Efficient LZ78 Factorization of Grammar Compressed Text

Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda

Department of Informatics, Kyushu University, Japan
{bannai,inenaga,takeda}@inf.kyushu-u.ac.jp

Abstract. We present an efficient algorithm for computing the LZ78 factorization of a text, where the text is represented as a straight line program (SLP), which is a context free grammar in the Chomsky normal form that generates a single string. Given an SLP of size n representing a text S of length N, our algorithm computes the LZ78 factorization of T in $O(n\sqrt{N} + m \log N)$ time and $O(n\sqrt{N} + m)$ space, where m is the number of resulting LZ78 factors. We also show how to improve the algorithm so that the $n\sqrt{N}$ term in the time and space complexities becomes either nL, where L is the length of the longest LZ78 factor, or $(N - \alpha)$ where $\alpha \geq 0$ is a quantity which depends on the amount of redundancy that the SLP captures with respect to substrings of S of a certain length. Since $m = O(N/\log_\sigma N)$ where σ is the alphabet size, the latter is asymptotically at least as fast as a linear time algorithm which runs on the uncompressed string when σ is constant, and can be more efficient when the text is compressible, i.e. when m and n are small.

1 Introduction

Large scale textual data are usually stored in compressed form, while it is later decompressed to be used. In order to circumvent the computational resources required to handle and process the cumbersome uncompressed string, the compressed string processing (CSP) approach has been gaining attention. The aim of CSP is to process text given in compressed form without explicitly decompressing the entire text, therefore allowing space efficient, as well as time efficient processing of the text when it is sufficiently compressed.

Many CSP algorithms work on a representation of the compressed text called *straight line programs* (SLPs). An SLP is a context free grammar in the Chomsky normal form that derives a single string. SLPs can efficiently model the outputs of many different types of compression algorithms (e.g.: grammar based [22,17], dictionary based [28,29]), and hence, an algorithm that works on an SLP can be applied to texts compressed by various compression algorithms. On the other hand, there are many CSP algorithms which make use of specific properties that are implicit in the compressed representation $C(S)$ of text S obtained by using a certain compression algorithm C [4,9,10,11]. Such CSP algorithms cannot be applied to representations produced by any arbitrary compression algorithm. To overcome this problem, we consider the problem of computing the compressed representation $C(S)$ from an arbitrary SLP representing S, without completely decompressing the SLP.

In this paper, we focus on the well known LZ78 compression algorithm [29]. LZ78 compresses a given text based on a dynamic dictionary which is constructed by partitioning the input string, the process of which is called LZ78 factorization. Other than its obvious use for compression, the LZ78 factorization is an important concept used in various string processing algorithms and applications [7,19,18,20]. The contribution of this paper is an $O(n\sqrt{N} + m \log N)$ time and $O(n\sqrt{N} + m)$ space algorithm to compute the LZ78 factorization of a string given as an SLP, where N is the length of the string, n is the size of the SLP, and m is the number of LZ78 factors.

We further show how to improve the $n\sqrt{N}$ term in the time and space complexities in two ways. An application of doubling search enables the term to be reduced to nL, where L is the longest LZ78 factor. Also, by applying the recent techniques of [13], the term can be reduced to $N - \alpha$, where $\alpha \geq 0$ is a quantity which depends on the amount of redundancy that the SLP captures with respect to substrings of S of a certain length. Since it is known that $m = O(N/\log_\sigma N)$ [29], where σ is the alphabet size, our approach is guaranteed to be asymptotically at least as fast as a linear time algorithm which runs on the uncompressed string if σ is considered constant, and can be even more efficient when the text is compressible, i.e. when m and n are small.

As a byproduct of the above results, we also obtain an efficient algorithm which converts a given *LZ77 factorization* of a string [28] to the corresponding LZ78 factorization without explicit decompression. We conclude the paper by mentioning several other interesting potential applications of our algorithm.

Related Work

An efficient algorithm for computing the LZ78 factorization was presented in [14]. Their algorithm requires only $O(N(\log \sigma + \log \log_\sigma N)/\log_\sigma N)$ bits of working space and runs in $O(N(\log\log N)^2/(\log_\sigma N \log\log\log N))$ worst-case time which is sub-linear when $\sigma = 2^{o(\log N \frac{\log\log\log N}{(\log\log N)^2})}$. However, their input assumes the uncompressed text and it is unknown how to apply their algorithm without completely decompressing the SLP.

2 Preliminaries

2.1 Strings

Let Σ be a finite *alphabet* and $\sigma = |\Sigma|$. An element of Σ^* is called a *string*. The length of a string S is denoted by $|S|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $S = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of S, respectively. The set of all substrings of a string S is denoted by *Substr*(S). The i-th character of a string S is denoted by $S[i]$ for $1 \leq i \leq |S|$, and the substring of a string S that begins at position i and ends at position j is denoted by $S[i : j]$ for $1 \leq i \leq j \leq |S|$. For convenience, let $S[i : j] = \varepsilon$ if $j < i$. For a string S and integer $q \geq 0$, let $\mathit{pre}(S, q)$ and $\mathit{suf}(S, q)$ represent respectively, the length-q prefix and suffix of T, that is, $\mathit{pre}(S, q) = S[1 : \min\{q, |S|\}]$ and $\mathit{suf}(S, q) = S[\max\{1, |S| - q + 1\} : |S|]$. We also assume that the last character of the string is a special character '\$' that does not occur anywhere else in the string.

Our model of computation is the word RAM: We shall assume that the computer word size is at least $\log |S|$, and hence, standard operations on values representing lengths and positions of string S can be manipulated in constant time. Space complexities will be determined by the number of computer words (not bits).

2.2 Straight Line Programs

A *straight line program* (*SLP*) is a set of assignments $\mathcal{T} = \{X_1 \to \mathit{expr}_1, X_2 \to \mathit{expr}_2, \ldots, X_n \to \mathit{expr}_n\}$, where each X_i is a distinct non-terminal variable and each expr_i is an expression that can be either $\mathit{expr}_i = a$ ($a \in \Sigma$), or $\mathit{expr}_i = X_{\ell(i)}X_{r(i)}$ ($i > \ell(i), r(i)$). An SLP is essentially a context free grammar in the Chomsky normal form, that derives a single string. Let $\mathit{val}(X_i)$ represent the string derived from variable X_i. To ease notation, we sometimes associate $\mathit{val}(X_i)$ with X_i and denote $|\mathit{val}(X_i)|$ as $|X_i|$. An SLP $\mathcal{T}$ *represents* the string $T = \mathit{val}(X_n)$. The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$.

The derivation tree of SLP $\mathcal{T}$ is a labeled ordered binary tree where each internal node is labeled with a non-terminal variable in $\{X_1, \ldots, X_n\}$, and each leaf is labeled with a terminal character in Σ. The root node has label X_n. Let $\mathcal{V}$ denote the set of internal nodes in the derivation tree. For any internal node $v \in \mathcal{V}$, let $\langle v \rangle$ denote the index of its label $X_{\langle v \rangle}$. Node v has a single child which is a leaf labeled with c when $(X_{\langle v \rangle} \to c) \in \mathcal{T}$ for some $c \in \Sigma$, or v has a left-child and right-child respectively denoted $\ell(v)$ and $r(v)$, when $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$. Each node v of the tree derives $\mathit{val}(X_{\langle v \rangle})$, a substring of T, whose corresponding interval $\mathit{itv}(v) = [b : e]$, with $T[b : e] = \mathit{val}(X_{\langle v \rangle})$, can be defined recursively as follows. If v is the root node, then $\mathit{itv}(v) = [1 : |T|]$. Otherwise, if $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$, then, $\mathit{itv}(\ell(v)) = [b_v : b_v + |X_{\langle \ell(v) \rangle}| - 1]$ and $\mathit{itv}(r(v)) = [b_v + |X_{\langle \ell(v) \rangle}| : e_v]$, where $[b_v : e_v] = \mathit{itv}(v)$. Let $\mathit{vOcc}(X_i)$ denote the number of times a variable X_i occurs in the derivation tree, i.e., $\mathit{vOcc}(X_i) = |\{v \mid X_{\langle v \rangle} = X_i\}|$.

For any interval $[b : e]$ of $T (1 \leq b < e \leq |T|)$, let $\xi_{\mathcal{T}}(b, e)$ denote the deepest node v in the derivation tree, which derives an interval containing $[b : e]$, that is, $\mathit{itv}(v) \supseteq [b : e]$, and no proper descendant of v satisfies this condition. We say that node v *stabs* interval $[b : e]$, and $X_{\langle v \rangle}$ is called the variable that stabs the interval. We have $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle}X_{\langle r(v) \rangle}) \in \mathcal{T}$, $b \in \mathit{itv}(\ell(v))$, and $e \in \mathit{itv}(r(v))$. When it is not confusing, we will sometimes use $\xi_{\mathcal{T}}(b, e)$ to denote the variable $X_{\langle \xi_{\mathcal{T}}(b,e) \rangle}$.

SLPs can be efficiently pre-processed to hold various information. $|X_i|$ and $\mathit{vOcc}(X_i)$ can be computed for all variables X_i ($1 \leq i \leq n$) in a total of $O(n)$ time by a simple dynamic programming algorithm.

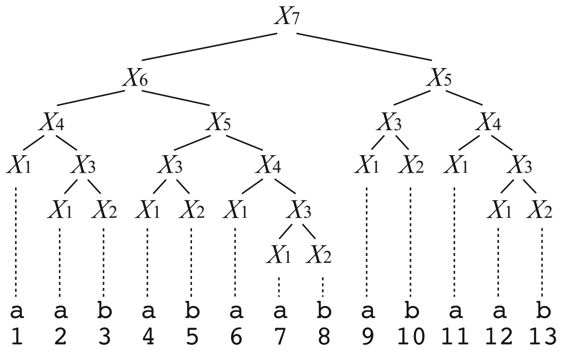

Fig. 1. The derivation tree of SLP $\mathcal{T} = \{X_1 \to \mathtt{a}, X_2 \to \mathtt{b}, X_3 \to X_1X_2, X_4 \to X_1X_3, X_5 \to X_3X_4, X_6 \to X_4X_5, X_7 \to X_6X_5\}$. $T = val(X_7) =$ aababaababaab.

2.3 LZ78 Encoding

Definition 1 (LZ78 factorization). *The LZ78-factorization of a string S is the factorization $f_1 \cdots f_m$ of S, where each LZ78-factor $f_i \in \Sigma^+$ $(1 \le i \le m)$ is the longest prefix of $f_i \cdots f_m$, such that $f_i \in \{f_jc \mid 1 \le j < i, c \in \Sigma\} \cup \Sigma$.*

For a given string S, let m denote the number of factors in its LZ78 factorization. The LZ78 factorization of the string can be encoded by a sequence of pairs, where the pair for factor f_i consists of the ID j of the previous factor f_j ($j = 0$ and $f_0 = \varepsilon$ when there is none) and the new character $S[|f_1 \cdots f_i|]$. Regarding this pair as a parent and edge label, the factors can also be represented as a trie. (See Fig. 2.)

By using this trie, the LZ78 factorization of a string of length N can be easily computed incrementally in $O(N \log \sigma)$ time and $O(m)$ space; Start from an empty tree with only the root. For $1 \le i \le m$, to calculate f_i, let v be the node of the trie reached by traversing the tree with $S[p : q]$, where $p = |f_0 \cdots f_{i-1}| + 1$, and $q \ge p$ is the smallest position after p such that v does not have an outgoing edge labeled with $S[q + 1]$. Naturally, v represents the longest previously used LZ78-factor that is a prefix of $S[p : |S|]$. Then, we can insert an edge labeled with $S[q+1]$ to a new node representing factor f_i, branching from v. The update for each factor f_i can be done in $O(|f_i| \log \sigma)$ time for the traversal and in $O(\log \sigma)$ time for the insertion, with a total of $O(N \log \sigma)$ time for all the factors. Since each node of the trie except the root corresponds to an LZ78 factor, the size of the trie is $O(m)$.

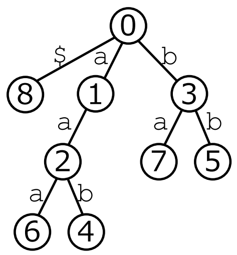

Fig. 2. The LZ78 dictionary for the string aaabaabbbaaaaaaaba\$. Each node numbered i represents the factor f_i of the LZ78 factorization, where f_i is the path label from the root to the node, e.g.: $f_2 =$ aa, $f_4 =$ aab.

Example 1. The LZ78 factorization of string aaabaabbbaaaaaaaba\$ is a, aa, b, aab, bb, aaa, aaaa, ba, \$, and can be represented as $(0, \mathtt{a})$, $(1, \mathtt{a})$, $(0, \mathtt{b})$, $(2, \mathtt{b})$, $(3, \mathtt{b})$, $(2, \mathtt{a})$, $(6, \mathtt{a})$, $(3, \mathtt{a})$, $(0, \$)$.

2.4 Suffix Trees

We give the definition of a very important and well known string index structure, the suffix tree. To assure property 3 for the sake of presentation, we assume that the string ends with a unique symbol that does not occur elsewhere in the string.

Definition 2 (Suffix Trees [26]). *For any string S, its suffix tree, denoted $ST(S)$, is a labeled rooted tree which satisfies the following:*

1. *each edge is labeled with an element in Σ^+;*
2. *there exist exactly n leaves, where $n = |S|$;*
3. *for each string $s \in \mathit{Suffix}(S)$, there is a unique path from the root to a leaf which spells out s;*
4. *each internal node has at least two children;*
5. *the labels x and y of any two distinct out-going edges from the same node begin with different symbols in Σ*

Since any substring of S is a prefix of some suffix of S, positions in the suffix tree of S correspond to a substring of S that is represented by the string spelled out on the path from the root to the position. We can also define a *generalized* suffix tree of a set of strings, which is simply the suffix tree that contains all suffixes of all the strings in the set.

It is well known that suffix trees can be represented and constructed in linear time [26,21,25], even independently of the alphabet size for integer alphabets [8]. Generalized suffix trees for a set of strings $\mathbf{S} = \{S_1, \ldots, S_k\}$, can be constructed in linear time in the total length of the strings, by simply constructing the suffix tree of the string $S_1\$_1 \cdots S_k\$_k$, and pruning the tree below the first occurrence of any $\$_i$, where $\$_i$ $(1 \leq i \leq k)$ are unique characters that do not occur elsewhere in strings of $\mathbf{S}$.

3 Algorithm

We describe our algorithm for computing the LZ78 factorization of a string given as an SLP in two steps. The basic structure of the algorithm follows the simple LZ78 factorization algorithm for uncompressed strings that uses a trie as mentioned in Section 2.3. Although the space complexity of the trie is only $O(m)$, we need some way to accelerate the traversal of the trie in order to achieve the desired time bounds.

3.1 Partial Decompression

We use the following property of LZ78 factors which is straightforward from its definition.

Lemma 1. *For any string S of length N and its LZ78-factorization $f_1 \cdots f_m$, $m \geq c_N$ and $|f_i| \leq c_N$ for all $1 \leq i \leq m$, where $c_N = \sqrt{2N + 1/4} - 1/2$.*

Proof. Since a factor can be at most 1 character longer than a previously used factor, $|f_i| \leq i$. Therefore, $N = \sum_{i=1}^{m} |f_i| \leq \sum_{i=1}^{m} i$, and thus $m \geq \sqrt{2N + 1/4} - 1/2$. For any factor of length $x = |f_{i_x}|$, there exist distinct factors $f_{i_1}, \ldots, f_{i_{x-1}}$ whose lengths are respectively $1, \ldots, x-1$. Therefore, $N = \sum_{i=1}^{m} |f_i| \geq \sum_{i=1}^{x} i$, and $x \leq \sqrt{2N + 1/4} - 1/2$. □

The lemma states that the length of an LZ78-factor is bounded by c_N. To utilize this property, we use ideas similar to those developed in [12,13] for counting the frequencies of all substrings of a certain length in a string represented by an SLP; For simplicity, assume $c_N \geq 2$. For each variable $X_i \to X_{\ell(i)} X_{r(i)}$, any length c_N substring that is stabbed by X_i is a substring of $t_i = \mathit{suf}(\mathit{val}(X_{\ell(i)}), c_N - 1)\mathit{pre}(\mathit{val}(X_{r(i)}), c_N - 1)$. On the other hand, all length c_N substrings are stabbed by some variable. This means that if we consider the set of strings consisting of t_i for all variables such that $|X_i| \geq c_N$, any length c_N substring of S is a substring of at least one of the strings. We can compute all such strings $T_S = \{t_i \mid |X_i| \geq c_N\}$ where $(X_i \to X_{\ell(i)} X_{r(i)}) \in \mathcal{T}$ in time linear in the total length, i.e. $O(nc_N)$ time by a straightforward dynamic programming [12].

All length c_N substrings of S occur as substrings of strings in T_S, and by Lemma 1, it follows that T_S contains all LZ78-factors of S as substrings.

3.2 Finding the Next Factor

In the previous subsection, we described how to partially decompress a given SLP of size n representing a string S of length N, to obtain a set of strings T_S with total length $O(n\sqrt{N})$, such that any LZ78-factor of S is a substring of at least one of the strings in T_S. We next describe how to identify these substrings.

We make the following key observation: since the LZ78-trie of a string S is a trie composed by substrings of S, it can be superimposed on a suffix tree of S, and be completely contained in it, with the exception that some nodes of the trie may correspond to implicit nodes of the suffix tree (in the middle of an edge of the suffix tree). Furthermore, this superimposition can also be done to the generalized suffix tree constructed for T_S. (See Fig. 3.)

Suppose we have computed the LZ78 factorization $f_1 \cdots f_{i-1}$, up to position $p - 1 = |f_1 \cdots f_{i-1}|$, and wish to calculate the next LZ78-factor starting at position p. Let $v = \xi_{\mathcal{T}}(p, p + c_N - 1)$, let $X_j = X_{\langle v \rangle}$ be the variable that stabs the interval $[p : p + c_N - 1]$, let q be the offset of p in t_j, and let w be the leaf of the generalized suffix tree that corresponds to the suffix $t_j[q : |t_j|]$. The longest previously used factor that is a prefix of $S[p : |S|]$ is the longest common prefix between $t_j[q : |t_j|]$ and all possible paths on the LZ78-trie built so far. If we

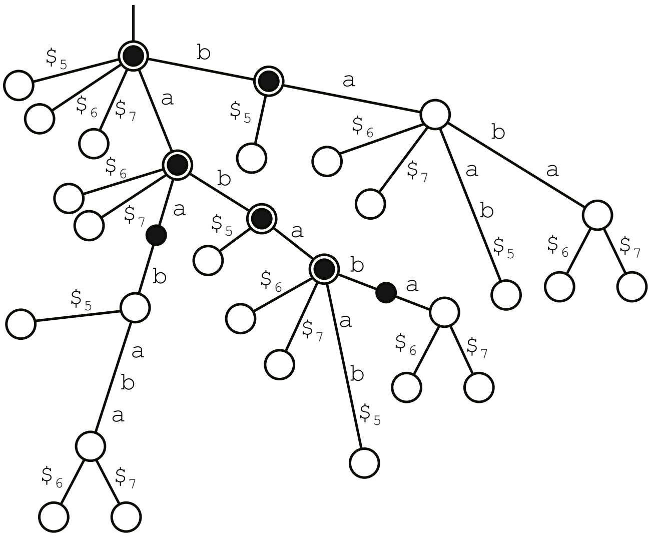

Fig. 3. The LZ78-trie of string $S = \texttt{aababaababaab}$, superimposed on the generalized suffix tree of $T_S = \{t_5, t_6, t_7\} = \{\texttt{abaab}\$_5, \texttt{aababa}\$_6, \texttt{aababa}\$_7\}$ for the SLP of Fig. 1. Here, $\$_5, \$_6, \$_7$ are end markers of each string in T_S, introduced so that each position in a string of T_s corresponds to a leaf of the suffix tree. The subtree consisting of the dark nodes is the LZ78-trie, derived from the LZ78-factorization: a, ab, aba, abab, aa, b, of S. Since any length $\lfloor c_N \rfloor = 4$ substring of S is a substring of at least one string in T_S, any LZ78-factor of S is a substring of some string of T_S, and the generalized suffix tree of T_S completely includes the LZ78-trie.

consider the suffix tree as a semi-dynamic tree, where nodes corresponding to the superimposed LZ78-trie are dynamically added and marked, the node x we seek is the *nearest marked ancestor* of w.

The generalized suffix tree for T_S can be computed in $O(n\sqrt{N})$ time. We next describe how to obtain the values v, q (and therefore w), and x as well as the computational complexities involved.

A naïve algorithm for obtaining v and q would be to traverse down the derivation tree of the SLP from the root, checking the decompressed lengths of the left and right child of each variable to determine which child to go down, in order to find the variables that correspond to positions p and $p + c_N - 1$. By doing the search in parallel, we can find v as the node at which the search for each position diverges, i.e., the lowest common ancestor of leaves in the derivation tree corresponding to positions p and $p + c_N - 1$. This traversal requires $O(h)$ time, where h is the height of the SLP, which can be as large as $O(n)$. To do this more efficiently, we can apply the algorithm of [5], which allows random access to arbitrary positions of the SLP in $O(\log N)$ time, with $O(n)$ time and space of preprocessing.

Theorem 1 ([5]). *For an SLP of size n representing a string of length N, random access can be supported in time $O(\log N)$ after $O(n)$ preprocessing time and space in the RAM model.*

Their algorithm basically constructs data structures in order to simulate the traversal of the SLP from the root, but reduces the time complexity from $O(h)$ to $O(\log N)$. Therefore, by running two random access operations for positions p and $p + c_N - 1$ in parallel until they first diverge, we can obtain v in $O(\log N)$ time. We note that this technique is the same as the first part of their algorithm for decompressing a substring $S[i : j]$ of length $m = j - i + 1$ in $O(m + \log N)$ time. The offset of p from the beginning of $X_{\langle v \rangle}$ can be obtained as a byproduct of the search for position p, and therefore, q can also be computed in $O(\log N)$ time.

For obtaining x, we use a data structure that maintains a rooted dynamic tree with marked/unmarked nodes such that the nearest marked ancestor in the path from a given node to the root can be found very efficiently. The following result allows us to find x – the nearest marked ancestor of w – in amortized constant time.

Lemma 2 ([27,1]). *A semi-dynamic rooted tree can be maintained in linear space so that the following operations are supported in amortized $O(1)$ time: 1) find the nearest marked ancestor of any node; 2) insert an unmarked node; 3) mark an unmarked node.*

For inserting the new node for the new LZ78-factor, we simply move down the edge of the suffix tree if x was an implicit node and has only one child. When x is branching, we can move down the correct suffix tree using level ancestor queries of the leaf w, therefore not requiring an $O(\log \sigma)$ factor.

Lemma 3 (Level ancestor query [3,2]). *Given a static rooted tree, we can preprocess the tree in linear time and space so that the ℓth node in the path from any node to the root can be found in $O(1)$ time for any integer $\ell \geq 0$, if such exists.*

Technically, our suffix tree is semi-dynamic in that new nodes are created since the LZ78-trie is superimposed. However, since we are only interested in level ancestor queries at branching nodes, we only need to answer them for the original suffix tree. Therefore, we can preprocess the tree in $O(n\sqrt{N})$ time and space to answer the level ancestor queries in $O(1)$ time.

The main result of this section follows:

Theorem 2. *Given an SLP of size n representing a string S of length N, we can compute the LZ78 factorization of S in $O(n\sqrt{N} + m \log N)$ time and $O(n\sqrt{N} + m)$ space, where m is the size of the LZ78 factorization.*

A better bound can be obtained by employing a simple doubling search on the length of partial decompressions.

Corollary 1. *Given an SLP of size n representing a string S of length N, we can compute the LZ78 factorization of S in $O(nL + m \log N)$ time and $O(nL + m)$*

space, where m is the size of the LZ78 factorization, and L is the length of the longest LZ78 factor.

Proof. Instead of using c_N for the length of partial decompressions, we start from length 2. For some length 2^{i-1}, if the LZ78 trie *outgrows* the suffix tree and reaches a leaf, we rebuild the suffix tree and the embedded LZ78 trie for length 2^i and continue with the factorization. This takes $O(n2^i)$ time, and the total asymptotic complexity becomes $n(2+\cdots+2^{\lceil \log_2 L \rceil}) = O(nL)$. Notice that the $m \log N$ term does not increase, since the factorization itself is not restarted, and also since the data structure of [5] is reused and only constructed once. □

3.3 Reducing Partial Decompression

By using the same techniques of [13], we can reduce the partial decompression conducted on the SLP, and reduce the complexities of our algorithm. Let $I = \{i \mid |X_i| \geq c_N\} \subseteq [1:n]$. The technique exploits the overlapping portions of each of the strings in T_S. The algorithm of [13] shows how to construct, in time linear of its size, a trie of size $(c_N - 1) + \sum_{i \in I}(|t_i| - (c_N - 1)) = N - \alpha = N_\alpha$ such that there is a one to one correspondence between a length c_N path on the trie and a length c_N substring of a string in T_S. Here,

$$\alpha = \sum_{i \in I}((vOcc(X_i) - 1) \cdot (|t_i| - (c_N - 1))) \geq 0 \tag{1}$$

can be seen as a quantity which depends on the amount of redundancy that the SLP captures with respect to length c_N substrings.

Furthermore, a suffix tree of a trie can be constructed in linear time:

Lemma 4 ([24]). *Given a trie, the suffix tree for the trie can be constructed in linear time and space.*

The generalized suffix tree for T_S used in our algorithm can be replaced with the suffix tree of the trie, and we can reduce the $O(n\sqrt{N})$ term in the complexity to $O(N_\alpha)$, thus obtaining an $O(N_\alpha + m \log N)$ time and $O(N_\alpha + m)$ space algorithm. Since N_α is also bounded by $O(n\sqrt{N})$, we obtain the following result:

Theorem 3. *Given an SLP of size n representing a string S of length N, we can compute the LZ78 factorization of S in $O(N_\alpha + m \log N)$ time and $O(N_\alpha + m)$ space, where m is the size of the LZ78 factorization, $N_\alpha = O(\min\{N - \alpha, n\sqrt{N}\})$, and $\alpha \geq 0$ is defined as in Equation (1).*

Since $m = O(N / \log_\sigma N)$ [29], our algorithms are asymptotically at least as fast as a linear time algorithm which runs on the uncompressed string when the alphabet size is constant. On the other hand, N_α can be much smaller than $O(n\sqrt{N})$ when $vOcc(X_i) > 1$ for many of the variables. Thus our algorithms can be faster when the text is compressible, i.e., n and m are small.

3.4 Conversion from LZ77 Factorization to LZ78 Factorization

As a byproduct of the algorithm proposed above, we obtain an efficient algorithm that converts a given *LZ77 factorization* [28] of a string to the corresponding LZ78 factorization, without explicit decompression.

Definition 3 (LZ77 factorization). *The LZ77-factorization of a string S is the factorization $f_1, \ldots, f_r$ of S such that for every $i = 1, \ldots, r$, factor f_i is the longest prefix of $f_i \cdots f_r$ with $f_i \in F_i$, where $F_i = \mathit{Substr}(f_1 \cdots f_{i-1}) \cup \Sigma$.*

It is known that the LZ77-factorization of string S can be efficiently transformed into an SLP representing S.

Theorem 4 ([23]). *Given the LZ77 factorization of size r for a string S of length N, we can compute in $O(r \log N)$ time an SLP representing S, of size $O(r \log N)$ and of height $O(\log N)$.*

The following theorem is immediate from Corollary 1 and Theorem 4.

Theorem 5. *Given the LZ77 factorization of size r for a string S of length N, we can compute the LZ78 factorization for S in $O(rL \log N + m \log N)$ time and $O(rL \log N + m)$ space, where m is the size of the LZ78 factorization for S, and L is the length of the longest LZ78 factor.*

It is also possible to improve the complexities of the above theorem using Theorem 3, so that the conversion from LZ77 to LZ78 can be conducted in $O(N_\alpha + m \log N)$ time and $O(N_\alpha + m)$ space, where N_α here is defined for the SLP generated from the input LZ77 factorization. This is significant since the resulting algorithm is at least as efficient as a naïve approach which requires decompression of the input LZ77 factorization, and can be faster when the string is compressible.

4 Discussion

We showed an efficient algorithm for calculating the LZ78 factorization of a string S, from an arbitrary SLP of size n which represents S. The algorithm is guaranteed to be asymptotically at least as fast as a linear time algorithm that runs on the uncompressed text, and can be much faster when n and m are small, i.e., the text is compressible.

It is easy to construct an SLP of size $O(m)$ that represents string S, given its LZ78 factorization whose size is m [16]. Thus, although it was not our primary focus in this paper, the algorithms we have developed can be regarded as a *re-compression* by LZ78, of strings represented as SLPs. The concept of re-compression was recently used to speed up fully compressed pattern matching [15]. We mention two other interesting potential applications of re-compression, for which our algorithm provides solutions:

Maintaining Dynamic SLP Compressed Texts

Modification to the SLP corresponding to edit operations on the string that it represents, e.g.: character substitutions, insertions, deletions can be conducted in $O(h)$ time, where h is the height of the SLP. However, these modifications are *ad-hoc*, and there are no guarantees as to how compressed the resulting SLP is, and repeated edit operations will inevitably cause degradation on the compression ratio. By periodically re-compressing the SLP, we can maintain the compressed size (w.r.t. LZ78) of the representation, without having to explicitly decompress the entire string during the maintenance process.

Computing the NCD w.r.t. LZ78 without Explicit Decompression

The Normalized Compression Distance (NCD) [6] measures the distance between two data strings, based on a specific compression algorithm. It has been shown to be effective for various clustering and classification tasks, while not requiring in-depth prior knowledge of the data. NCD between two strings S and T w.r.t. compression algorithm A is determined by the values $C_A(ST)$, $C_A(S)$, and $C_A(T)$, which respectively denote the sizes of the compressed representation of strings ST, S, and T when compressed by algorithm A.

When S and T are represented as SLPs, we can compute $C_{\mathrm{LZ78}}(S)$ and $C_{\mathrm{LZ78}}(T)$ without explicitly decompressing all of S and T, using the algorithms in this paper. Furthermore, the SLP for the concatenation ST can be obtained by simply considering a new single variable and production rule $X_{ST} \to X_S X_T$, where X_S and X_T are respectively the roots of the SLP which derive S and T. Thus, by applying our algorithm on this SLP, we can compute $C_{\mathrm{LZ78}}(ST)$ without explicit decompression as well. Therefore it is possible to compute NCD w.r.t. LZ78 between strings represented as SLPs, and therefore even cluster or classify them, without explicit decompression.

Acknowledgements. We thank the anonymous reviewers for helpful comments to improve the paper.

References

1. Amir, A., Farach, M., Idury, R.M., Poutré, J.A.L., Schäffer, A.A.: Improved dynamic dictionary matching. Information and Computation 119(2), 258–282 (1995)
2. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)
3. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. System Sci. 48(2), 214–230 (1994)
4. Bille, P., Fagerberg, R., Gørtz, I.L.: Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts. ACM Transactions on Algorithms 6(1) (2009)
5. Bille, P., Landau, G.M., Raman, R., Sadakane, K., Satti, S.R., Weimann, O.: Random access to grammar-compressed strings. In: Proc. SODA 2011, pp. 373–389 (2011)

6. Cilibrasi, R., Vitányi, P.M.: Clustering by compression. IEEE Transactions on Information Theory 51(4), 1523–1545 (2005)
7. Crochemore, M., Landau, G.M., Ziv-Ukelson, M.: A subquadratic sequence alignment algorithm for unrestricted scoring matrices. SIAM J. Comput. 32(6), 1654–1673 (2003)
8. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proc. FOCS 1997, pp. 137–143 (1997)
9. Freschi, V., Bogliolo, A.: A faster algorithm for the computation of string convolutions using LZ78 parsing. Information Processing Letters 110(14-15), 609–613 (2010)
10. Gawrychowski, P.: Optimal pattern matching in LZW compressed strings. In: Proc. SODA 2011, pp. 362–372 (2011)
11. Gawrychowski, P.: Tying up the loose ends in fully LZW-compressed pattern matching. In: Proc. STACS 2012, pp. 624–635 (2012)
12. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Fast q-gram Mining on SLP Compressed Strings. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 278–289. Springer, Heidelberg (2011)
13. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Speeding Up q-Gram Mining on Grammar-Based Compressed Texts. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 220–231. Springer, Heidelberg (2012)
14. Jansson, J., Sadakane, K., Sung, W.-K.: Compressed Dynamic Tries with Applications to LZ-Compression in Sublinear Time and Space. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 424–435. Springer, Heidelberg (2007)
15. Jeż, A.: Faster Fully Compressed Pattern Matching by Recompression. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part I. LNCS, vol. 7391, pp. 533–544. Springer, Heidelberg (2012)
16. Kida, T., Shibata, Y., Takeda, M., Shinohara, A., Arikawa, S.: Collage system: A unifying framework for compressed pattern matching. Theor. Comput. Sci. 298(1), 253–272 (2003)
17. Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Proc. DCC 1999, pp. 296–305. IEEE Computer Society (1999)
18. Li, M., Sleep, R.: Genre classification via an LZ78-based string kernel. In: Proc. ISMIR 2005, pp. 252–259 (2005)
19. Li, M., Sleep, R.: An LZ78 Based String Kernel. In: Li, X., Wang, S., Dong, Z.Y. (eds.) ADMA 2005. LNCS (LNAI), vol. 3584, pp. 678–689. Springer, Heidelberg (2005)
20. Li, M., Zhu, Y.: Image Classification Via LZ78 Based String Kernel: A Comparative Study. In: Ng, W.-K., Kitsuregawa, M., Li, J., Chang, K. (eds.) PAKDD 2006. LNCS (LNAI), vol. 3918, pp. 704–712. Springer, Heidelberg (2006)
21. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of ACM 23(2), 262–272 (1976)
22. Nevill-Manning, C.G., Witten, I.H., Maulsby, D.L.: Compression by induction of hierarchical grammars. In: Proc. DCC 1994, pp. 244–253 (1994)
23. Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1-3), 211–222 (2003)
24. Shibuya, T.: Constructing the suffix tree of a tree with a large alphabet. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E86-A(5), 1061–1066 (2003)
25. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)

26. Weiner, P.: Linear pattern-matching algorithms. In: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, pp. 1–11. Institute of Electrical Electronics Engineers, New York (1973)
27. Westbrook, J.: Fast Incremental Planarity Testing. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 342–353. Springer, Heidelberg (1992)
28. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3), 337–349 (1977)
29. Ziv, J., Lempel, A.: Compression of individual sequences via variable-length coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)