Compressed Range Minimum Queries

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

[1]University of Siegen

[2]Interdisciplinary Center Herzliya

[3]University of Haifa

SPIRE 2018

Slides by Seungbum Jo

Range Minimum Query (RMQ)

Given a string S of n integers in $[1, \sigma)$, a *range minimum query* $\mathrm{RMQ}(i, j)$ asks for the index of the smallest integer in $S[i \ldots j]$ (if there is a tie, we choose the *first* position).

Range Minimum Query (RMQ)

Given a string S of n integers in $[1, \sigma)$, a *range minimum query* RMQ(i, j) asks for the index of the smallest integer in $S[i \ldots j]$ (if there is a tie, we choose the *first* position).

10	8	4	2	5	2	9	3	7	1

$$\text{RMQ}(4, 7) = 4$$

Range Minimum Query (RMQ)

Given a string S of n integers in $[1, \sigma)$, a *range minimum query* $\text{RMQ}(i, j)$ asks for the index of the smallest integer in $S[i \ldots j]$ (if there is a tie, we choose the *first* position).

10	8	4	2	5	2	9	3	7	1

$$\text{RMQ}(4, 7) = 4$$

Goal : Design a data structure for answering RMQ efficiently using sublinear space.

Cartesian tree [Vuillemin 80]

Given a string S of size n, the cartesian tree C of S is defined as follows.

- Root node of C corresponds to $S[\mathrm{RMQ}(1,n)]$, and its left (resp. right) child is the cartesian tree of $S[1\ldots\mathrm{RMQ}(1,n)-1]$ (resp. $S[\mathrm{RMQ}(1,n)+1\ldots n]$).
- Each node in C with in-order number i corresponds to $S[i]$.
  (we refer the node with in-order number i as node i).

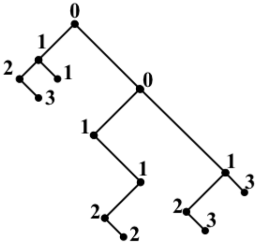


Figure: Catesian tree of $S =$ "2 3 1 1 0 1 2 2 1 0 2 3 1 3"

Cartesian tree [Vuillemin 80]

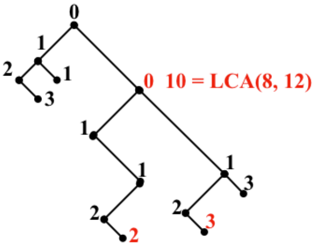


Figure: Catesian tree of $S =$ "2 3 1 1 0 1 2 2 1 0 2 3 1 3"

Properties of Cartesian trees

- For any two nodes i and j, RMQ(i, j) corresponds to the *nearest common ancestor* (LCA) of node i and j.
- For any two strings, all of their answers of RMQ are same if and only if their corresponding cartesian trees are identical.
  (Answering RMQ on $S =$ answering LCA on the cartesian tree of S)

Previous Results (with constant query time)

1. Systematic data structures (indexing model):
   - The query algorithm can access the input data.
   - Size of the data structure = size of (input + index).
   - $|S| + O(n \lg \sigma)$ bits [AGKR04], $|S| + 2n/c(n)$ bits [FH11]...
   - $|S| + O(n/c)$ bits with $O(c)$ query time is optimal [BDS12].

Previous Results (with constant query time)

1. Systematic data structures (indexing model):
   - The query algorithm can access the input data.
   - Size of the data structure = size of (input + index).
   - $|S| + O(n \lg \sigma)$ bits [AGKR04], $|S| + 2n/c(n)$ bits [FH11]...
   - $|S| + O(n/c)$ bits with $O(c)$ query time is optimal [BDS12].
2. Non-systematic data structures (encoding model):
   - The query algorithm cannot access the input data after preprocessing.
   - $4n + o(n)$ bits [Sadakane 07], $2n + o(n)$ bits [FH11, DRS12] (by storing the *Cartesian tree* of S (or its variant) efficiently).
   - Information-theoretical lower bound : $2n - O(\lg n)$ bits.
     $\rightarrow 2n + o(n)$-bit data structure is optimal for the worst case.

Our results

1. *Sublinear* space data structure for compressible inputs.
   - There are some sublinear data structures for answering RMQ for compressible inputs (BFN12 (for well-sorted permutation), DRS12 (for (entropy-based) compressible succinct-tree representation)...).
   - In this paper, we consider two approaches.
     1. Using string compression (compress input string S).
     2. Using tree compression (compress the cartesian tree of S).

Our results

Using string compression

- We consider a data structure for answering RMQ on a *grammar compression* of S. i.e., a context-free grammar that only generates S.
- Wlog, we assume that grammars are given as straight-line programs (SLP).
  - The right-hand side of each rule in S either consists of the concatenations of two non-terminals or of a single terminal symbol.
  - Size of SLP = total number of symbols in the rules.
  - LZ family, Re-Pair, Bisection...

ex)

$$\underbrace{aaaa...aaaaa}_{2^n \text{ a's}} \quad \rightarrow \quad \begin{array}{c} S \rightarrow A_n A_n \\ A_n \rightarrow A_{n-1} A_{n-1} \\ \vdots \\ A_2 \rightarrow A_1 A_1 \\ A_1 \rightarrow a \end{array}$$

Our results

Using string compression

By extending the Bille et al.'s data structure [BLRSSW 15] for random-accessing to the SLP-grammar compression $\mathcal{S}'$ of S, we obtain a data structure for answering RMQ on $\mathcal{S}'$.

Theorem

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.

Our results

Using tree compression

- We consider a data structure for answering LCA queries on a *top-tree compression* [BGLW 15] of the Cartesian tree $\mathcal{C}$ of S.
- The original top-tree compression paper [BGLW 15] gives a data structure for answering pre-order number of LCA queries.
- We showed that their data structure can be easily adjusted to work with in-order numbers instead of pre-order (note that $S[i]$ corresponds to the node in $\mathcal{C}$ with in-order number i).

Theorem

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.

Our results

2. Size comparison between two approaches
(using string compression vs tree compression)

- Top-tree compression can be exponentially better than any SLP of S. (i.e., when $S = 1\ 2\ 3 \ldots\ n$, the size of SLP is $O(n)$ whereas the size of $\mathcal{T}$ is $O(\log n)$.)
- On the opposite side, top-tree compression never worse by more than an $O(\sigma)$ factor compared to the SLP of S.

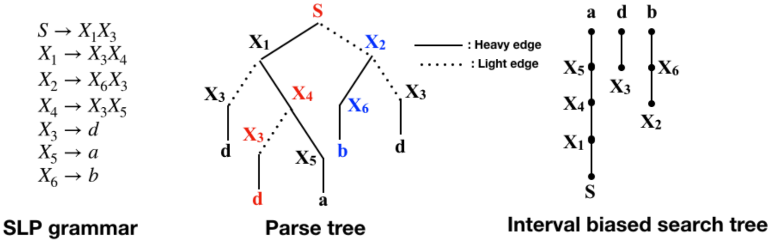
Theorem

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 $O(\mathit{depth}(\mathcal{S}') \cdot \log \sigma)$.

Approach 1. Using string compression

RMQ on the SLP-compressed string

Bille et al.'s random-access data structure (2015)



$S \to X_4 \to X_3 \to d$

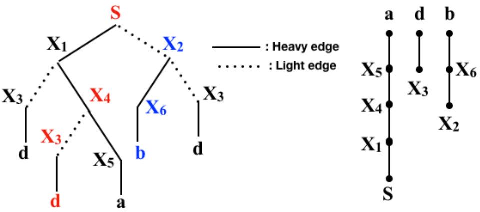
$S \to X_2 \to b$

- For each node v in the parse tree, they select the child of v that derives the longer string to be a *heavy node*.
- Using their data structure, for any position i, one can return the path form the root node to i (as components of heavy paths) in $\log n$ time, using *interval biased search tree*.

RMQ on the SLP-compressed string

$S \to X_1X_3$
$X_1 \to X_3X_4$
$X_2 \to X_6X_3$
$X_4 \to X_3X_5$
$X_3 \to d$
$X_5 \to a$
$X_6 \to b$

SLP grammar



Parse tree **Interval biased search tree**

$S \to X_4 \to X_3 \to d$
$S \to X_2 \to b$

Extension for supporting RMQ

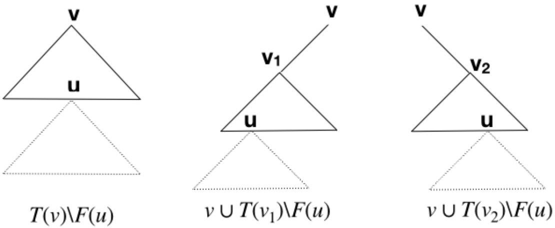
- For each node in the interval biased search tree, we store the location of the minimum value leaf (and value of the leaf).
- On the interval biased search tree, build standard linear-space constant query-time RMQ data structure over the left (resp. right) hanging subtree minimums (connected with light edge).

Approach 2. Using tree compression

RMQ using compressed-cartesian tree

Top-tree compression (Bille et al. 2015)

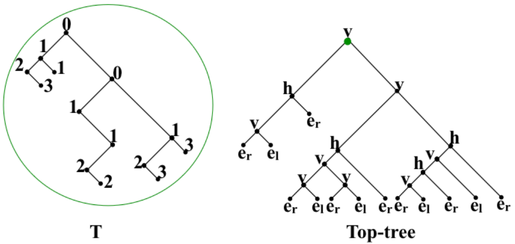
- For vertex $v \in T$ with children v_1 and v_2, Let $T(v)$ be the subtree of T rooted at v, and $F(v)$ to be the forest $T(v)$ without v. Then a *cluster* with *top boundary node* v and *bottom boundary node* u is a tree pattern which can be either (1) $T(v)\backslash F(u)$, (2) $v \cup T(v_1)\backslash F(u)$, or (3) $v \cup T(v_2)\backslash F(u)$.



RMQ using compressed-cartesian tree

Top-tree compression (Bille et al. 2015)

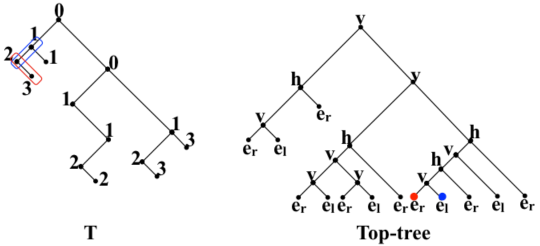
- The top-tree of a tree T is a hierarchical decomposition of T into clusters.
  1. The root of the top-tree is the cluster T itself.
  2. The leaves of the top-tree are clusters corresponding to the edges (v, u) of T, these edges are labeled with e_r (if u is a right child of v) or e_l (if u is a left child of v).
  3. Each internal node of the top-tree is a merged cluster of its two children, and labeled with h (horizontal merge) or v vertical merge.



RMQ using compressed-cartesian tree

Top-tree compression (Bille et al. 2015)

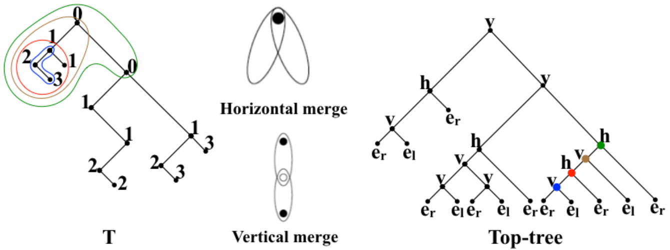
- The top-tree of a tree T is a hierarchical decomposition of T into clusters.
  1. The root of the top-tree is the cluster T itself.
  2. The leaves of the top-tree are clusters corresponding to the edges (v, u) of T, these edges are labeled with e_r (if u is a right child of v) or e_l (if u is a left child of v).
  3. Each internal node of the top-tree is a merged cluster of its two children, and labeled with h (horizontal merge) or v vertical merge.



RMQ using compressed-cartesian tree

Top-tree compression (Bille et al. 2015)

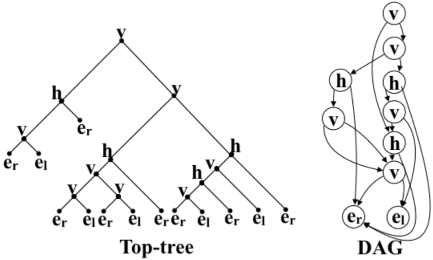
- The top-tree of a tree T is a hierarchical decomposition of T into clusters.
  1. The root of the top-tree is the cluster T itself.
  2. The leaves of the top-tree are clusters corresponding to the edges (v, u) of T, these edges are labeled with e_r (if u is a right child of v) or e_l (if u is a left child of v).
  3. Each internal node of the top-tree is a merged cluster of its two children, and labeled with h (horizontal merge) or v vertical merge.



RMQ using compressed-cartesian tree

Top-tree compression (Bille et al. 2015)

- After constructing the top-tree, compress the tree to DAG using a algorithm of Downey et al. (1980).



RMQ using compressed-cartesian tree

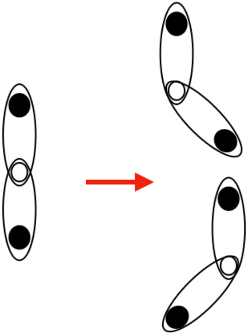
Top-tree compression (Bille et al. 2015)

- Using the Bille et al.'s top-tree compression algorithm, one can compress the cartesian tree of size n to the compression form $\mathcal{T}$ of size at most $O(n/\log n)$ with depth $O(\log n)$ (LRS17, DG18).
- If the pre-order number of v and u are given, Bille et al. showed that one can answer the pre-order number of LCA of v and u in $O(depth)$ time using $O(|\mathcal{T}|)$ space (idea : compute *local pre-order number* for each cluster).

RMQ using compressed-cartesian tree

Q : How to support LCA queries when v and u are given as in-order?

A : We maintain the same data data structure as the pre-order case, except we need to consider two cases for each vertical merging (left or right subtree).



Compressing the String vs. the Cartesian Tree

Compressing the String vs. the Cartesian Tree

Theorem

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 $O(\mathit{depth}(\mathcal{S}') \cdot \log \sigma)$.

Sketch of the proof : Construct $\mathcal{T}$ followed by the rules in $\mathcal{S}'$

Compressing the String vs. the Cartesian Tree

Theorem

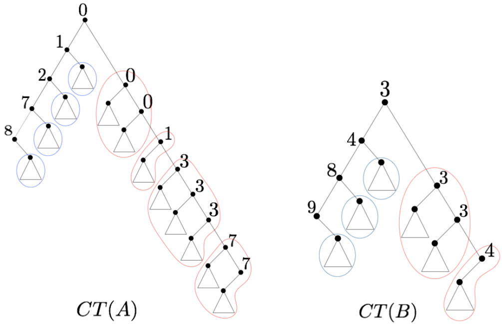
Given a string S of length n over an alphabet of size σ, for any SLP-grammar compression $\mathcal{S}'$ of S there is a top-tree compression $\mathcal{T}$ of the Cartesian tree $\mathcal{C}$ with size $O(|\mathcal{S}'| \cdot \sigma)$ and depth $O(\mathtt{depth}(\mathcal{S}') \cdot \log \sigma)$.

Sketch of the proof (cont.) :

- Let $CT(C)$ be a cartesian tree of the string derived by the SLP variable C.
- Consider the rule $C \to AB$ in $\mathcal{S}'$. How to construct a top-tree compression of $CT(C)$ when the top-tree compression of $CT(A)$ and $CT(B)$ are given?

Compressing the String vs. the Cartesian Tree

Sketch of the proof (cont.) :

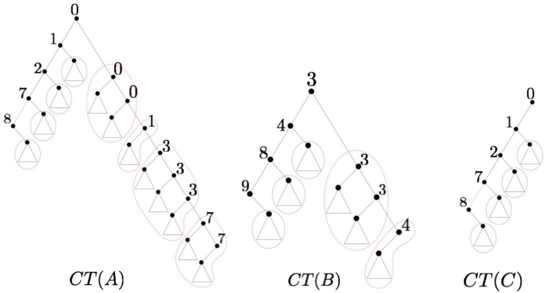


- Invariant : Whenever constructing the top-tree compression corresponding to the $CT(A)$ for any variable A in $\mathcal{S}'$, we maintain the clusters corresponding to the blue circles and red circles.
  - Blue circles : Subtrees hanging on the left spine.
  - Red circles : Set of subtrees hanging on the right spine

How to construct the clusters corresponding to the blue and red circles of $CT(C)$?

Compressing the String vs. the Cartesian Tree

Sketch of the proof (cont.) :



The strings corresponding to

A = 8..7..2..1..0..0..0..1..3..3..3..7..7

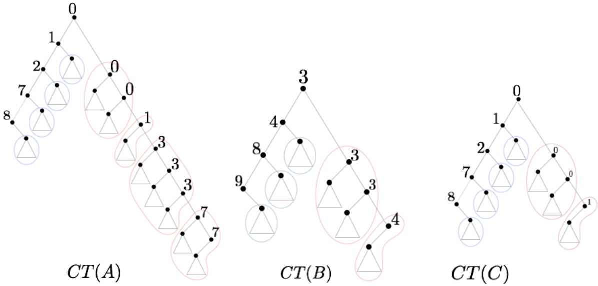
B = 9..8..4..3..3..3..4

C = 8..7..2..1..0..0..0..1..3..3..3..7..7 9..8..4..3..3..4

- Since the value corresponding to the root node in A is smaller than B, the root node of C is corresponding to the first 0 in A.
- The orange part of the string corresponding to A and C are identical $\rightarrow$ blue circles of $CT(C)$ are same as the blue circles in $CT(A)$.

Compressing the String vs. the Cartesian Tree

Sketch of the proof (cont.) :

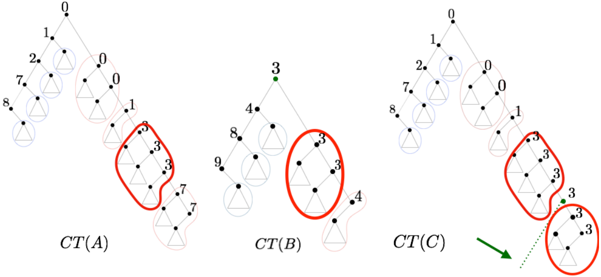


The strings corresponding to
$A = 8..7..2..1..0..0..0..1..3..3..3..7..7$
$B = 9..8..4..3..3..3..4$
$C = 8..7..2..1..0..0..0..1..3..3..3..7..7\ 9..8..4..3..3..4$

- Similarly, the first two red circles in $CT(C)$ are identical to the first two red circles in $CT(A)$.

Compressing the String vs. the Cartesian Tree

Sketch of the proof (cont.) :

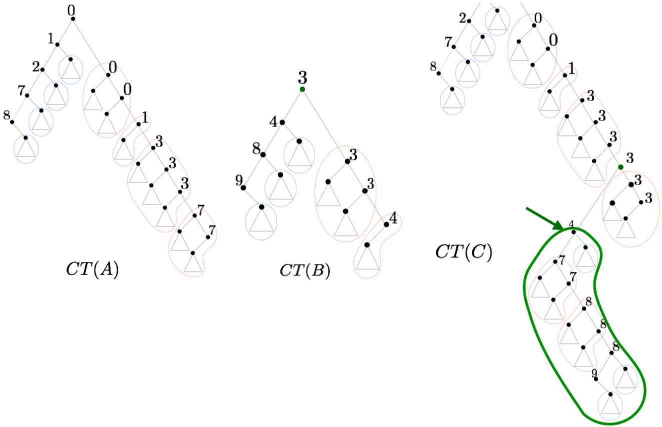


The strings corresponding to
$A = 8..7..2..1..0..0..0..1..3..3..3..7..7$
$B = 9..8..4..3..3..3..4$
$C = 8..7..2..1..0..0..0..1..3..3..3..7..7\ 9..8..4..3..3..3..4$

- The right child of the node corresponding to the 3rd 3 in $CT(C)$ is corresponding to the root node in $CT(B)$.
- How to construct the cluster corresponding to the left subtree in $CT(C)$ hanging on the node corresponding to the 4th 3, to construct the 3rd red circle of $CT(C)$?

Compressing the String vs. the Cartesian Tree

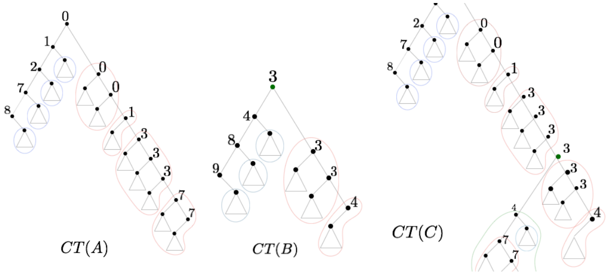
Sketch of the proof (cont.) :



- Key lemma : we can construct the cluster corresponding to the green circle using the red and blue clusters of $CT(A)$, and $CT(B)$, by adding $O(\sigma)$ extra clusters with increasing the height by $O(\log \sigma)$.

Compressing the String vs. the Cartesian Tree

Sketch of the proof (cont.) :



The strings corresponding to

$A = 8..7..2..1..0..0..0..1..3..3..3..7..7$

$B = 9..8..4..3..3..3..4$

$C = 8..7..2..1..0..0..0..1..3..3..3..7..7\ 9..8..4..3..3..3..4$

- The rest red circle in $CT(C)$ is identical to the corresponding red circles in $CT(B)$.
- Using the clusters corresponding to red and blue circles of $CT(C)$, we can construct the top-tree of $CT(C)$ by adding $O(\sigma)$ extra clusters with increasing the height by $O(\log \sigma)$.

Conclusion

- Data structure for compressed RMQ. We consider two approaches (i) using string compression, and (ii) using tree compression. Both data structures use sublinear size for compressible inputs.
- Compressing the cartesian tree can be exponentially better than compressing the string itself, and is never worse by more than an $O(\sigma)$ factor.
  - When $S = 1\ 2\ 3 \ldots\ n$, the size of SLP is $O(n)$ whereas the size of $\mathcal{T}$ is $O(\log n)$.
  - Using the Rytter's SLP construction algorithm, we can construct a top-tree compression of size $\min\left(O(n/\log n), O(\sigma|\mathcal{S}|\log n)\right)$, where $\mathcal{S}$ is the smallest possible SLP grammar of S.
  - Recently (see full version), we showed that the $O(\sigma)$ factor is tight.

Thank you!