Heaviest Induced Ancestors and Longest Common Substrings

Travis Gagie[*] Paweł Gawrychowski[†] Yakov Nekrich[‡]

Abstract

Suppose we have two trees on the same set of leaves, in which nodes are weighted such that children are heavier than their parents. We say a node from the first tree and a node from the second tree are induced together if they have a common leaf descendant. In this paper we describe data structures that efficiently support the following heaviest-induced-ancestor query: given a node from the first tree and a node from the second tree, find an induced pair of their ancestors with maximum combined weight. Our solutions are based on a geometric interpretation that enables us to find heaviest induced ancestors using range queries. We then show how to use these results to build an LZ-compressed index with which we can quickly find with high probability the longest substring common to the indexed string and a given pattern.

1 Introduction

In their paper "Range Searching over Tree Cross Products", Buchsbaum, Goodrich and Westbrook [4] considered how, given a forest of trees $T_1, \ldots, T_d$ and a subset E of the cross product of the trees' node sets, we can preprocess the trees such that later, given a d-tuple u consisting of one node from each tree, we can, e.g., quickly determine whether there is any d-tuple $e \in E$ that *induces* u — i.e., such that every node in e is a descendant of the corresponding node in u. (Unfortunately, some of their work was later found to be faulty; see [1].)

In this paper we assume we have two trees T_1 and T_2 on the same set of n leaves, in which each internal node has at least two children and nodes are weighted such that children are heavier than their parents. We assume E is the identity relation on the leaves. Following Buchsbaum et al., we say a node in T_1 and a node in T_2 are *induced* together if they have a common leaf descendant. We consider how, given a node v_1 in T_1 and a node v_2 in T_2, we can quickly find a pair of their *heaviest induced ancestors* (HIAs) — i.e., an ancestor u_1 of v_1 and ancestors u_2 of v_2 such that u_1 and u_2 are induced together and have maximum combined weight.

In Section 2 we give several tradeoffs for data structures supporting HIA queries: e.g., we describe an $\mathcal{O}(n)$-space data structure with $\mathcal{O}\big(\log^3 n(\log\log n)^2\big)$ query time. Our motivation is the problem of building LZ-compressed indexes with which we can quickly find a longest common substring (LCS) of the indexed string and a given pattern. Tree cross products and LZ-indexes may seem unrelated, until we compare figures from Buchbaum et al.'s paper and Kreft and Navarro's "On Compressing and Indexing Repetitive Sequences", shown in Figure 1. In Section 3 we show how, given a string S of length N whose LZ77 parse [23] consists of n phrases, we can build an $\mathcal{O}(n \log N)$-space index with which, given a pattern P of length m, we can find with high probability an LCS of P and S in $\mathcal{O}\big(m \log^2 n\big)$ time.

2 Heaviest Induced Ancestors

An obvious way to support HIA queries is to impose orderings on T_1 and T_2; for each node u, store u's weight and the numbers of leaves to the left of u's leftmost and rightmost leaf descendants; and store a range-emptiness data structure for the $n \times n$ grid on which there is a marker at point (x, y) if the x-th leaf from the left in T_1 is the y-th leaf from the left in T_2. Suppose there are $x_1 - 1$ and $x_2 - 1$ leaves to the left of the leftmost and rightmost leaf descendants of u_1 in T_1, and $y_1 - 1$ and $y_2 - 1$ leaves to the left of the leftmost and rightmost leaf descendants of u_2 in T_2. Then u_1 and u_2 are induced together if and only if the range $[x_1..x_2] \times [y_1..y_2]$ is non-empty. Chan, Larsen and Pătraşcu [5] showed how we can store the range-emptiness data structure in $\mathcal{O}(n)$ space with $\mathcal{O}(\log^\epsilon n)$ query time, or in $\mathcal{O}(n \log\log n)$ space with $\mathcal{O}(\log\log n)$ query time.

Given a node v_1 in T_1 and v_2 in T_2, we start with a pointer p to v_1 and a pointer q to the root of T_2. If the nodes u_1 and u_2 indicated by p and q are induced together, then we check whether u_1 and u_2 have greater combined weight than any induced pair we have seen so far and move q down one level toward v_2; otherwise, we move p up one level toward the root of T_1; we stop when p reaches the root of T_1 or q reaches v_2. This takes a total of $\mathcal{O}(\mathrm{depth}(v_1) + \mathrm{depth}(v_2))$ range-emptiness queries.

[*]University of Helsinki and HIIT
[†]Max-Planck-Institut für Informatik
[‡]University of Kansas

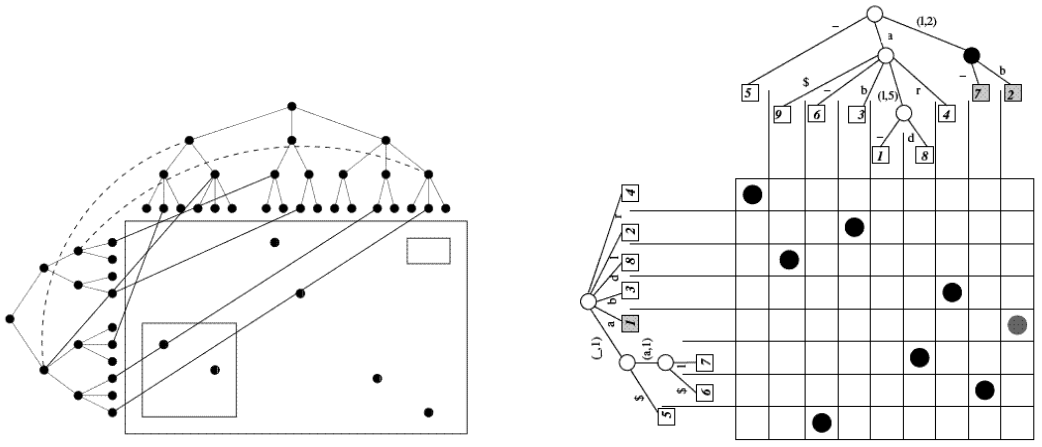

Figure 1: Figure 1 from Buchsbaum et al.'s "Range Searching over Tree Cross Products" and Figure 2b from Kreft and Navarro's "On Compressing and Indexing Repetitive Sequences", whose similarity suggests a link between the two problems. We exploit this link when we use HIA queries to implement LCS queries.

2.1 An $\mathcal{O}(n \log^2 n)$-space data structure with $\mathcal{O}(\log n \log \log n)$ query time

We now describe an $\mathcal{O}(n \log^2 n)$-space data structure with $\mathcal{O}(\log n \log \log n)$ query time; later we will show how to reduce the space via sampling, at the cost of increasing the query time. We first compute the heavy-path decompositions [20] of T_1 and T_2. These decompositions have the property that every root-to-leaf path consists of the prefixes of $\mathcal{O}(\log n)$ heavy paths. Therefore, for each leaf w there are $\mathcal{O}(\log^2 n)$ pairs (a, b) such that a and b are the lowest nodes in their heavy paths in T_1 and T_2, respectively, that are ancestors of w.

For each pair of heavy paths, one in T_1 and the other in T_2, we store a list containing each pair (a, b) such that a is a node in the first path, b is a node in the second path, a and b are induced together by some leaf, a's child in the first path is not induced with b by any leaf, and b's child in the second path is not induced with a by any leaf. We call this the paths' *skyline list*. Since there are n leaves and $\mathcal{O}(\log^2 n)$ pairs for each leaf, all the skyline lists have total length $\mathcal{O}(n \log^2 n)$. We store a perfect hash table containing the non-empty lists.

Let $L = (a_1, b_1), \ldots, (a_\ell, b_\ell)$ be the skyline list for a pair of heavy paths, sorted such that $\text{depth}(a_1) > \cdots > \text{depth}(a_\ell)$ and $\text{weight}(a_1) > \cdots > \text{weight}(a_\ell)$ or, equivalently, $\text{depth}(b_1) < \cdots < \text{depth}(b_\ell)$ and $\text{weight}(b_1) < \cdots < \text{weight}(b_\ell)$. (Notice that, if a is induced with b, then every ancestor of a is also induced with b. Therefore, if (a_i, b_i) and (a_j, b_j) are both pairs in L and a_i is deeper than a_j then, by our definition of a pair in a skyline list, b_j must be deeper than b_i.) Let v_1 be a node in the first path and v_2 be a node in the second path. Suppose we want to find the pair of induced ancestors in these paths of v_1 and v_2 with maximum combined weight. With the approach described above, we would start with a pointer p to v_1 and a pointer q to the highest node in the second path, then move p up toward the highest node in the first path and q down toward v_2.

A geometric visualization is shown in Figure 2: the filled markers (from right to left) have coordinates $(\text{weight}(a_1), \text{weight}(b_1)), \ldots, (\text{weight}(a_\ell), \text{weight}(b_\ell))$, the hollow marker has coordinates $(\text{weight}(v_1), \text{weight}(v_2))$, and we seek the point (x, y) that is dominated both by some filled marker and by the hollow marker, such that $x + y$ is maximized. Notice $(\text{weight}(a_1), \text{weight}(b_1)), \ldots, (\text{weight}(a_\ell), \text{weight}(b_\ell))$ is a skyline — i.e., no marker dominates any other marker. There are five cases to consider: neither v_1 nor v_2 are induced with any other nodes in the paths; v_1 is induced with some node in the second path, but v_2 is not induced with any node in the first path; v_1 is not induced with any node in the second path, but v_2 is induced with some node in the first path; both v_1 and v_2 are induced with some nodes in the paths, but not with each other; and v_1 and v_2 are induced together.

It follows that finding the pair of induced ancestors in these paths of v_1 and v_2 with maximum combined weight is equivalent to finding the interval $(a_i, b_i), \ldots, (a_j, b_j)$ in L such that $\text{depth}(a_{i-1}) > \text{depth}(v_1) \geq \text{depth}(a_i)$ and $\text{depth}(b_j) \leq \text{depth}(v_2) < \text{depth}(b_{j+1})$, then finding the maximum in

$$\begin{array}{rcl}
\text{weight}(v_1) & + & \text{weight}(b_{i-1}), \\
\text{weight}(a_i) & + & \text{weight}(b_i), \\
\text{weight}(a_{i+1}) & + & \text{weight}(b_{i+1}), \\
 & \vdots & \\
\text{weight}(a_{j-1}) & + & \text{weight}(b_{j-1}), \\
\text{weight}(a_j) & + & \text{weight}(b_j), \\
\text{weight}(a_{j+1}) & + & \text{weight}(v_2)\,.
\end{array}$$

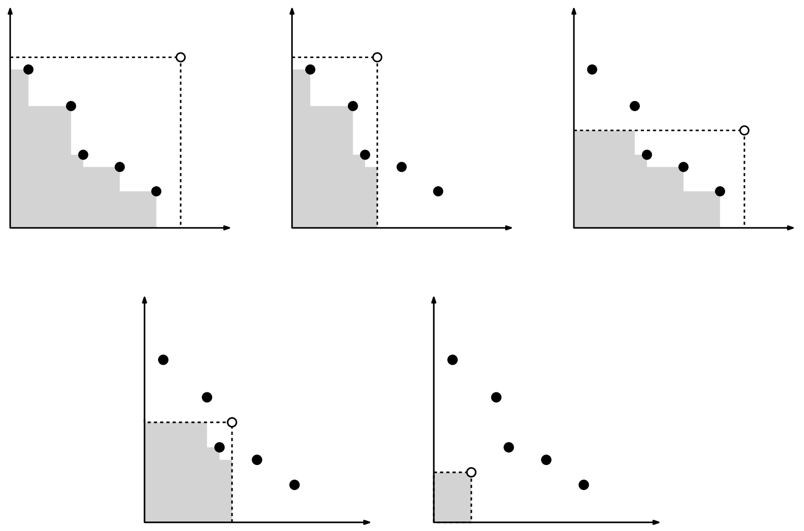

Figure 2: Finding the pair of induced ancestors of v_1 and v_2 with maximum combined weight is equivalent to storing a skyline such that, given a query point, we can quickly find the point (x, y) dominated both by some point on the skyline and by the query point, such that $x + y$ is maximized.

Therefore, if we store $\mathcal{O}(\ell)$-space predecessor data structures with $\mathcal{O}(\log\log n)$ query time [21] for $\text{depth}(a_1), \ldots, \text{depth}(a_\ell)$ and $\text{depth}(b_1), \ldots, \text{depth}(b_\ell)$ and an $\mathcal{O}(\ell)$-space range-maximum data with $\mathcal{O}(1)$ query time [8] for $\text{weight}(a_1) + \text{weight}(b_1), \ldots, \text{weight}(a_\ell) + \text{weight}(b_\ell)$, then in $\mathcal{O}(\log\log n)$ time we can find the pair of induced ancestors in these paths of v_1 and v_2 with maximum combined weight. Notice that we can assign v_1 and v_2 different weights when finding this pair of induced ancestors; this will be useful in Section 3.

Lemma 1 *We can store T_1 and T_2 in $\mathcal{O}(n \log^2 n)$ space such that, given nodes v_1 in T_1 and v_2 in T_2, in $\mathcal{O}(\log\log n)$ time we can find a pair of their induced ancestors in the same heavy paths with maximum combined weight, if such a pair exists.*

To find a pair of HIAs of v_1 and v_2, we consider the heavy-path decompositions of T_1 and T_2 as trees $\mathbf{T}_1$ and $\mathbf{T}_2$ of height $\mathcal{O}(\log n)$ in which each node is a heavy path and V is a child of U in $\mathbf{T}_1$ or $\mathbf{T}_2$ if the highest node in the path V is a child of a node in the path U in T_1 or T_2. We start with a pointer $\mathbf{p}$ to the path V_1 containing v_1 and a pointer $\mathbf{q}$ to the root of $\mathbf{T}_2$. If the skyline list for the nodes U_1 and U_2 indicated by $\mathbf{p}$ and $\mathbf{q}$ is non-empty, then we apply Lemma 1 to the deepest ancestors of v_1 and v_2 in U_1 and U_2, check whether the induced ancestors we find have greater combined weight than any induced pair we have seen so far and move $\mathbf{q}$ down one level toward V_2 (to execute the descent efficiently, in the very beginning we generate the whole path from V_2 containing v_2 to the root of $\mathbf{T}_2$); otherwise, we move $\mathbf{p}$ up one level toward the root of $\mathbf{T}_1$. This takes a total of $\mathcal{O}(\log n \log\log n)$ time. Again, we have the option of assigning v_1 and v_2 different weights for the purpose of the query.

Theorem 2 *We can store T_1 and T_2 in $\mathcal{O}(n \log^2 n)$ space such that, given nodes v_1 in T_1 and v_2 in T_2, in $\mathcal{O}(\log n \log\log n)$ time we can find a pair of their HIAs.*

In the full version of this paper we will reduce the query time in Theorem 2 to $\mathcal{O}(\log n)$ via fractional cascading [6]; however, this is not straightforward, as we need to modify our approach such that predecessor searches keep the same target as we change pairs of heavy paths and the hive or catalogue graph has bounded degree.

2.2 An $\mathcal{O}(n \log n)$-space data structure with $\mathcal{O}(\log^2 n)$ query time

To reduce the space bound in Theorem 2 to $\mathcal{O}(n \log n)$, we choose the orderings to impose on T_1 and T_2 such that each heavy path consists either entirely of leftmost children or entirely of rightmost children (except possibly for the highest nodes). We store an $\mathcal{O}(n \log^\epsilon n)$-space data structure [2] that supports $\mathcal{O}(\log\log n + k)$-time range-reporting queries on the grid described at the beginning of this section, where k is the number of points reported.

Notice that, if u_1 is an ancestor of w_1 in the same heavy path in T_1 and u_2 is an ancestor of w_2 in the same heavy path in T_2, then we can use a range-reporting query to find, e.g., the leaves that induce u_1 and u_2 together but not w_1 and w_2 together. Suppose there are $x_1 - 1$ and $x_2 - 1 > x_1 - 1$ leaves to the left of the

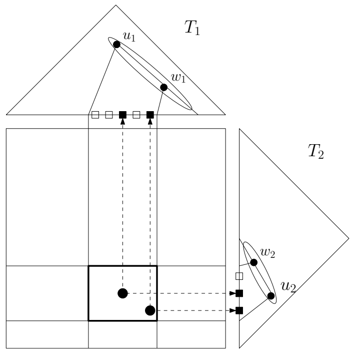

Figure 3: Suppose u_1 is an ancestor of w_1 in the same heavy path (shown as an oval) in T_1 and u_2 is an ancestor of w_2 in the same heavy path (also shown as an oval) in T_2. We can use a range-reporting query to find the leaves (shown as filled boxes) that induce u_1 and u_2 together but not w_1 and w_2 together.

leftmost leaf descendants of u_1 and w_1 in T_1, and $y_1 - 1$ and $y_2 - 1 > y_2 - 1$ leaves to the left of the rightmost leaf descendants of w_2 and u_2 in T_2; the cases when $x_2 < x_1$ or $y_2 < y_1$ are symmetric. Then the leaves that induce u_1 and u_2 together but not w_1 and w_2 together are indicated by markers in $[x_1..x_2 - 1] \times [y_1..y_2 - 1]$, as illustrated in Figure 3. That is, we query the cross product of the ranges of leaves in the subtrees of u_1 and u_2 but not w_1 and w_2. Similarly, we can find the leaves that induce u_1 and w_2 together but not u_2 and w_1 together (or vice versa), but then we query the cross product of the ranges of leaves in the subtrees of u_1 and w_2 but not w_1 (or of u_2 and w_1 but not w_2).

For each pair of heavy paths, we build a list containing each pair (a, b) such that, for some leaf x, a is the lowest ancestor of x in the first path and b is the lowest ancestor of x in the second path. We call this the paths' *extended list*, and consider it in decreasing order by the depth of the first component. Notice that an extended list is a supersequence of the the corresponding skyline list, but all the extended lists together still have total length $\mathcal{O}(n \log^2 n)$.

We do not store the complete extended lists; instead, we sample only every $(\log n)$-th pair, so the sampled lists take $\mathcal{O}(n \log n)$ space. We store a perfect hash function containing the non-empty sampled lists; we can still tell if a list was empty before sampling by using a range-reporting query to find any common leaf descendants of the highest nodes in the heavy paths. Given two consecutive sampled pairs from an extended list, in $\mathcal{O}(\log n)$ time we can recover the pairs between them using a range-reporting query, as described above.

With each sampled pair from an extended list, we store the preceding and succeeding pairs (possibly unsampled) that also belong to the corresponding skyline list; recall that the extended list is a supersequence of the skyline list. This gives us an irregular sampling (which may include duplicates) of pairs from the skyline lists, which has total size $\mathcal{O}(n \log n)$. Instead of storing predecessor and range-maximum data structures over the complete skyline lists, we store them over these sampled skyline lists, so we use a total of $\mathcal{O}(n \log n)$ space. Since these data structures are over sampled skyline lists, querying them indicates only which $(\log n)$-length block in a complete extended list contain the pair that would be returned by a query on a corresponding complete skyline list. We can recover any $(\log n)$-length block of a complete extended list in $\mathcal{O}(\log n)$ time with a range-reporting query, however, and then scan that block to find the pair with maximum combined weight.

If we sample only every $(\log^2 n)$-th pair from each extended list and use Chan et al.'s linear-space data structure for range reporting, then we obtain an even smaller (albeit slower) data structure for HIA queries.

Theorem 3 *We can store T_1 and T_2 in $\mathcal{O}(n \log n)$ space such that, given nodes v_1 in T_1 and v_2 in T_2, in $\mathcal{O}(\log^2 n)$ time we can find a pair of their HIAs. Alternatively, we can store T_1 and T_2 in $\mathcal{O}(n)$ space such that, given v_1 and v_2, in $\mathcal{O}(\log^{3+\epsilon} n)$ time we can find a pair of their HIAs.*

3 Longest Common Substrings

LZ-compressed indexes can use much less space than compressed suffix arrays or FM-indexes (see [3, 13, 14, 17]) when the indexed string is highly repetitive (e.g., versioned text documents, software repositories or databases of genomes of individuals from the same species). Although there is an extensive literature on the LCS problem, including Weiner's classic paper [22] on suffix trees and more recent algorithms for inputs compressed with the Burrows-Wheeler Transform (see [18]) or grammars (see [15]), we do not know of any grammar- or LZ-compressed indexes designed to support fast LCS queries.

Most LZ-compressed indexes are based on an idea by Kärkkäinen and Ukkonen [11]: we store a data structure supporting access to the indexed string $S[1..N]$; we store one Patricia tree [16] T_{rev} for the reversed phrases in the LZ parse, and another T_{suf} for the suffixes starting at phrase boundaries; we store a data structure for 4-sided range reporting for the grid on which there is a marker at point (x, y) if the x-th phrase in right-to-left lexicographic order is followed in the parse by the lexicographically y-th suffix starting at a phrase boundary; and we store a data structure for 2-sided range reporting for the grid on which there is a marker at point (x, y) if a phrase source begins at position x and ends at position y.

Given a pattern $P[1..m]$, for $1 \leq i \leq m$ we search for $(P[1..i])^{\mathrm{rev}}$ in T_{rev} (where the superscript rev indicates that a string is reversed) and for $P[i+1..m]$ in T_{suf}; access S to check that the path labels of the nodes where the searches terminate really are prefixed by $(P[1..i])^{\mathrm{rev}}$ and $P[i+1..m]$; find the ranges of leaves that are descendants of those nodes; and perform a 4-sided range-reporting query on the cross product of those ranges. This gives us the locations of occurrences of P in S that touch phrase boundaries. We then use recursive 2-sided range-reporting queries to find the phrase sources covering the occurrences we have found so far.

Rytter [19] showed how, if the LZ77 parse of S consists of n phrases, then we can build a balanced straight-line program (BSLP) for S with $\mathcal{O}(n \log N)$ rules. A BSLP for S is a context-free grammar in Chomsky normal form that generates S and only S such that, in the parse tree of S, every node's height is logarithmic in the size of its subtree. We showed in a previous paper [9, 10] how we can store a BSLP for S in $\mathcal{O}(n \log N)$ space such that extracting a substring of length m from around a phrase boundary takes $\mathcal{O}(m)$ time. Using this data structure for access to S and choosing the rest of the data structures appropriately, we can store S in $\mathcal{O}(n \log N)$ space such that listing all the occ occurrences of P in S takes $\mathcal{O}\big(m^2 + \mathrm{occ} \log \log N\big)$ time.

Our solution can easily be modified to find the LCS of P and S in $\mathcal{O}\big(m^2 \log \log n\big)$ time: we store the BSLP for S; the two Patricia trees T_{rev} and T_{suf}, with the nodes weighted by the lengths of their path labels; and an instance of Chan et al.'s $\mathcal{O}(n \log \log n)$-space range-emptiness data structure with $\mathcal{O}(\log \log n)$ query time, instead of the data structure for 4-sided range range reporting. All these data structures together take a total of $\mathcal{O}(n \log N)$ space. By the definition of the LZ77 parse, the first occurrence of every substring in S touches a phrase boundary. It follows that we can find the LCS of P and S by finding, for $1 \leq i \leq m$, values h and j such that some phrase ends with $P[h..i]$ and the next phrase starts with $P[i+1..j]$ and $j-h+1$ is maximum.

For $1 \leq i \leq m$ we search for $(P[1..i])^{\mathrm{rev}}$ in T_{rev} and for $P[i+1..m]$ in T_{suf}, as before; access S to find the longest common prefix (LCP) of $(P[1..i])^{\mathrm{rev}}$ and the path label of the node where the search in T_{rev} terminates, and the LCP of $P[i+1..m]$ and the path label of the node where the search in T_{suf} terminates; take v_1 and v_2 to be the loci of those LCPs, and treat them as having weights equal to the lengths of the LCPs; and then use the range-emptiness data structure and the simple HIA algorithm described at the beginning of Section 2 to find h and j for this choice of i. For each choice of i this takes $\mathcal{O}(m \log \log n)$ time, so we use $\mathcal{O}\big(m^2 \log \log n\big)$ time in total.

Lemma 4 *We can store S in $\mathcal{O}(n \log N)$ space such that, given a pattern P of length m, we can find the LCS of P and S in $\mathcal{O}\big(m^2 \log \log n\big)$ time.*

We now show how to use our data structure for HIA queries to reduce the dependence on m in Lemma 4 from quadratic to linear.

Ferragina [7] showed how, by storing path labels' Karp-Rabin hashes [12] and rebalancing the Patricia trees via centroid decompositions, in a total of $\mathcal{O}(m \log n)$ time we can find with high probability the nodes where the searches for $(P[1..i])^{\mathrm{rev}}$ and $P[i+1..m]$ terminate, for all choices of i. In our previous paper we showed how, by storing the hash of the expansion of each non-terminal in the BSLP for S, in $\mathcal{O}(m \log m)$ time we can then verify with high probability that the path labels of the nodes where the searches terminate really are prefixed by $(P[1..i])^{\mathrm{rev}}$ and $P[i+1..m]$.

Using the same techniques, in $\mathcal{O}(m \log m)$ time we can find with high probability for all choices of i, the LCP of $(P[1..i])^{\mathrm{rev}}$ and any reversed phrase, and the LCP of $P[i+1..m]$ and any suffix starting at a phrase boundary. If $m = n^{\mathcal{O}(1)}$ then $\mathcal{O}(m \log m) = \mathcal{O}(m \log n)$. If $m = n^{\omega(1)}$, then we can preprocess P and batch the searches for the LCPs, to perform them all in $\mathcal{O}(m)$ time.

More specifically, to find the LCPs of $P[2..m], \ldots, P[m..m]$ and suffixes starting at phrase boundaries, we first build the suffix array and LCP array of P. For $1 \leq i \leq m$ we use Ferragina's data structure to find the suffix starting at a phrase boundary whose LCP with $P[i+1..m]$ is maximum. For each phrase boundary, we use the suffix array and LCP array of P to build a Patricia tree for the suffixes of P whose LCPs we will seek at that phrase boundary. We then balance these Patricia trees via centroid decompositions. For each phrase boundary, we determine the length of the LCP of any suffix of P and the suffix starting at that phrase boundary. We then use the LCP array of P to find the LCPs of $P[2..m], \ldots, P[m..m]$ and suffixes starting at phrase boundaries. This takes a total of $\mathcal{O}(m)$ time. Finding the LCPs of $P[1], (P[1..2])^{\mathrm{rev}}, \ldots, P^{\mathrm{rev}}$ is symmetric.

Suppose we already know the LCPs of $P[1], (P[1..2])^{\mathrm{rev}}, \ldots, P^{\mathrm{rev}}$ and the reversed phrases, and the LCPs of $P[2..m], \ldots, P[m]$ and the suffixes starting at phrase boundaries. Then in a total of $\mathcal{O}\big(m \log^2 n\big)$ time we can find with high probability values h and j such that some phrase ends with $P[h..i]$ and the next phrase starts with $P[i+1..j]$ and $j-h+1$ is maximum, for each choice of i. To do this, we use m applications of Theorem 3 to T_{rev} and T_{suf}, one for each partition of P into a prefix and a suffix. This gives us the following result:

Theorem 5 *Let S be a string of length N whose LZ77 parse consists of n phrases. We can store S in $\mathcal{O}(n \log N)$ space such that, given a pattern P of length*

m, we can find with high probability a longest substring common to P and S in $\mathcal{O}\big(m \log^2 n\big)$ time.

We can reduce the time bound in Theorem 5 to $\mathcal{O}(m \log n \log\log n)$ at the cost of increasing the space bound to $\mathcal{O}\big(n(\log N + \log^2 n)\big)$, by using the data structure from Theorem 2 instead of the one from Theorem 3. In fact, as we noted in Section 2, in the full version of this paper we will also eliminate the $\log\log n$ factor here.

References

[1] A. Amir, G. M. Landau, M. Lewenstein and D. Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2), 2007.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proc. Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[3] D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1–2):54–101, 2012.

[4] A. L. Buchsbaum, M. T. Goodrich, and J. Westbrook. Range searching over tree cross products. In *Proc. European Symposium on Algorithms*, pages 120–131, 2000.

[5] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proc. Symposium on Computational Geometry*, pages 1–10, 2011.

[6] B. Chazelle and L. J. Guibas. Fractional cascading. *Algorithmica*, 1(2):133–191, 1986.

[7] P. Ferragina. On the weak prefix-search problem. *Theoretical Computer Science*, 483:75–84, 2013.

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

[9] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proc. Conference on Language and Automata Theory and Applications*, pages 240–251, 2012.

[10] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. Technical Report 1109.3954v6, arxiv.org, 2012.

[11] J. Kärkkäinen and E. Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. South American Workshop on String Processing*, pages 141–155, 1996.

[12] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[13] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[14] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[15] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoretical Computer Science*, 410(8–10):900–913, 2009.

[16] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

[17] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007.

[18] E. Ohlebusch, S. Gog, and A. Kügel. Computing matching statistics and maximal exact matches on compressed full-text indexes. In *Proc. Symposium on String Processing and Information Retrieval*, pages 347–358, 2010.

[19] W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1-3):211–222, 2003.

[20] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. In *Proc. Symposium on Theory of Computing*, pages 114–122, 1981.

[21] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[22] P. Weiner. Linear pattern matching algorithms. In *Proc. Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[23] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3), 1977.