Extracting the Sparse Longest Common Prefix Array from the Suffix Binary Search Tree

Tomohiro I[1][0000−0001−9106−6192], Robert W. Irving[2], Dominik Koeppl[3][0000−0002−8721−4444], and Lorna Love[2]

[1] Department of Artificial Intelligence, Kyushu Institute of Technology, Japan
tomohiro@ai.kyutech.ac.jp
[2] School of Computing Science, University of Glasgow, Glasgow, UK
Rob.Irving@glasgow.ac.uk, lornalove75@googlemail.com
[3] M&D Data Science Center, Tokyo Medical and Dental University, Japan
koeppl.dsc@tmd.ac.jp

Abstract. Given a text T of length n, the sparse suffix sorting problem asks for the lexicographic order of suffixes starting at m selectable text positions P. The suffix binary search tree [Irving and Love, JDA'03] is a dynamic data structure that can answer this problem dynamically in the sense that insertions and deletions of positions in P are allowed. While a standard binary search tree on strings needs to store two longest-common prefix (LCP) values per node for providing the same query bounds, each suffix binary search tree node only stores a single LCP value and a bit flag. Its tree topology induces the sorting of the m suffixes by an Euler tour in $\mathcal{O}(m)$ time. However, it has not been addressed how to compute the lengths of the longest common prefixes of two suffixes with neighboring ranks with this data structure. We show that we can compute these lengths again by an Euler tour in $\mathcal{O}(m)$ time.

Keywords: suffix binary search tree; sparse suffix sorting; longest common prefixes; Euler tour

1 Introduction

While common full-text indexing data structures provide interfaces answering pattern matching queries for all positions in the underlying text, the sparse variation of such data structures index the text only at certain m positions with the aim to improve the space and construction time bounds to a complexity related to m. Such a technique can make sense if we work with large data sets for which the maintenance of a full-text data structure is prohibitive with respect to its space or construction time. In such a case, when only certain positions are of interest such as word beginnings in natural language texts, we can resort to a sparse text index. One of the most well-studied sparse text indices is the sparse suffix array. Given a text $T[1..n]$ of length n, and a set of text positions $P = \{p_1, \ldots, p_m\}$ with $p_i \in [1..n]$, the sparse suffix array SSA determines the lexicographic order of all suffixes starting with the positions of P. Formally, the sparse suffix array $\mathsf{SSA}[1..m]$ is a ranking of P with respect to the suffixes starting at the respective positions, i.e., $T[\mathsf{SSA}[i]..] \prec T[\mathsf{SSA}[i+1]..]$ for $i \in [1..m-1]$ with $\{\mathsf{SSA}[1], \ldots, \mathsf{SSA}[m]\} = P$. To support pattern matching efficiently, the sparse suffix array can be augmented with the sparse longest common prefix (LCP) array SLCP to support the classic suffix array pattern matching algorithm of Manber and Myers [19]. The sparse LCP array SLCP stores the length of the LCP of each suffix stored in SSA with its preceding entry, i.e., $\mathsf{SLCP}[1] = 0$ and $\mathsf{SLCP}[i] := \mathsf{lcp}(T[\mathsf{SSA}[i-1]..], T[\mathsf{SSA}[i]..])$ for all integers $i \in [2..|\mathsf{SSA}|]$.

Other applications of SSA beyond plain pattern matching include the computation of the LCP array in external memory [12], a Burrows-Wheeler transform [4] variation [5], or finding maximal exact matches [14, 22]. There are algorithms that can compute SSA and SLCP for a given set P efficiently (e.g., [2, 3, 6–8, 13, 15, 17, 20, 21]), where most approaches resort to a data structure answering longest common extension (LCE) queries, i.e., a query that asks for the length of the LCP of two suffixes of the underlying text. As pointed out by Fischer et al. [7, Observation 1.2], given a data structure that answers LCE queries in $\mathcal{O}(t_{\mathrm{LCE}}(n))$ time for $t_{\mathrm{LCE}}(n) > 0$, we can solve the sparse suffix sorting problem for m positions in $\mathcal{O}(t_{\mathrm{LCE}}(n) \cdot m \cdot t_{\mathcal{D}}(m, n))$ time by inserting the m respective suffixes into a dynamic dictionary $\mathcal{D}$ with an insertion operation taking $t_{\mathcal{D}}(m, n)$ time when $\mathcal{D}$ stores m suffix starting positions. In particular, this observation generalizes the suffix sorting problem to be dynamic. By *dynamic* we mean that we allow insertions or deletions of positions in P, and hence have the additional need for updating $\mathcal{D}$. This problem is a fundamental task for dynamic pattern matching, where the user can (a) change P and (b) query an indexing data structure built on P and T, both in an arbitrary order. A straightforward approach is to use a binary search tree (BST) as the dictionary representation since it supports the necessary insertion operations. However, the time complexity can be a problem, since a suffix has $\mathcal{O}(n)$ characters and hence, we need $t_{\mathcal{D}}(m, n) = \mathcal{O}(nh)$ time for an insertion into a BST of height h. We can use a balanced representation such as the AVL-tree [1] to obtain $h = \mathcal{O}(\lg m)$, but the insertion time complexity has still a linear factor in n. Exactly for this use case scenario, Irving and Love [11] provided an augmentation of the standard BST to obtain $\mathcal{O}(m + h)$ time, which they called suffix BST. In the suffix BST, they augmented each BST node with the length of the LCEs with an ancestor node. They also proposed a variant with the virtues of the AVL-tree, which they called suffix AVL-tree, having $h = \mathcal{O}(\lg m)$. In the following we write SBST for the suffix BST or its variants, the suffix AVL-tree, built upon the m suffixes of our text. Although we can retrieve SSA from SBST with a simple Euler tour, we show in the following that retrieving SLCP is also possible:[4]

Theorem 1. *We can compute* SLCP *from* SBST *in time linear in the number of nodes stored in* SBST*.*

Theorem 1 is important in use cases where we need to compute the order of the suffixes dynamically, but then need SSA and SLCP for one of the aforementioned applications. The problem tackled by Thm. 1 can be made trivial with a variation of SBST that stores two LCP values per node [16, Sect. 4.6], and thus this variant is a constant factor larger than SBST. Another way to solve the problem naively would be to use an LCE data structure to compute the LCPs of two neighboring entries in SSA, after extracting SSA from SBST, as done in [7, Corollary 4.2].

2 Preliminaries

We work in the pointer machine model. Let $T[1..n]$ be a text of length n whose characters are drawn from an ordered alphabet Σ. We assume that we can compare two characters of Σ in constant time. We write $T[i]$ for the i-th character of T, for $i \in [1..n]$. For $X, Y, Z \in \Sigma^*$ with $T = XYZ$, X, Y, and Z are called a *prefix*, *substring*, and *suffix* of S, respectively. Since X, Y, or Z may be empty, X and Z are also substrings of S at the same time by this definition.

Let SSA and SLCP be defined as in the introduction. The sparse inverse suffix array SISA$[1..n]$ is (partially) defined by SISA[SSA$[i]] = i$. We only need SISA conceptually, and only care about the entries determined by this equation. The idea is that SISA$[i]$ for $i \in P$ is the rank of the suffix $T[i..]$ among all suffixes starting with a position in P. See Fig. 1 for an example of the defined

[4] A precursor of this research is the technical report [10, Section 5] and a PhD thesis [18].

arrays for $n = m$. There, the entries of rules are sorted in suffix order, i.e., rules$[i]$ is the rule for the node representing SSA$[i]$.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	c	a	a	t	c	a	c	g	g	t	c	g	g	a	c
SSA	2	14	6	3	15	1	5	11	7	13	12	8	9	4	10
SISA	6	1	4	14	7	3	9	12	13	15	8	11	10	2	5
SLCP	0	1	2	1	0	1	2	1	3	0	1	2	1	0	2
rules	E	A	L	L	A	D	R	A	L	A	L	L	R	D	A

Fig. 1. The example string $T =$ `caatcacggtcggac` used in [11, Fig. 2]. The row rules shows from which rule or scenario (cf. Sect. 4) the SLCP value was obtained.

3 The Suffix AVL Tree

Given a set of text positions P, the *suffix AVL tree* represents each suffix $T[p..]$ starting at a text position $p \in P$ by a node. The nodes are arranged in a binary search tree topology such that reading the nodes with an in-order traversal gives the sparse suffix array. To support fast operations, each node is augmented with the following extra information:

Given a node v of SBST, cla_v (resp. cra_v) is the lowest node having v as a descendant in its left (resp. right) subtree. We write $T[v..]$ for the suffix represented by the node v, i.e., we identify nodes with their respective suffix starting positions. Each node v stores a tuple (d_v, m_v), where m_v is

- $\mathsf{lcp}(T[v..], T[\mathsf{cla}_v..])$ if $d_v =$ `left` and cla_v exists,
- $\mathsf{lcp}(T[v..], T[\mathsf{cra}_v..])$ if $d_v =$ `right` and cra_v exists, or
- 0 if $d_v = \bot$.

The value of $d_v \in \{\texttt{left}, \texttt{right}, \bot\}$ is set such that m_v is maximized. Let ca_v be cla_v (resp. cra_v) if $d_v =$ `left` (resp. $d_v =$ `right`). If $d_v = \bot$, then ca_v as well as cla_v and cra_v are not defined. See Fig. 2 for an example.

4 Computing the Sparse LCP Array

Since an SBST node does not necessarily store the LCP with the lexicographically preceding suffix, it is not obvious how to compute SLCP from SBST. For computing SLCP from SBST, we use the following two facts and a lemma:

Fact 1: We have $T[\mathsf{cra}_v..] \prec T[v..] \prec T[\mathsf{cla}_v..]$ in case that cla_v and cra_v exist.

Fact 2: During an Euler tour (an in-order traversal), we can compute SSA by reading the text positions represented by the nodes in the order we visit the nodes for the first time. We can additionally keep track of the in-order rank SISA$[v]$ of each node v.

Lemma 1 ([11, Lemma 1]). *Given three strings X, Y, Z with the lexicographic order $X \prec Y \prec Z$, we have* $\mathsf{lcp}(X, Z) = \min(\mathsf{lcp}(X, Y), \mathsf{lcp}(Y, Z))$.

With the following three rules Rule L, R, and E (as abbreviations for left, right and non-existing), we can partially compute SLCP:

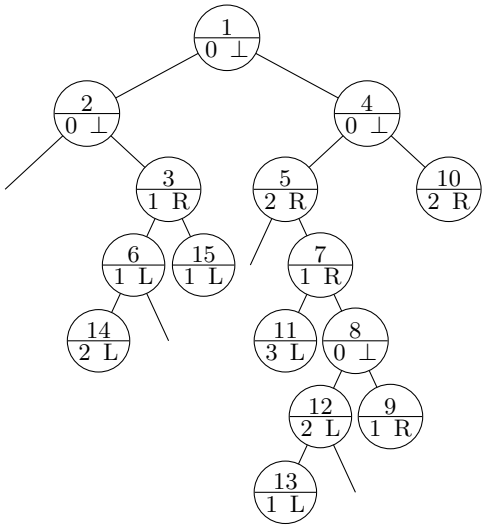

Fig. 2. The unbalanced SBST of the string T = `caatcacggtcggac` defined in Fig. 1 when inserting all text positions in increasing order. A node consists of its position v (top), m_v (bottom left) and d_v (bottom right) abbreviated to L and R for `left` and `right`, respectively.

Rule L If $d_v =$ `left` and the right sub-tree of v is empty, then $\mathsf{SLCP}[\mathsf{SISA}[v]+1] = m_v$. That is because $\mathsf{ca}_v = \mathsf{cla}_v = \mathsf{SSA}[\mathsf{SISA}[v]+1]$ is the starting position of the lexicographically next larger suffix with respect to the suffix starting with v.

Rule R If $d_v =$ `right` then $\mathsf{SLCP}[\mathsf{SISA}[v]] \geq m_v$ since v shares a prefix of at least m_v characters with the lexicographically (not necessarily next) smaller suffix cra_v. In particular, if v does not have a left child, then $\mathsf{SLCP}[\mathsf{SISA}[v]] = m_v$ since $\mathsf{cra}_v = \mathsf{SSA}[\mathsf{SISA}[v]-1]$ in this case.

Rule E If v does not have a left child and cra_v does not exist, then $\mathsf{SLCP}[\mathsf{SISA}[v]] = 0$. This is the case when $T[v..]$ is the smallest suffix stored in SBST.

To compute all SLCP values, there remain the two scenarios Scenario D and A (the letters D and A have the meaning to compare with a descendent or ancestor node):

Scenario D If a node v has a left child, then we have to compare v with the rightmost leaf in v's left subtree because this leaf corresponds to the lexicographically preceding suffix of the suffix starting with v.

Scenario A Otherwise, this lexicographically preceding suffix corresponds to cra_v, such that we have to compare cra_v with v. If $d_v =$ `right`, we are already done due to Rule R since $\mathsf{ca}_v = \mathsf{cra}_v$ in this case (such that the answer is already stored in m_v).

Figure 1 lists which rule or scenario has been applied to compute the SLCP value of a specific node of the SBST instance shown in Fig. 2. We cope with both scenarios by an Euler tour on SBST. For Scenario D, we want to know $\mathsf{lcp}(T[v..], T[\mathsf{cla}_v..])$ for each leaf v regardless of whether $d_v =$ `left` or not. For Scenario A, we want to know $\mathsf{lcp}(T[v..], T[\mathsf{cra}_v..])$ for each node v regardless of whether $d_v =$ `right` or not. We can obtain this LCP information by the following lemma:

Lemma 2. *Given* $\mathsf{lcp}(T[\mathsf{cla}_v..], T[\mathsf{cra}_v..])$, $\mathsf{lcp}(T[v..], T[\mathsf{ca}_v..])$, *and* d_v, *we can compute* $\mathsf{lcp}(T[v..], T[\mathsf{cla}_v..])$ *and* $\mathsf{lcp}(T[v..], T[\mathsf{cra}_v..])$ *in constant time.*

Proof. If $d_v =$ `left`,

- $\mathsf{lcp}(T[v..], T[\mathsf{cla}_v..]) = \mathsf{lcp}(T[v..], T[\mathsf{ca}_v..])$ since $\mathsf{cla}_v = \mathsf{ca}_v$, and
- $\mathsf{lcp}(T[v..], T[\mathsf{cra}_v..]) = \mathsf{lcp}(T[\mathsf{cla}_v..], T[\mathsf{cra}_v..])$.

The latter is because of Fact 1 (assuming cla_v and cra_v exist) and

$$\begin{aligned}\mathsf{lcp}(T[\mathsf{cra}_v..], T[\mathsf{cla}_v..]) &= \min(\mathsf{lcp}(T[\mathsf{cra}_v..], T[v..]), \mathsf{lcp}(T[v..], T[\mathsf{cla}_v..])) \\ &= \mathsf{lcp}(T[v..], T[\mathsf{cra}_v..]) \leq \mathsf{lcp}(T[v..], T[\mathsf{cla}_v..])\end{aligned}$$

according to Lemma 1. The case $d_v = \texttt{right}$ is symmetric:

- $\mathsf{lcp}(T[v..], T[\mathsf{cla}_v..]) = \mathsf{lcp}(T[\mathsf{cla}_v..], T[\mathsf{cra}_v..])$, and
- $\mathsf{lcp}(T[v..], T[\mathsf{cra}_v..]) = \mathsf{lcp}(T[v..], T[\mathsf{ca}_v..])$. □

With Lemma 2, we can keep track of $\mathsf{lcp}(T[\mathsf{cla}_v..], T[\mathsf{cra}_v..])$ while descending the tree from the root: Suppose that we know $\mathsf{lcp}(T[\mathsf{cla}_v..], T[\mathsf{cra}_v..])$ and v's left and right children are x and y, respectively. Then the following holds for x and y:

- Since $\mathsf{cla}_x = v$ and $\mathsf{cra}_x = \mathsf{cra}_v$, $\mathsf{lcp}(T[\mathsf{cla}_x..], T[\mathsf{cra}_x..]) = \mathsf{lcp}(T[v..], T[\mathsf{cra}_v..])$.
- Since $\mathsf{cra}_y = v$ and $\mathsf{cla}_y = \mathsf{cla}_v$, $\mathsf{lcp}(T[\mathsf{cla}_y..], T[\mathsf{cra}_y..]) = \mathsf{lcp}(T[v..], T[\mathsf{cla}_v..])$.

During an Euler tour, we keep the values $\mathsf{lcp}(T[\mathsf{cla}_u..], T[\mathsf{cra}_u..])$ in a stack for the ancestors u of the current node. By applying the above rules and using the LCP information of Lemma 2 for both scenarios, we can compute SLCP during a single Euler tour. Since we did not make any assumptions on the height of SBST, this algorithm runs in linear time, regardless of whether the tree is balanced or not. Altogether, we obtain SSA and SLCP with a single Euler tour with constant time operations per node, and thus could prove the claim of Thm. 1.

Finally, one might be interested not in the complete arrays, but in the recovery of certain parts. By augmenting each node v with the size of the subtree rooted at v, we can answer $\mathsf{SSA}[i]$ in $\mathcal{O}(h)$ time by a top-down traversal in SBST. We select the left child if its subtree size s is at most i, otherwise we exchange i with $i - s - 1$ and either select the right child if $i > 1$, or stop ($i = 1$) since we arrived at a node representing the suffix starting position in question. Like with classic AVL trees, the subtree sizes can be maintained dynamically without additionally costs to the asymptotic time bounds.

For computing $\mathsf{SLCP}[i]$, we need to locate the nodes representing $\mathsf{SSA}[i]$ and $\mathsf{SSA}[i{+}1]$, which is done by two top-down traversals as above. If we additionally keep track of $\mathsf{lcp}(T[\mathsf{cla}_v..], T[\mathsf{cra}_v..])$ while descending the tree as in the aforementioned Euler tour, we gain information of all the above described cases for computing $\mathsf{SLCP}[i]$. By then continuing the Euler tour as described above we obtain:

Corollary 1. SBST *augmented by the subtree sizes can retrieve* $\mathsf{SSA}[i..i + \ell]$ *and* $\mathsf{SLCP}[i..i + \ell]$ *in* $\mathcal{O}(h + \ell)$ *time for any* $\ell \in [0..m - i]$.

5 Future Work

Future directions of research include the analysis of space efficient data structures that have the same capabilities as the suffix BST, but work in compressed or succinct space. For instance, it is possible to use the B tree of [9] whose topology can be succinctly represented in $o(nk)$ bits. Another benefit of this B tree would be that it can read SLCP from the satellite values stored in leaves from left to right. Finally, for B+ variants, the data can be practically faster accessed than in classic binary search trees due to memory locality.

Acknowledgments This work was supported by JSPS KAKENHI Grant Numbers JP21K17701 (DK) and JP19K20213 (TI). We thank the four anonymous reviewers of SPIRE'21 for their valuable comments on our manuscript. They give additional inspiration for Cor. 1 and proposed the problem of how to efficiently merge two suffix binary search tree instances. Tackling this problem could indeed be useful for building and updating FM-indexes and other related indexing data structures.

References

[1] Adelson-Velsky, G.M., Landis, E.M.: An algorithm for organization of information. Dokl. Akad. Nauk SSSR 146, 263–266 (1962)

[2] Bille, P., Fischer, J., Gørtz, I.L., Kopelowitz, T., Sach, B., Vildhøj, H.W.: Sparse text indexing in small space. ACM Trans. Algorithms 12(3), 39:1–39:19 (2016)

[3] Birenzwige, O., Golan, S., Porat, E.: Locally consistent parsing for text indexing in small space. In: Proc. SODA. pp. 607–626 (2020)

[4] Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equipment Corporation, Palo Alto, California (1994)

[5] Chien, Y., Hon, W., Shah, R., Thankachan, S.V., Vitter, J.S.: Geometric BWT: Compressed text indexing via sparse suffixes and range searching. Algorithmica 71(2), 258–278 (2015)

[6] Ferragina, P., Fischer, J.: Suffix arrays on words. In: Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings. LNCS, vol. 4580, pp. 328–339 (2007)

[7] Fischer, J., I, T., Köppl, D.: Deterministic sparse suffix sorting in the restore model. ACM Trans. Algorithms 16(4), 50:1–50:53 (2020)

[8] I, T., Kärkkäinen, J., Kempa, D.: Faster sparse suffix sorting. In: Proc. STACS. LIPIcs, vol. 25, pp. 386–396 (2014)

[9] I, T., Köppl, D.: Load-balancing succinct B trees. ArXiv CoRR abs/2104.08751 (2021)

[10] Irving, R.W., Love, L.: Suffix binary search trees and suffix arrays. Tech. rep., University of Glasgow (2001)

[11] Irving, R.W., Love, L.: The suffix binary search tree and suffix AVL tree. J. Discrete Algorithms 1(5-6), 387–408 (2003)

[12] Kärkkäinen, J., Kempa, D.: LCP array construction using $\mathcal{O}(sort(n))$ (or less) I/Os. In: Proc. SPIRE. LNCS, vol. 9954, pp. 204–217 (2016)

[13] Kärkkäinen, J., Ukkonen, E.: Sparse suffix trees. In: Proc. COCOON. LNCS, vol. 1090, pp. 219–230 (1996)

[14] Khan, Z., Bloom, J.S., Kruglyak, L., Singh, M.: A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays. Bioinform. 25(13), 1609–1616 (2009)

[15] Kolpakov, R., Kucherov, G., Starikovskaya, T.A.: Pattern matching on sparse suffix trees. In: Proc. CCP. pp. 92–97 (2011)

[16] Köppl, D.: Exploring regular structures in strings. Ph.D. thesis, TU Dortmund (2018)

[17] Kosolobov, D., Sivukhin, N.: Construction of sparse suffix trees and lce indexes in optimal time and space. ArXiv CoRR abs/2105.03782 (2021)

[18] Love, L.: The suffix binary search tree. Ph.D. thesis, University of Glasgow, UK (2001)

[19] Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)

[20] Prezza, N.: In-place sparse suffix sorting. In: Proc. SODA. pp. 1496–1508 (2018)

[21] Uemura, T., Arimura, H.: Sparse and truncated suffix trees on variable-length codes. In: Proc. CPM. LNCS, vol. 6661, pp. 246–260 (2011)

[22] Vyverman, M., Baets, B.D., Fack, V., Dawyndt, P.: essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. Bioinform. 29(6), 802–804 (2013)