Pattern matching in Lempel-Ziv compressed strings: fast, simple, and deterministic[*]

Paweł Gawrychowski

Institute of Computer Science,
University of Wrocław,
ul. Joliot-Curie 15, 50–383 Wroclaw, Poland
`gawry@cs.uni.wroc.pl`

Abstract. Countless variants of the Lempel-Ziv compression are widely used in many real-life applications. This paper is concerned with a natural modification of the classical pattern matching problem inspired by the popularity of such compression methods: given an uncompressed pattern $s[1 . . m]$ and a Lempel-Ziv representation of a string $t[1 . . N]$, does s occur in t? Farach and Thorup [6] gave a randomized $\mathcal{O}(n \log^2 \frac{N}{n} + m)$ time solution for this problem, where n is the size of the compressed representation of t. Building on the methods of [4] and [7], we improve their result by developing a faster and fully deterministic $\mathcal{O}(n \log \frac{N}{n} + m)$ time algorithm. Note that for highly compressible texts, $\log \frac{N}{n}$ might be of order n, so for such inputs the improvement is very significant. A (tiny) fragment of our method can be used to give an asymptotically optimal solution for the substring hashing problem considered by Farach and Muthukrishnan [5].

Key-words: pattern matching, compression, Lempel-Ziv

1 Introduction

Effective compression methods allow us to decrease the space requirements which is clearly worth pursuing on its own. On the other hand, we do not want to store the data just for the sake of having it: we want to process it efficiently on demand. This suggest an interesting direction: can we process the data without actually decompressing it? Or, in other words, can we speed up processing if the compression ratio is high? Answer to such questions clearly depends on the particular compression and processing method chosen. In this paper we focus on Lempel-Ziv (also known as LZ77, or simply LZ for the sake of brievity), one of the most commonly used compression methods being the basis of the widely popular `zip` and `gz` archive file formats, and on pattern matching, one of the most natural text processing problem we might encounter. More specifically, we deal with the compressed pattern matching problem: given an uncompressed pattern $s[1 . . m]$ and a LZ representation of a string $t[1 . . N]$, does s occur in t? This line of research has been addressed before quite a few times already. Amir, Benson,

---

and Farach [1] considered the problem with LZ replaced by Lempel-Ziv-Welch (a simpler and easier to implement specialization of LZ), giving two solutions with complexities $\mathcal{O}(n \log m + m)$ and $\mathcal{O}(n + m^2)$, where n is the size of the compressed representation. The latter has been soon improved [12] to $\mathcal{O}(n + m^{1+\epsilon})$. Then Farach and Thorup [6] considered the problem in its full generality and gave a (randomized) $\mathcal{O}(n \log^2 \frac{N}{n} + m)$ time algorithm for the LZ case. Their solution consists of two phases, called *winding* and *unwinding*, the first one uses a cleverly chosen potential function, and the second one adds fingerprinting in the spirit of string hashing first developed by Karp and Rabin [10]. While a recent result of [8] shows that the winding can be performed in just $\mathcal{O}(n \log \frac{N}{n})$, it is not clear how to use that to improve the whole running time (or remove randomization). In this paper we take a completely different approach, and manage to develop a $\mathcal{O}(n \log \frac{N}{n} + m)$ time algorithm. This complements our recent result from SODA'11 [7] showing that in case of Lempel-Ziv-Welch, the compressed pattern matching can be solved in optimal linear time. While we were not able to achieve linear time for the general LZ case, the algorithm developed in this paper not only significantly improves the previously known time bounds, but also is fully deterministic and (relatively) simple. Moreover, LZ compression allows for an exponential decrease in the size of the compressed text, while in LZW n is at least $\sqrt{N}$. In order to deal with such highly compressible texts efficiently we need to combine quite a few different ideas, and the nonlinear time of our (and the previously known) solution might be viewed as an evidence that LZ is substantially more difficult to deal with than LZW.

2 Overview of the algorithm

Our goal is to detect an occurrence of s in a given Lempel-Ziv compressed text $t[1 \mathinner{.\,.} N]$. The Lempel-Ziv representation is quite difficult to work with efficiently, even for a such simple task as extracting a single letter. The starting point of our algorithm is thus transforming the input into a *straight-line program*, which is a context-free grammar with each nonterminal generating exactly one string. For that we use the method of Charikar *et al.* [4] to construct a SLP of size $\mathcal{O}(n \log \frac{N}{n})$ with additional property that all productions are *balanced*, meaning that the right sides are of the form XY with $\frac{\alpha}{1-\alpha} \le \frac{|X|}{|Y|} \le \frac{1-\alpha}{\alpha}$ for some constant α, where $|X|$ is the length of the (unique) string generated by X. Note that Rytter gave a much simpler algorithm [13] with the same size guarantee, using the so-called AVL grammars. We need the grammar to be balanced, though. We also need to add one small modification to the method of [4] in order to allow self-referential LZ compression.

   After transforming the text into a balanced SLP, for each nonterminal we try to check if the string it represents occurs inside s, and if so, compute the position of (any) its occurrence. Otherwise we would like to compute the longest prefix (suffix) of this string which is a suffix (prefix) of s. Computing such data for each nonterminal separately seems rather expensive, and we try to process the nonterminals in $\mathcal{O}(\log N)$ groups corresponding to the (truncated) logarithm

of their length. Because of some technical difficulties, we cannot really afford to check if the represented string occurs in s for each nonterminal exactly, though. Nevertheless, we can compute some approximation of this information, and by using a tailored variant of binary search applied to all nonterminals in a single group at once, we manage to process the whole grammar in time proportional to its size. The final element of the whole puzzle is a constant time procedure which detects an occurrence of s inside concatenation of two its substrings.

Because of the space constraints, some proofs are in the appendix.

3   Preliminaries

As usually, $|w|$ stands for the length of w, $w[i \mathinner{\ldotp\ldotp} j]$ refers to its fragment of length $j - i + 1$ beginning at the i-th character, where characters are numbered starting from 1. All strings considered in the paper are over an alphabet Σ of polynomial cardinality, namely $\Sigma = \{1, 2, \ldots, (n + m)^c\}$. A border of a string $w[1 \mathinner{\ldotp\ldotp} |w|]$ is a fragment which is both a prefix and a suffix of w, i.e., $w[1 \mathinner{\ldotp\ldotp} i] = w[|w| - i + 1 \mathinner{\ldotp\ldotp} |w|]$. We identify such fragment with its length and say that $\text{border}(t) = \{i_1, \ldots, i_k\}$ is the set of all borders of t. A period of a string $w[1 \mathinner{\ldotp\ldotp} |w|]$ is an integer p such that $w[i] = w[i + p]$ for all $1 \le i \le |w| - p$. Note that p is a period of iff $|w| - p$ is a border. The following lemma is a well-known property of periods.

Lemma 1 (Periodicity lemma). *If p and q are both periods of w, and $p + q \le |w| + \gcd(p, q)$, then $\gcd(p, q)$ is a period as well.*

The Lempel-Ziv representation of a string $t[1 \mathinner{\ldotp\ldotp} N]$ is a sequence of triples $(start_i, len_i, next_i)$ for $i = 1, 2, \ldots, n$, where n is the size of the representation. $start_i$ and len_i are nonnegative integers, and $next_i \in \Sigma$. Such triple refers to a fragment of the text $t[start_i \mathinner{\ldotp\ldotp} start_i + len_i - 1]$ and defines $t[1 + \sum_{j<i} len_j \mathinner{\ldotp\ldotp} \sum_{j \le i} len_j] = t[start_i \mathinner{\ldotp\ldotp} start_i + len_i - 1]next_i$. We require that $start_i \le 1 + \sum_{j<i} len_j$ if $len_i > 0$. The representation is not self-referential if all fragments we are referring to are already defined, i.e., $start_i + len_i - 1 \le \sum_{j<i} len_j$ for all i. The sequence of triples is often called the *LZ parse* of text.

Straight-line program is a context-free grammar in the Chomsky normal form such that the nonterminals $X_1, X_2, \ldots, X_s$ can be ordered in such a way that each X_i occurs exactly once as a left side, and whenever $X_i \to X_j X_k$ it holds that $j, k < i$. Such grammar derives exactly one string, and we identify each nonterminal with the unique string it derives, so $|X|$ stands for the length of the string derived from X. We call a straight-line program (SLP) *balanced* if for each production $X \to YZ$ both $|Y|$ and $|Z|$ are bounded by a constant fraction of $|X|$.

We preprocess the pattern s using standard tools (suffix trees [14] built for both the pattern and the reversed pattern, and lowest common ancestor queries [2]) to get the following primitives.

Lemma 2. *Pattern s can be preprocessed in linear time so that given i, j, k representing any two fragments $s[i \mathinner{\ldotp\ldotp} i + k]$ and $s[j \mathinner{\ldotp\ldotp} j + k]$ we can find their longest common prefix (suffix) in constant time.*

Lemma 3. *Pattern s can be preprocessed in linear time so that given any fragment $s[i \mathinner{..} j]$ we can find its longest suffix (prefix) which is a prefix (suffix) of the whole pattern in constant time, assuming we know the (explicit or implicit) vertex corresponding to $s[i \mathinner{..} j]$ in the suffix tree built for s (reversed s).*

Proof. See the appendix.   □

We will also use the suffix array SA built for s [9]. For each suffix of s we store its position inside SA, and treat the array as a sequence of strings rather than a permutation of $\{1, 2, \ldots, |s|\}$. Given any word w, we will say that it occurs at position i in the SA if w begins $s[SA[i] \mathinner{..} |s|]$. Similarly, the fragment of SA corresponding to w is the (maximal) range of entries at which w occurs.

4   Snippets toolbox

In this section we develop a few efficient procedures operating on fragments of the pattern, which we call *snippets*:

Definition 1. *A snippet is a substring of the pattern $s[i \mathinner{..} j]$. If $i = 1$ we call it a prefix snippet, if $j = m$ a suffix snippet.*

We identify snippets with the substrings they represent, and use $|s|$ to denote the length of the string represented by s. A snippet is stored as a pair (i, j).

The two results of this section that we are going to use later build heavily on the contents of [7]. Specifically, Lemma 6 appears there as Lemma 5. To prove it, we first need the following simple and relatively well known property of borders.

Lemma 4. *If the longest border of t is of length $b \geq \frac{|t|}{2}$ then all borders of length at least $\frac{|t|}{2}$ create one arithmetic progression. More specifically,* $\mathrm{border}(t) \cap \left\{ \frac{|t|}{2}, \ldots, |t| \right\} = \left\{ |t| - \alpha p : 0 \leq \alpha \leq \frac{|t|}{2p} \right\}$, *where $p = |t| - b$ is the period of t. We call this set the long borders of t.*

We need to extract borders of prefix and suffix snippets efficiently.

Lemma 5. *Pattern s can be preprocessed in linear time so that we can find the longest border of each its prefix (suffix) in constant time.*

The first result tells how to detect an occurrence of the pattern in a concatenation of two snippets. We will perform a lot of such operations, and an efficient implementation is crucial.

Lemma 6 (see Lemma 5 of [7]). *Given a prefix snippet and a suffix snippet we can detect an occurrence of the pattern in their concatenation in constant time.*

Proof. See the appendix.   □

The second result concerning snippets that we need can be deduced from Lemma 6 and Lemma 8 of [7], but we prefer to present here an explicit proof for the sake of completeness.

Lemma 7. *Given a prefix snippet s_1 and a snippet s_2 for which we know the corresponding node in the suffix tree, we can compute the longest prefix of s which is a suffix of $s_1 s_2$ in time $\mathcal{O}\left(\max\left(1, \log \frac{|s_1|}{|s_2|}\right)\right)$.*

Proof. See the appendix. $\square$

Note that the running time from the above lemma stays constant as long as $|s_1|$ is bounded from above by a constant fraction of $|s_2|$.

5 Constructing balanced grammar

Recall that a LZ parse is a sequence of triples $(start_i, len_i, next_i)$ for $i = 1, 2, \ldots, n$. In the not self-referential variant considered in [4], we require that $start_i + len_i - 1 \leq \sum_{j=1}^{i} len_j$ so that each triple refers only to the prefix generated so far. Although such assumption is made by some LZ-based compressors, [6] deals with the compressed pattern matching problem in its full generality, allowing self-references. Thus for the sake of completeness we need to construct a balanced grammar from a potentially self-referential LZ parse. It turns out that a small modification of a known method is enough for this task.

Lemma 8 (see Theorem 1 of [4]). *Given a (potentially self-referential) LZ parse of size n, we can build a α-balanced SLP of size $\mathcal{O}(n \log \frac{N}{n})$ describing the same string of length N, for any constant $0 < \alpha \leq 1 - \frac{\sqrt{2}}{2}$. Running time of the construction is proportional to the size of the output.*

Proof. See the appendix. $\square$

As a result we get a context-free grammar in which all nonterminals derive exactly one string, and right sides of all productions are of the form XY with $\frac{\alpha}{1-\alpha} \leq \frac{|X|}{|Y|} \leq \frac{1-\alpha}{\alpha}$. The exact value of α is not important, we only need the fact that both $\frac{|X|}{|Y|}$ and $\frac{|Y|}{|X|}$ are bounded from above. For the sake of concreteness we assume $\alpha = 0.25$. We also need to compute $|X|$ for each nonterminal X, and to group the nonterminals according to the (rounded down) logarithm of their length, with the base of the logarithm to be chosen later. Note that taking logarithms is not necessarily a constant time operations in our model. Of course we could preprocess $\log_b x$ for each $x \leq N$, but it introduces an additional $\mathcal{O}(N^\epsilon)$ addend in the running time. We can use the fact that the grammar is balanced instead: if $X \to YZ$, then $\log_b |X| \leq \beta + \max\left(\log_b |Y|, \log_b |Z|\right)$ for some constant β depending only on α and b, and the logarithms can be computed for all nonterminals in a bottom-up fashion using just linear time.

6   Processing balanced grammar

For each nonterminal X we would like to check if the string it represents occurs inside s. This information seems rather time consuming to extract: it seems that in order to compute it efficiently we would need to implement a procedure for detecting if a concatenation of two substrings of s occurs in s as well. In order to get the claimed running time we would need to answer such queries in constant time after a linear preprocessing, which seems difficult to achieve. Thus we work with an approximation of this information instead.

Definition 2. *A cover of a nonterminal X is pair of snippets $s[i\mathbin{..}i+2^k-1]$ and $s[j\mathbin{..}j+2^k-1]$ such that $2^k < |X| \le 2^{k+1}$, $s[i\mathbin{..}i+2^k-1]$ is a prefix of the string represented by X, and $s[j\mathbin{..}j+2^k-1]$ is a suffix of the string represented by X. We call k the order of X's cover.*

We try to find the cover of each nonterminal X. If there is none, we know that the string it represents does not occur inside s. In such case we compute $\mathrm{prefix}(X)$ ($\mathrm{suffix}(X)$), its longest prefix (suffix) which is a suffix (prefix) of the whole s. More precisely, we either:

1. compute the cover, in such case the string represented by X might or might no occur in s,
2. do not compute the cover, in such case the string represented by X does not occur in s.

As we will see later, it is possible to extract $\mathrm{prefix}(X)$ and $\mathrm{suffix}(X)$ from the cover of X using Lemma 7 in constant time, and the information about $\mathrm{prefix}(X)$ and $\mathrm{suffix}(X)$ for each nonterminal X is enough to detect an occurrence:

Lemma 9. *If s occurs in a string represented by a SLP then there exists a production $X \to YZ$ such that s occurs in $\mathrm{suffix}(Y)\,\mathrm{prefix}(Z)$.*

Proof. See the appendix. □

We process the nonterminals in groups. Nonterminals in the k-th group $\mathcal{G}_\ell = \{X_1, X_2, \dots X_s\}$ are chosen so that $(\frac{4}{3})^\ell < |X_i| \le (\frac{4}{3})^{\ell+1}$. The groups are disjoint so $\sum_\ell |\mathcal{G}_\ell| = \mathcal{O}(n \log \frac{N}{n})$. We start with computing the covers of nonterminals in $\mathcal{G}_1$ naively. Then we assume that all nonterminal in $\mathcal{G}_{\ell-1}$ are already processed, and we consider $\mathcal{G}_\ell$. Because the grammar is 0.25-balanced, if $X_i \to Y_iZ_i$ then $|Y_i|, |Z_i| \le \frac{3}{4}|X_i|$, and Y_i, Z_i belong to already processed $\mathcal{G}_{\ell'}$ with $\ell - 5 \le \ell' < \ell$. If for some Y_i or Z_i we do not have the corresponding cover, neither must have the corresponding X_i, so we use Lemma 7 to calculate $\mathrm{prefix}(X_i)$, $\mathrm{suffix}(X_i)$, and remove X_i from $\mathcal{G}_\ell$. For all remaining X_i we are left with the following task: given the covers of Y_i and Z_i, compute the cover of X_i, or detect that the represented string does not occur in s and so we do not have to compute the cover. Note that the known covers are of order k with $k_{min} = \lfloor \ell \log \frac{4}{3} \rfloor - 3 \le k \le \lceil \ell \log \frac{4}{3} \rceil = k_{max}$.

We reduce computing covers to a sequence of batched queries of the form: given a sequence of pairs of snippets $s[i\mathbin{..}i+2^{k_1}-1]$, $s[j\mathbin{..}j+2^{k_2}-1]$ does

their concatenation occur in s, and if so, what is the corresponding snippet? We call this merging the pair. For each ℓ we will require solving a constant number of such problems with $k_{min} \leq k_1, k_2 \leq k_{max}$, each containing $\mathcal{O}(|\mathcal{G}_\ell|)$ queries. We call this problem BATCHED-POWERS-MERGE. Before we develop an efficient solution for such question, lets see how it can be used to compute covers.
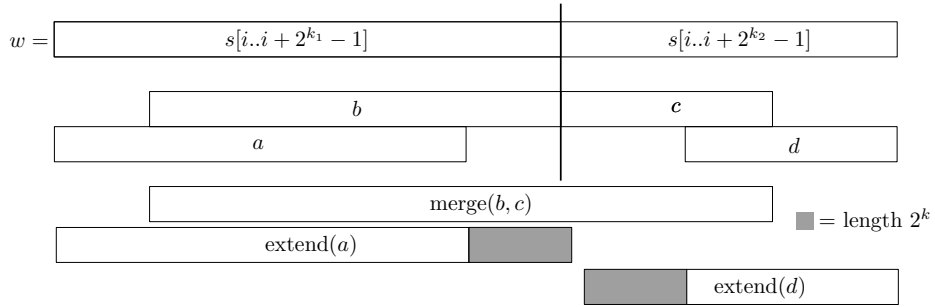

Fig. 1. Computing cover of a pair of snippets.

Lemma 10. *Computing covers of the nonterminals in any $\mathcal{G}_\ell$ can be reduced in linear time to a constant number of calls to BATCHED-POWERS-MERGE, with the number of pairs in each call bounded by $|\mathcal{G}_\ell|$.*

Proof. Recall that for each given pair of snippets we have their covers available, and the orders of those covers are from $\{k, k+1, \ldots, k+4\}$. Consider the situation for a single pair, see Figure 1. First we merge b and c to get merge(b, c). Then we extend a to the right and d to the left by merging with the corresponding fragments of merge(b, c) of length 2^k, and call the results extend(a) and extend(d). Then we would like iteratively extend both a and d with fragments of such length as long as it does not result in sticking out of the considered word w. To do that, we need to have the snippets corresponding to those fragments available. Consider the situation for a: first we extract the snippets from merge(b, c), then from extend(d). We claim that we are always able to perform such extraction: if the next 2^k characters fall outside merge(b, c), the distance to the left boundary of d does not exceed 2^k and thus we can use extend(d). If during this extending procedure the merging fails, the pair does not represent a substring of s. Otherwise we get the snippet corresponding to the prefix and suffix of w of lengths $|w| - |w| \bmod 2^k$, which allows us to extract the prefix and suffix of length $2^{k'}$ where $2^{k'} < |w| \leq 2^{k'+1}$, because $k \leq k'$.

To finish the proof, note that for a single pair we need a constant number of merges. Thus we can do the merging in parallel for all pairs in a constant number of calls to BATCHED-POWERS-MERGE. $\qquad\square$

Now we only have to develop the algorithm for BATCHED-POWERS-MERGE. A simple solution would be to do a binary search in the suffix array built for s for each pair separately: we can compare $s[i \,..\, i + 2^{k_1} - 1]s[j \,..\, j + 2^{k_2} - 1]$ with any suffix of s in constant time using at most two LCP queries so the search

takes $\mathcal{O}(\log m)$ time, which is way too slow. To get a better running time first observe that we can order all concatenations from a single problem efficiently.

Lemma 11. *Given $\mathcal{O}(|\mathcal{G}_\ell|)$ pairs of words of the form $s[i \mathinner{..} i + 2^{k_1} - 1]$, $s[j \mathinner{..} j + 2^{k_2} - 1]$ with $k_{min} \leq k_1, k_2 \leq k_{max}$ we can lexicographically sort their concatenations in time $\mathcal{O}(|\mathcal{G}_\ell| + m^\epsilon)$ if $|k_{max} - k_{min}| \in \mathcal{O}(1)$.*

Proof. We split the words to be sorted into a constant number of chunks of length $2^{k_{min}}$. Then we would like to assign numbers to those chunks so that $\mathrm{nr}(s[i \mathinner{..} i + 2^{k_{min}} - 1]) < \mathrm{nr}(s[j \mathinner{..} j + 2^{k_{min}} - 1])$ iff $s[i \mathinner{..} i + 2^{k_{min}} - 1]) <_{lex} s[j \mathinner{..} j + 2^{k_{min}} - 1])$. To compute all $\mathrm{nr}(s[i \mathinner{..} i + 2^{k_{min}} - 1])$ we retrieve the positions of $s[i \mathinner{..} m]$ in the suffix array. Then we sort the resulting list of $\mathcal{O}(|\mathcal{G}_\ell|)$ integers using radix sort, i.e., by $\frac{1}{\epsilon}$ rounds of counting sort. The time required by this sorting is linear plus $\mathcal{O}(m^\epsilon)$. After sorting we scan the list and identify different suffixes with the same prefix of length $2^{k_{min}}$, such suffixes belong to continuous blocks whose boundaries can be identified using LCP queries. Then the original task reduces to sorting a list of constant length vectors consisting of integers not exceeding m, which can be done efficiently using radix sort.   □

We apply the above lemma to all calls to BATCHED-POWERS-MERGE corresponding to nonempty $\mathcal{G}_\ell$. If $(\frac{4}{3})^\ell > m$ then clearly the corresponding $\mathcal{G}_\ell$ is empty, so the total running time of this part is just $\mathcal{O}(m^\epsilon \log m + \sum_\ell |\mathcal{G}_\ell|) = \mathcal{O}(m + n \log \frac{N}{n})$. Now that the queries are sorted, we can try to reuse the information acquired during binary search. For example, we can start the search for a given pair starting from the place where the lexicographically previous pair was found at. This might be still too slow though. To accelerate the search we develop a constant time procedure for locating the fragment of the suffix array corresponding to all occurrences of any $s[i \mathinner{..} i + 2^k - 1]$.

Lemma 12. *The pattern s can be processed in linear time so that given any $s[i \mathinner{..} i + 2^k - 1]$ we can compute its first and the last occurrence in the suffix array of s in constant time.*

Proof. It is enough to show that the suffix tree T built for s can be preprocessed in linear time so that we can locate the (implicit or explicit) vertex corresponding to any fragment which is a power of 2 in constant time. For that we should locate an ancestor of a given leaf which is at specified depth 2^k. This can be reduced to the so-called weighted ancestor queries: given a node-weighted tree, with the weights nondecreasing on any root-to-leaf path, preprocess it to find the predecessor of a given weight among the ancestors of v efficiently. Unfortunately, all known solutions for this problem [5,11] give nonconstant query time. We wish to improve this time by abusing the fact that only ancestors at depths 2^k are sought. First note that such ancestor is not necessarily an explicit vertex. We start with considering all edges of T. For each such edge e, we compute the smallest k such that e contains an implicit vertex at depth 2^k (there might be none), and split the edge to make it explicit. We call all original vertices at depths being powers of 2, and all new vertices, marked. For each leaf v we would

like to compute the depths of all its marked ancestors $marked(v)$. This can be done in linear time by a single top-bottom transversal, and the information can be stored in a single $\Theta(\log|s|)$-bit word. Then we construct $T' = \mathrm{compress}(T)$ containing only the leaves and marked vertices of T by collapsing all maximal fragments of T without such vertices. Then we build the level ancestor data structure for T' [3] allowing us to find the k-th ancestor of any vertex in constant time. Now given i and k we first locate the leaf v corresponding to $s[i..|s|]$ in T, then take a look at its bitvector $\mathrm{marked}(v)$. We can compute in constant time $\ell = \{k' > k : k' \in \mathrm{marked}(v)\}$ and retrieve the ℓ-th ancestor of v in T'. Going back to T we get a node with the same (lexicographically) smallest and largest suffix in its subtree as the node corresponding to $s[i..i+2^k-1]$. In fact can use a simpler solution because the depth of T' is just $\log n$. See the appendix.   □

Observe that the above lemma can be used to give an optimal solution for a slight relaxation of the *substring fingerprints* problem considered in [5]. This problem is defined as follows: given a string s, preprocess it to compute any *substring hash* $h_s(s[i..j])$ efficiently. We require that:

1. $h_s(s[i..j]) \in [1, \mathcal{O}(|s|^2)]$ so that the values can be operated on efficiently,
2. $h_s(s[i..j]) = h_s(s[k..l])$ iff $s[i..j] = s[k..l]$.

If we allow the range of h_s to be slightly larger, say $\mathcal{O}(|s|^3)$, a direct application of the above lemma allows us to evelute the fingerprints in constant time after a linear preprocessing.

Theorem 1. *Substring fingerprints of size $\mathcal{O}(|s|^3)$ can be computed in constant time after a linear time preprocessing.*

Proof. See the appendix.   □

   While the range of h_s is not optimal, it can be evaluated and operated on in constant time which should be enough to replace the results of [5].

   Now getting back to the original question, to locate $w = s[i..i+2^{k_1}-1]s[j..j+2^{k_2}-1]$ in the suffix array, we first look up the fragment corresponding to its prefix $s[i..i+2^{k_{min}}-1]$ using Lemma 12. Then we apply binary search in this fragment, with the exception that if the previous binary search was in this fragment as well, we start from the position it finished, not the beginning of the fragment. Additionally, the binary search is performed starting from the beginning and the end of the interval at the same time, see TWO-WAY-BINARY-SEARCH below. If the initial interval is $[a, b]$ and the position we are after is r, the running time of such modified search is $\mathcal{O}(\log\min(r-a+1, b-r+1))$ because we can compare w with any $s[i..|s|]$ in constant time using at most two LCP queries. The standard method results in a slightly worse $\mathcal{O}(\log(b-a+1))$ time, and it turns out that the decreasing this time is extremely useful.

   The goal of BATCHED-POWERS-MERGE is to search for the first occurrence of a string in the suffix array. A sequence of such searches should be seen as follows: we keep a partition of the whole $[1, |s|]$ into a number of disjoint intervals. Doing a single search splits at most one interval into two parts at the position

---

Algorithm 1 TWO-WAY-BINARY-SEARCH(a, b, w)

1: $x \leftarrow a$, $y \leftarrow b$
2: $k \leftarrow 1$
3: **while** $2^k \leq b - a$ **do**
4:     **if** $w <_{lex} s[SA[a + 2^k]]$ **then**
5:         $y \leftarrow a + 2^k$
6:         **break**
7:     **end if**
8:     **if** $s[SA[b - 2^k]] <_{lex} w$ **then**
9:         $x \leftarrow b - 2^k$
10:        **break**
11:     **end if**
12:     $k \leftarrow k + 1$
13: **end while**
14: $r \leftarrow$ binary search for w in $s[SA[x] \mathinner{..} |s|], s[SA[x + 1] \mathinner{..} |s|], \ldots, s[SA[y] \mathinner{..} |s|]$
15: **return** r

---

of the first occurrence. If the first occurrence is exactly at an already existing boundary, there is no split, otherwise we say that those two smaller intervals have been created in phase k_{min} (recall that k_{min} linearly depends on ℓ), and intervals created in phase k_{min} are kept in a list $I_{k_{min}}$. We will prune those lists to contain just the minimal under inclusion intervals.

Lemma 13. *All $\mathcal{O}(\log m)$ calls to* BATCHED-POWERS-MERGE *run in total time* $\mathcal{O}(m + \sum_\ell |\mathcal{G}_\ell|)$.

Proof. First note that the sorting in line 16 can be performed in time $\mathcal{O}(m^\epsilon + |I_{k_{min}}| + |\mathcal{G}_\ell|)$ using radix sort. Line 1 takes time $\mathcal{O}(m^\epsilon + |\mathcal{G}_\ell|)$ due to Lemma 11, and line 2 requires $\mathcal{O}(|I_{k_{min}}| + |\mathcal{G}_\ell|)$. All executions of line 7 take time $\mathcal{O}(|I_{k_{min}}|)$ because the words w_i are already sorted. For the time being assume that the binary search in line 13 is for free. Then the total complexity becomes $\mathcal{O}(\sum_i m^\epsilon + \left|I_{k_{min}}^{(i)}\right| + |\mathcal{G}_\ell|)$ where $\left|I_{k_{min}}^{(i)}\right|$ is the size of $I_{k_{min}}$ just before the i-th call to BATCHED-POWERS-MERGE. There is a constant number of those calls for each value of $1 \leq \ell \leq m$, and each k_{min} corresponds to at most constant number of different continuous values of ℓ, thus the sum is in fact $\mathcal{O}(m + \sum_\ell |\mathcal{G}_\ell|)$.

To finish the proof we have to bound the time taken by all binary searches. For that to happen we will view the intervals as vertices of a tree. Whenever we split an interval into two, we add a left and right child to the corresponding leaf v. The *rank* $\mathrm{rank}(v)$ of a vertex v is the rounded logarithm of its *weight*, which is the length of the corresponding interval. Then the cost of line 13 is simply $\mathcal{O}(1)$ plus $\min(\mathrm{rank}(\mathrm{left}(v)), \mathrm{rank}(\mathrm{right}(v)))$ where $\mathrm{left}(v)$ and $\mathrm{right}(v)$ are the left and right child of v, respectively. Hence we should bound the sum $\sum_v \min(\mathrm{left}(v), \mathrm{right}(v))$, where v is a non-leaf. We say that a vertex is *charged* when its weight does not exceed the weight of its brother. Now we claim that there are at most $\frac{m}{2^k}$ charged vertices of rank k: assume that there are u and v such that u is an ancestor of v, both are charged and of rank k, then weight of v

plus weight of its brother is at least twice as large as the weight of v alone, thus the rank of their parent is larger than the rank of v, contradiction. So all charged vertices of the same rank correspond to disjoint intervals, and there cannot be more than $\frac{m}{2^k}$ disjoint intervals of length at least 2^k on a segment of length m. Now we can bound the sum which gives the claim:

$$\sum_v \min(\operatorname{rank}(\operatorname{left}(v)), \operatorname{rank}(\operatorname{right}(v))) \leq \sum_{k \geq 0}^{\log m} k \frac{m}{2^k} \leq m \sum_{k \geq 0}^{\infty} \frac{k}{2^k} = 2m \qquad \square$$

Algorithm 2 BATCHED-POWERS-MERGE$(w_1, w_2, \ldots, w_{|\mathcal{G}_\ell|})$

1: sort all w_i ▷ **Lemma 11**
2: scan $I_{k_{min}}$ to find the intervals containing w_i
3: $L \leftarrow \emptyset$
4: $r_0 \leftarrow 1$
5: **for** $i \leftarrow 1$ to $|\mathcal{G}_\ell|$ **do**
6:     $[a, b] \leftarrow$ the interval corresponding to $w_i[1 \mathinner{.\,.} 2^{k_{min}}]$ in SA ▷ **Lemma 12**
7:     choose $[c, d] \in I_{k_{min}}$ containing the first occurrence of w_i in SA
8:     **if** $[c, d]$ is defined **then**
9:         $a \leftarrow \max(a, c)$
10:         $b \leftarrow \min(b, d)$
11:     **end if**
12:     $a \leftarrow \max(r_{i-1}, a)$
13:     $r_i \leftarrow$ TWO-WAY-BINARY-SEARCH(a, b, w_i)
14:     add $[a, r_i]$ and $[r_i, b]$ to L
15: **end for**
16: sort L and merge it with $I_{k_{min}}$, removing non-minimal intervals
17: **return** all answers r_i

If for a production $X \to YZ$ we cannot find the cover of X, we compute $\operatorname{prefix}(X)$ and $\operatorname{suffix}(X)$ given the covers of Y and Z.

Lemma 14. *Given the covers of Y and Z, we can compute $\operatorname{prefix}(X)$ and $\operatorname{suffix}(X)$ in constant time as long as $\frac{|Y|}{|Z|}$ and $\frac{|Z|}{|Y|}$ are bounded from above by a constant. To compute $\operatorname{prefix}(X)$ we can use $\operatorname{prefix}(Z)$ instead of the cover of Z, and $\operatorname{suffix}(X)$ can be replaced with $\operatorname{suffix}(Y)$ instead of the cover of Y.*

Proof. Use Lemma 7 with carefully chosen arguments, see the appendix. $\square$

Theorem 2. *Given a 0.25-balanced SLP of size $\mathcal{O}(n \log \frac{N}{n})$ and a pattern $s[1 \mathinner{.\,.} m]$, we can detect an occurrence of s in the represented text in time $\mathcal{O}(n \log \frac{N}{n} + m)$.*

Proof. By Lemma 10 and Lemma 13 we compute the covers of all nonterminals which represent subwords of s in time $\mathcal{O}(n \log \frac{N}{n} + m)$. For the remaining nonterminals X we use Lemma 14 to compute $\operatorname{prefix}(X)$ and $\operatorname{suffix}(X)$ in total linear time considering the nonterminals in bottom-up order. Then due to Lemma 9 if there is an occurrence of s, there is an occurrence in $\operatorname{prefix}(Y) \operatorname{suffix}(Z)$ for some production $X \to YZ$. We consider every nonterminal X, either lookup the already computed $\operatorname{prefix}(Y)$ and $\operatorname{suffix}(Z)$ or compute them using the known covers and Lemma 14, and use Lemma 6 to detect a possible occurrence. $\square$

By using Lemma 8 and Theorem 2 we get the final result.

Theorem 3. *Given a (potentially self-referential) Lempel-Ziv parse of size n describing a text $t[1 \mathinner{.\,.} N]$ and a pattern $s[1 \mathinner{.\,.} m]$, we can detect an occurrence of s inside t deterministically in time $\mathcal{O}(n \log \frac{N}{n} + m)$.*

References

1. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: pattern matching in z-compressed files. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 705–714, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.
2. M. A. Bender and M. Farach-Colton. The lca problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, London, UK, 2000. Springer-Verlag.
3. M. A. Bender and M. Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004.
4. M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Rasala, A. Sahai, and a. shelat. Approximating the smallest grammar: Kolmogorov complexity in natural models. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 792–801, New York, NY, USA, 2002. ACM.
5. M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *CPM '96: Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 130–140, London, UK, 1996. Springer-Verlag.
6. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 703–712, New York, NY, USA, 1995. ACM.
7. P. Gawrychowski. Optimal pattern matching in LZW compressed strings. In *SODA '11: Proceedings of the twenty-second annual ACM-SIAM symposium on Discrete algorithms*, to appear.
8. J. Iacono and Ö. Özkan. Mergeable dictionaries. In *ICALP (1)*, pages 164–175, 2010.
9. J. Kärkkäinen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.
10. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
11. T. Kopelowitz and M. Lewenstein. Dynamic weighted ancestors. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 565–574, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.
12. S. R. Kosaraju. Pattern matching in compressed texts. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 349–362, London, UK, 1995. Springer-Verlag.
13. W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theor. Comput. Sci.*, 302(1-3):211–222, 2003.
14. E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

Lemma 3. *Pattern s can be preprocessed in linear time so that given any fragment $s[i\mathinner{\ldotp\ldotp}j]$ we can find its longest suffix (prefix) which is a prefix (suffix) of the whole pattern in constant time, assuming we know the (explicit or implicit) vertex corresponding to $s[i\mathinner{\ldotp\ldotp}j]$ in the suffix tree built for s (reversed s).*

Proof. We assume that the suffix tree is built for s concatenated with a special terminating character, say \$. Each leaf in the suffix tree corresponds to some suffix of s, and is connected to its parent with an edge labeled with a single letter. If we mark all those parents, finding the longest prefix which is a suffix of the whole s reduces to finding the lowest marked vertex on a given path leading the root, which can be precomputed for all vertices in linear time.   □

Lemma 6 (see Lemma 5 of [7]). *Given a prefix snippet and a suffix snippet we can detect an occurrence of the pattern in their concatenation in constant time.*

Proof. We need to answer the following question: does s occur in $s[1\mathinner{\ldotp\ldotp}i]s[j\mathinner{\ldotp\ldotp}m]$? Or, in other words, is there $x \in border(s[1\mathinner{\ldotp\ldotp}i])$ and $y \in border(s[j\mathinner{\ldotp\ldotp}m])$ such that $x + y = m$? Note that either $x \geq \frac{|s[1\mathinner{\ldotp\ldotp}i]|}{2}$ or $y \geq \frac{|s[j\mathinner{\ldotp\ldotp}m]|}{2}$, and without losing the generality assume the former. From Lemma 4 we know that all such possible values of x create one arithmetic progression. More specifically, $x = i - \alpha p$, where $p \leq \frac{i}{2}$ is the period of $s[1\mathinner{\ldotp\ldotp}i]$ extracted using Lemma 5. We need to check if there is an occurrence of s in $s[1\mathinner{\ldotp\ldotp}i]s[j\mathinner{\ldotp\ldotp}m]$ starting after the αp-th character, for some $0 \leq \alpha \leq \frac{i}{p}$. For any such possible interesting shift, there will be no mismatch in $s[1\mathinner{\ldotp\ldotp}i]$. There might be a mismatch in $s[j\mathinner{\ldotp\ldotp}m]$, though.

Let $k \geq i$ be the longest prefix of s for which p is a period (such k can be calculated efficiently by looking up the longest common prefix of $s[p+1\mathinner{\ldotp\ldotp}m]$ and the whole s). We shift $s[1\mathinner{\ldotp\ldotp}k]$ by $\left\lfloor \frac{\min(i,i-j+1)}{p} \right\rfloor p$ characters. Note this is the maximal shift of the form αp which, after extending $s[1\mathinner{\ldotp\ldotp}k]$ to the whole s, does not result in sticking out of the right end of $s[j\mathinner{\ldotp\ldotp}m]$. Then compute the leftmost mismatch of the shifted $s[1\mathinner{\ldotp\ldotp}k]$ with $s[j\mathinner{\ldotp\ldotp}m]$, see Figure 2. Position of the first mismatch, or its nonexistence, allows us to eliminate all but one interesting shift. More precisely, we have two cases to consider.

1. There is no mismatch. If $k = m$ we are done, otherwise $s[k+1] \neq s[k+1-p]$, meaning that choosing any smaller interesting shift results in a mismatch.
2. There is a mismatch. Let the conflicting characters be a and b and call the position at which a occurs in the concatenation the *obstacle*. Observe that we must choose a shift αp so that $s[1\mathinner{\ldotp\ldotp}k]$ shifted by αp is completely on the left of the obstacle. On the other hand, if $s[1\mathinner{\ldotp\ldotp}k]$ shifted by $(\alpha+1)p$ is completely on the left as well, shifting $s[1\mathinner{\ldotp\ldotp}k]$ by αp results in a mismatch because $s[k+1] \neq s[k+1-p]$ and $s[k+1-p]$ matches with the corresponding character in $s[j\mathinner{\ldotp\ldotp}m]$. Thus we may restrict our attention to the largest shift for which $s[1\mathinner{\ldotp\ldotp}k]$ is on the left of the obstacle.

Having identified the only interesting shift, we verify if there is a match using one longest common prefix query on s. More precisely, if the shit is αp, we check

if the common prefix of $s[i - \alpha p \mathinner{.\,.} m]$ and $s[j \mathinner{.\,.} m]$ is of length $|s[i - \alpha p \mathinner{.\,.} m]|$. Overall, the whole procedure takes constant time. $\square$
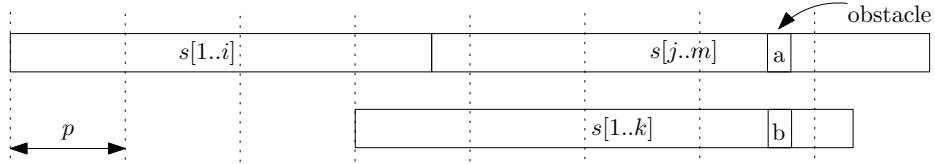

Fig. 2. Detecting an occurrence in a concatenation of two snippets.

Lemma 7. *Given a prefix snippet s_1 and a snippet s_2 for which we know the corresponding node in the suffix tree, we can compute the longest prefix of s which is a suffix of $s_1 s_2$ in time $\mathcal{O}\left(\max\left(1, \log \frac{|s_1|}{|s_2|}\right)\right)$.*

Proof. We try to find the longest border of $s_1 = s[1 \mathinner{.\,.} i]$ which can be extended with s_2. If there is none, we use Lemma 3 on s_2 to extract the answer. Of course s_1 might happen to have quite a lot of borders, and we do not have enough time to go through each of them separately. We try to abuse Lemma 4 instead: there are just $\log |s_1|$ groups of borders, and we are going to process each of them in constant time. It is not enough though, we need something faster when $|s_2|$ is relatively big compared to $|s_1|$. The whole method works as follows: as long as $|s_2|$ is smaller than $2|s_1|$, we check if it is possible to extend any of the long borders of s_1. If it is not possible, we replace s_1 with the longest prefix of s which ends $s_1[\frac{|s_1|}{2} \mathinner{.\,.} |s_1|]$ (we can preprocess such information for all prefixes of s in linear time). When $|s_2|$ exceeds $2|s_1|$, we look for an occurrence of s_2 in a prefix of s of length $|s_1| + |s_2|$. All such occurrences create one arithmetic progression due to Lemma 4, and it is possible to detect which one is preceded by a suffix of s_1 in constant time. More specifically, we show how to implement in constant time the following two primitives. In both cases the method resembles the one from Lemma 6.

1. Computing the longest long border of s_1 which can be extended with s_2 to form a prefix of s, if any. First we compute the period p of s_1 in constant time due to Lemma 3, then $p \leq \frac{|s_1|}{2}$ and any long border begins after the αp-th letter, for some $\alpha \geq 0$. We compute how far the period extends in both s and s_2, this gives us a simple arithmetic condition on the smallest value of α. More explicitly, there is either at most one valid α, or all are correct.
2. Detecting the rightmost occurrence of s_2 in s preceded by a suffix of s_1, assuming $|s_2| \geq 2|s_1|$. We begin with finding the first and the second occurrence of s_2 in s. Assuming we have the corresponding vertex in the suffix tree available, this takes just constant time. We check those (at most) two occurrences naively. There might be many more of them, though. But if the two

first occurrences begin before the $|s_1|$-th character, we know that all other interesting occurrences form one arithmetic progression with the known period of s_2. We check how far the period extends in s_1 (starting from the right end) and s (starting from the first occurrence of s_2), this again gives us a simple arithmetic condition on the best possible shift.

$\square$

Lemma 8 (see Theorem 1 of [4]). *Given a (potentially self-referential) LZ parse of size n, we can build a α-balanced SLP of size $\mathcal{O}(n \log \frac{N}{n})$ describing the same string of length N, for any constant $0 < \alpha \leq 1 - \frac{\sqrt{2}}{2}$. Running time of the construction is proportional to the size of the output.*

Proof. At a very high level, the idea of [4] is to process the parse from left-to-right. When processing a triple $(start_i, len_i, next_i)$, we already have an α-balanced SLP describing the prefix of the whole text corresponding to the previously encountered triples. Because the grammar is balanced, we can define $t[start_i \,.\, start_i + len_i - 1]$ by introducing a relatively small number of new nonterminals (with small actually meaning small in the amortized sense). Now if we allow the parse to be self-referential, it might happen that $t[start_i \,.\, start_i + len_i - 1]$ sticks out from the right end of $t[1 \,.\, \sum_{j=1}^{i-1} len_j]$. In such case we do as follows: let $L = \sum_{j=1}^{i-1} len_j$, and split the fragment corresponding to the current triple into three parts. First we have $t[start_i \,.\, L]$, then some repetitions of the same fragment, and then $t[start_i \,.\, len_i \bmod (L - start_i + 1)]$ followed by a single letter $next_i$. After defining a nonterminal deriving $t[start_i \,.\, L]$, we can define a nonterminal deriving the repetitions at the expense of introducing at most $2 \log len_i$ new nonterminals. Then we define a nonterminal deriving $t[start_i \,.\, len_i \bmod (L - start_i + 1)]next_i$. The only change in the analysis of this method is that we might end up adding $\sum_{i=1}^{n} \log len_i$ new nonterminals, which by the concavity of log is at most $\mathcal{O}(n \log \frac{N}{n})$, and thus does not change the asymptotic upper bound.

Note that the algorithm in [4] contains one special case: if the compression ratio is at most $2e$, the trivial grammar is returned. We do the same. $\square$

Lemma 9. *If s occurs in a string represented by a SLP then there exists a production $X \to YZ$ such that s occurs in $\mathrm{suffix}(Y)\,\mathrm{prefix}(Z)$.*

Proof. Consider the leftmost occurrence of s. Take the starting symbol $X = S$ and its production $X \to YZ$. If the leftmost occurrence is completely inside Y or Z, repeat with X replaced with Y or Z. Otherwise the occurrence crosses the boundary between Y and Z, in other words there is a prefix snippet $s[1 \,.\, i]$ ending Y and a suffix snippet $s[i+1 \,.\, m]$ starting Z. Then $|\mathrm{suffix}(Y)| \geq i$ and $|\mathrm{prefix}(Z)| \geq m - i$, and s occurs in $\mathrm{suffix}(Y)\,\mathrm{prefix}(Z)$. $\square$

Lemma 12. *The pattern s can be processed in linear time so that given any $s[i \,.\, i + 2^k - 1]$ we can compute its first and the last occurrence in the suffix array of s in constant time.*

Proof. First we use the standard micro-macro tree decomposition, which gives us a top fragment containing just $\frac{n}{\log n}$ leaves, and a collection of small trees on at most $\log n$ leaves. Note that in this particular case, the total number of vertices cannot be much larger than the number of leaves: the original tree contained vertices with outdegree 1, then we introduced at most one such vertex at each edge, and then we collapsed some parts of the tree. For each node in the top tree we store all $\log n$ answers explicitly. For each small tree we do as follows: first number its nodes in a depth-first order, then for each node compute a single bitvector containing the numbers of all its ancestors. To find the k-th ancestor of a given vertex v, we consider two cases.

1. v belongs to the top tree. Then we have the answer available.
2. v belongs to some small tree. We first check in constant time if its depth in this small tree does not exceed k. If it does, we can use the precomputed answers stored for the parent (in the top tree) of the root. Otherwise we take a look at the bitvector corresponding to v, and find its k-th highest bit set to 1. Then we retrieve the node corresponding to this depth-first number.   □
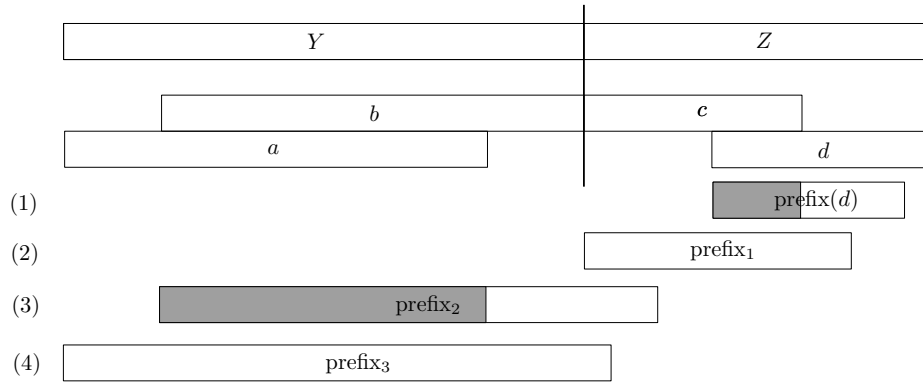

Fig. 3. Computing $\text{prefix}(X)$ given the covers of Y and Z.

Theorem 1. *Substring fingerprints of size $\mathcal{O}(|s|^3)$ can be computed in constant time after a linear time preprocessing.*

Proof. First we apply the preprocessing from Lemma 12 to s. We also store $\lfloor \log x \rfloor$ for any $1 \leq x \leq |s|$. Then given a query $s[i \mathbin{..} j]$ we compute $k = \lfloor \log(j - i + 1) \rfloor$ and using constant time level ancestors queries we locate the lowest existing ancestors of both $s[i \mathbin{..} i + 2^k - 1]$ and $s[j - 2^k + 1 \mathbin{..} j]$ in the suffix tree. Then $h_s(s[i \mathbin{..} j])$ is a triple containing $j - i + 1$ and those two ancestors.   □

Lemma 14. *Given the covers of Y and Z, we can compute $\text{prefix}(X)$ and $\text{suffix}(X)$ in constant time as long as $\frac{|Y|}{|Z|}$ and $\frac{|Z|}{|Y|}$ are bounded from above by a constant. To compute $\text{prefix}(X)$ we can use $\text{prefix}(Z)$ instead of the cover of Z, and $\text{suffix}(X)$ can be replaced with $\text{suffix}(Y)$ instead of the cover of Y.*

Proof. It is enough to consider $\text{prefix}(X)$. The idea is to use a few application of Lemma 7 with carefully chosen arguments, see Figure 3. More specifically, let a, b and c, d be the covers of Y and Z, respectively. First we locate the vertex corresponding to d in the suffix tree, due to Lemma 12 and $|d| = 2^k$ it takes constant time, then:

(1) apply Lemma 3 to compute $\text{prefix}(d)$ if we have the cover of Z, otherwise take the known $\text{prefix}(Z)$ and go to (3),
(2) apply Lemma 7 to c and $\text{prefix}(d)$ without the first $|c| + |d| - |Z|$ letters to get prefix_1,
(3) apply Lemma 7 to b and prefix_1 to get prefix_2,
(4) apply Lemma 7 to a and prefix_2 without the first $|a| + |b| - |Y|$ letters to get the desired answer prefix_3.

Note that whenever we apply the lemma to two words u and v, $|v|$ is a power of 2 and so we can use Lemma 12 to locate its corresponding node in constant time. Also, it holds that $|u| \geq \frac{\min(|Y|,|Z|)}{2}$ and $|v| \leq |Y| + |Z|$ and so the running time is bounded by:

$$\max\left(1, \log\frac{|v|}{|u|}\right) \leq \max\left(1, \log\left(\frac{|Y| + |Z|}{\min(|Y|, |Z|)}\right)\right) = \log\left(1 + \frac{\max(|Y|, |Z|)}{\min(|Y|, |Z|)}\right)$$

which is $\mathcal{O}(1)$. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$