Compact q-gram Profiling of Compressed Strings

Philip Bille
phbi@dtu.dk

Patrick Hagge Cording
phaco@dtu.dk

Inge Li Gørtz
inge@dtu.dk

May 23, 2018

Abstract

We consider the problem of computing the q-gram profile of a string T of size N compressed by a context-free grammar with n production rules. We present an algorithm that runs in $O(N - \alpha)$ expected time and uses $O(n + q + k_{T,q})$ space, where $N - \alpha \leq qn$ is the exact number of characters decompressed by the algorithm and $k_{T,q} \leq N - \alpha$ is the number of distinct q-grams in T. This simultaneously matches the current best known time bound and improves the best known space bound. Our space bound is asymptotically optimal in the sense that any algorithm storing the grammar and the q-gram profile must use $\Omega(n + q + k_{T,q})$ space. To achieve this we introduce the q-gram graph that space-efficiently captures the structure of a string with respect to its q-grams, and show how to construct it from a grammar.

1 Introduction

Given a string T, the q-gram profile of T is a data structure that can answer substring frequency queries for substrings of length q (q-grams) in $O(q)$ time. We study the problem of computing the q-gram profile from a string T of size N compressed by a context-free grammar with n production rules. We assume that the model of computation is the standard w-bit word RAM where each word is capable of storing a character of T, i.e., the characters of T are drawn from an alphabet $\{1, \ldots, 2^w\}$, and hence $w \geq \log N$ [8]. The space complexities are measured by the number of words used.

The generalization of string algorithms to grammar-based compressed text is currently an active area of research. Grammar-based compression is studied because it offers a simple and strict setting and is capable of modelling many commonly used compression schemes, such as those in the Lempel-Ziv family [21, 22], with little expansion [2, 15]. The problem of computing the q-gram profile has its applications in bioinformatics, data mining, and machine learning [4, 12, 14]. All are fields where handling large amount of data effectively is crucial. Also, the q-gram distance can be computed from the q-gram profiles of two strings and used for filtering in string matching [1, 9, 17–20].

Recently the first dedicated solution to computing the q-gram profile from a grammar-based compressed string was proposed by Goto et al. [7]. Their algorithm runs in $O(qn)$ expected time[1] and uses $O(qn)$ space. This was later improved by the same authors [6] to an algorithm that takes $O(N - \alpha)$ expected time and uses $O(N - \alpha)$ space, where N is the size of the uncompressed string, and α is a parameter depending on how well T is compressed with respect to its q-grams.

---

[1]The bound in [7] is stated as worst-case since they assume alphabets of size $O(N^c)$ for fast suffix sorting, where c is a constant. We make no such assumptions and without it hashing can be used to obtain the same bound in expectation.

$N - \alpha \leq \min(qn, N)$ is in fact the exact number of characters decompressed by the algorithm in order to compute the q-gram profile, meaning that the latter algorithm excels in avoiding decompressing the same character more than once.

We present a Las Vegas-type randomized algorithm that gives Theorem 1.

Theorem 1 *Let T be a string of size N compressed by a grammar of size n. The q-gram profile can be computed in $O(N - \alpha)$ expected time and $O(n + q + k_{T,q})$ space, where $k_{T,q} \leq N - \alpha$ is the number of distinct q-grams in T.*

Hence, our algorithm simultaneously matches the current best known time bound and improves the best known space bound. Our space bound is asymptotically optimal in the sense that any algorithm storing the grammar and the q-gram profile must use $\Omega(n + q + k_{T,q})$ space.

A straightforward approach to computing the q-gram profile is to first decompress the string and then use an algorithm for computing the profile from a string. For instance, we could construct a compact trie of the q-grams using an algorithm similar to a suffix tree construction algorithm as mentioned in [10], or use Rabin-Karp fingerprints to obtain a randomized algorithm [20]. However, both approaches are impractical because the time and space usage associated with a complete decompression of T is linear in its size $N = O(2^n)$. To achieve our bounds we introduce the q-gram graph, a data structure that space efficiently captures the structure of a string in terms of its q-grams, and show how to compute the graph from a grammar. We then transform the graph to a suffix tree containing the q-grams of T. Because our algorithm uses randomization to construct the q-gram graph, the answer to a query may be incorrect. However, as a final step of our algorithm, we show how to use the suffix tree to verify that the fingerprint function is collision free and thereby obtain Theorem 1.

2 Preliminaries and Notation

2.1 Strings and Suffix Trees

Let T be a string of length $|T|$ consisting of characters from the alphabet Σ. We use $T[i : j]$, $0 \leq i \leq j < |T|$, to denote the substring starting in position i of T and ending in position j of T. We define $socc(s, T)$ to be the number of occurrences of the string s in T.

The suffix tree of T is a compact trie containing all suffixes of T. That is, it is a trie containing the strings $T[i : |T| - 1]$ for $i = 0..|T| - 1$. The suffix tree of T can be constructed in $O(|T|)$ time and uses $O(|T|)$ space [3]. The generalized suffix tree is the suffix tree for a set of strings. It can be constructed using time and space linear in the sum of the lengths of the strings in the set. The set of strings may be compactly represented as a common suffix tree (CS-tree). The CS-tree has the characters of the strings on its edges, and the strings start in the leaves and end in the root. If two strings have some suffix in common, the suffixes are merged to one path. In other words, the CS-tree is a trie of the reversed strings, and is not to be confused with the suffix tree. For CS-trees, the following is known.

Lemma 1 (Shibuya [16]) *Given a set of strings represented by a CS-tree of size n and comprised of characters from an alphabet of size $O(n^c)$, where c is a constant, the generalized suffix tree of the set of strings can be constructed in $O(n)$ time using $O(n)$ space.*

For a node v in a suffix tree, the string depth $sd(v)$ is the sum of the lengths of the labels on the edges from the root to v. We use $parent(v)$ to get the parent of v, and $nca(v, u)$ is the nearest common ancestor of the nodes v and u.

2.2 Straight Line Programs

A Straight Line Program (SLP) is a context-free grammar in Chomsky normal form that derives a single string T of length N over the alphabet Σ. In other words, an SLP $\mathcal{S}$ is a set of n production rules of the form $X_i = X_l X_r$ or $X_i = a$, where a is a character from the alphabet Σ, and each rule is reachable from the start symbol X_n. Our algorithm assumes without loss of generality that the compressed string given as input is compressed by an SLP.

It is convenient to view an SLP as a directed acyclic graph (DAG) in which each node represents a production rule. Consequently, nodes in the DAG have exactly two outgoing edges. An example of an SLP is seen in Figure 2(a). When a string is decompressed we get a derivation tree which corresponds to the depth-first traversal of the DAG.

We denote by t_{X_i} the string derived from production rule X_i, so $T = t_{X_n}$. For convenience we say that $|X_i|$ is the length of the string derived from X_i, and these values can be computed in linear time in a bottom-up fashion using the following recursion. For each $X_i = X_l X_r$ in $\mathcal{S}$,

$$|X_i| = \begin{cases} |X_l| + |X_r| & \text{if } X_i \text{ is a nonterminal,} \\ 1 & \text{otherwise.} \end{cases}$$

Finally, we denote by $occ(X_i)$ the number of times the production rule X_i occurs in the derivation tree. We can compute the occurrences using the following linear time and space algorithm due to Goto et al. [7]. Set $occ(X_n) = 1$ and $occ(X_i) = 0$ for $i = 1..n-1$. For each production rule of the form $X_i = X_l X_r$, in decreasing order of i, we set $occ(X_l) = occ(X_l) + occ(X_i)$ and similarly for $occ(X_r)$.

2.3 Fingerprints

A Rabin-Karp fingerprint function ϕ takes a string as input and produces a value small enough to let us determine with high probability whether two strings match in constant time. Let s be a substring of T, c be some constant, $2N^{c+4} < p \leq 4N^{c+4}$ be a prime, and choose $b \in \mathbb{Z}_p$ uniformly at random. Then,

$$\phi(s) = \sum_{k=1}^{|s|} s[k] \cdot b^k \mod p.$$

Lemma 2 (Rabin and Karp [11]) *Let ϕ be defined as above. Then, for all $0 \leq i, j \leq |T| - q$,*

$$\phi(T[i:i+q]) = \phi(T[j:j+q]) \quad \text{iff} \quad T[i:i+q] = T[j:j+q] \quad w.h.p.$$

We denote the case when $T[i:i+q] \neq T[j:j+q]$ and $\phi(T[i:i+q]) = \phi(T[j:j+q])$ for some i and j a collision, and say that ϕ is collision free on substrings of length q in T if $\phi(T[i:i+q]) = \phi(T[j:j+q])$ iff $T[i:i+q] = T[j:j+q]$ for all i and j, $0 \leq i, j < |T| - q$.

Besides Lemma 2, fingerprints exhibit the useful property that once we have computed $\phi(T[i : i + q])$ we can compute the fingerprint $\phi(T[i + 1 : i + q + 1])$ in constant time using the update function,

$$\phi(T[i + 1 : i + q + 1]) = \phi(T[i : i + q])/b - T[i] + T[i + q + 1] \cdot b^q \mod p.$$

3 Key Concepts

3.1 Relevant Substrings

Consider a production rule $X_i = X_l X_r$ that derives the string $t_{X_i} = t_{X_l} t_{X_r}$. Assume that we have counted the number of occurrences of q-grams in t_{X_l} and t_{X_r} separately. Then the relevant substring r_{X_i} is the smallest substring of t_{X_i} that is necessary and sufficient to process in order to detect and count q-grams that have not already been counted. In other words, r_{X_i} is the substring that contains q-grams that start in t_{X_l} and end in t_{X_r} as shown in Figure 1. Formally, for a production rule $X_i = X_l X_r$, the relevant substring is $r_{X_i} = t_{X_i}[\max(0, |X_l| - q + 1) : \min(|X_l| + q - 2, |X_i| - 1)]$. We want the relevant substrings to contain at least one q-gram, so we say that a production rule X_i only has a relevant substring if $|X_i| \geq q$. The size of a relevant substring is $q \leq |r_{X_i}| \leq 2(q - 1)$.
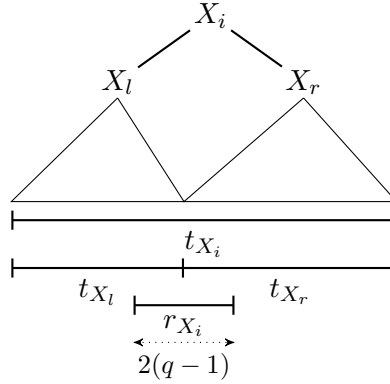

Figure 1: The derivation tree for $X_i = X_l X_r$ and the relevant susbtring r_{X_i} of X_i.

The concept of relevant substrings is the backbone of our algorithm because of the following. If X_i occurs $occ(X_i)$ times in the derivation tree for $\mathcal{S}$, then the substring t_{X_i} occurs at least $occ(X_i)$ times in T. It follows that if a q-gram s occurs $socc(s, t_{X_i})$ times in some substring t_{X_i} then we know that it occurs at least $socc(s, t_{X_i}) \cdot occ(X_i)$ times in T. Using our description of relevant substrings we can rewrite the latter statement to $socc(s, t_{X_i}) \cdot occ(X_i) = socc(s, t_{X_l}) \cdot occ(X_l) + socc(s, t_{X_r}) \cdot occ(X_r) + socc(s, r_{X_i}) \cdot occ(X_i)$ for the production rule $X_i = X_l X_r$. By applying this recursively to the root X_n of the SLP we get the following lemma.

Lemma 3 (Goto et al. [6]) *Let $\mathcal{S}_q = \{X_i \mid X_i \in \mathcal{S} \text{ and } |X_i| \geq q\}$ be the set of production rules that have a relevant substring, and let s be some q-gram. Then,*

$$socc(s, T) = \sum_{X_i \in \mathcal{S}_q} socc(s, r_{X_i}) \cdot occ(X_i).$$

4

3.2 Prefix and Suffix Decompression

The following Lemma states a result that is crucial to the algorithm presented in this paper.

Lemma 4 (Gąsieniec et al. [5]) *An SLP $\mathcal{S}$ of size n can be preprocessed in $O(n)$ time using $O(n)$ extra space such that, given a pointer to a variable X_i in $\mathcal{S}$, the prefix and suffix of t_{X_i} of length j can be decompressed in $O(j)$ time.*

Gąsieniec et al. give a data structure that supports linear time decompression of prefixes, but it is easy to extend the result to also hold for suffixes. Let s be some string and s^R the reversed string. If we reverse the prefix of length j of s^R this corresponds to the suffix of length j of s. To obtain an SLP for the reversed string we swap the two variables on the right-hand side of each nonterminal production rule. The reversed SLP $\mathcal{S}'$ contains n production rules and the transformation ensures that $t_{X_{i'}} = t_{X_i}^R$ for each production rule $X_{i'}$ in $\mathcal{S}'$. A proof of this can be found in [13]. Producing the reversed SLP takes linear time and in the process we create pointers from each variable to its corresponding variable in the reversed SLP. After both SLP's are preprocessed for linear time prefix decompression, a query for the length-j suffix of t_{X_i} is handled by following the pointer from X_i to its counterpart in the reversed SLP, decompressing the prefix of length j of this, and reversing the prefix.

3.3 The q-gram Graph

We now describe a data structure that we call the q-gram graph. It too will play an important role in our algorithm. The q-gram graph $G_q(T)$ captures the structure of a string T in terms of its q-grams. In fact, it is a subgraph of the De Bruijn graph over Σ^q with a few augmentations to give it some useful properties. We will show that its size is linear in the number of distinct q-grams in T, and we give a randomized algorithm to construct the graph in linear time in N.

A node in the graph represents a distinct $(q-1)$-gram, and the label on the node is the fingerprint of the respective $(q-1)$-gram. The graph has a special node that represents the first $(q-1)$-gram of T and which we will denote the start node. Let x and y be characters and α a string such that $|\alpha| = q-2$. There is an edge between two nodes with labels $\phi(x\alpha)$ and $\phi(\alpha y)$ if $x\alpha y$ is a substring of T. The graph may contain self-loops. Each edge has a label and a counter. The label of the edge $\{\phi(x\alpha), \phi(\alpha y)\}$ is y, and its counter indicates the number of times the substring $x\alpha y$ occurs in T. Since $|x\alpha y| = q$ this data structure contains information about the frequencies of q-grams in T.

Lemma 5 *The q-gram graph of T, $G_q(T)$, has $O(k_{T,q})$ nodes and $O(k_{T,q})$ edges.*

Proof. Each node represents a distinct $(q-1)$-gram and its outgoing edges have unique labels. The combination of a node and an outgoing edge thus represents a distinct q-gram, and therefore there can be at most $k_{T,q}$ edges in the graph. For every node with label $\phi(T[i : i+q-1])$, $i = 1..|T|-q-1$, the graph contains a node with label $\phi(T[i+1 : i+q])$ with an edge between the two. The graph is therefore connected and has at most has at most $k_{T,q} + 1$ nodes. $\square$

The graph can be constructed using the following online algorithm which takes a string T, an integer $q \geq 2$, and a fingerprint function ϕ as input. Let the start node of the graph have the fingerprint $\phi(T[0 : (q-1)-1])$. Assume that we have built the graph $G_q(T[0 : k+(q-1)-1])$ and that we keep its nodes and edges in two dictionaries implemented using hashing. We then compute

the fingerprint $\phi(T[k+1:k+(q-1)])$ for the $(q-1)$-gram starting in position $k+1$ in T. Recall that since this is the next successive q-gram, this computation takes constant time. If a node with label $\phi(T[k+1:k+(q-1)])$ already exists we check if there is an edge from $\phi(T[k:k+(q-1)-1])$ to $\phi(T[k+1:k+(q-1)])$. If such an edge exists we increment its counter by one. If it does not exist we create it and set its counter to 1. If a node with label $\phi(T[k+1:k+(q-1)])$ does not exist we create it along with an edge from $\phi(T[k:k+(q-1)-1])$ to it.

Lemma 6 *For a string T of length N, the algorithm is a Monte Carlo-type randomized algorithm that builds the q-gram graph $G_q(T)$ in $O(N)$ expected time.*

4   Algorithm

Our main algorithm is comprised of four steps: preparing the SLP, constructing the q-gram graph from the SLP, turning it into a CS-tree, and computing the suffix tree of the CS-tree. Ultimately the algorithm produces a suffix tree containing the reversed q-grams of T, so to answer a query for a q-gram s we will have to lookup s^R in the suffix tree. Below we will describe the algorithm and we will show that it runs in $O(qn)$ expected time while using $O(n+q+k_{T,q})$ space; an improvement over the best known algorithm in terms of space usage. The catch is that a frequency query to the resulting data structure may yield incorrect results due to randomization. However, we show how to turn the algorithm from a Monte Carlo to a Las Vegas-type randomized algorithm with constant overhead. Finally, we show that by decompressing substrings of T in a specific order, we can construct the q-gram graph by decompressing exactly the same number of characters as decompressed by the best known algorithm.

The algorithm is as follows. Figure 2 shows an example of the data structures after each step of the algorithm.

Preprocessing.   As the first step of our algorithm we preprocess the SLP such that we know the size of the string derived from a production rule, $|X_i|$, and the number of occurrences in the derivation tree, $occ(X_i)$. We also prepare the SLP for linear time prefix and suffix decompressions using Lemma 4.

Computing the q-gram graph.   In this step we construct the q-gram graph $G_q(T)$ from the SLP $\mathcal{S}$. Initially we choose a suitable fingerprint function for the q-gram graph construction algorithm and proceed as follows. For each production rule $X_i = X_l X_r$ in $\mathcal{S}$, such that $|X_i| \geq q$, we decompress its relevant substring r_{X_i}. Recall from the definition of relevant substrings that r_{X_i} is the concatenation of the $q-1$ length suffix of t_{X_l} and the $q-1$ length prefix of t_{X_r}. If $|X_l| \leq q-1$ we decompress the entire string t_{X_l}, and similarly for t_{X_r}. Given r_{X_i} we compute the fingerprint of the first $(q-1)$-gram, $\phi(r_{X_i}[0:(q-1)-1])$, and find the node in $G_q(T)$ with this fingerprint as its label. The node is created if it does not exist. Now the construction of $G_q(T)$ can continue from this node, albeit with the following change to the construction algorithm. When incrementing the counter of an edge we increment it by $occ(X_i)$ instead of 1.

The q-gram graph now contains the information needed for the q-gram profile; namely the frequencies of the q-grams in T. The purpose of the next two steps is to restructure the graph to a data structure that supports frequency queries in $O(q)$ time.

6

Transforming the q-gram graph to a CS-tree. The CS-tree that we want to create is basically the depth-first tree of $G_q(T)$ with the extension that all edges in $G_q(T)$ are also in the tree. We create it as follows. Let the start node of $G_q(T)$ be the node whose label match the fingerprint of the first $q - 1$ characters of T. Do a depth-first traversal of $G_q(T)$ starting from the start node. For a previously unvisited node, create a node in the CS-tree with an incoming edge from its predecessor. When reaching a previously visited node, create a new leaf in the CS-tree with an incoming edge from its predecessor. Labels on nodes and edges are copied from their corresponding labels in $G_q(T)$. We now create a path of length $q - 1$ with the first $q - 1$ characters of T as labels on its edges. We set the last node on this path to be the root of the depth-first tree. The first node on the path is the root of the final CS-tree.

Computing the suffix tree of the CS-tree. Recall that a suffix in the CS-tree starts in a node and ends in the root of the tree. Usually we store a pointer from a leaf in the suffix tree to the node in the CS-tree from which the particular suffix starts. However, when we construct the suffix tree, we store the value of the counter of the first edge in the suffix as well as the label of the first node on the path of the suffix.
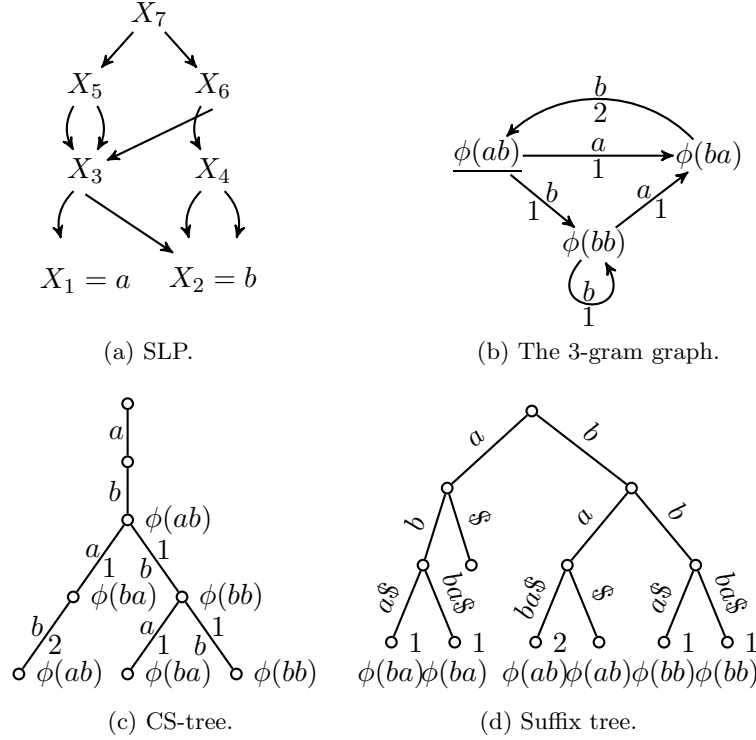

Figure 2: An SLP compressing the string `ababbbab`, and the data structures after each step of the algorithm executed with $q = 3$.

Our algorithm resembles the one by Goto et al. [6]. The main difference between the algorithms is that Goto et al. use the so-called neighbour graph to capture the q-grams of T where we use

the q-gram graph. This is also the key to the improvement in space usage. If a q-gram occurs in several relevant substrings it will occur several times in the neighbour graph but only once in the q-gram graph.

4.1 Correctness

Before showing that our algorithm is correct, we will prove some crucial properties of the q-gram graph, the CS-tree, and the suffix tree of the CS-tree subsequent to their construction in the algorithm.

Lemma 7 *The q-gram graph $G_q(T)$ constructed from the SLP is connected.*

Proof. Consider a production rule $X_i = X_l X_r$. If $|X_i| \leq 2(q-1)$ we decompress the entire string t_{X_i} and insert it into the q-gram graph, and we know that $G_q(t_{X_i})$ is connected. Assume that $G_q(t_{X_l})$ and $G_q(t_{X_r})$ are both connected. We know from Lemma 3 that if we insert all the relevant substrings of the nodes reachable from X_l (including X_l) into the graph, then it will contain all $(q-1)$-grams of t_{X_l}. Since the first $q-1$ characters of r_{X_i} is a suffix of t_{X_l}, the subgraphs $G_q(t_{X_l})$ and $G_q(r_{X_i})$ will have at least one node in common, and similarly for $G_q(t_{X_r})$ and $G_q(r_{X_i})$. Therefore, $G_q(X_i)$ is connected. $\square$

Lemma 8 *Assuming that we are given a fingerprint function ϕ that is collision free for substrings of length $q-1$ in T, then the CS-tree built by the algorithm contains each distinct q-gram in T exactly once.*

Proof. Let v be a node with an outgoing edge e in $G_q(T)$. The combination of the label of v followed by the character on e is a distinct q-gram and occurs only once in $G_q(T)$ due to the way we construct it. There may be several paths of length $q-1$ ending in v spelling the same string s, and because the fingerprint function is deterministic, there can not be a path spelling s ending in some other node. Since the depth-first traversal of $G_q(T)$ only visits e once, the resulting CS-tree will only contain the combination of the labels on v and e once. $\square$

Lemma 9 *Assuming that we are given a fingerprint function ϕ that is collision free for substrings of length $q-1$ in T, then any node v in the suffix tree of the CS-tree with $sd(v) \geq q$ is a leaf.*

Proof. Each suffix of length $\geq q$ in the CS-tree has a distinct q length prefix (Lemma 8), so therefore each node in the suffix tree with string depth $\geq q$ is a leaf. $\square$

We have now established the necessary properties to prove that our algorithm is correct.

Lemma 10 *Assuming that we are given a fingerprint function ϕ that is collision free on all substrings of length $q-1$ of T, our algorithm correctly computes a q-gram profile for T.*

Proof. Our algorithm inserts each relevant substring r_{X_i} exactly once, and if a q-gram s occurs $socc(s, r_{X_i})$ times in r_{X_i}, the counter on the edge representing s is incremented by exactly $socc(s, r_{X_i}) \cdot occ(X_i)$. From Lemma 3 we then know that when $G_q(T)$ is fully constructed, the counters on its edges correspond to the frequencies of the q-grams in T. Since $G_q(T)$ is connected (Lemma 7) the tree created by the algorithm is a CS-tree that contains each q-gram from $G_q(T)$ exactly once (Lemma 8). Finally, we know from Lemma 9 that a node v with $sd(v) \geq q$ in the suffix tree is a leaf, so searching for a string of length q in the suffix tree will yield a unique result and can be done in $O(q)$ time. $\qquad\square$

4.2 Analysis

Theorem 2 *The algorithm runs in $O(qn)$ expected time and uses $O(n + q + k_{T,q})$ space.*

Proof. Let $\mathcal{S}_q = \{X_i \mid X_i \in \mathcal{S} \text{ and } |X_i| \geq q\}$ be the set of production rules that have a relevant substring. For each production rule $X_i = X_l X_r \in \mathcal{S}_q$ we decompress its relevant substring of size $|r_{X_i}|$ and insert it into the q-gram graph. Since r_{X_i} is comprised of the suffix of t_{X_l} and the prefix of t_{X_r} we know from Lemma 4 that r_{X_i} can be decompressed in $O(|r_{X_i}|)$ time. Inserting r_{X_i} into the q-gram graph can be done in $O(|r_{X_i}|)$ expected time (Lemma 6). Since $|\mathcal{S}_q| = O(n)$ and $q \leq |r_{X_i}| \leq 2(q - 1)$ this step of the algorithm takes $O(qn)$ time. When transforming the q-gram graph to a CS-tree we do one traversal of the graph and add $q - 1$ nodes, so this step takes $O(q + k_{T,q})$ time. Constructing the suffix tree takes expected linear time in the size of the CS-tree if we hash the characters of the alphabet to a polynomial range first (Lemma 1). Finally, observe that since our algorithm is correct, it detects all q-grams in T and therefore there can be at most $k_{T,q} \leq \sum_{X_i \in \mathcal{S}_q} |r_{X_i}| = O(qn)$ distinct q-grams in T. Thus, the expected running time of our algorithm is $O(qn)$.

In the preprocessing step of our algorithm we use $O(n)$ space to store the size of the derived substrings and the number of occurrences in the derivation tree as well as the data structure needed for linear time prefix and suffix decompressions (Lemma 4). The space used by the q-gram graph is $O(k_{T,q})$, and when transforming it to a CS-tree we add at most one new node per edge in the graph and extend it by $q-1$ nodes and edges. Thus, its size is $O(q+k_{T,q})$. The CS-tree contains $O(q+k_{T,q})$ suffixes, so the size of the suffix tree is $O(q+k_{T,q})$. In total our algorithm uses $O(n+q+k_{T,q})$ space. $\square$

4.3 Verifying the fingerprint function

Until now we have assumed that the fingerprints used as labels for the nodes in the q-gram graph are collision free. In this section we describe an algorithm that verifies if the chosen fingerprint function is collision free using the suffix tree resultant from our algorithm.

If there is a collision among fingerprints, the q-gram graph construction algorithm will add an edge such that there are two paths of length $q - 1$ ending in the same node while spelling two different strings. This observation is formalized in the next lemma.

Lemma 11 *For each node v in $G_q(T)$, if every path of length $q - 1$ ending in v spell the same string, then the fingerprint function used to construct $G_q(T)$ is collision free for all $(q - 1)$-grams in T.*

Proof. From the q-gram graph construction algorithm we know that we create a path of characters in the same order as we read them from T. This means that every path of length $q-1$ ending in a node v represents the $q-1$ characters generating the fingerprint stored in v, regardless of what comes before those $q-1$ characters. If all the paths of length $q-1$ ending in v spell the same string s, then we know that there is no other substring $s' \neq s$ of length $q-1$ in T that yields the fingerprint $\phi(s)$. $\square$

It is not straightforward to check Lemma 11 directly on the q-gram graph without using too much time. However, the error introduced by a collision naturally propagates to the CS-tree and the suffix tree of the CS-tree, and as we shall now see, the suffix tree offers a clever way to check for collisions. First, recall that in a leaf v in the suffix tree, we store the fingerprint of the reversed prefix of length $q-1$ of the suffix ending in v. Now consider the following property of the suffix tree.

Lemma 12 *Let v_ϕ be the fingerprint stored in a leaf v in the suffix tree. The fingerprint function ϕ is collision free for $(q-1)$-grams in T if $v_\phi \neq u_\phi$ or $sd(nca(v, u)) \geq q-1$ for all pairs v, u of leaves in the suffix tree.*

Proof. Consider the contrapositive statement: If ϕ is not collision free on T then there exists some pair v, u for which $v_\phi = u_\phi$ and $sd(nca(v, u)) < q-1$. Assume that there is a collision. Then at least two paths of length $q-1$ spelling the same string end in the same node in $G_q(T)$. Regardless of the order of the nodes in the depth-first traversal of $G_q(T)$, the CS-tree will have two paths of length $q-1$ spelling different strings and yet starting in nodes storing the same fingerprint. Therefore, the suffix tree contains two suffixes that differ by at least one character in their $q-1$ length prefix while ending in leaves storing the same fingerprint, which is what we want to show. $\square$

Checking if there exists a pair of leaves where $v_\phi = u_\phi$ and $sd(nca(v, u)) < q-1$ is straightforward. For each leaf we store a pointer to its ancestor w that satisfies $sd(w) \geq q-1$ and $sd(parent(w)) < q-1$. Then we visit each leaf v again and store v_ϕ in a dictionary along with the ancestor pointer just defined. If the dictionary already contains v_ϕ and the ancestor pointer points to a different node, then it means that $v_\phi = u_\phi$ and $sd(nca(v, u)) < q-1$ for some two leaves.

The algorithm does two passes of the suffix tree which has size $O(q + k_{T,q})$. Using a hashing scheme for the dictionary we obtain an algorithm that runs in $O(q + k_{T,q})$ expected time.

4.4 Eliminating redundant decompressions

We now present an alternative approach to constructing the q-gram graph from the SLP. The resulting algorithm decompresses fewer characters.

In our first algorithm for constructing the q-gram graph we did not specify in which order to insert the relevant substrings into the graph. For that reason we do not know from which node to resume construction of the graph when inserting a new relevant substring. So to determine the node to continue from, we need to compute the fingerprint of the first $(q-1)$-gram of each relevant substring. In other words, the relevant substrings are overlapping, and consequently some characters are decompressed more than once. Our improved algorithm is based on the following observation. Consider a production rule $X_i = X_l X_r$. If all relevant substrings of production rules reachable from X_l (including r_{X_l}) have been inserted into the graph, then we know that all q-grams

in t_{X_l} are in the graph. Since the $q - 1$ length prefix of r_{X_i} is also a suffix of t_{X_l}, then we know that a node with the label $\phi(r_{X_i}[0 : (q - 1) - 1])$ is already in the graph. Hence, after inserting all relevant substrings of production rules reachable from X_l we can proceed to insert r_{X_i} without having to decompress $r_{X_i}[0 : (q - 1) - 1]$.

Algorithm. First we compute and store the size of the relevant substring $|r_{X_i}| = \min(q-1, |X_l|) + \min(q - 1, |X_r|)$ for each production rule $X_i = X_l X_r$ in the subset $\mathcal{S}_q = \{X_i \mid X_i \in \mathcal{S} \text{ and } X_i \geq q\}$ of the production rules in the SLP. We maintain a linked list L with a pointer to its head and tail, denoted by $head(L)$ and $tail(L)$. The list is initially empty.

We now start decompressing T by traversing the SLP depth-first, left-to-right. When following a pointer from X_i to a right child, and $X_i \in \mathcal{S}_q$, we add X_i and the sentinel value $|r_{X_i}| - (q-1)$ to the back of L. As characters are decompressed they are fed to the q-gram graph construction algorithm, and when a counter on an edge in $G_q(T)$ is incremented, we increment it by $occ(head(L))$. For each character we decompress, we decrement the sentinel value for $head(L)$, and if this value becomes 0 we remove the head of the list and set $head(L)$ to be the next production rule in the list. Note that when L is empty in the beginning of the execution of the algorithm we do not alter any values.

When leaving a node $X_i \in \mathcal{S}_q$ we mark it as visited and store a pointer from X_i to the node with label $\phi(t_{X_i}[|X_i| - (q - 1) : |X_i| - 1])$ in $G_q(T)$, i.e., the node labelled with the suffix of length $q - 1$ of t_{X_i}. To do this we need to consider two cases. Let $X_i = X_l X_r$. If $X_r \in \mathcal{S}_q$ then we copy the pointer from X_r. If $X_r \notin \mathcal{S}_q$ then $\phi(t_{X_i}[|X_i| - (q - 1) : |X_i| - 1])$ is the most recently visited node in $G_q(T)$.

If we encounter a node that has been marked as visited, we decompress its prefix of length $q - 1$ using the data structure of Lemma 4, set the node with label $\phi(t_{X_i}[|X_i| - (q - 1) : |X_i| - 1])$ to be the node from where construction of the q-gram graph should continue, and do not proceed to visit its children nor add it to L.

Analysis. Assume without loss of generality that the algorithm is at a production rule deriving the string $t_{X_i} = t_{X_l} t_{X_r}$ and all q-grams in t_{X_l} are in $G_q(T)$. There is always such a rule, since we start by decompressing the string derived by the left child of the leftmost rule in $\mathcal{S}_q$. For each variable X_i added to L we decompress $|r_{X_i}| - (q - 1)$ characters before X_i is removed from the list. We only add a variable once to the list, so the total number of characters decompressed is at most $(q - 1) + \sum_{X_i \in \mathcal{S}_q} |r_{X_i}| - (q - 1) = O(N - \alpha)$, and we hereby obtain our result from Theorem 1. This is fewer characters than our first algorithm that require $\sum_{X_i \in \mathcal{S}_q} |r_{X_i}|$ characters to be decompressed. Furthermore, it is exactly the same number of characters decompressed by the fastest known algorithm due to Goto et al. [6].

References

[1] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (QUASAR). In *Proc. 3rd RECOMB*, pages 77–83, 1999.

[2] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Trans. Inf. Theory*, 51(7):2554–2576, 2005.

[3] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th FOCS*, pages 137–143, 1997.

[4] T. Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, 2003.

[5] L. Gąsieniec, R. Kolpakov, I. Potapov, and P. Sant. Real-time traversal in grammar-based compressed files. In *Proc. 15th DCC*, page 458, 2005.

[6] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. Speeding up q-gram mining on grammar-based compressed texts. In *Proc. 23rd CPM*, pages 220–231, 2012.

[7] K. Goto, H. Bannai, S. Inenaga, and M. Takeda. Fast q-gram mining on SLP compressed strings. *J. Discrete Algorithms*, 18(0):89–99, 2013.

[8] T. Hagerup. Sorting and searching on the word ram. In *Proc. 15th STACS*, pages 366–398, 1998.

[9] P. Jokinen and E. Ukkonen. Two algorithms for approximate string matching in static texts. In *Proc. 16th MFCS*, pages 240–248, 1991.

[10] J. Kärkkäinen and E. Sutinen. Lempel–Ziv index for q-grams. *Algorithmica*, 21(1):137–154, 1998.

[11] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.

[12] C. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: A string kernel for SVM protein classification. In *Proc. PSB*, volume 7, pages 566–575, 2002.

[13] W. Matsubara, S. Inenaga, A. Ishino, A. Shinohara, T. Nakamura, and K. Hashimoto. Efficient algorithms to compute compressed longest common substrings and compressed palindromes. *Theoret. Comput. Sci.*, 410(8):900–913, 2009.

[14] G. Paaß, E. Leopold, M. Larson, J. Kindermann, and S. Eickeler. SVM classification using sequences of phonemes and syllables. In *Proc. 6th PKDD*, pages 373–384, 2002.

[15] W. Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoret. Comput. Sci.*, 302(1):211–222, 2003.

[16] T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. *IEICE Trans. Fundamentals*, 86(5):1061–1066, 2003.

[17] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proc. 3rd ESA*, pages 327–340, 1995.

[18] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. 7th CPM*, pages 50–63, 1996.

[19] T. Takaoka. Approximate pattern matching with samples. In *Proc. 5th ISAAC*, pages 234–242, 1994.

[20] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theoret. Comput. Sci.*, 92(1):191–211, 1992.

[21] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.

[22] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Trans. Inf. Theory*, 24(5):530–536, 1978.