Compressed Range Minimum Queries*

Seungbum Jo[1], Shay Mozes[2], and Oren Weimann[3]

[1] University of Haifa
seungbum.jo@uni-siegen.de
[2] Interdisciplinary Center Herzliya
smozes@idc.ac.il
[3] University of Haifa
oren@cs.haifa.ac.il

Abstract. Given a string S of n integers in $[0, \sigma)$, a range minimum query $\mathsf{RMQ}(i, j)$ asks for the index of the smallest integer in $S[i \dots j]$. It is well known that the problem can be solved with a succinct data structure of size $2n + o(n)$ and constant query-time. In this paper we show how to preprocess S into a *compressed representation* that allows fast range minimum queries. This allows for *sublinear* size data structures with logarithmic query time. The most natural approach is to use string compression and construct a data structure for answering range minimum queries directly on the compressed string. We investigate this approach using grammar compression. We then consider an alternative approach. Even if S is not compressible, its Cartesian tree necessarily is. Therefore, instead of compressing S using string compression, we compress the Cartesian tree of S using tree compression. We show that this approach can be exponentially better than the former, and is never worse by more than an $O(\sigma)$ factor (i.e. for constant alphabets it is never asymptotically worse).

1 Introduction

Given a string S of n integers in $[0, \sigma)$, a range minimum query $\mathsf{RMQ}(i, j)$ returns the index of the smallest integer in $S[i \dots j]$. A range minimum data structure consists of a preprocessing algorithm and a query algorithm. The preprocessing algorithm takes as input the string S, and constructs the data structure, whereas the query algorithm takes as input the indices i, j and, by accessing the data structure, returns $\mathsf{RMQ}(i, j)$. The range minimum problem is one of the most fundamental problems in stringology, and as such has been extensively studied, both in theory and in practice (see e.g. [11] and references therein).

Range minimum data structures fall into two categories. *Systematic* data structures store the input string S, whereas *non-systematic* data structures do not. A significant amount of attention has been devoted to devising RMQ data structures that answer queries in constant time and require as little space as possible. There are succinct systematic structures that answer queries in constant

* Supported in part by Israel Science Foundation grant 592/17

time and require fewer than $2n$ bits in addition to the $n \log \sigma$ bits required to represent S [11]. Similarly, there are succinct non-systematic structures that answer queries in constant time, and require $2n + o(n)$ bits [8, 11].

The *Cartesian tree* $\mathcal{C}$ of S is a rooted ordered binary tree with n nodes. It is defined recursively. Let i be the index of the smallest element of S (if the smallest element appears multiple times in S, let i be the first such appearance). The Cartesian tree of S is composed of a root node whose left subtree is the Cartesian tree of $S[1, i-1]$, and whose right subtree is the Cartesian tree of $S[i+1, n]$. See Fig. 1. By definition, the character $S[i]$ corresponds to the i'th node in an inorder traversal of $\mathcal{C}$ (we will refer to this node as node i). Furthermore, for any nodes i and j in $\mathcal{C}$, their lowest common ancestor $\mathsf{LCA}(i, j)$ in $\mathcal{C}$ corresponds to $\mathsf{RMQ}(i, j)$ in S. It follows that the Cartesian tree of S completely characterizes S in terms of range minimum queries. Indeed, two strings return the same answers for all possible range minimum queries if and only if their Cartesian trees are identical. This well known property has been used by many RMQ data structures including the succinct structures mentioned above. Since there are $2^{2n-O(\log n)}$ distinct rooted binary trees with n nodes, there is an information theoretic lower bound of $2n - O(\log n)$ bits for RMQ data structures. In this sense, the above mentioned $2n + o(n)$ bits data structures [8, 11] are nearly optimal.

1.1 Our results and techniques

In this work we present RMQ data structures whose size can be *sublinear* in the size of the input string that answer queries in $O(\log n)$ time. This is achieved by using compression techniques, and developing data structures that can answer RMQ/LCA queries directly on the compressed objects. Since we aim for sublinear size data structures, we focus on non-systematic data structures. We consider two different approaches to achieve this goal. The first approach is to use string compression to compress S, and devise an RMQ data structure on the compressed representation. This approach has also been suggested in [1, Section 7.1] in the context of compressed suffix arrays. See also [8, Theorem 2], [11, Theorem 4.1], and [3] for steps in this direction. The second approach is to use tree compression to compress the Cartesian tree $\mathcal{C}$, and devise an LCA data structure on the compressed representation. To the best of our knowledge, this is the first time such approach has been suggested. Note that the two approaches are not equivalent. For example, consider a sorted sequence of an arbitrary subset of n different integers from $[1, 2n]$. As a string this sorted sequence is not compressible, but its Cartesian tree is an (unlabeled) path, which is highly compressible. In a nutshell, we show that the tree compression approach can exponentially outperform the string compression approach. Furthermore, it is never worse than the string compression approach by more than an $O(\sigma)$ factor. We next elaborate on these two approaches.

Using string compression. In Section 2.1, we show how to answer range minimum queries on a *grammar compression* of the input string S. A grammar compression is a context-free grammar that generates only S. The grammar is represented as a *straight line program* (SLP) $\mathcal{S}$. I.e., the right-hand side of each rule in $\mathcal{S}$ either consists of the concatenations of two non-terminals or of a single terminal symbol. The size $|\mathcal{S}|$ of the SLP $\mathcal{S}$ is defined as the number of rules in $\mathcal{S}$. Ideally, $|\mathcal{S}| \ll |S|$. Computing the smallest possible SLP is NP-hard [7], but there are many theoretically and practically efficient compression schemes for constructing $\mathcal{S}$ [7, 12, 13, 15] that reasonably approximate the optimal SLP. In particular, Rytter [14] showed an SLP $\mathcal{S}$ of depth $\log n$ (the depth of an SLP is the depth of its parse tree) whose size is larger than the optimal SLP by at most a multiplicative $\log n$ factor.

In [1], it was shown how to support range minimum queries on S with a data structure of size $O(|\mathcal{S}|)$ in time proportional to the depth of the SLP $\mathcal{S}$. Bille et al. [6] designed a data structure of size $O(|\mathcal{S}|)$ that supports random-access to S (i.e. retrieve the i'th symbol in S) in $O(\log n)$ time (i.e. regardless of the depth of the SLP $\mathcal{S}$). We show how to simply augment their data structure within the same $O(|\mathcal{S}|)$ size bound to answer range minimum queries in $O(\log n)$ time (i.e., how to avoid the logarithmic overhead incurred by using the solution of [1] on Rytter's SLP).

Theorem 1. *Given a string S of length n and an SLP-grammar compression $\mathcal{S}$ of S, there is a data structure of size $O(|\mathcal{S}|)$ that answers range minimum queries on S in $O(\log n)$ time.*

Using tree compression. In Section 2.2, we give a data structure for answering LCA queries on a compressed representation of the Cartesian tree $\mathcal{C}$. By the discussion above, this is equivalent to answering range minimum queries on S. We use DAG compression of the top-tree of the Cartesian tree $\mathcal{C}$ of S. We now explain these concepts.

A *top-tree* [2] of a tree T is a hierarchical decomposition of the edges of T into clusters. Each cluster is a connected subgraph of T with the property that any two crossing clusters (i.e., clusters whose intersection is nonempty and neither cluster contains the other) share at most two vertices; the root of the cluster (called the top boundary node) and a leaf of the cluster (called a bottom boundary node). Such a decomposition can be described by a rooted ordered binary tree $\mathcal{T}$, called a top-tree, whose leaves correspond to clusters with individual edges of T, and whose root corresponds to the entire tree T. The cluster corresponding to a non-leaf node of $\mathcal{T}$ is obtained from the clusters of its two children by either identifying their top boundary nodes (horizontal merge) or by identifying the top boundary node of the left child with the bottom boundary node of the right child (vertical merge). See Fig. 1.

A *DAG compression* [9] of a tree T is a representation of T by a DAG whose nodes correspond to nodes of T. All nodes of T with the same subtree are represented by the same node of the DAG. Thus, the DAG has two sinks, corresponding to the two types of leaf nodes of T (a single edge cluster, either left or right), and a single source, corresponding the root of T. If u is the parent of ℓ and r in T, then the node in the DAG representing the subtree of T rooted at u has edges leading to the two nodes of the DAG representing the subtree of

T rooted at ℓ and the subtree of T rooted at r. Thus, repeating rooted subtrees in T are represented only once in the DAG. See Fig. 1.

A *top-tree compression* [5] of a tree T is a DAG compression of T's top-tree $\mathcal{T}$. Bille et al. [5] showed how to construct a data structure whose size is linear in the size of the DAG of $\mathcal{T}$ and supports navigational queries on T in time linear in the depth of $\mathcal{T}$. In particular, given the preorder numbers of two vertices u, v in T, their data structure can return the preorder number of $\mathsf{LCA}(u, v)$ in T. We show that their data structure can be easily adjusted to work with inorder numbers instead of preorder, so that, given the inorder numbers i, j of two vertices in T one can return the inorder number of $\mathsf{LCA}(i, j)$ in T. This is precisely $\mathsf{RMQ}(i, j)$ when T is taken to be the Cartesian tree $\mathcal{C}$ of S.

Theorem 2. *Given a string S of length n and a top-tree compression $\mathcal{T}$ of the Cartesian tree $\mathcal{C}$, there is a data structure of size $O(|\mathcal{T}|)$ that answers range minimum queries on S in $O(\texttt{depth}(\mathcal{T}))$ time.*

By combining Theorem 2 with the greedy construction of $\mathcal{T}$ given in [5] (in which $\texttt{depth}(\mathcal{T}) = O(\log n)$), we can obtain an $O(|\mathcal{T}|)$ space data structure that answers RMQ in $O(\log n)$ time.

We already mentioned that, on some RMQ instances, top-tree compression can be much better than any string compression technique. As an example, consider the string $S = 123 \cdots n$. Its Cartesian tree is a single (rightmost, and unlabeled) path, which compresses using top-tree compression into size $|\mathcal{T}| = O(\log n)$. On the other hand, since $\sigma = n$, S is uncompressible with an SLP. By Theorem 2, this shows that the tree compression approach to the RMQ problem can be exponentially better than the string compression approach. In fact, for any string over an alphabet of size $\sigma = \Omega(n)$, any SLP must have $|\mathcal{S}| = \Omega(n)$ while for top-trees $|\mathcal{T}| = O(n/\log n)$ [5]. In Section 3 we show that, for small alphabets, $\mathcal{T}$ cannot be much larger nor much deeper than $\mathcal{S}$ for any SLP $\mathcal{S}$.

Theorem 3. *Given a string S of length n over an alphabet of size σ, for any SLP-grammar compression $\mathcal{S}$ of S there is a top-tree compression $\mathcal{T}$ of the Cartesian tree $\mathcal{C}$ with size $O(|\mathcal{S}| \cdot \sigma)$ and depth $O(\texttt{depth}(\mathcal{S}) \cdot \sigma)$.*

Plugging Rytter's [14] SLP into Theorem 3 shows that, at least for small alphabets σ, the top-tree compression approach to RMQ is never far worse than the SLP approach.

Corollary 1. *Given a string S of length n over an alphabet of size σ, let $\mathcal{S}$ denote the smallest possible SLP-grammar compression of S. There is a top-tree compression $\mathcal{T}$ of the Cartesian tree $\mathcal{C}$ of S with size at most $|\mathcal{T}| = \min(O(n/\log n), O(|\mathcal{S}| \cdot \sigma))$, and there is a data structure of size $O(|\mathcal{T}|)$ that answers range minimum queries on S in $O(\log n \cdot \log \sigma)$ time.*

2 RMQ on Compressed Representations

2.1 Compressing the string

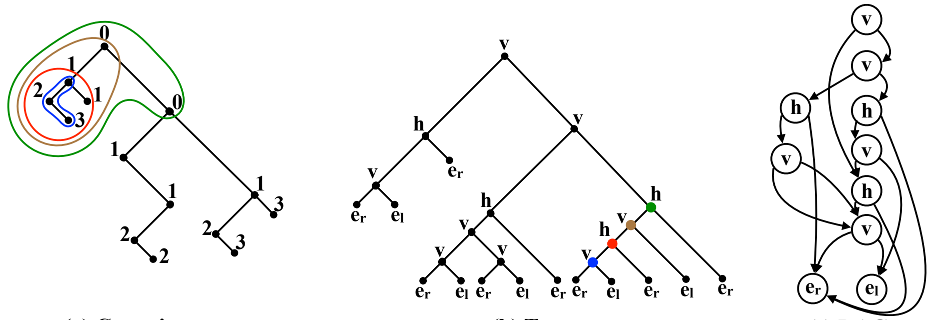
Given an SLP compression $\mathcal{S}$ of S, Bille et al. [6] presented a data structure of size $O(|\mathcal{S}|)$ that can report any $S[i]$ in $O(\log n)$ time. The proof of Theorem 1 is a rather straightforward extension of this data structure to support range minimum queries.

The key technique used in [6] is an efficient representation of the *heavy path decomposition* of the SLP's parse tree. For each node v in the parse tree, we select the child of v that derives the longer string to be a *heavy* node. The other child is *light*. Heavy edges are edges going into a heavy node and light edges are edges going into a light node. The heavy edges decompose the parse tree into *heavy paths*. The number of light edges on any path from a node v to a leaf is $O(\log |v|)$ where $|v|$ denotes the length of the string derived from v. A traversal of the parse tree from its root to the i'th leaf $S[i]$ enters and exists at most $\log n$ heavy paths. Bille et al. show how to simulate this traversal in $O(\log n)$ time on a representation of the heavy path decomposition that uses only $O(|\mathcal{S}|)$ space (note that we cannot afford to store the entire parse tree as its size is n which can be exponentially larger than $|\mathcal{S}|$). We do not go into the internals of their representation but it is important to note that for each heavy path P encountered during the traversal their structure computes the total size (number of leaves) of all subtrees hanging with light edges from the left (respectively right) of P between the entry point and exit point in P. This is achieved with a binary search tree (called an *interval biased search tree*) that ensures that collecting these values (as well as finding the entry and exit points) on all encountered heavy paths telescopes to a total of $O(\log n)$ time (rather than $O(\log^2 n)$).

In order to extend their structure to support range minimum queries we need only the following two changes: (1) in the interval biased search tree, apart from storing for each node the number of leaves in its subtree, we also store the location of the minimum value leaf. This means that apart from accumulating subtree sizes we can also compare their minimums. (2) for each heavy path in their representation we add a standard linear-space constant query-time RMQ data structure [4] over the left (respectively right) hanging subtree minimums. This RMQ structure will be queried only on the unique heavy path containing the lowest common ancestor of the i'th and j'th leaves in the parse tree.

2.2 Compressing the Cartesian tree

We next prove Theorem 2, i.e. how to support range minimum queries on S using a compressed representation of the *Cartesian tree* [16]. Recall that the Cartesian tree $\mathcal{C}$ of S is defined as follows: If the smallest character in S is $S[i]$ (in case of a tie we choose a leftmost position) then the root of $\mathcal{C}$ corresponds to $S[i]$, its left child is the Cartesian tree of $S[1, i-1]$ and its right child is the Cartesian tree of $S[i+1, n]$. By definition, the i'th character in S corresponds to the node in $\mathcal{C}$ with inorder number i (we will refer to this node as node i).



(a) Cartesian tree **(b) Top-tree** **(c) DAG**

Fig. 1. The string S = "23110122102313" and its corresponding (a) Cartesian tree , (b) top-tree, and (c) DAG representation of the top-tree. In (a), each node is labeled by its corresponding character in S (these labels are for illustration only, the top-tree construction treats the Cartesian tree as an unlabeled tree). In (b) and (c), each node is labeled by e_l or e_r (atomic edge clusters), v (a vertical merge), or h (a horizontal merge). Four clusters are marked with matching colors in (a) and in (b).

Observe that for any nodes i and j in $\mathcal{C}$, the lowest common ancestor $\mathsf{LCA}(i,j)$ of these nodes in $\mathcal{C}$ corresponds to $\mathsf{RMQ}(i,j)$ in S. This implies that without storing S explicitly, one can answer range minimum queries on S by answering LCA queries on $\mathcal{C}$. In this section, we show how to support LCA queries on $\mathcal{C}$ on a *top-tree* compression [5] $\mathcal{T}$ of $\mathcal{C}$. The query time is $O(\texttt{depth}(\mathcal{T}))$ which can be made $O(\log n)$ using the (greedy) construction of Bille et al. [5] that gives $\texttt{depth}(\mathcal{T}) = O(\log n)$. We first briefly restate the construction of Bille et al., and then extend it to support LCA queries.

The *top-tree* of a tree T (in our case T will be the Cartesian tree $\mathcal{C}$) is a hierarchical decomposition of T into *clusters*. Let v be a node in T with children v_1, v_2.[4] Define $T(v)$ to be the subtree of T rooted at v. Define $F(v)$ to be the forest $T(v)$ without v. A *cluster* with *top boundary node* v can be either (1) $T(v)$, (2) $\{v\} \cup T(v_1)$, or (3) $\{v\} \cup T(v_2)$. For any node $u \neq v$ in a cluster with top boundary node v, deleting from the cluster all descendants of u (not including u itself) results in a *cluster* with *top boundary node* v and *bottom boundary node* u. The top-tree is a binary tree defined as follows (see Fig. 1):

- The root of the top-tree is the cluster T itself.
- The leaves of the top-tree are (atomic) clusters corresponding to the edges of T. An edge $(v, \mathit{parent}(v))$ of T is a cluster where $\mathit{parent}(v)$ is the top boundary node. If v is a leaf then there is no bottom boundary node, otherwise v is a bottom boundary node. If v is the right child of $\mathit{parent}(v)$ then we label the $(v, \mathit{parent}(v))$ cluster as e_r and otherwise as e_ℓ.

[4] Bille et al. considered trees with arbitrary degree, but since our tree T is a Cartesian tree we can focus on binary trees.

– Each internal node of the top-tree is a *merged* cluster of its two children. Two edge disjoint clusters A and B whose nodes overlap on a single boundary node can be merged if their union $A \cup B$ is also a cluster (i.e. contains at most two boundary nodes). If A and B share their top boundary node then the merge is called *horizontal*. If the top boundary node of A is the bottom boundary node of B then the merge is called *vertical* and in the top-tree A is the left child and B is the right child.

Bille et al. [5] proposed a greedy algorithm for constructing the top-tree: Start with n identical clusters, one for each edge of T, and at each step merge all possible clusters. More precisely, at each step, first do all possible horizontal merges and then do all possible vertical merges. After constructing the top-tree, the actual compression $\mathcal{T}$ is obtained by representing the top-tree as a directed acyclic graph (DAG) using the algorithm of [9]. Namely, all nodes in the top-tree that have a child with subtree X will point to the same subtree X (see Fig. 1). Bille et al. [5] showed that using the above greedy algorithm, one can construct $\mathcal{T}$ of size $|\mathcal{T}|$ that can be as small as $\log n$ (when the input tree T is highly repetitive) and in the worst-case is at most $O(n/\log_\sigma^{0.19} n)$. Dudek and Gawrychowski [10] have recently improved the worst-case bound to $O(n/\log_\sigma n)$ by merging in the i'th step only clusters whose size is at most α^i for some constant α. Using either one of these merging algorithms to obtain the top-tree and its DAG representation $\mathcal{T}$, a data structure of size $O(|\mathcal{T}|)$ can then be constructed to support various queries on T. In particular, given nodes i and j in T (specified by their position in a *preorder* traversal of T) Bille et al. showed how to find the (preorder number of) node $\mathsf{LCA}(i,j)$ in $O(\log n)$ time. Therefore, the only change required in order to adapt their data structure to our needs is the representation of nodes by their *inorder* rather than preorder numbers.

The *local preorder number* u_C of a node u in T and a cluster C in $\mathcal{T}$ is the preorder number of u in a preorder traversal of the cluster C. To find the preorder number of $\mathsf{LCA}(i,j)$ in $O(\log n)$ time, Bille et al. showed it suffices if for any node u and any cluster C we can compute u_C in constant time from u_A or u_B (the local preorder numbers of u in the clusters A and B whose merge is the cluster C) and vice versa. In Lemma 6 of [5] they show that indeed they can compute this in constant time. The following lemma is a modification of that lemma to work when u_A, u_B and u_C are *local inorder numbers*.

Lemma 1 (Modified Lemma 6 of [5]). *Let C be an internal node in $\mathcal{T}$ corresponding to the cluster obtained by merging clusters A and B. For any node u in C, given u_C we can tell in constant time if u is in A (and obtain u_A) in B (and obtain u_B) or in both. Similarly, if u is in A or in B we can obtain u_C in constant time from u_A or u_B.*

Proof. We show how to obtain u_A or u_B when u_C is given. Obtaining u_C from u_A or u_B is done similarly. For each node C, we store a following information:

– $\ell(A)$ $(r(A))$: the first (last) node visited in an inorder traversal of C that is also a node in A.

– $\ell(B)$ ($r(B)$): the first (last) node visited in an inorder traversal of C that is also a node in B.
– the number of nodes in A and in B.
– u'_C, where u' is the common boundary node of A and B.

Consider the case where C is obtained by merging A and B vertically (when the bottom boundary node of A is the top boundary node of B), and where B includes vertices that are in the left subtree of this boundary node, the other case is handled similarly:

– if $u_C < \ell(B)$ then u is a node in A and $u_A = u_C$.
– if $\ell(B) \leq u_C \leq r(B)$ then u is a node in B and $u_B = u_C - \ell(B) + 1$. For the special case when $u_C = u'_C$ then u is also the bottom boundary node in A and $u_A = \ell(B)$.
– if $u_c > r(B)$ then u is a node in A visited after visiting all the nodes in B then $u_A = u_C - |B| + 1$.

When C is obtained by merging A and B horizontally (when A and B share their top boundary node and A is to the left of B):

– if $u_C < r(A)$ then u is a node in A and $u_A = u_C$.
– if $u_C \geq r(A)$ then u is a node in B and $u_B = u_C - |A| + 1$. For the special case when $u_C = u'_C$ then u is also the top boundary node in A and $u_A = |A|$.

3 Compressing the String vs. the Cartesian Tree

In this section we compare the sizes of the SLP compression $\mathcal{S}$ and the top-tree compression $\mathcal{T}$. We show that given any SLP $\mathcal{S}$ of height h we can construct a top-tree compression $\mathcal{T}$ based on $\mathcal{S}$ (i.e. non-greedily) such that $|\mathcal{T}| = O(|\mathcal{S}| \cdot \sigma)$ and the height of $\mathcal{T}$ is $O(h \log \sigma)$. Using $\mathcal{T}$, we can then answer range minimum queries on S in time $O(h \log \sigma)$ as done in Section 2.2. Furthermore, we can construct $\mathcal{T}$ using Rytter's SLP [14] as $\mathcal{S}$. Then, the height of $\mathcal{S}$ is $h = \log n$ and the size of $\mathcal{S}$ is larger than the optimal SLP by at most a multiplicative $\log n$ factor. Combined with Rytter's SLP, and since every unlabeled tree has a top-tree compression $\mathcal{T}$ of size $O(n/\log n)$ and height $\log n$ [5], we obtain Theorem 3.

Consider a rule $C \rightarrow AB$ in the SLP. We will construct a top-tree (a hierarchy of clusters) of C (i.e. of the Cartesian tree of the string derived by the SLP variable C) assuming we have the top-trees of A and of B. We show that the top-tree of C contains only $O(\sigma)$ new clusters that are not clusters in the top-trees of A and of B, and that the height of the top-tree is only $O(\log \sigma)$ larger than the height of the top tree of A or the top tree of B. To achieve this, for any variable A of the SLP, we will make sure that certain clusters (associated with its rightmost and leftmost paths) must be present in its top-tree. See Fig. 2.

We first describe how the Cartesian tree $CT(C)$ of the string derived by variable C can be described in terms of the Cartesian trees $CT(A)$ and $CT(B)$. We label each node in a Cartesian tree with its corresponding character in the

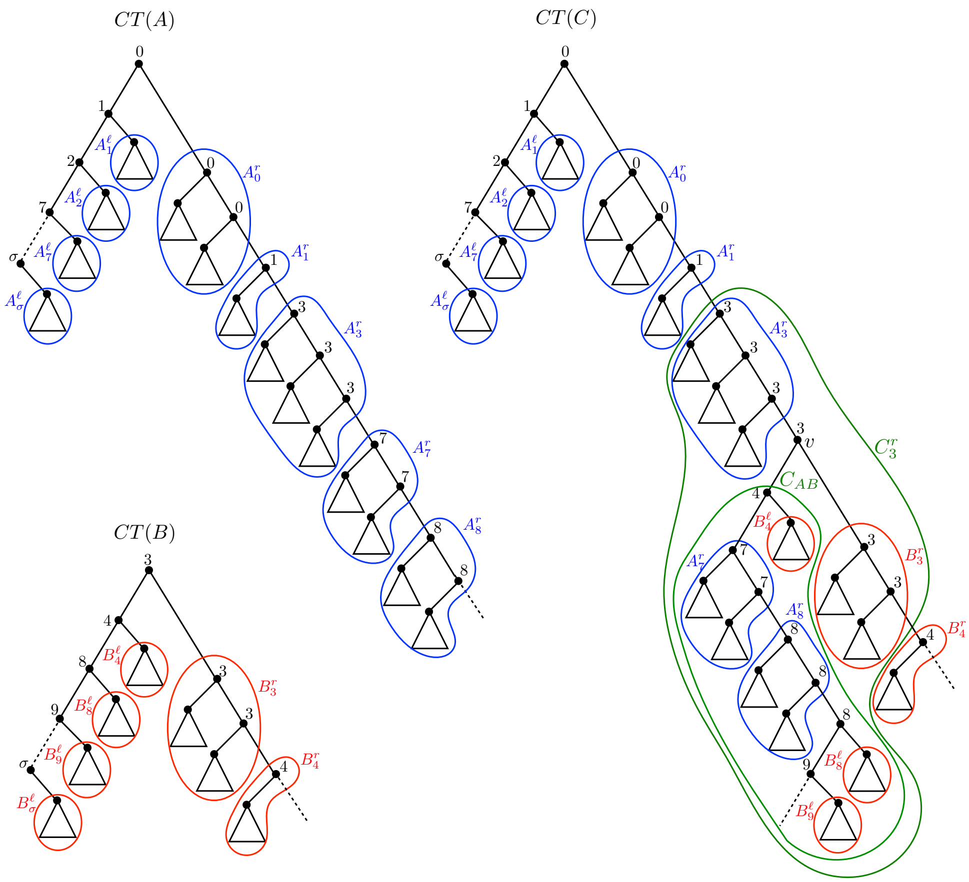


Fig. 2. The Cartesian tree of SLP variables A, B, C where $C \to AB$. The single additional clusters of C_3^r (in green) is formed by merging existing clusters from A (in blue) and from B (in red). First, cluster C_{AB} is formed by alternating subpaths of the leftmost path in $CT(B)$ and the rightmost path in $CT(A)$ (here, $x = 3, y = 7$, and $z = 8$). Then, C_{AB} is merged with B_3^r, v, and A_3^r. In this example, $A_s = \{A_i^r \mid i > 3\}$ and $B_p = \{B_i^\ell \mid i > 3\}$.

string. These labels are only used for the sake of this description, the actual Cartesian tree is an unlabeled tree. By definition of the Cartesian tree, the labels are monotonically non-decreasing as we traverse any root-to-leaf path. Let $\ell(A)$ (respectively $r(A)$) denote the path in $CT(A)$ starting from the root and following left (respectively right) edges. Since we break ties by taking the leftmost occurrence of the same character we have that the path $\ell(A)$ is strictly increasing (the path $r(A)$ is just non-decreasing).

Let x be the label of the root of $CT(B)$. To simplify the presentation we assume that the label of the root of $CT(A)$ is smaller or equal to x (the other case is handled similarly). Split $CT(A)$ by deleting the edge connecting the last occurrence of x on $r(A)$ with its right child (again, for simplicity of presentation we assume without loss of generality that this node exists). The resulting two subtrees are the Cartesian trees $CT(A_p)$ and $CT(A_s)$ of a prefix A_p and a suffix of A_s of A whose concatenation is A. Split $CT(B)$ by deleting the edge connecting the root to its left child. The resulting two subtrees are the Cartesian trees $CT(B_p)$ and $CT(B_s)$ of a prefix and a suffix of B. The Cartesian tree $CT(C)$ of the concatenation $C = AB$ is obtained as follows. Compute recursively the Cartesian tree $CT(A_sB_p)$ of the concatenation of A_s and B_p, and attach $CT(A_sB_p)$ as the left child of the rightmost leaf in $CT(A_p)$. Then attach $CT(B_s)$ as the right child of the rightmost leaf in $CT(A_p)$. See Fig. 2.

We move on to describing the clusters of the top-tree. For a node with label i appearing in $\ell(A)$ we define A_i^ℓ to be the subtree rooted at the node's right child. We do this for all nodes except for the first node of $\ell(A)$ (i.e. the root of $CT(A)$). Next consider the path $r(A)$. For every label i there can be multiple vertices with label i that are consecutive on $r(A)$. We define A_i^r to be the union of all vertices of $r(A)$ that have label i together with the subtrees rooted at their left children. Again, we treat the first node of $r(A)$ (i.e. the root of $CT(A)$) differently: if its label is i then A_i^r does not include this vertex (the root) nor its left subtree. See Fig. 2 (left).

We define the top-tree recursively by describing how to obtain the clusters for the top-tree of the Cartesian tree $CT(C)$ from the top-trees of $CT(A)$ and $CT(B)$. For each variable (say A) of the SLP $\mathcal{S}$ of S, we require that in the top-tree of S there is a cluster for every A_i^ℓ and every A_i^r. We will show how to construct all the C_i^ℓ and C_i^r clusters of C by merging clusters of A and B while introducing only $O(\sigma)$ new clusters, and with $O(\log\sigma)$ increase in height. First observe that for every i we have that $C_i^\ell = A_i^\ell$ so we already have these clusters. Next consider the clusters C_i^r. Let x denote the label of the root of $CT(B)$. It is easy to see that $C_i^r = A_i^r$ for every $i < x$ and that $C_i^r = B_i^r$ for every $i > x$. Therefore, the only new cluster we need to create is C_x^r.

The cluster C_x^r is composed of the following components: First, it contains the cluster A_x^r. Then, the root of $CT(B)$ (denoted v, and whose label is x) is connected as the right child of the bottom boundary node of A_x^r. The right child of v in C_x^r is the top boundary node of B_x^r and all of B_x^r is contained in C_x^r. The left child of v in C_x^r is the top boundary node of a single new cluster C_{AB} consisting of $O(\sigma)$ existing clusters.

The cluster C_{AB} consist of all clusters B_i^ℓ and the clusters A_i^r for $i > x$. More precisely, let y denote the smallest number larger than x such that A_y^r appears in $r(A)$. Starting from top to bottom, C_{AB} first contains a leftmost path that is a prefix of $\ell(B)$. More precisely, it is the prefix of $\ell(B)$ containing all nodes with labels i for $x < i < y$. For each such node, its right subtree is the cluster B_i^ℓ. After this leftmost path C_{AB} then continues with a rightmost path that is a subpath of $r(A)$ consisting of all nodes in $r(A)$ with labels i for $y \leq i \leq z$. Here z is the smallest number greater or equal to y such that B_z^ℓ appears in $\ell(B)$. In this way, C_{AB} keeps alternating between subpaths of $\ell(B)$ and of $r(A)$ (along with the subtrees hanging from these subpaths). Overall, C_{AB} composes to $O(\sigma)$ clusters consisting of single edges, clusters A_i^r, and clusters B_i^ℓ. We merge these clusters into the single cluster C_{AB} by first doing a horizontal merge for every B_i^ℓ with a single edge cluster and then greedily doing vertical merges for all $O(\sigma)$ clusters of the path. This adds $O(\sigma)$ new clusters and adds $O(\log \sigma)$ to the height of the cluster's hierarchy. Finally, we obtain C_x^r by merging C_{AB}, A_x^r, and B_x^r.

To conclude, once we have all clusters of the SLP's start variable, we merge them into a single cluster (i.e. obtain the top-tree of the entire Cartesian tree of S) by greedily merging all its $O(\sigma)$ clusters (introducing $O(\sigma)$ new clusters and increasing the height by $O(\log \sigma)$) similarly to the above.

4 Conclusions

In this paper we have investigated compressed RMQ. We have shown that compressing the Cartesian tree can be exponentially better than compressing the string itself, and is never worse by more than an $O(\sigma)$ factor. Improving this $O(\sigma)$ factor or finding a counter example that actually exhibits an $\Omega(\sigma)$ factor remains an interesting open question.

References

1. Andrés Abeliuk, Rodrigo Cánovas, and Gonzalo Navarro. Practical compressed suffix trees. *Algorithms*, 6(2):319–351, 2013.
2. Stephen Alstrup, Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Maintaining information in fully-dynamic trees with top trees. *ACM Transactions on Algorithms*, 1:243–264, 2003.
3. Jérémy Barbay, Johannes Fischer, and Gonzalo Navarro. Lrm-trees: Compressed indices, adaptive sorting, and compressed permutations. *Theor. Comput. Sci.*, 459:26–41, 2012.
4. Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, pages 88–94, 2000.
5. Philip Bille, Inge Li Gørtz, Gad M. Landau, and Oren Weimann. Tree compression with top trees. *Inf. Comput.*, 243:166–177, 2015.
6. Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann. Random access to grammar-compressed strings and trees. *SIAM J. Comput.*, 44(3):513–539, 2015.

7. Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *IEEE Trans. Information Theory*, 51(7):2554–2576, 2005.
8. Pooya Davoodi, Rajeev Raman, and Srinivasa Rao Satti. Succinct representations of binary trees for range minimum queries. In *COCOON 2012, Proceedings*, pages 396–407, 2012.
9. Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980.
10. Bartlomiej Dudek and Pawel Gawrychowski. Slowing down top trees for better worst-case bounds. *CoRR*, abs/1801.01059, 2018.
11. Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011.
12. Keisuke Goto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Fast q-gram mining on SLP compressed strings. *J. Discrete Algorithms*, 18:89–99, 2013.
13. Artur Jez and Markus Lohrey. Approximation of smallest linear tree grammar. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014*, pages 445–457, 2014.
14. Wojciech Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.
15. Yoshimasa Takabatake, Tomohiro I, and Hiroshi Sakamoto. A space-optimal grammar compression. In *25th Annual European Symposium on Algorithms, ESA 2017*, pages 67:1–67:15, 2017.
16. Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.