. Approximating the Median of Minima

It remains to find an approximation of the median of minima in the union of the five disjoint subarrays A_1, A_2, A_3, A_4, and A_5 defined by the queries q_1, q_2 and q_3, q_4 and q_5, q_6 and q_7, and q_8, respectively. For the total minimum of these 5 subarrays, let m_i denote the exact number of minima in A_i. From Sect. 4.1, we get a lower bound n_i on the number of minima in A_i, with $n_i \leq m_i \leq (1 + \frac{2}{r})n_i$.[6] Let $n := \sum_{i=1}^{5} n_i$ be the lower bound on the total number of minima, and let $m := \sum_{i=1}^{5} m_i$ be the exact number of minima in the whole array.

In what follows, we describe how the approximate median of minima is chosen and compute its exact rank in C. We first select the subarray A_ℓ fulfilling $\sum_{i=1}^{\ell-1} n_i < \lceil n/2 \rceil \leq \sum_{i=1}^{\ell} n_i$. Then we select the pivot of C_p in subarray A_ℓ where $\sum_{i=1}^{p-1} \lambda(C_i) < \lceil n/2 \rceil - \sum_{i=1}^{\ell-1} n_i \leq \sum_{i=1}^{p} \lambda(C_i)$. Here $C_1, \ldots, C_j, \ldots, C_{j+r}$ is the partition of A_ℓ as described in Sect. 4.1, and $\lambda(C_i)$ denotes the known lower bound on the number of minima in C_i, which is also an upper bound except for C_j. By construction, the actual rank of the selected element is at most

$$\left(1 + \frac{2}{r}\right) \cdot \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n_j}{r} \right\rceil + \left\lceil \frac{m}{r} \right\rceil \ .$$

If $n < m$, we have $\lceil \frac{n}{2} \rceil \leq \frac{m}{2}$. Using $n_j \leq m$, we get

$$\left(1 + \frac{2}{r}\right) \cdot \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{n_j}{r} \right\rceil + \left\lceil \frac{m}{r} \right\rceil \leq \left(1 + \frac{2}{r}\right) \cdot \frac{m}{2} + \frac{2m+2}{r} \leq \left(\frac{1}{2} + \frac{4}{r}\right) \cdot m \ .$$

On the other hand, if $n = m$, then our lower bound on the number of minima is tight and we get exact numbers. Thus, the rank of the selected minimum is at most $\lceil \frac{n}{2} \rceil + \lceil \frac{n_j}{r} \rceil$. If $n = m$ and $m \geq r$, then the selected minimum has a rank of

[6]For subarrays A_1 and A_5, we actually know the *exact* numbers of minima, because these are stored along with table P (see end of Sect. 3.1.3).

at most

$$\left\lceil \frac{m}{2} \right\rceil + \left\lceil \frac{m_j}{r} \right\rceil \le \frac{m}{2} + \frac{1}{2} + \frac{m+r-1}{r} \le \left(\frac{1}{2} + \frac{5}{2r}\right) \cdot m \le \left(\frac{1}{2} + \frac{3}{r}\right) \cdot m \ .$$

If $n = m$ and $m < r$, then all minima are known, and the exact median of minima can easily be selected.

Since we know that at least $\lceil \frac{n}{2} \rceil \ge \frac{n}{2}$ elements are smaller or equal to the selected minimum and that $\frac{m}{2} \le \left(1 + \frac{2}{r}\right) \frac{n}{2}$, the rank of the chosen minimum is at least $\left(1 + \frac{2}{r}\right)^{-1} \cdot \frac{m}{2}$. A brief calculation shows then that the rank of the chosen element is at least

$$\frac{1}{1 + \frac{2}{r}} \cdot \frac{m}{2} \ge \left(1 - \frac{2/r}{1 + 2/r}\right) \cdot \frac{m}{2} = \left(1 - \frac{2}{r+2}\right) \cdot \frac{m}{2} = \left(\frac{1}{2} - \frac{1}{r+2}\right) \cdot m \ .$$

Choosing $\alpha = \frac{4}{r}$, we get an approximation of the median of minima, where its rank is contained in $[(\frac{1}{2} - \alpha)m, (\frac{1}{2} + \alpha)m]$. This proves Thm. 1.3.

5. Applications in Compressed Suffix Trees and Arrays

5.1. Improvements in Compressed Suffix Trees

It is well-known that RMQs in general are a versatile tool for many string matching tasks. The most important application of this kind is to preprocess the array containing the lengths of the *longest common prefixes* [1] of lexicographically adjacent suffixes (LCP-array for short) for constant-time range minimum queries. Then the longest common prefix between *arbitrary* suffixes can be found in constant time. This, in turn, can be used for many tasks in approximate and exact string matching, and also for navigational operations in (compressed) suffix trees, such as computing so-called *suffix links*, when the suffix tree is only represented *implicitly* by the suffix- and LCP-array [1].

The most fundamental navigational operation, however, is to locate the outgoing edge of a node v that is labeled with a given character $c \in \Sigma$ (Σ is the alphabet), called *getChild*(v, c). It can be shown [9, 24] that one can retrieve all outgoing edges by performing subsequent queries of the form $\text{RMQ}_{\mathsf{LCP}}(\ell, r)$, where the query indices ℓ and r are the interval in LCP that represent node v [1]. If the RMQs return the leftmost minimum, then *getChild*(v, c) takes $O(|\Sigma|)$ time, as in the worst case all $|\Sigma|$ minima have to be visited.

To speed up this search, Sadakane [24] proposes to use RMQ^{med} instead of plain RMQs in his Compressed Suffix Tree (CST), such that the interval $[\ell, r]$ can be binary-searched in $O(\log |\Sigma|)$ time. However, he does not give the details how the structures for RMQ^{med} can be constructed efficiently, and a naive approach would give $O(n^2)$ construction time. As the construction times of the other structures in the CST are at most $O(n \log^{\delta} n)$ for any constant $0 < \delta \le 1$ [28], in order to maintain this time bound it is important to have an efficient preprocessing algorithm for range median of minima queries — a gap that we close in this paper.

Lemma 5.1. *There is a linear-time preprocessing scheme such that the number of search steps performed by getChild(v, c) in Compressed Suffix Trees is* $\log_{\frac{16}{15}} |\Sigma| \approx 10.74 \log |\Sigma|$ *in the worst case.*

We mention that the constant 1/12 in Lemma 5 of Sadakane's paper [24] should also be 1/16, as there the queries are also decomposed into 8 different subqueries.[7] We also remark that the $2.54\,n$ bits from our array T (Sect. 3.1.4) are *not* necessary in the CST, but only our tables M and M' for the long queries (Sec. 3.2): The balanced parentheses sequence (using $4n$ bits) describing the suffix tree topology can be re-used to index into the table of precomputed in-block-queries (our P), as a length-s sequence of parentheses uniquely describes the distribution of the minima of the corresponding part in the depth sequence [24, Sect. 4.2–3].

We note two further applications of our new scheme. First, as RMQ$^{\text{med}}$ completely substitutes the so-called *child table* in Abouelhoda et al.'s *Enhanced Suffix Array* [1] and in Kim et al.'s Compressed Suffix Tree [16], we also improve on their space consumption, as our structures are much smaller than the child table. Further, our preprocessing from Sect. 3.2 is also necessary for Fischer et al.'s entropy bounded CST [10].

5.2. Pattern Matching in (Compressed) Suffix Arrays

For a given pattern P of length m, the most common task in pattern matching is to check whether P is a substring of T. We now show that our scheme for RMQ$^{\text{med}}$ leads to a $O(t_{\mathsf{SA}} m \log |\Sigma|)$ search-method in suffix arrays, where t_{SA} is the time to access an element from the *suffix array* [24, Tbl. 1]. This method is simply obtained by calling *getChild* subsequently for all characters in P, each time invoking an evaluation of the suffix array, hence the additional factor t_{SA}.

Let us first consider uncompressed suffix arrays, where $t_{\mathsf{SA}} = O(1)$. The best results on pattern matching in uncompressed suffix arrays are due to Kim and Park [16], who give a succinct version of Kim et al.'s data structure [15] that allows $O(m \log |\Sigma|)$-time pattern searches. This table [16] requires $5n + o(n)$ bits of space; with our scheme for RMQ$^{\text{med}}$, this is reduced to $\approx 2.54\,n + o(n)$ bits, a space reduction by a factor of about 2. Given the practical importance of pattern matching, this is not negligible. Our new technique can also be used to get an $O(m \log |\Sigma|)$-pattern search-algorithm in suffix arrays on words [6].

Naturally, our scheme for RMQ$^{\text{med}}$ can also be combined with compressed versions of the suffix array. For example, combining Grossi and Vitter's Compressed Suffix Array [13, Thm. 2] with our search strategy, where $t_{\mathsf{SA}} = O(\log^{\alpha}_{|\Sigma|} n)$ with constant $0 < \alpha \leq 1$, matching takes $O(\log^{\alpha}_{|\Sigma|}(n) m \log |\Sigma|)$ time, for *any* alphabet size $|\Sigma|$, while needing only $\alpha^{-1} H_0 n + O(n)$ bits in total (H_0 being the empirical order-0 entropy of the input text [18]). Although this cannot be directly compared to more recent advances in compressed text indexing, such as Ferragina et al.'s

[7]Confirmed by K. Sadakane (personal communication, June 2007).

Alphabet-Friendly FM-Index [7], it is interesting to note that the size of our structure for RMQ$^{\text{med}}$ is independent of the alphabet size, which stands in contrast to all other structures in compressed suffix arrays [21].

5.3. More Applications

We finally mention two recent developments that make use of the techniques introduced in this article. Navarro and Sadakane [26, Sect. 7] give a new data structure for arbitrary dynamic trees, using $2n + o(n)$ bits of space, and supporting almost all operations, including insertion and deletion of nodes, in sub-logarithmic time. The Super-Cartesian Tree is used as a main tool for this data structure. Also, Gog and Ohlebusch [22] have developed a new compressed suffix tree, which is entirely based on the Super-Cartesian Tree.

References

[1] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. Discrete Algorithms*, 2(1):53–86, 2004.

[2] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe. Nearest common ancestors: A survey and a new distributed algorithm. In *Proc. SPAA*, pages 258–264. ACM Press, 2002.

[3] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[4] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[5] D. R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[6] P. Ferragina and J. Fischer. Suffix arrays on words. In *Proc. CPM*, volume 4580 of *LNCS*, pages 328–339. Springer, 2007.

[7] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):Article No. 20, 2007.

[8] J. Fischer and V. Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Proc. CPM*, volume 4009 of *LNCS*, pages 36–48. Springer, 2006.

[9] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *Proc. ESCAPE*, volume 4614 of *LNCS*, pages 459–470. Springer, 2007.

[10] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proc. CPM*, volume 5029 of *LNCS*, pages 152–165. Springer, 2008.

[11] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proc. STOC*, pages 135–143. ACM Press, 1984.

[12] I. M. Gessel. Super ballot numbers. *J. Symbolic Computation*, 14(2–3):179–194, 1992.

[13] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM J. Comput.*, 35(2):378–407, 2005.

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

[15] D. K. Kim, J. E. Jeon, and H. Park. An efficient index data structure with the capabilities of suffix trees and suffix arrays for alphabets of non-negligible size. In *Proc. SPIRE*, volume 3246 of *LNCS*, pages 138–149. Springer, 2004.

[16] D. K. Kim and H. Park. A new compressed suffix tree supporting fast search and its construction algorithm using optimal working space. In *Proc. CPM*, volume 3537 of *LNCS*, pages 33–44. Springer, 2004.

[17] D. K. Kim, J. S. Sim, H. Park, and K. Park. Constructing suffix arrays in linear time. *J. Discrete Algorithms*, 3(2–4):126–142, 2005.

[18] G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

[19] D. Merlini, R. Sprugnoli, and M. C. Verri. Waiting patterns for a printer. *Discrete Applied Mathematics*, 144(3):359–373, 2004.

[20] J. I. Munro. Tables. In *Proc. FSTTCS*, volume 1180 of *LNCS*, pages 37–42. Springer, 1996.

[21] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):Article No. 2, 2007.

[22] E. Ohlebusch and S. Gog. A compressed enhanced suffix array supporting fast string matching. Submitted, 2009.

[23] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *ACM Transactions on Algorithms*, 3(4):Article No. 43, 2007.

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

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

[26] K. Sadakane and G. Navarro. Fully-functional static and dynamic succinct trees. CoRR, abs/0905.0768v1, 2009.

[27] R. P. Stanley. *Enumerative Combinatorics*, volume 2. Cambridge University Press, 1999.

[28] N. Välimäki, W. Gerlach, K. Dixit, and V. Mäkinen. Engineering a compressed suffix tree implementation. In *Proceeding of the 6th Workshop on Experimental Algorithms (WEA 2007)*, LNCS 4525, pages 217–228. Springer-Verlag, 2007.

[29] J. Vuillemin. A unifying look at data structures. *Comm. ACM*, 23(4):229–239, 1980.

Johannes Fischer
Center for Bioinformatics (ZBIT), Universität Tübingen,
Sand 14, 72076 Tübingen, Germany
e-mail: `fischer@informatik.uni-tuebingen.de`

Volker Heun
Institut für Informatik, Ludwig-Maximilians-Universität München,
Amalienstr. 17, 80333 München, Germany
e-mail: `Volker.Heun@bio.ifi.lmu.de`