Text Compression and Compressed String Mining

後藤, 啓介

https://doi.org/10.15017/1441265

出版情報：九州大学, 2013, 博士（理学）, 課程博士
バージョン：

Text Compression and Compressed String Mining

Keisuke Goto

January, 2014

Abstract

Due to the rapid advance in computer technology and global growth of computer networks, we can utilize a large amount of machine-readable data today. Most of such data can be seen as sequences of characters, or strings, and the demand for mining valuable information from them is increasing. To mine valuable information from them, efficient string mining algorithms applicable to large-scale string data are needed. In this thesis, we develop fast and space efficient string mining algorithms for enormous string data, using text compression as a core technology. We focus on compressed string processing, which is an approach that directly processes compressed data without explicit decompression.

We present simple and efficient algorithms for calculating all frequencies of q-grams that occur in a string T represented in compressed form, namely, as a straight line program (SLP). Our algorithm runs in $O(qn)$ time and space, where n is the size of the SLP. Computational experiments show that our algorithm and its variation are practical for small q, actually running faster on various real string data, compared to algorithms that work on the uncompressed text. We also discuss applications in data mining and classification of string data, for which our algorithms can be useful.

We then improve the algorithm so that it can handle larger q. We propose an $O(\min\{qn, N - dup(q, \mathcal{T})\})$ algorithm improving on our previous $O(qn)$ algorithm when $q = \Omega(N/n)$, where N is the length of T and $dup(q, \mathcal{T})$ is a quantity that represents the amount of redundancy that the SLP captures with respect to q-grams in T. The algorithm is asymptotically always at least as fast and better in many cases compared to working on the uncompressed strings.

We further consider the extended problem which computes non-overlapping occurrence frequency of all q-grams. The non-overlapping occurrence frequency of a string P in a string T is defined as the maximum number of non-overlapping occurrences of P in T. We present the first algorithm for calculating the non-overlapping occurrence frequency of all q-grams, that works for any $q \geq 2$, and runs in $O(q^2 n)$ time and $O(qn)$ space.

Since the runtime of compressed string processing algorithms depends on the size of an input SLP, it is important to develop algorithms to compute, from a given text, an SLP of small size that derives it. It is known that the computation of the smallest sized grammar of a string is NP-hard, and therefore several approximation algorithms have been proposed. Rytter pro-

posed an $O(\log N)$ approximation algorithm (Rytter 2003), which is one of several algorithms which achieve the best known approximation ratio running in linear time. The algorithm firstly computes the LZ77 factorization of a given string T and then transforms it into an SLP. The bottleneck here is the computation of the LZ77 factorization from T.

To eliminate the above bottleneck, we propose linear time LZ77 factorization algorithms that are fast in practice. Computational experiments on various data sets show that our algorithms constantly outperform LZ_OG (Ohlenbusch and Gog 2011) which is one of the fastest existing linear time algorithms, and can be up to 2 to 3 times faster in the processing after obtaining the suffix array.

We also propose space efficient linear time LZ77 factorization algorithms. Our new algorithms use $N \log N + O(\sigma \log N)$ bits of working space, where σ is the alphabet size. Computational experiments show that our algorithms are only about 2-3 times as slow as KKP2 (Kärkkäinen et al. 2013), which is the fastest algorithm among linear time algorithms using $2N \log N$ bits of working space, despite the intricacies introduced in order to use less space.

Acknowledgments

I would like to thank everyone who supported me in my research and life at Kyushu University.

First, I would express my appreciation to my supervisor, Professor Masayuki Takeda. He kindly directed me and taught many things to me, how to research, how to think, and so on. I will never forget anything that I learned from him. I would also express my appreciation to Professor Hideo Bannai and Professor Shunsuke Inenaga. They always supported and encouraged me in various situations. Without their support, most of my research would not have gone well. I would like to thank Professor Eiji Takimoto and Professor Masafumi Yamashita, who are the members of the committee of my thesis.

I would like to express my appreciation to all students in the laboratory. I enjoyed very much the discussion with them, and their comments were very helpful for my research. Thanks to them, I was able to live a satisfying research life.

This research was partly supported by JSPS (Japan Society for the Promotion of Science). The results in the thesis were partially published in the Proc. of SPIRE'11, the Proc. of SOFSEM'12, the Proc. of CPM'12, the Proc. of DCC'13, and the Proc. of DCC'14. I am thankful for all editors, committees, anonymous referees, and publishers.

Last, but not least, I really thank my parents and wife Narumi for their support.

Contents

Chapter 1

Introduction

1.1 Background and Motivation

Due to the progress and spread of computer and sensor technologies, we can now obtain enormous sized machine-readable data. Most of such data can be seen as sequences of characters, or strings, and the demand for mining valuable information from them is increasing. To this end, it is necessary to develop efficient string mining algorithms applicable to large-scale string data. In this thesis, we develop fast and space efficient string mining algorithms for enormous string data, using text compression as a core technology.

Text compression is a widely used technology that allows us to represent strings more compactly by detecting and removing the redundancies in them. Though text compression is useful for reducing storage and communication costs, we usually need to decompress the compressed representation of the data before utilizing and analyzing them. It takes extra time compared with processing from uncompressed strings. Compressed string processing (CSP) is an approach that directly processes compressed data without explicit decompression in order to reduce the above mentioned overhead. One goal of CSP is to develop algorithms such that **[Goal 1]** *processing time on compressed strings* $<$ *decompression time + processing time on uncompressed strings*. The algorithms in [54, 56–58] achived **Goal 1** for the exact and approximate pattern matching problems. A more difficult and challenging goal is to develop algorithms such that **[Goal 2]** *processing time on compressed strings* $<$ *processing time on uncompressed strings.* The algorithms in [51, 64] achieved **Goal 2** for the exact pattern matching problem. CSP has been studied especially for pattern matching problems [9, 20, 34, 39, 49, 54–58, 68, 73]. On the other hand, other CSP algorithms, e.g. for problems of equality testing and edit distance, have also been proposed [21, 30, 30, 33, 53], but they consider only theoretical aspects.

Since there exist many different text compression schemes, it is not realistic to develop different algorithms for respective schemes. Thus, it is common to consider algorithms on strings represented as *straight line programs* (SLPs) [30, 39, 48]. An SLP is a context free grammar in the Chomsky normal form that derives a single string. Outputs of various compression algorithms, e.g. Sequitur [59], Re-Pair [46], and Lempel-Ziv family [66, 72, 74, 75], can be regarded as or quickly transformed to SLPs. Thus many CSP algorithms assume that the input is given as an SLP. SLPs can represent strings of length exponential in its size. Therefore in such extreme case, if we can develop polynomial time algorithms on SLPs, they can run exponentially faster than the algorithms on uncompressed strings, and consume exponentially less space as well. Recently, even compressed self-indices based on SLPs have appeared [13], and SLPs are a promising representation of compressed strings for conducting various operations.

Since the runtime of CSP algorithms depends on the input SLP size, it is important to develop algorithms to compute, from a given text, an SLP of small size that derives it. However, it is known that the computation of the smallest sized grammar of a string is NP-hard [11]. There is no polynomial time algorithm to compute the smallest sized grammar unless P=NP. Therefore many approximation algorithms have been proposed that guarantee the output size is proportional in approximation of the smallest grammar size [42, 46, 52, 59, 75]. We only focus on linear time approximation algorithms since we want to treat enormous data.

The runtime of some CSP algorithms on SLPs depends not only on the size of a given SLP but also on other properties of the SLP. For example, the algorithm in [13] depends on the height of the derivation tree of a given SLP to locate the occurrences of a given pattern in the decompressed string, and the algorithm [21] computing edit distance depends on, for each internal node of the derivation tree, the balancedness with respect to the decompressed strings for its left and right children. Therefore, it is also important to develop compression algorithms which obtain SLPs having good properties for CSP.

1.2 Our Contribution

In this thesis, we explore more advanced fields of applications for CSP, and especially study CSP for string mining, we call *compressed string mining*. There are two ways to speed up CSP. The first is to develop fast and space efficient algorithms running on compressed strings, and the second is to develop efficient approximation algorithms which output smaller SLPs. Our main contributions are the following two.

(A) Developing efficient algorithms to compute q-gram frequencies on SLPs. We consider the problem of computing all frequencies of q-grams that occur in a string T when give an SLP of size n representing T of size N over an alphabet Σ. Frequencies of q-grams are important features of string data, widely used in many fields such as text and natural language processing [8], machine learning [3], and bioinformatics [6].

Inenaga and Bannai proposed [33] an $O(|\Sigma|^2 n^2)$-time $O(n^2)$-space algorithm for finding the most frequent 2-gram from an SLP. Claude and Navarro [13] mentioned that the most frequent 2-gram can be found in $O(|\Sigma|^2 n \log n)$ time and $O(n \log N)$ space, if the SLP is pre-processed and a self-index is built. It is possible to extend these two algorithms to handle q-grams for $q > 2$, but would respectively require $O(|\Sigma|^q qn^2)$ and $O(|\Sigma|^q qn \log n)$ time, since they must essentially enumerate and count the occurrences of all substrings of length q, regardless of whether the q-gram occurs in the string. Note also that any algorithm that first decompresses the SLP obtaining the entire text T, and then works on the decompressed text, requires exponential time in the worst case, since N can be as large as $O(2^n)$.

We propose an $O(qn)$-time and space algorithm that computes all frequencies of q-grams that occur in a string when given an SLP of size n representing the string. We prove that the q-gram frequencies problem on SLPs can be reduced to the weighted q-gram frequencies problem on a single uncompressed string of size at most $2(q-1)n$, and the reduced problem can be solved in time proportional to the input size of $2(q-1)n$. The algorithm solves the more general problem and greatly improves the computational complexity compared to previous work, moreover the algorithm achieved **Goal 2**. Computational experiments show that our algorithm and its variation are practical for small q, actually running faster on various real string data, compared to algorithms that work on the uncompressed text. Our algorithms have profound applications in the field of string mining and classification. We discuss several applications and extensions. For example, our algorithm leads to an $O(q(n_1 + n_2))$-time algorithm for computing the q-gram spectrum kernel [47] between SLP compressed texts of size n_1 and n_2. It also leads to an $O(qn)$-time algorithm for finding the optimal q-gram (or emerging q-gram) that discriminates between two sets of SLP compressed strings, when n is the total size of the SLPs.

We then improve the algorithm to be able to handle larger q. The improved algorithm is asymptotically always at least as fast and better in many cases compared to working on the uncompressed strings. Though the $O(qn)$ algorithm runs faster than algorithms on uncompressed strings when q is small, it is slower when q is large. This is because the total length of the partial decompressions in the algorithm becomes longer than the uncompressed string T. Theoretically, q can be as large as $O(N)$, hence in such a case the algorithm requires $O(Nn)$ time, which is worse than a trivial $O(N)$ solution that first decompresses the given SLP and runs a linear time algorithm for q-gram frequencies computation on T. We propose an $O(\min\{qn, N - dup(q, \mathcal{T})\})$ algorithm improving on our previous $O(qn)$ algorithm when $q = \Omega(N/n)$. The computational experiments show that our new approach achieves a practical speed up as well, for all values of q.

We further consider the extended problem which computes *non-overlapping occurrence frequency* of all q-grams. The *non-overlapping occurrence frequency* of a string P in a string T is defined as the maximum number of non-overlapping occurrences of P in T [2]. The problem for SLP was first considered in [32], where an algorithm for $q = 2$ running in $O(n^4 \log n)$ time and $O(n^3)$ space was presented. However, the algorithm cannot be readily extended to handle $q > 2$. Intuitively, the problem for $q = 2$ is much easier compared to larger values of q, since there is only one way for a 2-gram to overlap, while there can be many ways that a longer q-gram can overlap. We present the first algorithm for calculating the non-overlapping occurrence frequency of all q-grams, that works for any $q \geq 2$, and runs in $O(q^2 n)$ time and $O(qn)$ space. Not only do we solve a more general problem, but the complexity is greatly improved compared to previous work.

(B) Developing efficient compression algorithms with high compression ratio. We consider the problem of computing a smaller sized SLP representing a given string T. Rytter [63] proposed an algorithm that, given the LZ77 factorization of T, computes an SLP of size $O(z \log N)$ representing T in output linear time, where z is the size of the LZ77 factorization of T and N is the length of T. This is one of several algorithms which achieve the best known approximation ratio running in linear time. Moreover, the SLP by Rytter's algorithm has good feature that the height of derivation tree is $O(\log N)$ since the derivation tree of the SLP is of form AVL trees. For a string T, we can obtain an SLP of T by firstly computing the LZ77 factorization of T, and then computing an SLP from the LZ77 factorization using Rytter's algorithm. The bottleneck here is the computation of the LZ77 factorization from T. We propose several fast and space efficient algorithms to compute the LZ77 factorization of a given string T in linear time.

Fast linear time LZ77 factorization algorithm. Most recent efficient linear time algorithms are off-line, and use $O(N)$ space for integer alphabets [12, 15–17, 36, 62]. They first construct the suffix arrays [50] of the string, and then compute an array called the Longest Previous Factor (LPF) array from which the LZ77 factorization can be easily computed [1, 12, 16, 17, 62]. Many algorithms of this family first compute the longest common prefix (LCP) array prior to the computation of the LPF array. However, the computation of the LCP array is also costly. The algorithm CI1 (COMPUTE_LPF) of [15], and the algorithm LZ_OG [62] cleverly avoids its computation and directly computes the LPF array.

An important observation here is that the LPF array is actually more information than is required for the computation of the LZ77 factorization, i.e., if our objective is the LZ77 factorization, we only use a subset of the entries in the LPF array . However, the above algorithms focus on computing the entire LPF array, perhaps since it is difficult to determine beforehand, which entries of LPF are actually required. Although some algorithms such as a variant of CPS1 or CPS2 in [12] avoid computation of LPF, they either require the LCP array, or do not run in linear worst case time and are not as efficient. (See [1] for a survey.)

We propose a new approach to avoid the computation of LCP and LPF arrays altogether, and our algorithms run in linear time and using three to four integer arrays of length N. The resulting algorithms are surprisingly both simple and efficient. Computational experiments on various data sets show that our algorithms constantly outperform LZ_OG [62] which is one of the fastest among existing linear time algorithms, and can be up to 2 to 3 times faster in the processing after obtaining the suffix array, while requiring the same or a little more space.

Space efficient linear time LZ77 factorization algorithm. Kärkkäinen et al. [36] independently and almost simultaneously proposed 3 algorithms which avoid the computation of LPF array. Their algorithms are called KKP3, KKP2, and KKP1, which respectively store and utilize 3, 2, and 1 auxiliary integer arrays of length N kept in main memory. KKP3 can be seen as reorganization of one of our algorithms, but is modified so that array access are more cache friendly, thus making the algorithm run faster. KKP2 is based on KKP3, but further reduces one integer array by an elegant technique that rewrites values on the integer array. KKP1 is the same as KKP2, except that it assumes that the suffix array is stored on disk, but since the values of the suffix array are only accessed sequentially, the suffix array is streamed from the disk. Thus, KKP1 can be regarded as requiring only a single integer array to be held in memory. In this sense, KKP1 is the most space efficient linear time algorithm, and has been shown to be faster than KKP2, if we assume that the suffix array is already computed and exists on disk [36]. However, note that the *total* space requirement of KKP1 is still two integer arrays, one existing in memory and the other existing on disk.

We propose new algorithms for computing the LZ77 factorization that uses only $N \log N + O(\sigma \log N)$ bits of working space. We achieve this by introducing a series of techniques for rewriting the various auxiliary integer arrays from one to another, in linear time and in-place, i.e., using only $O(\sigma \log N)$ bits of working space. Computational experiments show that our algorithm is at most around twice as slow as previous algorithms, but in turn, uses only half the total space, and may be a viable alternative when the total space (including disk) is a limiting factor due to the enormous size of data.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows.

In Chapter 2 we define some notations and introduce several important data structures such as SLPs, Suffix Arrays, LCP arrays, and LZ77 factorization.

In Chapter 3, we present algorithms that computes all q-gram frequencies of a string from a given SLP repreesnting the string without explicit decompression. We also explain applications of q-gram frequencies to several data mining tasks, and describe efficient CSP solutions based on the above algorithm.

In Chapter 4, we show how to improve the algorithm in Chapter 3 in order to handle large q.

In Chapter 5, we present an algorithm that computes all non-overlapping q-gram frequencies of a string from a given SLP representing the string without explicit decompression.

In Chapter 6, we present fast linear time LZ77 factorization algorithms which avoid the computation of the whole LPF array. We show that our approach is very effective compared with the previous approach that firstly computes LPF array.

In Chapter 7, we describe new space efficient linear time LZ77 factorization algorithms, which are the most space efficient among all existing linear time algorithms when the alphabet size is small.

In Chapter 8, we present the conclusion of the thesis, and give future perspectives.

Chapter 2

Preliminaries

2.1 Intervals and Strings

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. For any integer $q > 0$, an element of Σ^q is called an q-*gram*. The length of a string T is denoted by $|T|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $T = XYZ$, X, Y and Z are called a *prefix*, *substring*, and *suffix* of T, respectively. The i-th character of a string T is denoted by $T[i]$ for $1 \leq i \leq |T|$, and the substring of a string T that begins at position i and ends at position j is denoted by $T[i : j]$ for $1 \leq i \leq j \leq |T|$. For convenience, let $T[i : j] = \varepsilon$ if $j < i$. Let T^R denote the reversal of T, namely, $T^R = T[N]T[N-1] \cdots T[1]$, where $N = |T|$. For a string T and $q \geq 0$, let $pre(T, q)$ and $suf(T, q)$ represent respectively, the length-q prefix and suffix of T. That is, $pre(T, q) = T[1 : \min(q, N)]$ and $suf(T, q) = T[\max(1, N - q + 1) : N]$.

For integers $i \leq j$, let $[i : j]$ denote the interval of integers $\{i, \ldots, j\}$. For an interval $[i : j]$ and integer $q > 0$, let $pre([i : j], q)$ and $suf([i : j], q)$ represent respectively, the length-q prefix and suffix interval, that is, $pre([i : j], q) = [i : \min(i + q - 1, j)]$ and $suf([i : j], q) = [\max(i, j - q + 1) : j]$. The substrings of T is also denoted by the combination of T and interval. For a string T and interval $[i : j](1 \leq i \leq j \leq N)$, $T([i : j])$ denote $T[i : j]$, and $T([i : j]) = T[i : j] = \varepsilon$ if $i < j$.

For an integer i and a set of integers A, let $i \oplus A = \{i + x \mid x \in A\}$ and $i \ominus A = \{i - x \mid x \in A\}$. If $A = \emptyset$, then let $i \oplus A = i \ominus A = \emptyset$. Similarly, for a pair of integers (x, y), let $i \oplus (x, y) = (i + x, i + y)$.

For the computation model, we use the word RAM model with word-length $\Theta(\log N)$, where any arithmetic operation for a number represented by $\Theta(\log N)$ bits, and read and write of the number to memory are achieved in constant time. For convenience, we omit $O(\log N)$ terms when describing space complexities in bits, i.e. ignore a constant number of integers.

2.2 Occurrences and Frequencies

For any strings T and P, let $Occ(T, P)$ be the set of occurrences of P in T, i.e.,

$$Occ(T, P) = \{k > 0 \mid T[k : k + |P| - 1] = P\}.$$

The number of occurrences of P in T, or the *frequency* of P in T is, $|Occ(T, P)|$. Any two occurrences $k_1, k_2 \in Occ(T, P)$ with $k_1 < k_2$ are said to be *overlapping* if $k_1 + |P| - 1 \geq k_2$. Otherwise, they are said to be *non-overlapping*. The *non-overlapping frequency* $nOcc(T, P)$ of P in T is defined as the size of a largest subset of $Occ(T, P)$ where any two occurrences in the set are non-overlapping. For any strings X, Y, we say that an occurrence i of a string Z in XY, with $|Z| \geq 2$, *crosses* X and Y, if $i \in [|X| - |Z| + 2 : |X|] \cap Occ(XY, Z)$.

For any strings T and P, we define the sets of *right and left priority non-overlapping occurrences* of P in T, respectively, as follows:

$$\begin{aligned} RnOcc(T, P) &= \begin{cases} \emptyset & \text{if } Occ(T, P) = \emptyset, \\ \{i\} \cup RnOcc(T[1 : i - 1], P) & \text{otherwise,} \end{cases} \\ LnOcc(T, P) &= \begin{cases} \emptyset & \text{if } Occ(T, P) = \emptyset, \\ \{j\} \cup j + |P| - 1 \oplus LnOcc(T[j + |P| : |T|], P) & \text{otherwise,} \end{cases} \end{aligned}$$

where $i = \max Occ(T, P)$ and $j = \min Occ(T, P)$. For all $k \in RnOcc(T, P)$, it is trivially said that $RnOcc(T[k : |T|], P) \subseteq RnOcc(T, P)$. It can be said to $LnOcc$ similarly. Note that $RnOcc(T, P) \subseteq Occ(T, P)$, $LnOcc(T, P) \subseteq Occ(T, P)$, and $LnOcc(T, P) = |T| - |P| + 2 \ominus RnOcc(T^R, P^R)$.

Lemma 1. $nOcc(T, P) = |RnOcc(T, P)| = |LnOcc(T, P)|$

Proof. We prove $nOcc(T[1 : i], P) = |LnOcc(T[1 : i], P)|$ by induction on i. For $i \leq 1$, the statement clearly holds. Now, assume that the statement holds for $i < k$, where $k \geq 2$. For $i = k$, notice that $0 \leq nOcc(T[1 : k], P) - |LnOcc(T[1 : k], P) \leq 1$, since there can be at most one new occurrence of P ending at position i, which may or may not be counted for $nOcc(T[1 : k], P)$. If we assume on the contrary that the statement does not hold for $i = k$, then $nOcc(T[1 : k], P) - nOcc(T[1 : k - 1], P) = nOcc(T[1 : k], P) - |LnOcc(T[1 : k], P)| = 1$. Since the change was caused by the new occurrence, we have $nOcc(T[1 : k]) = nOcc(T[1 : k - |P|]) + 1$. By the inductive hypothesis, we have $nOcc(T[1 : k - |P|], P) = |LnOcc(T[1 : k - |P|], P)|$. Also, $|LnOcc(T[1 : k], P)| = |LnOcc(T[1 : k - |P|], P)| + 1$, since the new occurrence does not overlap with any occurrences in $LnOcc(T[1 : k - |P|])$. This leads to $nOcc(T[1 : k]) = |LnOcc(T[1 : k], P)|$, a contradiction. $nOcc(T, P) = |RnOcc(T, P)|$ can be shown symmetrically. $\square$

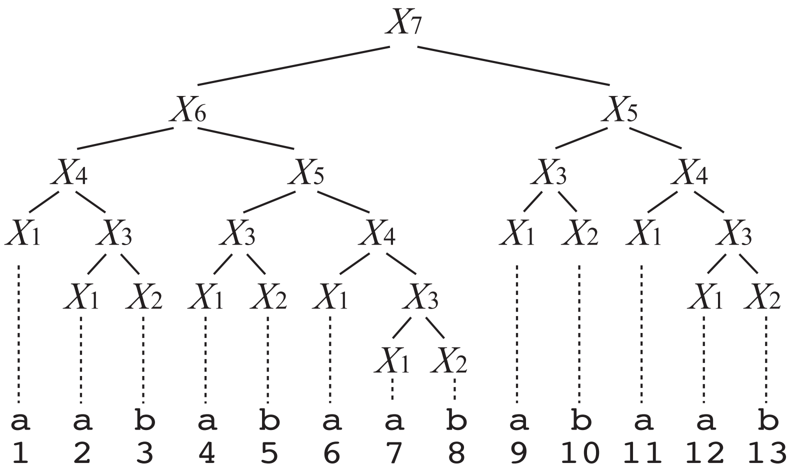


Figure 2.1: The derivation tree of SLP $\mathcal{T} = \{X_1 \to \mathtt{a},\ X_2 \to \mathtt{b},\ X_3 \to X_1X_2,\ X_4 \to X_1X_3,\ X_5 \to X_3X_4,\ X_6 \to X_4X_5,\ X_7 \to X_6X_5\}$, representing string $T = val(X_7) =$ `aababaababaab`.

Lemma 2. *For any strings T and P, and any integer i with $1 \leq i \leq |T|$, let $u_1 = \max LnOcc(T[1 : i-1], P) + |P| - 1$ and $u_2 = i - 1 + \min RnOcc(T[i : |T|], P)$. Then $nOcc(T, P) = |LnOcc(T[1 : u_1], P)| + nOcc(T[u_1 + 1 : u_2 - 1], P) + |RnOcc(T[u_2 : |T|], P)|$.*

Proof. By Lemma 1 and the definitions of u_1, u_2, $LnOcc$ and $RnOcc$, we have

$$\begin{aligned}
&nOcc(T, P) \\
&= |LnOcc(T[1 : u_1], P)| + |LnOcc(T[u_1 + 1 : |T|], P)| \\
&= |LnOcc(T[1 : u_1], P)| + |RnOcc(T[u_1 + 1 : |T|], P)| \\
&= |LnOcc(T[1 : u_1], P)| + |RnOcc(T[u_1+1 : u_2-1], P)| + |RnOcc(T[u_2 : |T|], P)| \\
&= |LnOcc(T[1 : u_1], P)| + nOcc(T[u_1+1 : u_2 - 1], P) + |RnOcc(T[u_2 : |T|], P)|.
\end{aligned}$$

□

2.3 Straight Line Programs

A *straight line program* (*SLP*) is a set of assignments $\mathcal{T} = \{X_1 \to expr_1, X_2 \to expr_2, \ldots, X_n \to expr_n\}$, where each X_i is a variable and each $expr_i$ is an expression, where $expr_i = a$ ($a \in \Sigma$), or $expr_i = X_{\ell(i)}X_{r(i)}$ ($i > \ell(i), r(i)$). It is essentially a context free grammar in the Chomsky normal form, that derives a single string. Let $val(X_i)$ represent the string derived from variable X_i. To ease notation, we sometimes associate $val(X_i)$ with X_i and denote $|val(X_i)|$ as $|X_i|$, and $val(X_i)([u : v])$ as $X_i([u : v])$ for any interval $[u : v]$. $pre(X_i, q)$ and $suf(X_i, q)$ respectively denotes the length-q prefix and suffix of $val(X_i)$. An SLP $\mathcal{T}$ *represents* the string $T = val(X_n)$.

The *size* of the program $\mathcal{T}$ is the number n of assignments in $\mathcal{T}$. Note that N can be as large as $\Theta(2^n)$. However, we assume as in various previous work on SLP, that the computer word size is at least $\log N$, and hence, values representing lengths and positions of T in our algorithms can be manipulated in constant time.

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 $val(X_{\langle v \rangle})$, a substring of T, whose corresponding interval $itv(v)$, with $T(itv(v)) = val(X_{\langle v \rangle})$, can be defined recursively as follows. If v is the root node, then $itv(v) = [1 : N]$. Otherwise, if $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle} X_{\langle r(v) \rangle}) \in \mathcal{T}$, then, $itv(\ell(v)) = [b_v : b_v + |X_{\langle \ell(v) \rangle}| - 1]$ and $itv(r(v)) = [b_v + |X_{\langle \ell(v) \rangle}| : e_v]$, where $[b_v : e_v] = itv(v)$. Let $vOcc(X_i)$ denote the number of times a variable X_i occurs in the derivation tree, i.e., $vOcc(X_i) = |\{v \mid X_{\langle v \rangle} = X_i\}|$. We assume that any variable X_i is used at least once, that is $vOcc(X_i) > 0$.

For any interval $[b : e]$ of $T (1 \leq b \leq e \leq N)$, let $\xi_{\mathcal{T}}(b, e)$ denote the deepest node v in the derivation tree, which derives an interval containing $[b : e]$, that is, $itv(v) \supseteq [b : e]$, 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. If $b = e$, we have that $(X_{\langle v \rangle} \to c) \in \mathcal{T}$ for some $c \in \Sigma$, and $itv(v) = b = e$. If $b < e$, then we have $(X_{\langle v \rangle} \to X_{\langle \ell(v) \rangle} X_{\langle r(v) \rangle}) \in \mathcal{T}$, $b \in itv(\ell(v))$, and $e \in itv(r(v))$. 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 $vOcc(X_i)$ 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. Also, the following Lemma is useful for partial decompression of a prefix of a variable.

Lemma 3 ([22]). *Given an SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^n$, it is possible to pre-process $\mathcal{T}$ in $O(n)$ time and space, so that for any variable X_i and $1 \leq j \leq |X_i|$, $X_i([1 : j])$ can be computed in $O(j)$ time.*

2.4 Suffix Arrays and LCP Arrays

The suffix array [50] SA of any string T is an array of length N such that for any $1 \leq i \leq N$, $SA[i] = j$ indicates that $suf(j)$ is the i-th lexicographically smallest suffix of T. For convenience, we assume that $SA[0] = SA[N + 1] = 0$. The inverse suffix array SA^{-1} of SA is an array of length N such that $SA^{-1}[SA[i]] = i$. As in [37], let Φ be an array of length N such

Table 2.1: Suffix array and LCP array for string T=abracadabra

i	$T[i:N]$	$SA[i]$	$LCP[i]$	$T[SA[i]:N]$
1	abracadabra	11	0	a
2	bracadabra	8	1	abra
3	racadabra	1	4	abracadabra
4	acadabra	4	1	acadabra
5	cadabra	6	1	adabra
6	adabra	9	0	bra
7	dabra	2	3	bracadabra
8	abra	5	0	cadabra
9	bra	7	0	dabra
10	ra	10	0	ra
11	a	3	2	racadabra

that $\Phi[SA[1]] = N$ and $\Phi[SA[i]] = SA[i-1]$ for $2 \leq i \leq N$, i.e., for any suffix $j = SA[i]$, $\Phi[j] = SA[i-1]$ is the immediately preceding suffix in the suffix array. The suffix array SA for any string of length N can be constructed in $O(N)$ time regardless of the alphabet size, assuming an integer alphabet (e.g. [38, 61]). Furthermore, there exists a linear time suffix array construction algorithm for a constant alphabet using $O(1)$ working space [60].

Although our algorithms will not utilize the following array, we shall introduce it for completeness. The LCP array is an array of length N such that $LCP[i]$ is the length of the longest common prefix of $T[SA[i-1]:N]$ and $T[SA[i]:N]$ for $2 \leq i \leq N$, and $LCP[1] = 0$. Given the text and suffix array, the LCP array can also be calculated in $O(N)$ time [40]. (See Table 2.4 shows suffix array and lcp array for string $T = abracadabra$.)

2.5 LZ77 Factorization

LZ77 factorization is dynamic dictionary based encodings with many variants. The variant we consider is also known as the s-factorization [14].

Definition 1 (LZ77-factorization)**.** *The s-factorization of a string T is the factorization $T = f_1 \cdots f_n$ where each s-factor $f_k \in \Sigma^+$ $(k = 1, \ldots, n)$ starting at position $i = |f_1 \cdots f_{k-1}| + 1$ in T is defined as follows: If $T[i] = c \in \Sigma$ does not occur before i then $f_k = c$. Otherwise, f_k is the longest prefix of $suf(i)$ that occurs at least once before i.*

Note that each LZ77 factor can be represented in constant space, i.e., a pair of integers where the first and second elements respectively represent the length and position of a previous occurrence of the factor. If the factor is a new character and the length of its previous occurrence is 0, the second element will encode the new character instead of the position. For example the s-factorization of the string T = abaababababaaaaabbabab is a, b, a, aba, baba, aaaa, b, babab. This can be represented as $(0, \mathtt{a})$, $(0, \mathtt{b})$, $(1, 1)$, $(3, 1)$, $(4, 5)$, $(4, 10)$, $(1, 2)$, $(5, 5)$.

We define two functions LPF and $PrevOcc$ below. For any $1 \leq i \leq N$, $LPF(i)$ is the longest length of longest common prefix between $suf(i)$ and $suf(j)$ for any $1 \leq j < i$, and $PrevOcc(i)$ is a position j which gives $LPF(i)$[1]. More precisely,

$$LPF(i) = \max(\{0\} \cup \{lcp(suf(i), suf(j)) \mid 1 \leq j < i\})$$

and

$$PrevOcc(i) = \begin{cases} -1 & \text{if } LPF(i) = 0 \\ j & \text{otherwise} \end{cases}$$

where j satisfies $1 \leq j < i$, and $T[i : i + LPF(i) - 1] = T[j : j + LPF(i) - 1]$. Let $p_k = |f_1 \cdots f_{k-1}| + 1$. Then, f_k can be represented as a pair $(LPF(p_k), PrevOcc(p_k))$ if $LPF(p_k) > 0$, and $(0, T[p_k])$ otherwise.

[1]There can be multiple choices of j, but here, it suffices to fix one.

Chapter 3

Algorithm for q-gram Frequencies

Toward compressed string mining, in this chapter we focus on the q-gram frequencies problem. The q-gram frequencies problem is an important fundamental problem which appears in machine learning [3] and data mining [6]. Our interest is how to compute frequencies of all q-grams that occur in T when given an SLP $\mathcal{T}$ representing a string T. The definition of the problem is as follows.

Problem 1 (q-gram frequencies on SLP)**.** *Given an integer $q \geq 1$ and an SLP $\mathcal{T}$ of size n that represents string T, output $(i, |Occ(T, P)|)$ for all $P \in \Sigma^q$ where $Occ(T, P) \neq \emptyset$, and some $i \in Occ(T, P)$.*

When $q = 1$, the problem is very simple because we only have to compute how many terminal variable is used in the derivation tree of X_n, which is namely $vOcc(X_i)$ for $X_i = a$ and $a \in \Sigma$. The computation can be done in $O(n)$ time as shown in Section 2.3.

When $q = 2$, the subproblem of finding the most frequent 2-gram from an SLP was previously considered by Inenaga and Bannai [33], and they proposed an $O(|\Sigma|^2 n^2)$-time $O(n^2)$-space algorithm. Claude and Navarro mentioned that the most frequent 2-gram can be found in $O(|\Sigma|^2 n \log n)$ time and $O(n \log N)$ space [13], if the SLP is pre-processed and a self-index is built.

It is possible to extend these two algorithms to handle q-grams for $q > 2$, but would respectively require $O(|\Sigma|^q qn^2)$ and $O(|\Sigma|^q qn \log n)$ time, since they essentially enumerate and count the occurrences of all substrings of length q, regardless of whether the q-gram occurs in the string. Both the algorithms are not practical when we want to apply them for larger values of q since they need time and space exponential in q.

In this chapter we propose an $O(qn)$ algorithm to compute q-gram frequencies in T when an SLP $\mathcal{T}$ of size n representing T is given. The key point of our algorithm to reduce the q-gram frequencies problem from SLPs to the weighted q-gram frequencies problem from uncompressed strings of size $O(qn)$ by partially decompressing the SLPs in $O(qn)$ time. The

Algorithm 1: A naïve algorithm for computing q-gram frequencies.

Input: string T, integer $q \geq 1$
Report: $(P, |Occ(T,P)|)$ for all $P \in \Sigma^q$ where $Occ(T,P) \neq \emptyset$.
$\mathbf{S} \leftarrow \emptyset$; `// empty associative array`
for $i \leftarrow 1$ **to** $N - q + 1$ **do**
  $qgram \leftarrow T[i : i + q - 1]$;
  if $qgram \in \text{keys}(\mathbf{S})$ **then** $\mathbf{S}[qgram] \leftarrow \mathbf{S}[qgram] + 1$;
  else $\mathbf{S}[qgram] \leftarrow 1$; `// new q-gram`
for $qgram \in \text{keys}(\mathbf{S})$ **do Report** $(qgram, \mathbf{S}[qgram])$

weighted q-gram frequencies problem from uncompressed strings can be solved in linear time with a slight modification of a linear time algorithm which solves the normal q-gram frequencies problem on uncompressed strings. According to the computational experiments, the $O(qn)$ algorithm tends to be faster than the linear time algorithm on uncompressed strings when q is small, more precisely, when the total length of partially decompressed strings is shorter than T. Our new algorithm is theoretically superior to the previous ones, moreover it runs faster in practice than the algorithm on uncompressed strings, thus achieving **Goal 2**.

This result primarily appeared in [25, 28].

3.1 $O(N)$ time Algorithm on Uncompressed Strings

We describe two algorithms (Algorithm 1 and Algorithm 2) for computing the q-gram frequencies of a given uncompressed string T.

A naïve algorithm for computing the q-gram frequencies is given in Algorithm 1. The algorithm constructs an associative array, where keys consist of q-grams, and the values correspond to the occurrence frequencies of the q-grams. The time complexity depends on the implementation of the associative array, but requires at least $O(qN)$ time since each q-gram is considered explicitly, and the associative array is accessed $O(N)$ times: e.g. $O(qN \log|\Sigma|)$ time and $O(qN)$ space using a simple trie.

The q-gram frequencies of string T can be calculated in $O(N)$ time using suffix array SA and lcp array LCP, as shown in Algorithm 2. For each $1 \leq i \leq N$, the suffix $SA[i]$ represents an occurrence of q-gram $T[SA[i] : SA[i]+q-1]$, if the suffix is long enough, i.e. $SA[i] \leq N-q+1$. The key is that since the suffixes are lexicographically sorted, intervals on the suffix array where the values in the lcp array are at least q represent occurrences of the same q-gram. The algorithm runs in $O(N)$ time, since SA and LCP can be constructed in $O(N)$ time. The rest is a simple $O(N)$ loop. A technicality is that we encode the output for a q-gram as one of the positions in the text where the q-gram occurs, rather than the q-gram itself. This is because there can be a total of $O(N)$ different q-grams, and if we output them as length-q strings, it would require at

Algorithm 2: A linear time algorithm for computing q-gram frequencies.

Input: string T, integer $q \geq 1$
Report: $(i, |Occ(T, P)|)$ for all $P \in \Sigma^q$ and some position $i \in Occ(T, P)$.
$SA \leftarrow SUFFIXARRAY(T)$; $LCP \leftarrow LCPARRAY(T, SA)$; $count \leftarrow 1$;
for $i \leftarrow 2$ **to** $N + 1$ **do**
  if $i = N + 1$ **or** $LCP[i] < q$ **then** `// end of interval where lcp` $\geq q$
    if $count > 0$ **then**
      Report $(SA[i - 1], count)$;
      $count \leftarrow 0$;
  if $i \leq N$ **and** $SA[i] \leq N - q + 1$ **then** `// count current suffix if valid`
    $count \leftarrow count + 1$;

least $O(qN)$ time.

3.2 $O(qn)$ time Algorithm on SLPs

We now describe the core idea of our algorithms, and explain two variations which utilize variants of the two algorithms for uncompressed strings presented in Section 3.1. For $q = 1$, the 1-gram frequencies are simply the frequencies of the alphabet, and the output is $(a, \sum\{vOcc(X_i) \mid X_i = a\})$ for each $a \in \Sigma$, which takes only $O(n)$ time. For $q \geq 2$, we make use of Lemma 4 below. The idea is similar to the *mk Lemma* [11], but the statement is more specific.

Lemma 4. *Let $\mathcal{T} = \{X_i = expr_i\}_{i=1}^n$ be an SLP that represents string T. For an interval $[u : v]$ $(1 \leq u < v \leq N)$, there exists exactly one variable $X_i = X_{\ell(i)}X_{r(i)}$ such that for some $[u_i : v_i] \in itv(X_i)$, the following holds: $[u : v] \subseteq [u_i : v_i]$, $u \in [u_i : u_i + |X_{\ell(i)}| - 1] \in itv(X_{\ell(i)})$ and $v \in [u_i + |X_{\ell(i)}| : v_i] \in itv(X_{r(i)})$.*

Proof. Any interval is a subinterval of the interval $[1 : N]$ derived by X_n. For a given variable, if the interval $[u : v]$ is a subinterval of the interval derived by either of its children, we recursively consider the child variable. Each time, the interval derived by the variable is divided into two parts and becomes smaller. Hence, a variable $X_i = X_{\ell(i)}X_{r(i)}$ satisfying the condition will eventually be obtained. Any other variable $X_{i'} = X_{\ell(i')}X_{r(i')}$ cannot satisfy the condition, since if the interval derived by $X_{i'}$ is to contain the given interval, it must be a descendant or an ancestor of X_i. Either way, this contradicts the condition that the given interval is not a subinterval of any of the intervals derived from the children variables $X_{\ell(i')}$, $X_{r(i')}$, $X_{\ell(i)}$, $X_{r(i)}$.

Note that if we consider length 1 intervals $[u : u]$ and $[v : v]$ corresponding to leaves in the derivation tree, X_i corresponds to the lowest common ancestor of these intervals in the derivation tree.

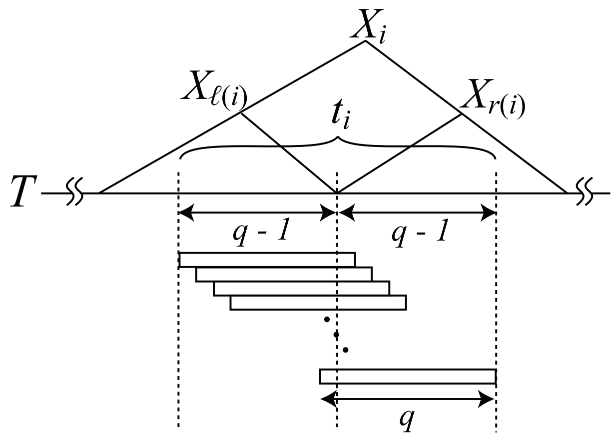


Figure 3.1: Length-q intervals which are stabbed by $X_i = X_{\ell(i)}X_{r(i)}$.

From Lemma 4, each occurrence of a q-gram ($q \geq 2$) represented by some length-q interval of T, corresponds to a single variable $X_i = X_{\ell(i)}X_{r(i)}$, and is split in two by intervals corresponding to $X_{\ell(i)}$ and $X_{r(i)}$. On the other hand, consider all length-q intervals that correspond to a given variable. Counting the frequencies of the q-grams they represent, and summing them up for all variables give the frequencies of all q-grams of T.

For variable $X_i = X_{\ell(i)}X_{r(i)}$, let $t_i = \mathit{suf}(X_{\ell(i)}, q-1)\mathit{pre}(X_{r(i)}, q-1)$. Then, all q-grams represented by length q intervals that correspond to X_i are those in t_i. (Figure 3.1.) If we obtain the frequencies of all q-grams in t_i, and then multiply each frequency by $vOcc(X_i)$, we obtain frequencies for the q-grams occurring in all intervals derived by X_i. It remains to sum up the q-gram frequencies of t_i for all $1 \leq i \leq n$. We can regard it as obtaining the weighted q-gram frequencies in the set of strings $\{t_1, \ldots, t_n\}$, where each q-gram in t_i is weighted by $vOcc(X_i)$.

We further reduce this problem to a weighted q-gram frequencies problem for a single string z as in Algorithm 3. String z is constructed by concatenating each t_i satisfying $q \leq |t_i| \leq 2(q-1)$, and the weights of q-grams starting at each position in z is held in array w. On line 8, 0's instead of $vOcc(X_i)$ are appended to w for the last $q-1$ values corresponding to t_i. This is to avoid counting unwanted q-grams that are generated by the concatenation of t_i to z on line 6, which are not substrings of each t_i. The weighted q-gram frequency problem for a single string (Line 9) can be solved with a slight modification of Algorithm 1 or 2. The modified algorithms are shown respectively in Algorithms 4 and 5.

Theorem 1. *Given an SLP $\mathcal{T} = \{X_i = expr_i\}_{i=1}^{n}$ of size n representing a string T, the q-gram frequencies of T can be computed in $O(qn)$ time for any $q > 0$.*

Proof. Consider Algorithm 3. The correctness is straightforward from the above arguments, so we consider the time complexity. Line 1 can be computed in $O(n)$ time. Line 2 can be computed in $O(qn)$ time by a simple dynamic programming. For $pre()$: If $X_i = a$ for some

Algorithm 3: Calculating q-gram frequencies of an SLP for $q \geq 2$

Input: SLP $\mathcal{T} = \{X_i = expr_i\}_{i=1}^{n}$ representing string T, integer $q \geq 2$.
Report: all q-grams and their frequencies which occur in T.
1 Calculate $vOcc(X_i)$ for all $1 \leq i \leq n$;
2 Calculate $pre(X_i, q-1)$ and $suf(X_i, q-1)$ for all $1 \leq i \leq n-1$;
3 $z \leftarrow \varepsilon$; $w \leftarrow []$;
4 **for** $i \leftarrow 1$ **to** n **do**
5   **if** $X_i = X_{\ell(i)} X_{r(i)}$ **and** $|X_i| \geq q$ **then**
6         $t_i = suf(X_{\ell(i)}, q-1) pre(X_{r(i)}, q-1)$; z.append(t_i);
7   **for** $j \leftarrow 1$ **to** $|t_i| - q + 1$ **do** w.append($vOcc(X_i)$);
8   **for** $j \leftarrow 1$ **to** $q - 1$ **do** w.append(0);
9 **Report** q-gram frequencies in z, where each q-gram $z[i : i+q-1]$ is weighted by $w[i]$.

Algorithm 4: A variant of Algorithm 1 for weighted q-gram frequencies.

Input: string T, array of integers w of length N, integer $q \geq 1$
Report: $(P, \sum_{i \in Occ(T,P)} w[i])$ for all $P \in \Sigma^q$ where $\sum_{i \in Occ(T,P)} w[i] > 0$.
$\mathbf{S} \leftarrow \emptyset$; `// empty associative array`
for $i \leftarrow 1$ **to** $N - q + 1$ **do**
    $qgram \leftarrow T[i : i+q-1]$;
    if $qgram \in$ keys($\mathbf{S}$) **then** $\mathbf{S}[qgram] \leftarrow \mathbf{S}[qgram] + w[i]$;
    else if $w[i] > 0$ **then** $\mathbf{S}[qgram] \leftarrow w[i]$; `// new q-gram`
for $qgram \in$ keys($\mathbf{S}$) **do Report** $(qgram, \mathbf{S}[qgram])$

$a \in \Sigma$, then $pre(X_i, q-1) = a$. If $X_i = X_{\ell(i)} X_{r(i)}$ and $|X_{\ell(i)}| \geq q-1$, then $pre(X_i, q-1) = pre(X_{\ell(i)}, q-1)$. If $X_i = X_{\ell(i)} X_{r(i)}$ and $|X_{\ell(i)}| < q-1$, then $pre(X_i, q-1) = pre(X_{\ell(i)}, q-1) pre(X_{r(i)}, q-1-|X_{\ell(i)}|)$. The strings $suf()$ can be computed similarly. The computation amounts to copying $O(q)$ characters for each variable, and thus can be done in $O(qn)$ time. For the loop at line 4, since the length of string t_i appended to z, as well as the number of elements appended to w is at most $2(q-1)$ in each loop, the total time complexity is $O(qn)$. Finally, since the length of z and w is $O(qn)$, line 9 can be calculated in $O(qn)$ time using the weighted version of Algorithm 2 (Algorithm 5).

Note that the time complexity for using the weighted version of Algorithm 1 for line 9 of Algorithm 3 would be at least $O(q^2 n)$: e.g. $O(q^2 n \log |\Sigma|)$ time and $O(q^2 n)$ space using a trie.

Algorithm 5: A variant of Algorithm 2 for weighted q-gram frequencies.

Input: string T, array of integers w of length N, integer $q \geq 1$

Output: $(i, \sum_{i \in Occ(T,P)} w[i])$ for all $P \in \Sigma^q$ where $\sum_{i \in Occ(T,P)} w[i] > 0$ and some position $i \in Occ(T, P)$.

```
SA ← SUFFIXARRAY(T); LCP ← LCPARRAY(T, SA); count ← 1;
for i ← 2 to N + 1 do
    if i = N + 1 or LCP[i] < q then      // end of interval where lcp ≥ q
        if count > 0 then
            Report (SA[i − 1], count);
            count ← 0;
    if i ≤ N and SA[i] ≤ N − q + 1 then      // count current suffix if valid
        count ← count + w[SA[i]];
```

3.3 Computational Experiments

We implemented 4 algorithms (NMP, NSA, SMP, SSA) that count the frequencies of all q-grams in a given text. NMP (Algorithm 1) and NSA (Algorithm 2) work on the uncompressed text. SMP (Algorithm 3 + Algorithm 4) and SSA (Algorithm 3 + Algorithm 5) work on SLPs. The algorithms were implemented using the C++ language, and source codes are available at `http://code.google.com/p/qshi/`. We used `std::map` from the Standard Template Library (STL) for the associative array implementation. For constructing suffix arrays, we used the divsufsort library version 2.0.0[1] developed by Yuta Mori. This implementation is not linear time in the worst case, but has been empirically shown to be one of the fastest implementations on various data.

All computations were conducted on a Mac Pro (Mid 2010) with MacOS X Lion 10.7.2, and 2 x 2.93GHz 6-Core Xeon processors and 64GB Memory, only utilizing a single process/thread at once. The program was compiled using the GNU C++ compiler (`g++`) 4.6.2 with the `-Ofast` option for optimization. The running times are measured in seconds, starting from after reading the uncompressed text into memory for NMP and NSA, and after reading the SLP that represents the text into memory for SMP and SSA. Each computation is repeated at least 3 times, and the average is taken.

3.3.1 Fibonacci Strings

The i th Fibonacci string F_i can be represented by the following SLP: $X_1 = \mathtt{b}$, $X_2 = \mathtt{a}$, $X_i = X_{i-1}X_{i-2}$ for $i > 2$, and $F_i = val(X_i)$. Figure 3.2 (Up) shows the running times on Fibonacci strings $F_{20}, F_{25}, \ldots, F_{95}$, for $q = 50$. Although this is an extreme case since Fibonacci strings can be exponentially compressed, we can see that SMP and SSA that work on the SLP are clearly faster than NMP and NSA which work on the uncompressed string.

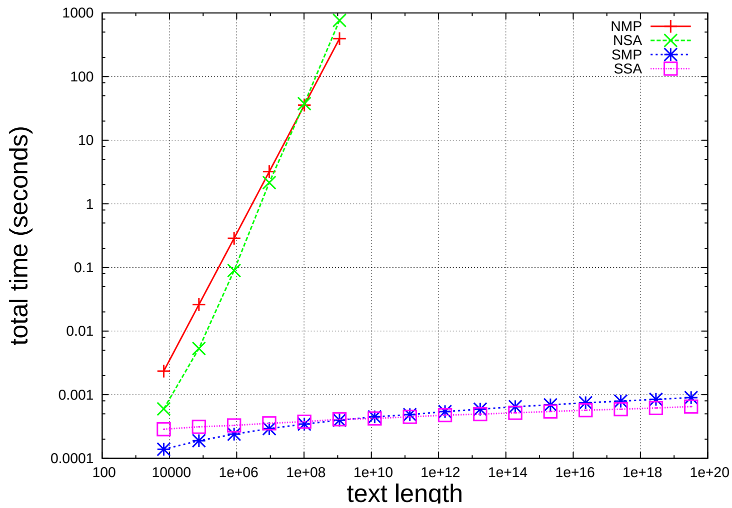
3.3.2 Pizza & Chili Corpus

We also applied the algorithms on texts XML, DNA, ENGLISH, and PROTEINS, with sizes 50MB, 100MB, and 200MB, obtained from the Pizza & Chili Corpus[2]. We used two variations of SLP data, which are generated by RE-PAIR [45] and LCA [52].

Table 3.1 shows the running times for all algorithms and data which are generated by RE-PAIR, where q is varied from 2 to 10. We see that for all corpora, SMP and SSA running on SLPs are actually faster than NMP and NSA running on uncompressed text, when q is small. Furthermore, SMP is faster than SSA when q is smaller. Interestingly for XML, the SLP

[1] `http://code.google.com/p/libdivsufsort/`

[2] `http://pizzachili.dcc.uchile.cl/texts.html`



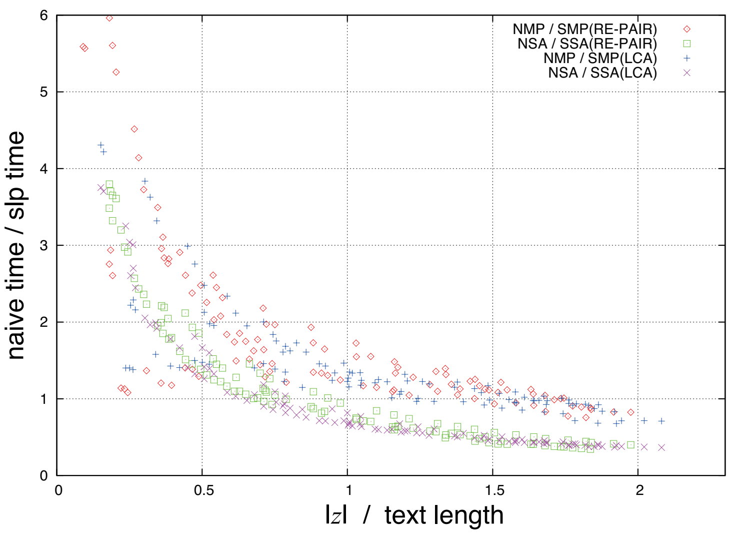


Figure 3.2: (Up) Running times of NMP, NSA, SMP, SSA on Fibonacci strings for $q = 50$. (Down) Time ratios NMP/SMP and NSA/SSA plotted against ratio $|z|/N$, for the Pizza & Chili Corpus.

versions are faster even for q up to 9. Table 3.2 shows the running times for the data which are generated by LCA. The SLPs which are generated by LCA consist of more variables than the SLPs which are generated by RE-PAIR. The size of string z which is generated by Algorithm 3 generally increases with respect to the size of the SLP, so the results for SLPs generated by RE-PAIR tend to have better performance compared to those for LCA.

Figure 3.2 (Down) shows the same results as time ratio: NMP/SMP and NSA/ SSA, plotted against ratio: (length of z in Algorithm 3)/(length of uncompressed text). As expected, the SLP versions are basically faster than their uncompressed counterparts, when $|z|/$(text length) is less than around 0.7. This is because the SLP versions run the weighted versions of the uncompressed algorithms on a text of length $|z|$, with some overhead for constructing z and for handling the weights. Results with SLPs generated by both RE-PAIR and LCA show similar tendencies.

Table 3.3 and 3.4 show the memory usage of the algorithms measured by the *getrusage*() function. We see that in terms of memory usage, NMP is the best when q is not too large. However, NMP is never the fastest choice. NSA can be more space efficient and faster than SMP or SSA when q is not so small. On the other hand, the memory usage of NSA is fairly large even when q is small, and SMP and SSA can both be faster and more space efficient in this case.

Table 3.1: Running times in seconds for the Pizza & Chili Corpus. Each data is compressed by RE-PAIR [45]. Bold numbers represent the fastest time for each data and q. Times for SMP and SSA are prefixed with ▷, if they become fastest when all algorithms start from the SLP representation, i.e., NMP and NSA require time for decompressing the SLP (denoted by decompression time). The bold horizontal lines show the boundary where $|z|$ in Algorithm 3 exceeds the uncompressed text length.

XML(RE-PAIR)															
	50MB					100MB					200MB				
	SLP Size: 2,702,383					SLP Size: 5,059,578					SLP Size: 9,541,590				
	decompression time: 0.85 secs					decompression time: 1.72 secs					decompression time: 3.66 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	5,404,574	5.6	8.9	**1.0**	1.4	10,118,964	11.3	19.2	**2.0**	3.1	19,082,988	22.9	41.7	**4.1**	6.5
3	10,713,906	12.5	8.9	**2.4**	2.5	20,103,632	26.4	19.3	**4.7**	5.3	37,966,315	55.7	41.7	**9.3**	11.0
4	15,680,270	19.7	8.9	5.3	**3.8**	29,544,225	42.8	19.3	10.3	**7.9**	55,983,397	93.3	41.8	20.7	**16.3**
5	20,223,744	26.7	8.9	9.5	**5.0**	38,287,472	58.1	19.3	18.7	**10.4**	72,878,965	129.3	41.7	37.0	**21.3**
6	24,428,612	32.7	9.0	13.8	**6.2**	46,436,350	71.8	19.3	27.5	**12.8**	88,786,480	158.7	41.7	54.6	**25.8**
7	28,354,144	36.9	8.9	18.2	**7.1**	54,094,679	81.9	19.4	36.3	**14.8**	103,862,589	181.1	41.7	73.1	**30.1**
8	32,052,358	41.0	8.9	23.6	**8.1**	61,340,059	90.2	19.4	49.1	**16.6**	118,214,023	198.3	41.9	95.5	**34.2**
9	35,525,151	45.6	9.0	28.0	**8.9**	68,175,926	98.4	19.4	56.3	**18.3**	131,868,777	218.9	41.6	118.2	**37.9**
10	38,838,107	48.8	**9.0**	33.5	▷9.7	74,690,539	107.2	**19.3**	65.4	▷19.9	144,946,389	235.7	41.8	133.1	**41.3**

DNA(RE-PAIR)															
	50MB					100MB					200MB				
	SLP Size: 6,406,324					SLP Size: 12,233,978					SLP Size: 23,171,463				
	decompression time: 1.15 secs					decompression time: 2.43 secs					decompression time: 5.02 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	12,812,616	2.0	12.5	**1.8**	4.3	24,467,924	4.1	27.5	**3.7**	9.2	46,342,894	8.6	61.7	**7.5**	19.3
3	25,624,572	3.8	12.4	**2.9**	6.7	48,935,122	7.9	27.3	**5.7**	14.1	92,684,656	16.4	61.7	**11.7**	29.1
4	38,427,472	6.0	12.5	**4.5**	9.9	73,391,054	12.7	27.4	**8.8**	20.5	139,011,475	26.4	61.4	**17.4**	42.2
5	51,148,851	8.3	12.4	**6.7**	13.1	97,743,073	17.4	27.3	**13.3**	27.0	185,200,662	36.2	61.4	**26.9**	56.3
6	63,566,979	10.8	12.4	**10.3**	16.8	121,657,437	22.1	27.4	**19.8**	34.7	230,769,162	46.2	61.7	**40.2**	73.1
7	75,366,779	15.0	**12.4**	15.0	20.7	144,600,769	30.6	**27.3**	▷29.1	42.8	274,845,524	62.3	61.4	**56.8**	90.9
8	86,058,072	20.5	**12.4**	22.4	24.7	165,661,494	41.0	**27.5**	43.3	51.9	315,811,932	83.4	**61.7**	89.3	110.3
9	95,468,332	28.9	**12.4**	38.3	27.9	184,445,080	57.3	**27.5**	73.9	59.5	352,780,338	116.1	**61.2**	139.2	127.3
10	103,563,590	49.5	**12.4**	60.0	30.9	200,915,121	98.5	**27.4**	119.0	66.3	385,636,192	199.8	**61.5**	231.8	143.3

ENGLISH(RE-PAIR)															
	50MB					100MB					200MB				
	SLP Size: 4,861,619					SLP Size: 10,063,953					SLP Size: 18,945,126				
	decompression time: 1.06 secs					decompression time: 2.31 secs					decompression time: 4.86 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	9,722,886	5.1	11.7	**1.7**	3.2	20,127,476	10.5	25.6	**4.0**	7.7	37,889,802	21.5	56.8	**7.8**	16.3
3	19,371,594	10.9	11.8	**3.8**	5.4	40,135,705	23.1	25.6	**8.4**	12.5	75,611,002	48.5	56.6	**16.4**	25.6
4	28,806,795	18.7	11.7	**7.6**	8.1	59,789,962	40.0	25.6	**17.2**	18.3	112,835,471	84.8	57.0	**32.5**	37.5
5	37,815,947	29.0	11.7	14.7	**10.8**	78,702,809	63.7	25.6	32.4	**24.4**	148,938,576	137.4	56.6	63.0	**49.8**
6	46,271,085	43.0	**11.8**	24.9	13.5	96,629,891	95.0	**25.6**	57.6	30.6	183,493,406	205.9	**56.4**	106.6	62.9
7	54,049,585	57.4	**11.8**	37.1	16.1	113,307,235	127.2	**25.8**	81.9	36.6	215,975,218	276.4	**56.7**	160.1	75.7
8	61,098,637	71.0	**11.7**	53.2	18.5	128,612,883	156.8	**25.8**	122.4	41.8	246,127,485	341.7	**56.6**	242.2	87.9
9	67,333,842	83.2	**11.9**	70.3	20.5	142,376,652	185.8	**25.9**	156.2	47.0	273,622,444	405.6	**57.3**	298.8	100.2
10	72,766,008	95.5	**11.9**	84.6	22.3	154,559,225	213.9	**25.9**	190.1	51.6	298,303,942	469.4	**57.4**	381.9	110.9

PROTEINS(RE-PAIR)															
	50MB					100MB					200MB				
	SLP Size: 10,357,053					SLP Size: 18,806,316					SLP Size: 32,375,988				
	decompression time: 1.53 secs					decompression time: 3.33 secs					decompression time: 6.70 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	20,714,056	4.2	12.9	**3.6**	7.2	37,612,582	8.4	28.7	**6.9**	14.4	64,751,926	16.8	60.6	**12.3**	27.1
3	41,393,022	9.0	12.9	**7.4**	13.0	75,190,116	18.0	28.8	**14.0**	25.9	129,449,835	36.2	60.8	**24.2**	47.5
4	60,589,652	20.5	**12.9**	18.8	20.4	110,572,865	40.8	**28.7**	34.8	40.2	191,045,216	82.2	**60.8**	▷61.6	74.9
5	76,267,233	59.7	**12.9**	51.0	26.9	140,409,835	123.1	**28.7**	93.6	54.2	243,692,809	241.5	**60.6**	162.9	101.6
6	85,957,716	104.7	**13.1**	98.6	32.1	160,241,692	223.0	**28.9**	183.6	65.2	280,408,504	444.4	**61.0**	318.8	123.7
7	90,917,270	128.3	**13.0**	128.2	34.6	171,093,875	287.0	**29.1**	255.9	71.2	301,810,933	593.4	**61.0**	473.6	136.1
8	93,077,387	133.0	**13.0**	146.8	35.9	176,147,947	301.6	**28.9**	288.8	74.1	311,863,817	627.1	**60.9**	562.6	142.3
9	94,652,133	134.7	**13.0**	150.9	36.8	179,504,647	307.0	**28.9**	310.7	76.1	318,432,611	637.1	**61.2**	587.9	148.1
10	96,283,667	135.8	**12.9**	153.6	37.5	182,971,091	309.3	**28.8**	306.6	77.3	325,028,658	649.6	**61.2**	582.0	149.7

Table 3.2: Running times in seconds for the Pizza & Chili Corpus. Each data is compressed by LCA [52]. Bold numbers represent the fastest time for each data and q. Times for SMP and SSA are prefixed with $\triangleright$, if they become fastest when all algorithms start from the SLP representation, i.e., NMP and NSA require time for decompressing the SLP (denoted by decompression time). The bold horizontal lines show the boundary where $|z|$ in Algorithm 3 exceeds the uncompressed text length.

XML(LCA)															
	50MB					100MB					200MB				
	SLP Size: 4,523,711					SLP Size: 8,434,909					SLP Size: 15,924,230				
	decompression time: 0.67 secs					decompression time: 1.40 secs					decompression time: 2.95 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	9,046,908	5.6	8.9	**1.4**	2.4	16,869,304	11.3	19.2	**2.7**	5.2	31,847,946	22.9	41.7	**5.3**	11.1
3	18,049,781	12.5	8.9	**3.8**	4.7	33,684,488	26.4	19.3	**7.3**	9.8	63,630,712	55.7	41.7	**14.5**	20.3
4	26,600,275	19.7	8.9	7.9	**7.1**	49,821,349	42.8	19.3	15.5	**14.5**	94,397,662	93.3	41.8	31.2	**29.7**
5	34,296,630	26.7	**8.9**	13.7	$\triangleright$9.1	64,573,151	58.1	19.3	27.5	**18.6**	122,946,997	129.3	41.7	55.3	**38.5**
6	41,267,760	32.7	**9.0**	19.4	11.0	78,035,445	71.8	**19.3**	39.1	22.4	149,289,229	158.7	**41.7**	79.3	46.1
7	47,364,107	36.9	**8.9**	25.3	12.5	89,974,719	81.9	**19.4**	50.9	25.5	172,985,811	181.1	**41.7**	104.8	53.0
8	52,566,768	41.0	**8.9**	32.3	13.6	100,305,951	90.2	**19.4**	67.5	28.0	193,871,501	198.3	**41.9**	139.0	59.0
9	57,416,357	45.6	**9.0**	37.7	14.8	109,917,599	98.4	**19.4**	79.4	30.4	213,294,106	218.9	**41.6**	164.0	64.5
10	62,113,559	48.8	**9.0**	44.5	15.9	119,213,755	107.2	**19.3**	87.8	32.6	232,110,590	235.7	**41.8**	188.5	69.8

DNA(LCA)															
	50MB					100MB					200MB				
	SLP Size: 6,875,540					SLP Size: 13,130,252					SLP Size: 24,875,272				
	decompression time: 0.72 secs					decompression time: 1.51 secs					decompression time: 3.18 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	13,750,566	2.0	12.5	**1.4**	4.1	26,259,990	4.1	27.5	**2.9**	9.0	49,750,030	8.6	61.7	**6.1**	19.0
3	27,499,612	3.8	12.4	**2.6**	7.8	52,517,989	7.9	27.3	**5.4**	16.4	99,497,221	16.4	61.7	**10.9**	34.0
4	41,233,447	6.0	12.5	**4.5**	12.0	78,757,910	12.7	27.4	**9.2**	25.1	149,221,813	26.4	61.4	**18.1**	51.9
5	54,846,345	8.3	12.4	**7.1**	16.1	104,837,527	17.4	27.3	**14.2**	33.5	198,734,240	36.2	61.4	**29.2**	70.8
6	68,075,224	**10.8**	12.4	$\triangleright$11.1	20.5	130,354,024	22.1	27.4	**22.0**	43.2	247,434,478	46.2	61.7	**43.6**	91.5
7	80,317,216	15.0	**12.4**	17.1	24.7	154,216,217	30.6	**27.3**	33.3	52.3	293,437,541	62.3	**61.4**	64.7	111.9
8	91,336,539	20.5	**12.4**	25.4	28.3	175,953,553	41.0	**27.5**	49.5	60.5	335,812,032	83.4	**61.7**	97.7	131.3
9	100,964,579	28.9	**12.4**	42.7	31.6	195,238,735	57.3	**27.5**	83.9	67.7	373,922,825	116.1	**61.2**	162.4	147.9
10	109,112,377	49.5	**12.4**	69.8	33.9	211,943,374	98.5	**27.4**	137.8	73.8	407,523,732	199.8	**61.5**	280.0	162.6

ENGLISH(LCA)															
	50MB					100MB					200MB				
	SLP Size: 6,900,943					SLP Size: 14,188,706					SLP Size: 26,622,149				
	decompression time: 0.82 secs					decompression time: 1.75 secs					decompression time: 3.74 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	13,801,372	5.1	11.7	**2.2**	4.3	28,376,898	10.5	25.6	**4.9**	10.5	53,243,784	21.5	56.8	**9.7**	21.8
3	27,567,320	10.9	11.8	**5.5**	8.5	56,699,575	23.1	25.6	**11.8**	19.4	106,423,066	48.5	56.6	**22.8**	39.9
4	40,814,697	18.7	11.7	**11.6**	12.9	84,173,516	40.0	25.6	**24.6**	29.2	158,417,667	84.8	57.0	**48.3**	59.6
5	52,814,284	29.0	**11.7**	21.5	16.8	109,482,188	63.7	**25.6**	47.5	38.1	207,133,095	137.4	**56.6**	93.8	79.3
6	63,255,060	43.0	**11.8**	34.9	20.2	131,943,421	95.0	**25.6**	77.6	46.1	251,209,285	205.9	**56.4**	156.1	97.2
7	71,756,474	57.4	**11.8**	49.9	22.8	150,733,960	127.2	**25.8**	112.3	52.5	289,041,443	276.4	**56.7**	227.0	112.6
8	78,061,763	71.0	**11.7**	72.2	24.6	165,238,663	156.8	**25.8**	164.1	57.3	319,411,332	341.7	**56.6**	319.8	124.4
9	82,902,871	83.2	**11.9**	88.0	26.0	176,608,232	185.8	**25.9**	201.0	60.7	343,753,791	405.6	**57.3**	414.5	132.4
10	87,002,279	95.5	**11.9**	97.6	27.2	186,161,731	213.9	**25.9**	233.4	63.3	364,216,092	469.4	**57.4**	465.4	139.1

PROTEINS(LCA)															
	50MB					100MB					200MB				
	SLP Size: 11,080,596					SLP Size: 20,523,326					SLP Size: 35,664,074				
	decompression time: 0.84 secs					decompression time: 1.84 secs					decompression time: 3.77 secs				
q	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA	$\lvert z\rvert$	NMP	NSA	SMP	SSA
2	22,160,678	4.2	12.9	**3.0**	7.8	41,046,138	8.4	28.7	**5.9**	16.1	71,327,634	16.8	60.6	**10.7**	30.4
3	44,308,566	9.0	12.9	**7.8**	15.0	82,078,185	18.0	28.8	**14.7**	30.8	142,638,768	36.2	60.8	**25.8**	57.7
4	64,915,672	20.5	**12.9**	21.2	23.2	120,888,577	40.8	**28.7**	40.2	47.7	210,819,082	82.2	**60.8**	71.1	90.0
5	81,572,382	59.7	**12.9**	60.4	29.2	153,169,620	123.1	**28.7**	114.0	60.3	268,633,454	241.5	**60.6**	202.1	115.6
6	92,353,450	104.7	**13.1**	111.6	33.4	175,766,089	223.0	**28.9**	215.7	69.6	311,180,656	444.4	**61.0**	379.7	133.7
7	95,984,524	128.3	**13.0**	140.8	34.7	184,840,592	287.0	**29.1**	291.3	73.2	330,941,705	593.4	**61.0**	544.7	142.1
8	96,967,563	133.0	**13.0**	152.5	34.4	187,501,717	301.6	**28.9**	322.5	72.4	337,428,046	627.1	**60.9**	614.8	141.7
9	98,393,791	134.7	**13.0**	160.7	34.4	190,745,408	307.0	**28.9**	330.3	72.7	344,457,810	637.1	**61.2**	654.1	141.3
10	100,625,828	135.8	**12.9**	163.3	34.8	195,433,969	309.3	**28.8**	335.2	73.2	353,955,168	649.6	**61.2**	666.3	143.0

Table 3.3: Memory usage in Mega bytes for the computation of Most Frequent q-gram from the Pizza & Chili Corpus. Each data is compressed by RE-PAIR algorithm [45].

XML(RE-PAIR)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	156	244	228	1,428	296	460	382	2,728	566	874
3	118	714	166	332	233	1,428	309	626	371	2,728	582	1,187
4	135	714	204	422	258	1,428	365	794	374	2,728	666	1,503
5	170	714	271	506	312	1,428	472	953	454	2,728	835	1,806
6	217	714	362	586	391	1,428	624	1,105	586	2,728	1,088	2,097
7	269	714	457	658	482	1,428	790	1,244	745	2,728	1,378	2,368
8	322	714	622	726	576	1,428	1,078	1,377	911	2,728	1,875	2,626
9	375	714	745	799	672	1,428	1,299	1,517	1,082	2,728	2,269	2,899
10	430	714	862	860	771	1,428	1,512	1,637	1,259	2,728	2,647	3,135

DNA(RE-PAIR)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	357	565	228	1,428	688	1,085	377	2,728	1,277	2,029
3	114	714	357	773	228	1,428	688	1,482	370	2,728	1,277	2,780
4	114	714	358	981	228	1,428	689	1,879	378	2,728	1,278	3,532
5	114	714	360	1,190	228	1,428	692	2,277	380	2,728	1,283	4,285
6	114	714	374	1,404	228	1,428	711	2,683	379	2,728	1,310	5,050
7	115	714	398	1,618	229	1,428	748	3,091	380	2,728	1,366	5,820
8	117	714	445	1,834	231	1,428	825	3,505	377	2,728	1,492	6,604
9	126	714	534	2,058	241	1,428	971	3,937	371	2,728	1,733	7,426
10	162	714	653	2,237	277	1,428	1,136	4,296	378	2,728	1,980	8,133

ENGLISH(RE-PAIR)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	287	444	228	1,428	590	915	378	2,728	1,069	1,682
3	117	714	293	603	232	1,428	599	1,243	375	2,728	1,081	2,299
4	128	714	318	762	250	1,428	639	1,572	378	2,728	1,132	2,916
5	161	714	384	919	304	1,428	748	1,898	428	2,728	1,284	3,532
6	227	714	514	1,077	414	1,428	970	2,226	587	2,728	1,608	4,152
7	332	714	711	1,224	598	1,428	1,316	2,536	867	2,728	2,144	4,746
8	477	714	1,099	1,364	863	1,428	2,017	2,833	1,294	2,728	3,262	5,319
9	652	714	1,490	1,506	1,199	1,428	2,770	3,136	1,869	2,728	4,551	5,904
10	841	714	1,893	1,620	1,583	1,428	3,590	3,386	2,563	2,728	6,035	6,401

PROTEINS(RE-PAIR)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	602	938	228	1,428	1,061	1,671	376	2,728	1,734	2,784
3	115	714	604	1,275	229	1,428	1,063	2,281	379	2,728	1,736	3,834
4	124	714	665	1,631	238	1,428	1,146	2,922	380	2,728	1,848	4,928
5	259	714	987	1,983	383	1,428	1,548	3,566	497	2,728	2,376	6,046
6	1,020	714	2,422	2,306	1,504	1,428	3,712	4,184	2,026	2,728	5,454	7,169
7	1,633	714	3,511	2,454	2,905	1,428	6,176	4,488	4,722	2,728	10,173	7,743
8	1,757	714	4,317	2,541	3,273	1,428	7,896	4,661	5,577	2,728	13,511	8,069
9	1,791	714	4,489	2,670	3,365	1,428	8,270	4,907	5,778	2,728	14,250	8,513
10	1,813	714	4,572	2,736	3,418	1,428	8,452	5,038	5,886	2,728	14,598	8,752

Table 3.4: Memory usage in Mega bytes for the computation of Most Frequent q-gram from the Pizza & Chili Corpus. Each data is compressed by LCA algorihtm [52].

XML(LCA)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	272	418	228	1,428	515	788	382	2,728	858	1,374
3	118	714	280	565	233	1,428	525	1,062	371	2,728	870	1,891
4	135	714	320	716	258	1,428	584	1,342	374	2,728	957	2,417
5	170	714	396	858	312	1,428	704	1,611	454	2,728	1,146	2,929
6	217	714	499	996	391	1,428	878	1,871	586	2,728	1,437	3,427
7	269	714	605	1,114	482	1,428	1,064	2,100	745	2,728	1,767	3,878
8	322	714	781	1,217	576	1,428	1,371	2,301	911	2,728	2,297	4,275
9	375	714	921	1,330	672	1,428	1,625	2,519	1,082	2,728	2,747	4,697
10	430	714	1,047	1,423	771	1,428	1,853	2,700	1,259	2,728	3,155	5,057

DNA(LCA)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	379	602	228	1,428	729	1,155	377	2,728	1,362	2,168
3	114	714	379	825	228	1,428	729	1,581	370	2,728	1,362	2,975
4	114	714	380	1,048	228	1,428	730	2,007	378	2,728	1,363	3,782
5	114	714	383	1,272	228	1,428	735	2,435	380	2,728	1,370	4,591
6	114	714	401	1,504	228	1,428	760	2,873	379	2,728	1,406	5,417
7	115	714	442	1,742	229	1,428	829	3,328	380	2,728	1,519	6,274
8	117	714	496	1,970	231	1,428	922	3,768	377	2,728	1,676	7,113
9	126	714	586	2,198	241	1,428	1,068	4,209	371	2,728	1,918	7,954
10	162	714	708	2,381	277	1,428	1,239	4,578	378	2,728	2,174	8,682

ENGLISH(LCA)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	381	604	228	1,428	779	1,238	378	2,728	1,449	2,312
3	117	714	386	828	232	1,428	786	1,698	375	2,728	1,458	3,175
4	128	714	420	1,057	250	1,428	840	2,168	378	2,728	1,530	4,053
5	161	714	505	1,283	304	1,428	984	2,633	428	2,728	1,736	4,928
6	227	714	667	1,505	414	1,428	1,267	3,095	587	2,728	2,165	5,806
7	332	714	890	1,689	598	1,428	1,668	3,494	867	2,728	2,802	6,589
8	477	714	1,300	1,839	863	1,428	2,418	3,828	1,294	2,728	4,018	7,264
9	652	714	1,731	1,999	1,199	1,428	3,258	4,179	1,869	2,728	5,474	7,965
10	841	714	2,147	2,105	1,583	1,428	4,108	4,416	2,563	2,728	7,021	8,456

PROTEINS(LCA)												
	50MB				100MB				200MB			
q	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA	NMP	NSA	SMP	SSA
2	114	714	636	995	228	1,428	1,146	1,812	376	2,728	1,897	3,053
3	115	714	637	1,354	229	1,428	1,147	2,477	379	2,728	1,898	4,210
4	124	714	699	1,735	238	1,428	1,231	3,174	380	2,728	2,011	5,411
5	259	714	1,033	2,115	383	1,428	1,662	3,887	497	2,728	2,581	6,656
6	1,020	714	2,448	2,436	1,504	1,428	3,800	4,523	2,026	2,728	5,593	7,808
7	1,633	714	3,540	2,565	2,905	1,428	6,271	4,805	4,722	2,728	10,323	8,365
8	1,757	714	4,362	2,648	3,273	1,428	8,027	4,976	5,577	2,728	13,722	8,695
9	1,791	714	4,543	2,785	3,365	1,428	8,426	5,245	5,778	2,728	14,502	9,187
10	1,813	714	4,630	2,864	3,418	1,428	8,616	5,404	5,886	2,728	14,866	9,489

3.4 Applications and Extensions

We showed that for an SLP $\mathcal{T}$ of size n representing string T, q-gram frequencies problems on T can be reduced to weighted q-gram frequencies problems on a string z of length $O(qn)$, which can be much shorter than T. This idea can further be applied to obtain efficient compressed string processing algorithms for interesting problems which we briefly introduce below.

3.4.1 q-gram Spectrum Kernel

A string kernel is a function that computes the inner product between two strings which are mapped to some feature space. It is used for classifying string or text data using methods such as Support Vector Machines (SVMs), and its computation is one of the dominating factors in the time complexity of learning and classification. A q-gram spectrum kernel [47] considers the feature space of q-grams. For string T_1 and T_2, the kernel function is defined as $K_q(T_1, T_2) = \sum_{p \in \Sigma^q} |Occ(T_1, p)||Occ(T_2, p)|$. The calculation of the kernel function amounts to summing up the product of occurrence frequencies in strings T_1 and T_2 for all q-grams which occur in both T_1 and T_2. This can be done in $O(|T_1| + |T_2|)$ time using suffix trees or arrays [67, 70].

Let two SLPs $\mathcal{T}_1$ and $\mathcal{T}_2$ of size n_1 and n_2 represent strings T_1 and T_2. First, we calculate strings z_1, z_2, and weight array w_1 and w_2, for SLPs $\mathcal{T}_1$ and $\mathcal{T}_2$ using Algorithm 3. Second, we construct the suffix array and lcp array of string $z_1 z_2$, and consider the weighted q-gram frequencies on this string with respect to weight array $w_1 w_2$. As we described previously, intervals where the values of the lcp array are at least q represent occurrences of the same q-gram. A subtle difference is that we must sum the occurrences of the q-grams separately for strings T_1 and T_2. We can obtain whether an occurrence of a q-gram is in T_1 or T_2 by checking the position of the q-gram: if it is less than $|z_1| - q + 2$ then it occurs in T_1, and if it is at least $|z_1| + 1$ then it occurs in T_2. (Note that q-grams generated by the concatenation of z_1 and z_2 are essentially ignored since they have weight 0 by the construction of w_1.) Finally, we can compute the q-gram spectrum kernel $K_q(T_1, T_2)$ by multiplying the number of occurrences of each q-gram for each string, and summing them up. This can be done in $O(q(n_1 + n_2))$ time since it is a simple scan of the suffix array and lcp arrays of length $|z_1 z_2| = O(q(n_1 + n_2))$.

3.4.2 Optimal Substring Patterns of Length q

Given two sets of strings, finding string patterns that are frequent in one set and not in the other, is an important problem in string data mining, with many problem formulations and types of patterns to be considered, e.g.: in Bioinformatics [6], Machine Learning (optimal patterns [3]), and more recently Knowledge Discovery in Databases (emerging patterns [10]). A simple optimal q-gram pattern discovery problem can be defined as follows: Let $\mathbf{T_1} = \{T_{1,1}, \ldots, T_{1,m_1}\}$

and $\mathbf{T_2} = \{T_{2,1}, \ldots, T_{2,m_2}\}$ be two multisets of strings. The problem is to find the q-gram p which gives the highest (or lowest) score according to some scoring function that depends only on $|\mathbf{T_1}|$, $|\mathbf{T_2}|$, and the number of strings respectively in $\mathbf{T_1}$ and $\mathbf{T_2}$ for which p is a substring. For uncompressed strings, the problem can be solved in $O(N)$ time, where N is the total length of the strings in both $\mathbf{T_1}$ and $\mathbf{T_2}$. This can be done by using a generalized suffix array of $\mathbf{T_1}$ and $\mathbf{T_2}$, which is a suffix array constructed for all suffixes of strings in $\mathbf{T_1}$ and $\mathbf{T_2}$, and each suffix is also identified with the index of the string it comes from. We can then simply scan this suffix array and its corresponding lcp array to identify intervals corresponding to q-grams as before, and for each interval, count the number of distinct strings that come respectively from $\mathbf{T_1}$ and $\mathbf{T_2}$. We prepare a bit array of size $m_1 + m_2$ where each bit corresponds to a string in either $\mathbf{T_1}$ or $\mathbf{T_2}$, and represents whether a suffix coming from the string has occurred in the interval. Then, the counting for each interval, as well as the re-setting of the bit array, can be conducted in time linear in the size of the interval, resulting in a total of $O(N)$ time.

For the SLP compressed version of this problem, the input is two multisets of SLPs, $\mathcal{T}_1 = \{\mathcal{T}_{1,1}, \ldots, \mathcal{T}_{1,m_1}\}$ and $\mathcal{T}_2 = \{\mathcal{T}_{2,1}, \ldots, \mathcal{T}_{2,m_2}\}$. For each SLP $\mathcal{T}_{i,j}$, we construct the string $z_{i,j}$ and weight array $w_{i,j}$ as in Algorithm 3. Notice that the number of occurrences of q-grams in $T_{i,j}$ correspond to the total weight of their occurrences in $z_{i,j}$ weighted by $w_{i,j}$. Therefore, the problem can be reduced to the problem of finding the optimal q-gram from two sets of weighted strings, $\{z_{1,1}, \ldots, z_{1,m_1}\}$ and $\{z_{2,1}, \ldots, z_{2,m_2}\}$. Since the total length of $z_{i,j}$ is $O(qM)$, where M is the total number of variables in $\mathcal{T}_1$ and $\mathcal{T}_2$, the problem can be solved in $O(qM)$ time by applying the algorithm mentioned above for the uncompressed case, that incorporates weights.

3.4.3 Different Lengths

Standard techniques on suffix trees [29] can be used to modify and extend our algorithm to consider all substrings of length not only q, but all lengths up-to and including q. Note that substrings of length less than q can be associated to a q-gram that starts at the same position. For example, an occurrence of q-gram $T[u : u + q - 1]$ implies an occurrence of its prefixes, 1-gram $T[u : u]$, 2-gram $T[u : u + 1]$, $\ldots$, and $(q - 1)$-gram $T[u : u + q - 2]$, and hence, these substrings can be counted with respect to the q-gram $T[u : u + q - 1]$. Here, although the q-gram $T[u : u + q - 1]$ contains other substrings of length less than q, such substrings will be counted with respect to a different q-gram. For example, the 1-gram $T[u + 1 : u + 1], \ldots,$ and $(q-1)$-gram $T[u+1 : u+q-1]$ will be counted with respect to q-gram $T[u+1 : u+q]$, 1-gram $T[u + 2 : u + 2], \ldots,$ and $(q - 2)$-gram $T[u + 2 : u + q - 1]$ with q-gram $T[u + 2 : u + q + 1]$, and so on. Therefore, occurrences of substrings with lengths at most q are all represented in the string z and weight array w as computed in Algorithm 3, where the weight of a substring that starts at position i is $w[i]$. A slight technicality is that the last $q - 1$ positions of the text do not have a corresponding q-gram which starts at that same position, and cannot be counted this way. This can be overcome simply by adding $T[N-q+2:N]\$$ to z, and $1^{q-1}0$ to w, where $\$$ is a character which does not appear elsewhere in T.

Next, consider a suffix tree of the modified z, where each leaf that corresponds to suffix $z[i:|z|]$ is weighted by $w[i]$. Then, for any (possibly implicit) node v in the suffix tree that represents string P of length at most q, the sum of the weights on the leaves in the subtree rooted at v is $Occ(T,P)$. For the applications discussed above, although the number of substrings of length at most q can be as large as $\Theta(q^2n)$, the $O(qn)$ time complexity can be maintained. This is because the size of the suffix tree is $O(qn)$, and there exist only $O(qn)$ substrings with distinct frequencies, which correspond to nodes of the suffix tree. Therefore, the computations of the extra substrings can be summarized with respect to them. The algorithm can also be simulated on suffix and LCP arrays [40].

When extending the problem of finding the optimal substring pattern mentioned in Section 3.4.2 to include all lengths up-to and including q, there is a technicality in counting the number of distinct strings that contain the pattern. This problem can be solved by applying the algorithm of [31] to two sets of strings.

Chapter 4

Faster Algorithm for q-gram Frequencies

In Chapter 3, we considered the problem of computing all q-gram frequencies in a string T of length N when given an SLP of size n representing T, and proposed an $O(qn)$ algorithm to solve the problem. In this Chapter, we improve the $O(qn)$ algorithm both theoretically and practically. The drawback of the $O(qn)$ algorithm is that it runs slowly when q is large since q can be $O(N)$ in theory, and the total length of the decompressed strings can be $O(Nn)$ and the algorithm requires $O(Nn)$ time in such situation. We introduce a q-gram neighbor relation on SLP variables, in order to reduce the redundancy in the partial decompression of T which is performed in the $O(qn)$ algorithm. Using this relation, we are able to convert the problem to a weighted q-gram frequencies problem on a weighted trie, whose size is at most $N - dup(q, \mathcal{T})$. Here, $dup(q, \mathcal{T})$ is a quantity that represents the amount of redundancy that the SLP captures with respect to q-grams. Since the size of the trie is also bounded by $O(qn)$, the time complexity of our new algorithm is $O(\min\{qn, N - dup(q, \mathcal{T})\})$, improving on our previous $O(qn)$ algorithm when $q = \Omega(N/n)$. The computational experiments show that our new approach achieves a practical speed up as well, for all values of q.

This result primarily appeared in [27].

4.1 $O(N - dup(q, \mathcal{T}))$ time Algorithm on SLPs

We now describe our new algorithm which solves the q-gram frequencies problem on SLPs. The new algorithm basically follows the previous $O(qn)$ algorithm, but is an elegant refinement. The reduction for the previous $O(qn)$ algorithm leads to a fairly large amount of redundantly decompressed regions of the text as q increases. This is due to the fact that the t_i's are considered independently for each variable X_i, while *neighboring* q-grams that are stabbed by different variables actually share $q - 1$ characters. The key idea of our new algorithm is to exploit this redundancy. (See Figure 4.1.) In what follows, we introduce the concept of q-gram neighbors, and reduce the q-gram frequencies problem on SLP to a weighted q-gram frequencies problem

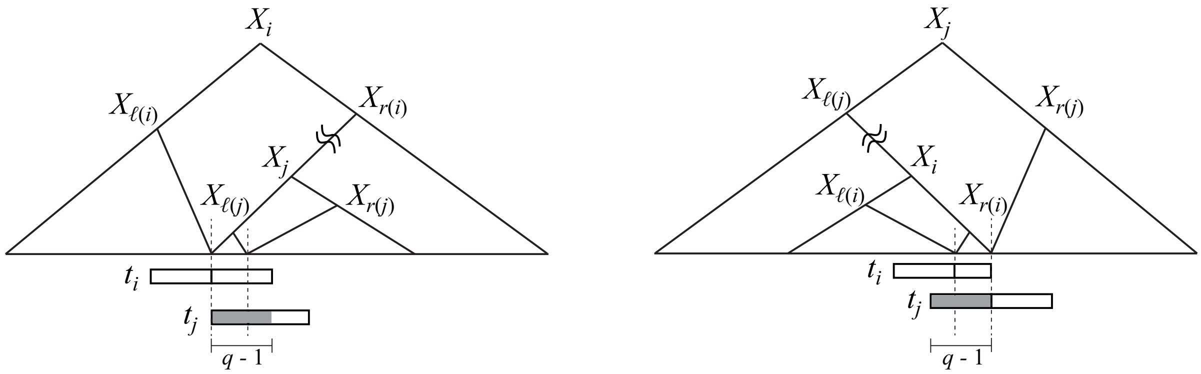


Figure 4.1: q-gram neighbors and redundancies. (Left) X_j is a right q-gram neighbor of X_i, and X_i is *a* left q-gram neighbor of X_j. Note that the right q-gram neighbor of X_i is uniquely determined since $|X_{r(i)}| \geq q$ and it must be a descendant on the left most path rooted at $X_{r(i)}$. However, X_j may have other left q-gram neighbors, since $|X_{\ell(j)}| < q$, and they must be ancestors of X_j. t_i (resp. t_j) represents the string corresponding to the union of intervals $[u : u+q-1]$ where $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_i$ (resp. $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_j$). The shaded region depicts the string which is redundantly decompressed, if both t_i and t_j are considered independently. (Right) Shows the reverse case, when $|X_{r(i)}| < q$.

on a weighted tree.

4.1.1 q-gram Neighbor Graph

We say that X_j is a *right q-gram neighbor* of X_i $(i \neq j)$, or equivalently, X_i is a *left q-gram neighbor* of X_j, if for some integer $u \in [1 : N - q]$, $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_i$ and $X_{\langle \xi_{\mathcal{T}}(u+1,u+q)\rangle} = X_j$. Notice that $|X_i|$ and $|X_j|$ are both at least q if X_i and X_j are right or left q-gram neighbors of each other.

Definition 2. *For $q \geq 2$, the right q-gram neighbor graph of SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^n$ is the directed graph $G_q = (V, E_r)$, where*

$$\begin{aligned} V &= \{X_i \mid i \in \{1, \ldots, n\}, |X_i| \geq q\} \\ E_r &= \{(X_i, X_j) \mid X_j \textit{ is a right } q\textit{-gram neighbor of } X_i\ \} \end{aligned}$$

Note that there can be multiple right q-gram neighbors for a given variable. However, the total number of edges in the neighbor graph is bounded by $2n$, as will be shown below.

Lemma 5. *Let X_j be a right q-gram neighbor of X_i. If, $|X_{r(i)}| \geq q$, then X_j is the label of the deepest node on the left-most path of the derivation tree rooted at a node labeled $X_{r(i)}$ whose length is at least q. Otherwise, if $|X_{r(i)}| < q$, then X_i is the label of the deepest node on the right-most path rooted at a node labeled $X_{\ell(j)}$ whose length is at least q.*

Proof. Suppose $|X_{r(i)}| \geq q$. Let u be a position, where $X_{\langle \xi_{\mathcal{T}}(u,u+q-1)\rangle} = X_i$ and $X_{\langle \xi_{\mathcal{T}}(u+1,u+q)\rangle} = X_j$. Then, since the interval $[u+1 : u+q]$ is a prefix of $itv(X_{r(i)})$, X_j must be on the left most path rooted at $X_{r(i)}$. Since $X_j = X_{\langle \xi_{\mathcal{T}}(u+1,u+q)\rangle}$, the lemma follows from the definition of $\xi_{\mathcal{T}}$. The case for $|X_{r(i)}| < q$ is symmetrical and can be shown similarly. □

Lemma 6. *For an arbitrary SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^n$ and integer $q \geq 2$, the number of edges in the right q-gram neighbor graph G_q of $\mathcal{T}$ is at most $2n$.*

Proof. Suppose X_j is a right q-gram neighbor of X_i. From Lemma 5, we have that if $|X_{r(i)}| \geq q$, the right q-gram neighbor of X_i is uniquely determined and that $|X_{\ell(j)}| < q$. (See Figure 4.1 (Left)) Similarly, if $|X_{r(i)}| < q$, $|X_{\ell(j)}| \geq q$ and the left q-gram neighbor of X_j is uniquely X_i. (See Figure 4.1 (Right)) Therefore,

$$\begin{aligned} & \sum_{i=1}^{n} |\{(X_i, X_j) \in E_r \mid |X_{r(i)}| \geq q\}| + \sum_{i=1}^{n} |\{(X_i, X_j) \in E_r \mid |X_{r(i)}| < q\}| \\ = & \sum_{i=1}^{n} |\{(X_i, X_j) \in E_r \mid |X_{r(i)}| \geq q\}| + \sum_{i=1}^{n} |\{(X_i, X_j) \in E_r \mid |X_{\ell(j)}| \geq q\}| \leq 2n. \end{aligned}$$

Since the number of unique left q-gram neighbors is bounded by n (one for each variable), the total number of right q-gram neighbors is $2n$. □

Lemma 7. *For an arbitrary SLP $\mathcal{T} = \{X_i \to expr_i\}_{i=1}^n$ and integer $q \geq 2$, the right q-gram neighbor graph G_q of $\mathcal{T}$ can be constructed in $O(n)$ time.*

Proof. For any variable X_i, let $lm_q(X_i)$ and $rm_q(X_i)$ respectively represent the index of the label of the deepest node with length at least q on the left-most and right-most path in the derivation tree rooted at X_i, or $null$ if $|X_i| < q$. These values can be computed for all variables in a total of $O(n)$ time based on the following recursion: If $(X_i \to a) \in \mathcal{T}$ for some $a \in \Sigma$, then $lm_q(X_i) = rm_q(X_i) = null$. For $(X_i \to X_{\ell(i)} X_{r(i)}) \in \mathcal{T}$,

$$lm_q(X_i) = \begin{cases} null & \text{if } |X_i| < q, \\ i & \text{if } |X_i| \geq q \text{ and } |X_{\ell(i)}| < q, \\ lm_q(X_{\ell(i)}) & \text{otherwise.} \end{cases}$$

$rm_q(X_i)$ can be computed similarly. Finally,

$$\begin{aligned} E_r \;=\; & \{(X_i, X_{lm_q(X_{r(i)})}) \mid lm_q(X_{r(i)}) \neq null, i = 1, \ldots, n\} \\ & \cup \{(X_{rm_q(X_{\ell(i)})}, X_i) \mid rm_q(X_{\ell(i)}) \neq null, i = 1, \ldots, n\}. \end{aligned}$$

□

Lemma 8. *Let $G_q = (V, E_r)$ be the right q-gram neighbor graph of SLP $\mathcal{T} = \{X_i = expr_i\}_{i=1}^n$ representing string T, and let $X_{i_1} = X_{\langle \xi_{\mathcal{T}}(1,q) \rangle}$. Any variable $X_j \in V (i_1 \neq j)$ is reachable from X_{i_1}, that is, there exists a directed path from X_{i_1} to X_j in G_q.*

Proof. Straightforward, since any q-gram of T except for the left most $T([1 : q])$ has a q-gram on its left. $\square$

4.1.2 Weighted q-gram Frequencies Over a Trie

From Lemma 8, we have that the right q-gram neighbor graph is connected. Consider an arbitrary directed spanning tree rooted at $X_{i_1} = X_{\langle \xi_{\mathcal{T}}(1,q) \rangle}$ which can be obtained in linear time by a depth first traversal on G_q from X_{i_1}. We define the label $label(X_i)$ of each node X_i of the q-gram neighbor graph, by

$$label(X_i) = t_i[q : |t_i|]$$

where $t_i = suf(X_{\ell(i)}, q-1)pre(X_{r(i)}, q-1)$ as before. For convenience, let X_{i_0} be a dummy variable such that $label(X_{i_0}) = T([1 : q-1])$, and $X_{r(i_0)} = X_{i_1}$ (and so $(X_{i_0}, X_{i_1}) \in E_r$).

Lemma 9. *Fix a directed spanning tree on the right q-gram neighbor graph of SLP $\mathcal{T}$, rooted at X_{i_0}. Consider a directed path $X_{i_0}, \ldots, X_{i_m}$ on the spanning tree. The weighted q-gram frequencies on the string obtained by the concatenation $label(X_{i_0})label(X_{i_1}) \cdots label(X_{i_m})$, where each occurrence of a q-gram that ends in a position in $label(X_{i_j})$ is weighted by $vOcc(X_{i_j})$, is equivalent to the weighted q-gram frequencies of strings $\{t_{i_1}, \ldots t_{i_m}\}$ where each q-gram in t_{i_j} is weighted by $vOcc(X_{i_j})$.*

Proof. Proof by induction: for $m = 1$, we have that $label(X_{i_0})label(X_{i_1}) = t_{i_1}$. All q-grams in t_{i_1} end in t_{i_1} and so are weighted by $vOcc(X_{i_1})$. When $label(X_{i_j})$ is added to $label(X_{i_0}) \cdots label(X_{i_{j-1}})$, $|label(X_{i_j})|$ new q-grams are formed, which correspond to q-grams in t_{i_j}, i.e. $|t_{i_j}| = q - 1 + |label(X_{i_j})|$, and t_{i_j} is a suffix of $label(X_{i_{j-1}})label(X_{i_j})$. All the new q-grams end in $label(X_{i_j})$ and are thus weighted by $vOcc(X_{i_j})$. $\square$

From Lemma 9, we can construct a weighted trie Υ based on a directed spanning tree of G_q and $label()$, where the weighted q-grams in Υ (represented as length-q paths) correspond to the occurrence frequencies of q-grams in T[1].

Lemma 10. Υ *can be constructed in time linear in its size.*

Proof. See Algorithm 6. Let G be the q-gram neighbor graph. We construct Υ in a depth first manner starting at X_{i_0}. The crux of the algorithm is that rather than computing $label()$ separately

[1] A minor technicality is that a node in Υ may have multiple children with the same character label, but this does not affect the time complexities of the algorithm.

Algorithm 6: Constructing weighted trie from SLP

Construct right q-gram neighbor graph $G = (V, E_r)$;
Calculate $vOcc(X_i)$ and $|label(X_i)|$ for $i = 1, \ldots, n$;
for $i = 0, \ldots, n$ **do** visited$[i]$ = false;
$X_{i_1} = X_{\langle \xi_{\mathcal{T}}(1,q) \rangle} = lm_q(X_n)$;
Define X_{i_0} so that $X_{r(i_0)} = X_{i_1}$ and $|label(X_{i_0})| = q - 1$;
$root \leftarrow$ new node; `// root of resulting trie`
BuildDepthFirst(i_0, $root$);
return $root$

Procedure BuildDepthFirst(i, $trieNode$)

`// add prefix of` $r(i)$ `to trieNode while right neighbors are unique`
$l \leftarrow 0$; $k \leftarrow i$;
while *true* **do**
  $l \leftarrow l + |label(X_k)|$;
  visited$[k] \leftarrow$ true;
  `// exit loop if right neighbor might be non-unique or is visited`
  if $|X_{r(k)}| < q$ **or** visited$[lm_q(X_{r(k)})]$ = true **then break**;
  $k \leftarrow lm_q(X_{r(k)})$;
add new branch from $trieNode$ with string $X_{r(i)}([1 : l])$;
let end of new branch be $newTrieNode$;
`// If` $|X_{r(k)}| < q$`, there may be multiple right neighbors.`
`// If` $|X_{r(k)}| \geq q$`, nothing is done because it was already visited.`
for $X_c \in \{X_j \mid (X_k, X_j) \in E_r\}$ **do**
  if visited$[c]$ = false **then** BuildDepthFirst(X_c, $newTrieNode$);

for each variable, we are able to aggregate the $label()$s and limit all partial decompressions of variables to prefixes of variables, so that Lemma 3 can be used.

Any directed acyclic path on G starting at X_{i_0} can be segmented into multiple sequences of variables, where each sequence $X_{i_j}, \ldots, X_{i_k}$ is such that j is the only integer in $[j : k]$ such that $j = 0$ or $|X_{r(i_{j-1})}| < q$. From Lemma 5, we have that $X_{i_{j+1}}, \ldots, X_{i_k}$ are uniquely determined. If $j > 0$, $label(X_{i_j})$ is a prefix of $val(X_{r(i_j)})$ since $|X_{r(i_{j-1})}| < q$ (see Figure 4.1 Right), and if $j = 0$, $label(X_{i_0})$ is again a prefix of $val(X_{r(i_0)}) = val(X_{i_1})$. It is not difficult to see that $label(X_{i_j}) \cdots label(X_{i_k})$ is also a prefix of $X_{r(i_j)}$ since $X_{i_{j+1}}, \ldots, X_{i_k}$ are all descendants of $X_{r(i_j)}$, and each $label()$ extends the partially decompressed string to consider consecutive q-grams in $X_{r(i_j)}$. Since prefixes of variables of SLPs can be decompressed in time proportional to the output size with linear time pre-processing (Lemma 3), the lemma follows. □

We only illustrate how the character labels are determined in the pseudo-code of Algorithm 6. It is straightforward to assign a weight $vOcc(X_k)$ to each node of Υ that corresponds to $label(X_k)$.

Lemma 11. *The number of edges in* Υ *is* $(q-1)+\sum\{|t_i|-(q-1) \mid |X_i| \geq q, i=1,\ldots,n\} = N - dup(q,\mathcal{T})$ *where*

$$dup(q,\mathcal{T}) = \sum\{(vOcc(X_i)-1)\cdot(|t_i|-(q-1)) \mid |X_i| \geq q, i=1,\ldots,n\}\}$$

Proof. $(q-1)+\sum\{|t_i|-(q-1) \mid |X_i| \geq q, i=1,\ldots,n\}$ is straight forward from the definition of $label(X_i)$ and the construction of Υ. Concerning dup, each variable X_i occurs $vOcc(X_i)$ times in the derivation tree, but only once in the directed spanning tree. This means that for each occurrence after the first, the size of Υ is reduced by $|label(X_i)| = |t_i| - (q-1)$ compared to T. Therefore, the lemma follows. □

To efficiently count the weighted q-gram frequencies on Υ, we can use suffix trees. A suffix tree for a trie is defined as a generalized suffix tree for the set of strings represented in the trie as leaf to root paths[2]. The following is known.

Lemma 12 ([65])**.** *Given a trie of size* m*, the suffix tree for the trie can be constructed in* $O(m)$ *time and space.*

With a suffix tree, it is a simple exercise to solve the weighted q-gram frequencies problem on Υ in linear time. In fact, it is known that the suffix array for the common suffix trie can also be constructed in linear time [19], as well as its longest common prefix array [43], which can also be used to solve the problem in linear time.

Corollary 1. *The weighted* q*-gram frequencies problem on a trie of size* m *can be solved in* $O(m)$ *time and space.*

From the above arguments, the theorem follows.

Theorem 2. *The* q*-gram frequencies problem on an SLP* $\mathcal{T}$ *of size* n*, representing string* T *can be solved in* $O(\min\{qn, N - dup(q,\mathcal{T})\})$ *time and space.*

Note that since each $q \leq |t_i| \leq 2(q-1)$, and $|label(X_i)| = |t_i| - (q-1)$, the total length of decompressions made by the algorithm, i.e. the size of the reduced problem, is at least halved and can be as small as $1/q$ (e.g. when all $|t_i| = q$), compared to the previous $O(qn)$ algorithm.

[2]When considering leaf to root paths on Υ, the direction of the string is the reverse of what is in T. However, this is merely a matter of representation of the output.

Table 4.1: A comparison of the size of Υ and the total length of strings t_i for SLPs that represent textual data from Pizza & Chili Corpus. The length of the original text is 209,715,200. The SLPs were constructed by RE-PAIR [45].

	XML		DNA		ENGLISH		PROTEINS	
q	$\sum \lvert t_i \rvert$	size of Υ	$\sum \lvert t_i \rvert$	size of Υ	$\sum \lvert t_i \rvert$	size of Υ	$\sum \lvert t_i \rvert$	size of Υ
2	19,082,988	9,541,495	46,342,894	23,171,448	37,889,802	18,944,902	64,751,926	32,375,964
3	37,966,315	18,889,991	92,684,656	46,341,894	75,611,002	37,728,884	129,449,835	64,698,833
4	55,983,397	27,443,734	139,011,475	69,497,812	112,835,471	56,066,348	191,045,216	93,940,205
5	72,878,965	35,108,101	185,200,662	92,516,690	148,938,576	73,434,080	243,692,809	114,655,697
6	88,786,480	42,095,985	230,769,162	114,916,322	183,493,406	89,491,371	280,408,504	123,786,699
7	103,862,589	48,533,013	274,845,524	135,829,862	215,975,218	103,840,108	301,810,933	127,510,939
8	118,214,023	54,500,142	315,811,932	153,659,844	246,127,485	116,339,295	311,863,817	129,618,754
9	131,868,777	60,045,009	352,780,338	167,598,570	273,622,444	126,884,532	318,432,611	131,240,299
10	144,946,389	65,201,880	385,636,192	177,808,192	298,303,942	135,549,310	325,028,658	132,658,662
15	204,193,702	86,915,492	477,568,585	196,448,347	379,441,314	157,558,436	347,993,213	138,182,717
20	255,371,699	104,476,074	497,607,690	200,561,823	409,295,884	162,738,812	364,230,234	142,213,239
50	424,505,759	157,069,100	530,329,749	206,796,322	429,380,290	165,882,006	416,966,397	156,257,977
100	537,677,786	192,816,929	536,349,226	207,838,417	435,843,895	167,313,028	463,766,667	168,544,608

4.2 Computational Experiments

We first evaluate the size of the trie Υ induced from the right q-gram neighbor graph, on which the running time of the new algorithm of Section 4.1 is dependent. We used data sets obtained from Pizza & Chili Corpus, and constructed SLPs using the RE-PAIR [45] compression algorithm. Each data is of size 200MB. Table 4.1 shows the sizes of Υ for different values of q, in comparison with the total length of strings t_i, on which the previous $O(qn)$-time algorithm of Section 3.2 works. We cumulated the lengths of all t_i's only for those satisfying $|t_i| \geq q$, since no q-gram can occur in t_i's with $|t_i| < q$. Observe that for all values of q and for all data sets, the size of Υ (i.e., the total number of characters in Υ) is smaller than those of t_i's and the original string.

The construction of the suffix tree or array for a trie, as well as the algorithm for Lemma 3, require various tools such as level ancestor queries [4, 5, 18] for which we did not have an efficient implementation. Therefore, we try to assess the practical impact of the reduced problem size using a simplified version of our new algorithm. We compared three algorithms (NSA, SSA, STSA) that count the occurrence frequencies of all q-grams in a text given as an SLP. NSA is the $O(N)$-time algorithm which works on the uncompressed text, using suffix and LCP arrays. SSA is our previous $O(qn)$-time algorithm [25], and STSA is a simplified version of our new algorithm. STSA further reduces the weighted q-gram frequencies problem on Υ, to a weighted q-gram frequencies problem on a single string as follows: instead of constructing Υ, each branch of Υ (on line 7 of BuildDepthFirst) is appended into a single string. The q-

Table 4.2: Running time in seconds for SLPs that represent textual data from Pizza & Chili Corpus. The SLPs were constructed by RE-PAIR [45]. Bold numbers represent the fastest time for each data and q. STSA is faster than SSA whenever $q > 3$.

	XML			DNA			ENGLISH			PROTEINS		
q	NSA	SSA	STSA	NSA	SSA	STSA	NSA	SSA	STSA	NSA	SSA	STSA
2	41.67	**6.53**	7.63	61.28	**19.27**	22.73	56.77	**16.31**	19.23	60.16	**27.13**	30.71
3	41.46	10.96	**10.92**	61.28	**29.14**	31.07	56.77	25.58	**25.57**	60.53	**47.53**	50.65
4	41.87	16.27	**14.5**	61.65	42.22	**41.69**	56.77	37.48	**34.95**	**60.86**	74.89	73.51
5	41.85	21.33	**17.42**	61.57	56.26	**54.21**	57.09	49.83	**45.21**	**60.53**	101.64	79.1
6	41.9	25.77	**20.07**	**60.91**	73.11	68.63	57.11	62.91	**55.28**	**61.18**	123.74	75.83
7	41.73	30.14	**21.94**	**60.89**	90.88	82.85	**56.64**	75.69	63.35	**61.14**	136.12	72.62
8	41.92	34.22	**23.97**	**61.57**	110.3	93.46	**57.27**	87.9	69.7	**61.39**	142.29	71.08
9	41.92	37.9	**25.08**	**61.26**	127.29	96.07	**57.09**	100.24	73.63	**61.36**	148.12	69.88
10	41.76	41.28	**26.45**	**60.94**	143.31	96.26	**57.43**	110.85	75.68	**61.42**	149.73	69.34
15	41.95	58.21	**32.21**	**61.72**	190.88	84.86	**57.31**	146.89	70.63	**60.42**	160.58	66.57
20	41.82	74.61	**39.62**	**61.36**	203.03	83.13	**57.65**	161.12	64.8	**61.01**	165.03	66.09
50	**42.07**	134.38	53.98	**61.73**	216.6	78.0	**57.02**	166.67	57.89	**61.05**	181.14	66.36
100	**41.81**	181.23	60.18	**61.46**	217.05	75.91	57.3	166.67	**56.86**	**60.69**	197.33	69.9

grams that are represented in the branching edges of Υ can be represented in the single string, by redundantly adding $suf(X_{r(i)}([1:l]), q-1)$ in front of the string corresponding to the next branch. This leads to some duplicate partial decompression, but the resulting string is still always shorter than the string produced by our previous algorithm [25]. The partial decompression of $X_{r(i)}([1:l])$ is implemented using a simple $O(h+l)$ algorithm, where h is the height of the SLP which can be as large as $O(n)$. These implementations are available at `http://code.google.com/p/qshi/` along with the implementations of Section 3.3.

All computations were conducted on a Mac Pro (Mid 2010) with MacOS X Lion 10.7.2, and 2 x 2.93GHz 6-Core Xeon processors and 64GB Memory, only utilizing a single process/thread at once. The program was compiled using the GNU C++ compiler (`g++`) 4.6.2 with the `-Ofast` option for optimization. The running times were measured in seconds, after reading the uncompressed text into memory for NSA, and after reading the SLP that represents the text into memory for SSA and STSA. Each computation was repeated at least 3 times, and the average was taken.

Table 4.2 summarizes the running times of the three algorithms. SSA and STSA computed weighted q-gram frequencies on t_i and Υ, respectively. Since the difference between the total length of t_i and the size of Υ becomes larger as q increases, STSA outperforms SSA when the value of q is not small. In fact, in Table 4.2 STSA was faster than SSA for all values of $q > 3$. STSA was even faster than NSA on the XML data whenever $q \leq 20$. What is interesting is that STSA outperformed NSA on the ENGLISH data when $q = 100$.

Chapter 5

Algorithm for Non-Overlapping q-gram Frequencies

In Chapter 3 and 4, we considered the q-gram frequencies problem on SLPs, and how to efficiently solve the problem when the input text is given as an SLP. In this Chapter, we further consider a variation of the problem, where we consider *non-overlapping occurrence frequencies* of q-grams. The *non-overlapping occurrence frequency* $nOcc(T, P)$ of a string P in a string T is defined as the maximum number of non-overlapping occurrences of P in T [2]. The precise definition of the new problem is defined as follows.

Problem 2 (Non-overlapping q-gram frequencies on SLP)**.** *Given an SLP $\mathcal{T}$ of size n that describes string T and a positive integer q, compute $nOcc(T, P)$ for all q-grams $P \in \Sigma^q$.*

For uncompressed texts, the problem can be solved in $O(N)$ time, by applying string indices such as suffix arrays. A similar problem is the *string statistics problem* [2], which asks for the non-overlapping occurrence frequency of a given string P in string T. The problem can be solved in $O(|P|)$ time for any P, provided that the string is pre-processed in $O(N \log N)$ time using the sophisticated algorithm of [7]. However, note that the preprocessing requires only $O(N)$ time if occurrences are allowed to overlap. This perhaps indicates the intrinsic difficulty that arises when considering overlaps.

For SLPs, if $q = 1$ then since no occurrences of a 1-gram overlap, the 1-gram non-overlapping frequency is simply the number of occurrences of the corresponding character in string T. This can be computed in a total of $O(n)$ time, since $nOcc(T, a) = \sum_{X_i = a} vOcc(X_i)$ for each $a \in \Sigma$. So we consider the problem for $q \geq 2$. For $q = 2$, the problem for SLP was first considered in [32], where an algorithm for $q = 2$ running in $O(n^4 \log n)$ time and $O(n^3)$ space was presented. However, the algorithm cannot be readily extended to handle $q > 2$. Intuitively, the problem for $q = 2$ is much easier compared to larger values of q, since there is only one way for a 2-gram to overlap, while there can be many ways that a longer q-gram can overlap. In this chapter, we present the first algorithm for calculating the non-overlapping occurrence frequencies of all q-grams, that works for any $q \geq 2$, and runs in $O(q^2 n)$ time and $O(qn)$ space. Not only do we solve a more general problem, but the complexity is greatly improved compared to previous work.

In the following sections, we first describe an alternative algorithm to compute 2-gram non-overlapping frequencies on SLPs, and then give an extended algorithm for $q \geq 3$.

This result primarily appeared in [26].

5.1 Simple Linear Time Algorithm on SLPs for $q = 2$

Note that for convenience $X_i[j]$ and $X_i[j:k]$ denote $val(X_i)[j]$ and $val(X_i)[j:k]$, respectively. (See Chapter 2.3.)

Problem 3 (Non-overlapping 2-gram frequencies on SLP). *Given an SLP $\mathcal{T}$ that describes string T, compute $nOcc(T, P)$ for all 2-grams $P \in \Sigma^2$.*

Let $plen(X_i) = \max\{k \mid X_i[j] = pre(X_i, 1), 1 \leq \forall j \leq k\}$ and $slen(X_i) = \min\{k \mid X_i[j] = suf(X_i, 1), k \leq \forall j \leq |X_i|\}$. That is, $plen(X_i)$ and $slen(X_i)$ are the length of the maximum runs of the first and the last characters of $val(X_i)$, respectively. We can compute $plen(X_i)$ for all variables X_i in a total of $O(n)$ time, as follows:

$$plen(X_i) = \begin{cases} 1 & \text{if } X_i = a, \\ |X_{\ell(i)}| + plen(X_{r(i)}) & \text{if } X_i = X_{\ell(i)} X_{r(i)}, plen(X_{\ell(i)}) = |X_{\ell(i)}|, X_{\ell(i)}[1] = X_{r(i)}[1], \\ plen(X_{\ell(i)}) & \text{otherwise.} \end{cases}$$

$slen(X_i)$ can be computed similarly in $O(n)$ time.

Theorem 3. *Problem 3 can be solved in $O(n)$ time.*

Proof. Algorithm 7 shows a pseudo-code of our algorithm to compute non-overlapping frequences of 2-grams from a given SLP. We initialize list z to be empty.

Firstly, let us consider a 2-gram of form ab, where $a \neq b \in \Sigma$. It is clear that no occurrences of such a 2-gram overlap. Therefore, we simply compute the number of occurrences of ab. By Lemma 4, we have $nOcc(T, ab) = \sum vOcc(X_i)$, where $X_i = X_{\ell(i)} X_{r(i)}$ is any variable such that $suf(X_{\ell(i)}, 1) = a$ and $pre(X_{r(i)}, 1) = b$. We append the pair (ab, $vOcc(X_i)$) to list z (in line 6).

Now we consider a 2-gram of form aa, where $a \in \Sigma$. A key idea is to find an interval that corresponds to a maximal repetition of a in T. Namely, if there is an interval $[u, v]$ ($1 \leq u \leq v \leq N$) such that $T[u:v] = a^{v-u+1}$, $T[u-1] \neq a$, and $T[v+1] \neq a$, then we know that there are at most $\lfloor (v-u+1)/2 \rfloor$ non-overlapping occurrences of aa in $T[u-1 : v+1]$. By summing up this value for all such intervals, we obtain $nOcc(T, aa)$. To find such intervals, we process variables $X_i = X_{\ell(i)} X_{r(i)}$ in increasing order of i. There are three cases to consider (see also Figure 5.1):

1. When $suf(X_{\ell(i)}, 1) = pre(X_{r(i)}, 1) = a$, $slen(X_{\ell(i)}) < |X_{\ell(i)}|$ and $plen(X_{r(i)}) < |X_{r(i)}|$ (line 13). For any interval $[u', v'] \in itv(X_i)$, let $j_1 = u' + |X_{\ell(i)}| - slen(X_{\ell(i)}) - 1$ and $j_2 = u' + |X_{\ell(i)}| + plen(X_{r(i)})$, it holds that $T[j_1] \neq a$ and $T[j_2] \neq a$. Since there are at most $\lfloor (slen(X_{\ell(i)}) + plen(X_{r(i)}))/2 \rfloor \geq 1$ non-overlapping occurrences of aa in $T[j_1+1 : j_2-1]$, we append pair (aa, $vOcc(X_i) \cdot \lfloor (slen(X_{\ell(i)}) + plen(X_{r(i)}))/2 \rfloor$) to list z.

2. When $suf(X_{\ell(i)}, 1) \neq pre(X_{r(i)}, 1)$ and $1 < slen(X_{\ell(i)}) < |X_{\ell(i)}|$ (line 9). Let $suf(X_{\ell(i)}, 1) = a$. For any interval $[u', v'] \in itv(X_i)$, it holds that $T[u' + |X_{\ell(i)}| - slen(X_{\ell(i)}) - 1] \neq a$ and $T[u' + |X_{\ell(i)}|] \neq a$. Since there are at most $\lfloor slen(X_{\ell(i)})/2 \rfloor \geq 1$ non-overlapping occurrences of aa in $T[u' + |X_{\ell(i)}| - slen(X_{\ell(i)}) - 1 : u' + |X_{\ell(i)}|]$, we append pair ($aa$, $vOcc(X_i) \cdot \lfloor slen(X_{\ell(i)})/2 \rfloor$) to list z.

3. When $suf(X_{\ell(i)}, 1) \neq pre(X_{r(i)}, 1)$ and $1 < plen(X_{r(i)}) < |X_{r(i)}|$ (line 11). This is symmetric to Case 2, and we append pair (bb, $vOcc(X_i) \cdot \lfloor plen(X_{r(i)})/2 \rfloor$) to list z, where $b = pre(X_{r(i)}, 1)$.

For convenience, we assume that T starts and ends with special characters $\#$ and $\$$ that do not occur anywhere else in T, respectively. Then we can cope with the last variable X_n as described above. By Lemma 4, we are guaranteed to obtain the non-overlapping frequencies for all 2-grams.

For all variables X_i, $pre(X_i, 1)$, $suf(X_i, 1)$, $plen(X_i)$, and $slen(X_i)$ can be computed in a total of $O(n)$ time, as descrived above. The amortized number of 2-grams appended to w for each variable is at most one, and hence the size of z does not exceed $2n$. Assuming an integer alphabet, sorting the elements in z using radix sort takes $O(n)$ time (line 14). Finally, since the same 2-gram will appear consecutively in z after the sort, we may scan z and sum up the occurrences for each distinct 2-gram in $O(n)$ time (line 15). □

5.2 $O(q^2 n)$ time Algorithm on SLPs for $q > 2$

5.2.1 Key Ideas

Solving Problem 2 for $q \geq 3$ is essentially more difficult than when $q \leq 2$, since q-grams with $q \geq 3$ can have more than 1 period. This implies that computing $plen(X_i)$ and $slen(X_i)$ does

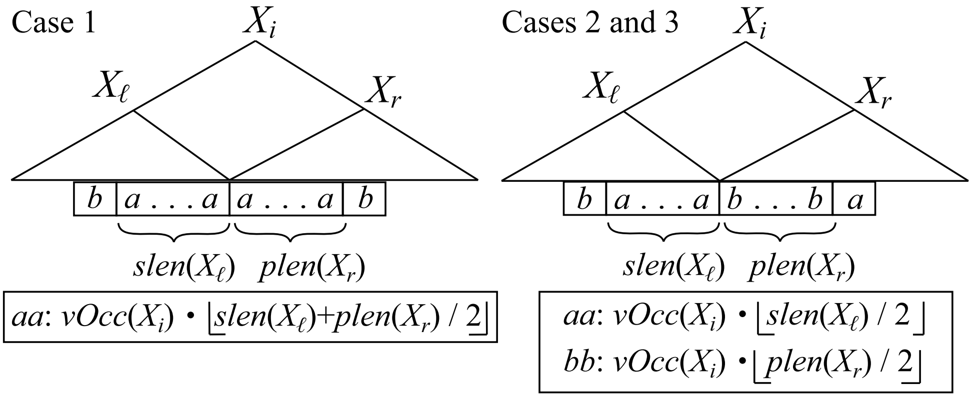


Figure 5.1: Non-overlapping frequencies corresponding to X_i

not help. To deal with the general case $q \geq 3$, we introduce an extended notion of $plen(X_i)$ and $slen(X_i)$, called *longest overlapping covers*.

For any string T and positive integers q and j ($1 \leq j \leq j+q-1 \leq N$), the *longest overlapping cover* of the q-gram $P = T[j : j+q-1]$ w.r.t. position j of T is an ordered pair $loc_q(T, j) = (b, e)$ of positions in T which is defined as:

$$loc_q(T,j) = \arg\max_{(b,e)} \left\{ (e-b) \;\middle|\; \begin{array}{l} (b,e) \in Occ(T,P) \times ((q-1) \oplus Occ(T,P)), \\ b \leq j \leq j+q-1 \leq e, \\ \forall k \in [b : e-q] \cap Occ(T,P), \\ \quad [k+1 : \min\{k+q-1, e-q+1\}] \cap Occ(T,P) \neq \emptyset \end{array} \right\}$$

Namely, $loc_q(T, j)$ represents the beginning and end positions of the maximum chain of overlapping occurrences of q-gram $T[j : j+q-1]$ that contains position j. For example, consider string $T =$ `aaabaabaaabaabaaaabaa` of length 21. For $q = 5$ and $j = 9$, we have $loc_q(T, j) = (2, 16)$, since $T[2 : 6] = T[5 : 9] = T[9 : 13] = T[12 : 16] =$ `aabaa`. Note that $T[17 : 21] =$ `aabaa` is not contained in this chain since it does not overlap with $T[12 : 16]$.

Lemma 13. *Given a string T and integers q, j, the longest cover $loc_q(T, j)$ can be computed in $O(N)$ time.*

Proof. Using, for example, the KMP algorithm [44], we can obtain a sorted list of $Occ(T, T[j : j+q-1])$ in $O(N)$ time. We can just scan this list forwards and backwards, to easily obtain b and e. $\square$

For a variable $X_i = X_{\ell(i)}X_{r(i)}$ and a position $1 \leq j \leq |X_i| - q + 1$, a longest overlapping cover $(b, e) = loc_q(X_i, j)$ is said to be *closed in* X_i if $q-1 < b < |X_{\ell(i)}|+q$ and $|X_{\ell(i)}|-q+1 <$

Algorithm 7: Algorithm for computing 2-gram non-overlapping frequencies from SLP

Input: SLP $\mathcal{T} = \{X_i\}_{i=1}^{n}$ representing string T.
Output: $nOcc(T, P)$ for all 2-grams $P \in \Sigma^2$.

1 Compute $plen(X_i)$, $slen(X_i)$, $pre(X_i, 1)$, and $suf(X_i, 1)$ for all $1 \leq i \leq n$;
2 $z \leftarrow []$; `// list to hold pairs: (2-gram, non-overlapping freq in` X_i`)`
3 **for** $i \leftarrow 1$ **to** n **do**
4   **if** $X_i = X_{\ell(i)} X_{r(i)}$ **then**
5         $a \leftarrow suf(X_{\ell(i)}, 1)$; $b \leftarrow pre(X_{r(i)}, 1)$;
6   **if** $a \neq b$ **then**
7   z.append((ab, $vOcc(X_i)$));
8   **if** $1 < slen(X_{\ell(i)}) < |X_{\ell(i)}|$ **then**
9   z.append((aa, $vOcc(X_i) \cdot \lfloor slen(X_{\ell(i)})/2 \rfloor$));
10   **if** $1 < plen(X_{r(i)}) < |X_{r(i)}|$ **then**
11   z.append((bb, $vOcc(X_i) \cdot \lfloor plen(X_{r(i)})/2 \rfloor$));
12   **else if** $slen(X_{\ell(i)}) < |X_{\ell(i)}|$ **and** $plen(X_{r(i)}) < |X_{r(i)}|$ **then** `// now` $a = b$
13   z.append((aa, $vOcc(X_i) \cdot \lfloor (slen(X_{\ell(i)}) + plen(X_{r(i)}))/2 \rfloor$));
14 $RadixSort(z)$; `// same 2-grams now appear consecutively in` z.
15 Scan z from beginning to end, to sum up occurrences of each distinct 2-gram;

$e < |X_i| - q + 2$. For the special case of $i = n$, we say that (b, e) is closed in X_n if $b < |X_{\ell(i)}| + q$ and $|X_{\ell(i)}| - q + 1 < e$.

Theorem 4. *Problem 2 can be solved in $O(q^2 n)$ time, provided that, for all variables X_i, $(b, e) = loc_q(X_i, j)$ and $nOcc(X_i[b : e], s)$ are already computed for all positions j s.t. $\max\{1, |X_{\ell(i)}| - 2q + 3\} \leq j \leq \min\{|X_{\ell(i)}| + q - 1, |X_i| - q + 1\}$, where $s = X_i[j : j + q - 1]$.*

Proof. Algorithm 8 shows a pseudo-code of our algorithm to solve Problem 2.

Consider q-gram $s = X_i[j : j + q - 1]$ at position j for which $(b, e) = loc_q(X_i, j)$ is closed in X_i. A key observation is that, if (b, e) is closed in X_i, then (b, e) is never closed in $X_{\ell(i)}$ or $X_{r(i)}$. Therefore, by summing up $vOcc(X_i) \cdot nOcc(X_i[b : e], s)$ for each closed (b, e) in X_i, for all such variables X_i, we obtain $nOcc(T, s)$. The range of j implies that all covers (b, e) that satisfy $b < |X_{\ell(i)}| + q$ and $|X_{\ell(i)}| - q + 1 < e$, are considered, and Line 14 is sufficient to check if (b, e) is closed.

For all $1 \leq i \leq n$, $vOcc(X_i)$ can be computed in $O(n)$ time, and $t_i = pre(X_i, 2q - 2) suf(X_i, 2q - 2)$ can be computed in $O(qn)$ time and space. The problem amounts to summing up the values of $vOcc(X_i) \cdot nOcc(X_i[b : e], s)$ for each q-gram s contained in each t_i, and can be reduced to a weighted q-gram frequencies problem on string z and integer array w of length $O(qn)$, which can be solved in $O(qn)$ time by Algorithm 5 in Section 3.2.

Algorithm 8: Computing q-gram non-overlapping frequencies from SLP

Input: SLP $\mathcal{T} = \{X_i\}_{i=1}^{n}$ representing string T, integer $q \geq 2$.
Output: $nOcc(T, P)$ for all q-grams $P \in \Sigma^q$ where $Occ(T, P) \neq \emptyset$.

1 Compute $vOcc(X_i)$ for all $1 \leq i \leq n$;
2 Compute $pre(X_i, 2q-2)$ and $suf(X_i, 2q-2)$ for all $1 \leq i \leq n-1$;
3 $z \leftarrow \varepsilon$; $w \leftarrow []$;
4 **for** $i \leftarrow 1$ **to** n **do**
5   **if** $|X_i| \geq q$ **then**
6   let $X_i = X_\ell X_r$;
7     $k \leftarrow |suf(X_\ell, 2q-2)|$;
8     $t_i = suf(X_\ell, 2q-2)pre(X_r, 2q-2)$;
9   z.append(t_i);
10     $w_i \leftarrow$ create integer array of length $|t_i|$, each element set to 0;
11   **for** $j \leftarrow \max\{1, |X_\ell| - 2q + 3\}$ **to** $\min\{|X_\ell| + q - 1, |X_i| - q + 1\}$ **do**
12       $s \leftarrow X_i[j : j + q - 1]$;
13       $(b, e) \leftarrow loc_q(X_i, j)$;
14   **if** $(q - 1 < b$ **and** $e < |X_i| - q + 2)$ **or** $i = n$ **then**
15   **if** $loc_q(X_i, h) \neq loc_q(X_i, j)$ *for any position* h *s.t.* $\max\{1, |X_\ell| - 2q + 3\} \leq h < j$ **then**
16           $w_i[k - |X_\ell| + j] \leftarrow vOcc(X_i) \cdot nOcc(X_i[b : e], s)$;
17   w.append(w_i);
18 Calculate q-gram frequencies in z, where each q-gram starting at position d is *weighted* by $w[d]$.

In line 15, we check if there is no previous position h ($\max\{1, |X_{\ell(i)}| - 2q + 3\} \leq h < j$) such that $X_i[h : h + q - 1] = X_i[j : j + q - 1]$ by $loc_q(X_i, h) = loc_q(X_i, j)$, so that we do not count the same q-gram more than once. If there is no such h, we set the value of $w_i[k - |X_{\ell(i)}| + j]$ to $vOcc(X_i) \cdot nOcc(X_i[b : e], s)$. This can be checked in $O(q^2 n)$ time for all X_i and j. Hence the theorem holds. □

5.2.2 Computing Longest Overlapping Covers

In this subsection, we will show how to compute longest overlapping cover $(b, e) = loc_q(X_i, j)$ where $s = X_i[j : j + q - 1]$ for all X_i and all j required for Theorem 4.

For any string T and integers q and j ($1 \leq j < q$), let

$$\overrightarrow{loc}_q(T, j) = \begin{cases} (j, be) & \text{if } j + q - 1 \leq N, \\ (j, N) & \text{otherwise,} \end{cases}$$

$$\overleftarrow{loc}_q(T, j) = \begin{cases} (eb, N - j + 1) & \text{if } N - j - q + 2 \geq 1, \\ (1, N - j + 1) & \text{otherwise,} \end{cases}$$

where $(j, be) = (j - 1) \oplus loc_q(T[j : N], 1)$ and $(eb, N - j + 1) = loc_q(T[1 : N - j + 1], N - j - q + 2)$. Namely, $\overrightarrow{loc}_q(T, j)$ is a suffix of the longest overlapping cover of the q-gram $T[j : j + q - 1]$ that begins at position j ($1 \leq j < q$) in T, and $\overleftarrow{loc}_q(T, j)$ is a prefix of the longest overlapping cover of the q-gram $T[N - j - q + 2 : N - j + 1]$ that ends at position $N - j + 1$ in T.

Lemma 14. *For all $1 \leq i \leq n$ and $1 \leq j \leq 2(q - 1)$, $\overrightarrow{loc}_q(X_i, j)$ can be computed in a total of $O(q^2 n)$ time.*

Proof. We use dynamic programming. Let $X_i = X_{\ell(i)} X_{r(i)}$, and assume $\overrightarrow{loc}_q(X_{\ell(i)}, j)$ and $\overrightarrow{loc}_q(X_{r(i)}, j)$ have been calculated for all $1 \leq j \leq 2(q-1)$. We examine the string $X_i[\max(j, |X_{\ell(i)}| - q + 2) : \min(|X_i|, |X_{\ell(i)}| + q - 1)]$ for occurrences of p_j that cross $X_{\ell(i)}$ and $X_{r(i)}$, obtain its longest overlapping cover (b_i, e_i), and check if it overlaps with $\overrightarrow{loc}_q(X_{\ell(i)}, j)$. Furthermore, let bb_r be the left most occurrence of p_j in $X_{r(i)}$ that has the possibility of overlapping with (b_i, e_i). Then, $\overrightarrow{loc}_q(X_i, j)$ is either $\overrightarrow{loc}_q(X_{\ell(i)}, j)$, or its end can be extended to e_i, or further to the end of $\overrightarrow{loc}_q(X_{r(i)}, bb_r)$, depending on how the covers overlap.

More precisely, let $(j, be_\ell) = \overrightarrow{loc}_q(X_{\ell(i)}, j)$, $(b_i, e_i) = \max(j - 1, |X_{\ell(i)}| - q + 1) \oplus loc_q(X_i[\max(j, |X_{\ell(i)}| - q + 2) : \min(|X_i|, |X_{\ell(i)}| + q - 1)], h)$ where $h \in Occ(X_i[\max(j, |X_{\ell(i)}| - q + 2) : \min(|X_i|, |X_{\ell(i)}| + q - 1)], p_j)$, and $(bb_r, be_r) = |X_{\ell(i)}| \oplus \overrightarrow{loc}_q(X_{r(i)}, k)$ where $k = \min Occ(pre(X_{r(i)}, 2(q - 1)), p_j)$. (Note that $(bb_r, be_r), (b_i, e_i)$ are not defined if occurrences h, k of p_j do not exist.) Then we have

$$\overrightarrow{loc}_q(X_i, j) = \begin{cases} (j, be_\ell) & \text{if } be_\ell < b_i \text{ or } \nexists h, \\ (j, e_i) & \text{if } b_i \leq be_\ell \text{ and } (e_i < bb_r \text{ or } \nexists k) \\ (j, be_r) & \text{otherwise.} \end{cases}$$

(See also Figure 5.2.) For all variables X_i we pre-compute $pre(X_i, 3(q-1))$ and $suf(X_i, 3(q-1))$. This can be done in a total of $O(qn)$ time. Then, each $\overrightarrow{loc}_q(X_i, j)$ can be computed in $O(q)$ time using the KMP algorithm, Lemma 13, and the above recursion, giving a total of $O(q^2 n)$ time for all $1 \leq i \leq n$ and $1 \leq j \leq 2(q - 1)$. $\square$

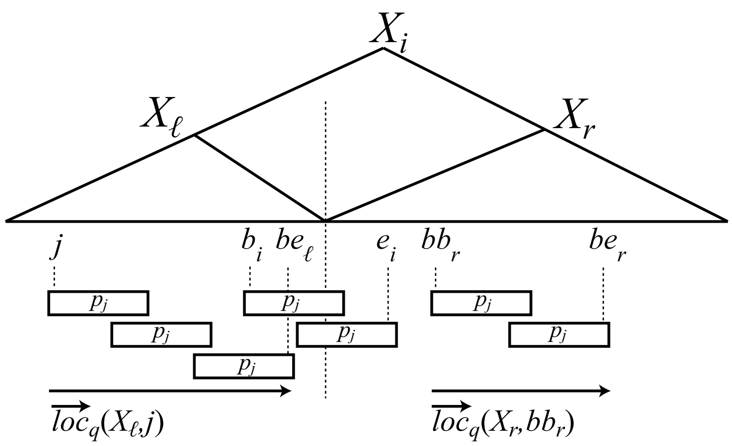


Figure 5.2: Illustration for Lemma 14. In this figure, $\overrightarrow{loc}_q(X_i, j) = (j, e_i)$.

Lemma 15. *For all* $1 \leq i \leq n$ *and* $1 \leq j \leq 2(q-1)$*,* $\overleftarrow{loc}_q(X_i, j)$ *can be computed in a total of* $O(q^2n)$ *time.*

Proof. The proof is essentially the same as the proof for $\overrightarrow{loc}_q(X_i, j)$ in Lemma 14.

Recall that we have assumed in Theorem 4 that $loc_q(X_i, j)$ are already computed. The following lemma describes how $loc_q(X_i, j)$ can actually be computed in a total of $O(q^2n)$ time.

Lemma 16. *For all* $1 \leq i \leq n$ *and* j *s.t.* $|X_{\ell(i)}|-2q+3 \leq j \leq |X_{\ell(i)}|+q-1$*,* $(b, e) = loc_q(X_i, j)$ *can be computed in a total of* $O(q^2n)$ *time.*

Proof.

Let $s_j = X_i[j : j+q-1]$. Firstly, we compute $(b_i, e_i) = loc_q(suf(X_{\ell(i)}, 2q-2)pre(X_{r(i)}, 2q-2), j)$ by Lemma 13, using the KMP algorithm in $O(q)$ time, and then $loc_q(X_i, j)$ can be computed based on (b_i, e_i), as follows: Let $(eb_\ell, ee_\ell) = \overleftarrow{loc}_q(X_{\ell(i)}, h)$ and $(bb_r, be_r) = |X_{\ell(i)}| \oplus \overrightarrow{loc}_q(X_{r(i)}, k)$, where $h = |suf(X_{\ell(i)}, 2q-2)| - (\max Occ(suf(X_{\ell(i)}, 2q-2), s_j) + q - 1) + 1$, $k = \min Occ(pre(X_{r(i)}, 2q-2), s_j)$.

1. If $b_i \leq |X_{\ell(i)}|$ and $e_i > |X_{\ell(i)}|$, then we have $b \leq b_i \leq |X_{\ell(i)}| < e_i \leq e$. $(b, e) = loc_q(X_i, j)$ can be computed by checking whether (eb_ℓ, ee_ℓ), (b_i, e_i), and (bb_r, be_r) are overlapping or not. (See also Figure 5.3.)

2. If $e_i \leq |X_{\ell(i)}|$, then trivially $b = eb_\ell$ and $e = e_i = ee_\ell$. (See also Figure 5.4.)

3. If $b_i > |X_{\ell(i)}|$, then trivially $b = b_i$ and $e = be_r$.

Each $ee_\ell = h$ and $bb_r = |X_{\ell(i)}| + k$ can be computed using the KMP algorithm in $O(q)$ time. By Lemmas 14 and 15, (eb_ℓ, ee_ℓ) and (bb_r, be_r) can be pre-computed in a total of $O(q^2n)$ time for all $1 \leq i \leq n$. Hence the lemma holds. □

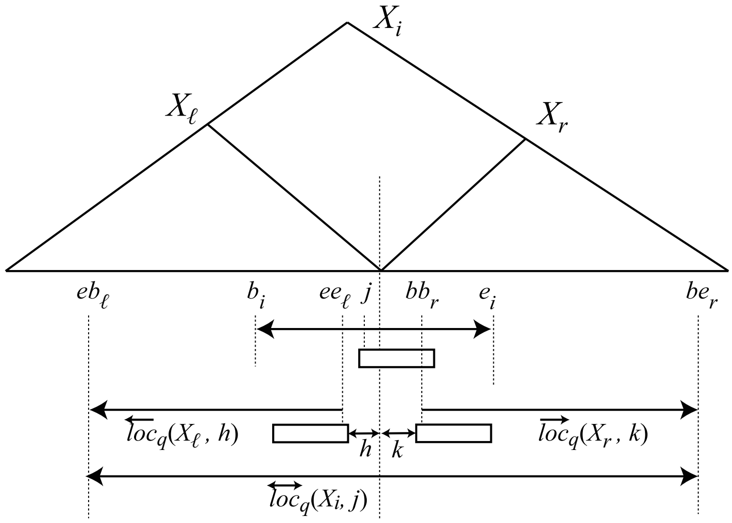


Figure 5.3: Illustration for Lemma 16 case 1. Rectangles show important occurrences of $X_i[j : j+q-1]$. In this case $b = eb_\ell$ and $e = be_r$.

5.2.3 Largest Left-Priority and Smallest Right-Priority Occurrences

In order to compute $nOcc(X_i[b : e], s)$ for all X_i and all j required for Theorem 4, where $(b, e) = loc_q(X_i, j)$ and $s = X_i[j : j+q-1]$, we will use the largest and second largest occurrences of $LnOcc$ and $RnOcc$.

For any set S of integers and integer $1 \leq k \leq |S|$, let $\max_k S$ and $\min_k S$ denote the k-th largest and the k-th smallest element of S.

For $1 \leq i \leq n$ and $1 \leq j \leq 2(q-1)$, consider computing $\max_k LnOcc(X_i[j : be_i], p_j)$ for $k = 1, 2$, where $(j, be_i) = \overrightarrow{loc}_q(X_i, j)$ and $p_j = X_i[j : j+q-1]$. Intuitively, difficulties in computing $\max_k LnOcc(X_i[j : be_i], p_j)$ come from the fact that the string $val(X_i)[j : be_i]$ can be as long as $O(2^n)$, but we only have prefix $pre(X_i, 3(q-1))$ and suffix $suf(X_i, 3(q-1))$ of $val(X_i)$ of length $O(q)$. Hence we cannot compute the value of be_i by simply running the KMP algorithm on those partial strings. For the same reason, the size of $LnOcc(X_i[j : be_i], p_j)$ can be as large as $O(2^n/q)$. Hence we cannot store $LnOcc(X_i[j : be_i], p_j)$ as is. Still, as will be seen in the following lemma, we can compute those values efficiently, only in $O(q^2 n)$ time.

Lemma 17. *For any* $1 \leq i \leq n$ *and* $1 \leq j \leq 2(q-1)$, *let* $(j, be_i) = \overrightarrow{loc}_q(X_i, j)$, $p_j = X_i[j : j+q-1]$. *We can compute the values* $\max_1 LnOcc(X_i[j : be_i], p_j)$ *and* $\max_2 LnOcc(X_i[j : be_i], p_j)$ *for all* $1 \leq i \leq n$ *and* $1 \leq j \leq 2(q-1)$, *in a total of* $O(q^2 n)$ *time.*

Proof. We compute the smallest occurrence b_i in $(j-1) \oplus LnOcc(X_i[j : be_i], p_j)$ that crosses $X_{\ell(i)}$ and $X_{r(i)}$, and does not overlap with the largest occurrence in $(j-1) \oplus LnOcc(X_{\ell(i)}[j : be_\ell], p_j)$, where $(j, be_\ell) = \overrightarrow{loc}_q(X_{\ell(i)}, j)$. Also, we compute the smallest occurrence bb_r in $(j-1) \oplus LnOcc(X_i[j : be_i], p_j)$ that is completely within $X_{r(i)}$ and does not overlap with b_i.

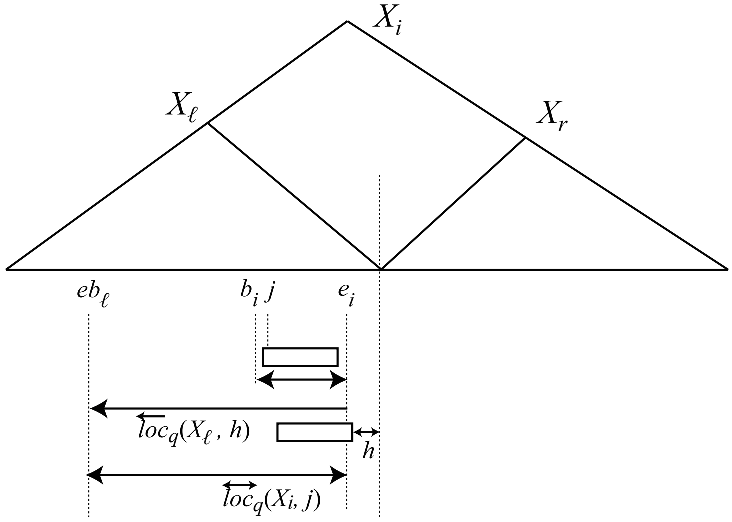


Figure 5.4: Illustration for Lemma 16 case 2. Rectangles show important occurrences of $X_i[j : j+q-1]$. In this case $b = eb_\ell$ and $e = e_i = ee_\ell$.

Then the desired value $\max_1 LnOcc(X_i[j : be_i], p_j)$ can be computed depending whether b_i and bb_r exist or not.

Formally, let Consider the set $S = ((j-1) \oplus LnOcc(X_i[j : be_i], p_j)) \cap [|X_{\ell(i)}|-q+2 : |X_{\ell(i)}|]$ of occurrence of p_j which is either empty or singleton. If S is singleton, then let b_i be its single element. Let $bb_r = \min\{k - |X_{\ell(i)}| \mid k \in (j-1) \oplus LnOcc(X_i[j : be_i], p_j) \cap [|X_{\ell(i)}|+1 : |X_{\ell(i)}|+q-1], \text{if } \exists b_i \text{ then } k \geq b_i + q\}$.

Then we have

$$\max_1 LnOcc(X_i[j : be_i], p_j) \\ = \begin{cases} \max_1 LnOcc(X_{\ell(i)}[j : be_\ell], p_j) & \text{if } \nexists b_i \text{ and } \nexists bb_r \\ b_i - j + 1 & \text{if } \exists b_i \text{ and } \nexists bb_r \\ bb_r - j + \max_1 LnOcc(X_{r(i)}[bb_r : be_r], p_j) & \text{if } \exists bb_r \end{cases}$$

(See also Figure 5.5.)

For all variables X_i we pre-compute $pre(X_i, 3(q-1))$ and $suf(X_i, 3(q-1))$. This can be done in a total of $O(qn)$ time. If b_i or bb_r exists, $|X_{\ell(i)}| - 3(q-1) \leq j - 1 + \max LnOcc(X_{\ell(i)}[j : be_\ell], j) \leq |X_{\ell(i)}| - q + 1$. Then, each b_i and bb_r can be computed from $LnOcc(X_i[(j-1+\max LnOcc(X_{\ell(i)}[j : be_\ell], j)) : |X_{\ell(i)}| + 3(q-1)], p_j)$ runnning the KMP algorithm on string $pre(X_i, 3(q-1))suf(X_i, 3(q-1))$.

Based on the above recursion, we can compute $\max_1 LnOcc(X_i[j : be_i], p_j)$ in a total of $O(q^2 n)$ time for all $1 \leq i \leq n$ and $1 \leq j \leq 2(q-1)$.

It is not difficult to see that similar claims, with slightly different conditions, can be made for

$\max_2 LnOcc(X_i[j : be_i], p_j)$ where the value corresponds to one of 4 values: $\max_2 LnOcc(X_{\ell(i)}[j : be_\ell], p_j)$, $\max_1 LnOcc(X_{\ell(i)}[j : be_\ell], p_j)$, b_i, or $\max_2 LnOcc(X_{r(i)}[bb_r : be_r], p_j)$, with appropriate offsets. □

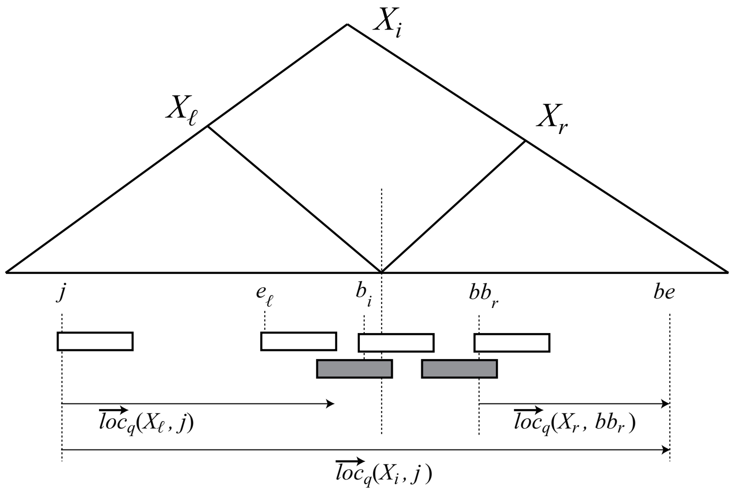


Figure 5.5: Illustration for Lemma 17, calculating $\max LnOcc(X_i[j : be], p_j)$. Shadowed occurrences are not in $LnOcc(X_i[j : be_i], p_j)$, while white ones are in $LnOcc(X_i[j : be_i], p_j)$.

The next lemma can be shown similarly to Lemma 17.

Lemma 18. *For any* $1 \leq i \leq n$ *and* $1 \leq j \leq 2(q-1)$*, let* $(eb, ee) = \overleftarrow{loc}_q(X_i, j)$*, and* $s_j = X_i[|X_i| - j - q + 2 : |X_i| - j + 1]$*. We can compute the values* $\min_1 RnOcc(X_i[eb : ee], s_j)$ *and* $\min_2 RnOcc(X_i[eb : ee], s_j)$ *for all* $1 \leq i \leq n$ *and* $1 \leq j \leq 2(q-1)$*, in a total of* $O(q^2 n)$ *time.*

Lemma 19. *For all* $1 \leq i \leq n$ *and* $1 \leq j < q$*,* $\max LnOcc(X_i[eb_i : ee_i], s_j)$ *can be computed in a total of* $O(q^2 n)$ *time, where* $(eb_i, ee_i) = \overleftarrow{loc}_q(X_i, j)$ *and* $s_j = X_i[|X_i| - j - q + 2 : |X_i| - j + 1]$*.*

Proof. Our basic strategy for computing $\max LnOcc(X_i[eb_i : ee_i], s_j)$ is as follows. Firstly we compute the largest element of $LnOcc(X_i[eb_i : ee_i], s_j)$ that occurs completely within $X_{\ell(i)}$. Secondly we compute the smallest element of $LnOcc(X_i[eb_i : ee_i], s_j)$ that crosses the boundary of $X_{\ell(i)}$ and $X_{r(i)}$. Let d be this occurrence, if such exists. Then the desired output $\max LnOcc(X_i[eb_i : ee_i], s_j)$ is given as either the largest or the second largest element of $LnOcc(X_{r(i)}[d + q : 1], s_j)$.

More formally: We consider the case where $eb_i + q - 1 \leq |X_{\ell(i)}|$. Let $ee_\ell = q - 1 + \max(Occ(X_i, s_j) \cap [|X_{\ell(i)}| - 2q + 2 : |X_{\ell(i)}| - q + 1])$, $m = eb_i - 1 + \max LnOcc(X_{\ell(i)}[eb_i : ee_\ell], s_j)$ where $(eb_i, ee_\ell) = \overleftarrow{loc}_q(X_{\ell(i)}, |X_{\ell(i)}| - (ee_\ell + q - 1) + 1)$. Let $d = m + q - 1 +$

$\min LnOcc(X_i[m+q : ee_i], s_j)$. Let

$$bb_r = \begin{cases} d & \text{if } ee_i - q + 1 \leq |X_{\ell(i)}| \text{ or } d > |X_{\ell(i)}|, \\ d + q - 1 + \min LnOcc(X_i[d+q : |X_i|], s_j) & \text{otherwise.} \end{cases}$$

Let $h' = \max_2 LnOcc(X_i[bb_r : be_r], s_j)$ and $h = \max_1 LnOcc(X_i[bb_r : be_r], s_j)$ where $(bb_r, be_r) = \overrightarrow{loc}_q(X_i, bb_r)$. (See also Figure 5.6.) Then

$$\max LnOcc(X_i[eb_i : ee_i], s_j) = \begin{cases} h & \text{if } h \leq ee_i - q + 1, \\ h' & \text{otherwise.} \end{cases}$$

The case where $eb_i + q - 1 > |X_{\ell(i)}|$ can be solved similarly.

Each ee_ℓ, d and bb_r can be computed in $O(q)$ time using the KMP algorithm, hence requiring a total of $O(q^2 n)$ time. By Lemmas 14 and 15, $\overleftarrow{loc}_q(X_{\ell(i)}, ee_\ell)$ and $\overrightarrow{loc}_q(X_i, bb_r)$ can be computed in $O(q^2 n)$ time for all $X_i = X_{\ell(i)} X_{r(i)}$ and $1 \leq j < n$. By Lemma 17, h' and h can be computed in a total of $O(q^2 n)$ time for all $X_i = X_{\ell(i)} X_{r(i)}$ and $1 \leq j < n$. Therefore, by dynamic programming we can compute $LnOcc(X_i[eb_i : ee_i], s_j)$ in a total of $O(q^2 n)$ time. □

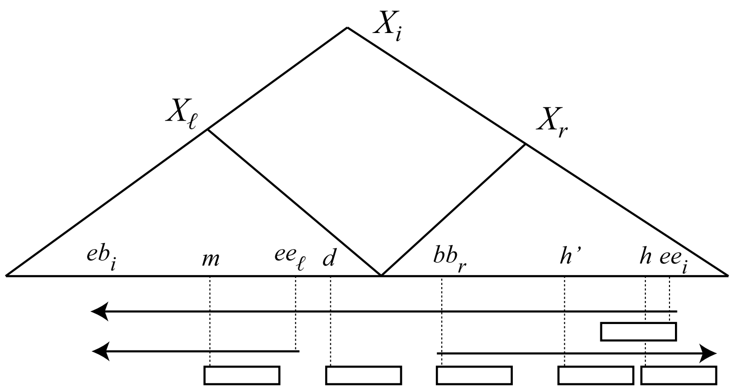


Figure 5.6: Illustration for Lemma 19. Rectangles show important occurrences of s_j. In this case $\max LnOcc(X_i[eb_i, ee_i], s_j) = h'$, as $h > ee_i - q + 1$.

Lemma 20. *For all $1 \leq i \leq n$ and $1 \leq j < q$, $\min RnOcc(X_i[bb_i : be_i], p_j)$ can be computed in a total of $O(q^2 n)$ time, where $(bb_i, be_i) = \overrightarrow{loc}_q(X_i, j)$ and $p_j = X_i[j : j + q - 1]$.*

Proof. The lemma can be shown in a similar way to Lemma 19, using Lemma 18 instead of Lemma 17. □

5.2.4 Counting Non-Overlapping Occurrences in Longest Overlapping Covers

Firstly, we show how to count non-overlapping occurrences of q-gram p_j in $X_i[j : be_i]$, for all i and j, where $p_j = X_i[j : j+q-1]$ and $(j, be_j) = \overrightarrow{loc}_q(X_i[j : be_i], p_j)$.

Lemma 21. *For any* $1 \le i \le n$ *and* $1 \le j \le 2(q-1)$*, let* $(j, be_i) = \overrightarrow{loc}_q(X_i, j)$ *and* $p_j = X_i[j : j+q-1]$*. We can compute* $nOcc(X_i[j : be_i], p_j)$ *for all* $1 \le i \le n$ *and* $1 \le j \le 2(q-1)$*, in a total of* $O(q^2 n)$ *time.*

Proof. By Lemma 1, we have $nOcc(X_i[j : be_i], p_j) = |LnOcc(X_i[j : be_i], p_j)|$. We compute the smallest occurrence b_i in $(j-1) \oplus LnOcc(X_i[j : be_i], p_j)$ that crosses $X_{\ell(i)}$ and $X_{r(i)}$, and does not overlap with the largest occurrence in $(j-1) \oplus LnOcc(X_{\ell(i)}[j : be_\ell], p_j)$, where $(j, be_\ell) = \overrightarrow{loc}_q(X_{\ell(i)}, j)$. Also, we compute the smallest occurrence bb_r in $(j-1) \oplus LnOcc(X_i[j : be_i], p_j)$ that is completely within $X_{r(i)}$ and does not overlap with b_i. Then the desired value $nOcc(X_i[j : be_i], p_j)$ can be computed depending whether b_i and bb_r exist or not.

Formally: Consider the set $S = ((j-1) \oplus LnOcc(X_i[j : be_i], p_j)) \cap [|X_{\ell(i)}| - q + 2 : |X_{\ell(i)}|]$ of occurrence of p_j which is either empty or singleton. If S is singleton, then let b_i be its single element. Let $bb_r = \min\{k - |X_{\ell(i)}| \mid k \in LnOcc(X_i[j : be_i], p_j) \cap [|X_{\ell(i)}| + 1 : |X_{\ell(i)}| + q - 1], \text{if } \exists b_i \text{ then } k \ge b_i + q\}$.

Then we have

$$
\begin{aligned}
&nOcc(X_i[j : be_i], p_j) \\
&= \begin{cases}
nOcc(X_{r(i)}[j - |X_{\ell(i)}| : be_i - |X_{\ell(i)}|], p_j) & \text{if } j > |X_{\ell(i)}|, \\
nOcc(X_{\ell(i)}[j : be_\ell], p_j) & \text{if } \nexists b_i \text{ and } \nexists bb_r, \\
nOcc(X_{\ell(i)}[j : be_\ell], p_j) + 1 & \text{if } \exists b_i \text{ and } \nexists bb_r \\
nOcc(X_{\ell(i)}[j : be_\ell], p_j) + nOcc(X_{r(i)}[b_r : be_r], p_j) & \text{if } \nexists b_i \text{ and } \exists bb_r, \\
nOcc(X_{\ell(i)}[j : be_\ell], p_j) + nOcc(X_{r(i)}[b_r : be_r], p_j) + 1 & \text{if } \exists b_i \text{ and } \exists bb_r,
\end{cases}
\end{aligned}
$$

where $(bb_r, be_r) = \overrightarrow{loc}_q(X_{r(i)}, bb_r)$.

For all variables X_i we pre-compute $pre(X_i, 3(q-1))$ and $suf(X_i, 3(q-1))$. This can be done in a total of $O(qn)$ time. If b_i or bb_r exists, $|X_{\ell(i)}| - 3(q-1) \le j - 1 + \max LnOcc(X_{\ell(i)}[j : be_\ell], j) \le |X_{\ell(i)}| - q + 1$. Then, each b_i and bb_r can be computed from $LnOcc(X_i[(j - 1 + \max LnOcc(X_{\ell(i)}[j : be_\ell], j)) : |X_{\ell(i)}| + 3(q-1)], p_j)$ running the KMP algorithm on string $pre(X_i, 3(q-1))suf(X_i, 3(q-1))$. Based on the above recursion, we can compute $nOcc(X_i[j : be_i], p_j)$ in a total of $O(q^2 n)$ time for all $1 \le i \le n$ and $1 \le j \le 2(q-1)$. $\square$

The next lemma can be shown similarly to Lemma 21.

Lemma 22. *For any* $1 \leq i \leq n$ *and* $1 \leq j \leq 2(q-1)$*, let* $(eb_i, ee_i) = \overleftarrow{loc}_q(X_i, j)$ *and* $s_j = X_i[|X_i| - j - q + 2 : |X_i| - j + 1]$*. We can compute* $nOcc(X_i[eb_i : ee_i], s_j)$ *for all* $1 \leq i \leq n$ *and* $1 \leq j \leq 2(q-1)$*, in a total of* $O(q^2 n)$ *time.*

We have also assumed in Theorem 4 that $nOcc(X_i[b : e], s_j)$ are already computed. This can be computed efficiently, as follows:

Lemma 23. *For all* $1 \leq i \leq n$ *and* j *s.t.* $|X_{\ell(i)}|-2q+3 \leq j \leq |X_{\ell(i)}|+q-1$, $nOcc(X_i[b : e], s_j)$ *can be computed in a total of* $O(q^2 n)$ *time, where* $(b, e) = loc_q(X_i, j)$ *and* $s_j = X_i[j : j+q-1]$*.*

Proof.

We consider the case where $|X_{\ell(i)}| - q + 2 \leq j \leq |X_{\ell(i)}|$, as the other cases can be shown similarly. Our basic strategy for computing $nOcc(X_i[b : e], s_j)$ is as follows. Firstly we compute the largest element of $LnOcc(X_i[b : e], s_j)$ that occurs completely within $X_{\ell(i)}$. Secondly we compute the smallest element of $RnOcc(X_i[b : e], s_j)$ that occurs completely within $X_{r(i)}$. Thirdly we compute an occurrence of s_j that crosses the boundary of $X_{\ell(i)}$ and $X_{r(i)}$, and do not overlap the above occurrences of s_j completely within $X_{\ell(i)}$ and $X_{r(i)}$.

Formally: Let $ee_\ell = b + q - 2 + \max Occ(X_i[b : |X_{\ell(i)}|], s_j)$, $bb_r = \min Occ(X_i[|X_{\ell(i)}| +1 : e], s_j)$, $u_1 = b+q-2+\max LnOcc(X_i[b : ee_\ell], s_j)$, and $u_2 = bb_r-1+\min RnOcc(X_i[bb_r : e], s_j)$. We consider the case where all these values exist, as other cases can be shown similarly. It follows from Lemmas 1 and 2 that

$$
\begin{aligned}
& nOcc(X_i[b : e], s_j) \\
= \;& |LnOcc(X_i[b : u_1], s_j)| + nOcc(X_i[u_1+1 : u_2-1], s_j) + |RnOcc(X_i[u_2 : e], s_j)| \\
= \;& nOcc(X_i[b : ee_\ell], s_j) + nOcc(X_i[u_1 + 1 : u_2 - 1], s_j) + nOcc(X_i[bb_r : e], s_j),
\end{aligned}
$$

(See also Figure 5.7.)

By Lemma 16, $(b, e) = loc_q(X_i, j)$ can be pre-computed in a total of $O(q^2 n)$ time. Since $b < ee_\ell$ and $bb_r < e$, ee_ℓ and bb_r can be computed in $O(q)$ time using the KMP algorithm. By Lemmas 21 and 22 $nOcc(X_i[b : ee_\ell], s_j)$ and $nOcc(X_i[bb_r : e], s_j)$ can be pre-computed in a total of $O(q^2 n)$ time (Notice $(b, ee_\ell) = \overleftarrow{loc}_q(X_{\ell(i)}, ee_\ell)$ and $(bb_r, e) = \overrightarrow{loc}_q(X_{r(i)}, bb_r - |X_{\ell(i)}|) \oplus |X_{\ell(i)}|$). By Lemmas 19 and 20, u_1 and u_2 can be pre-computed in a total of $O(q^2 n)$ time. Hence $nOcc(X_i[u_1 + 1 : u_2 - 1], s_j)$ can be computed in $O(q)$ time using the KMP algorithm for each i and j. The lemma thus holds. □

5.2.5 Main Result

The following theorem concludes this whole section.

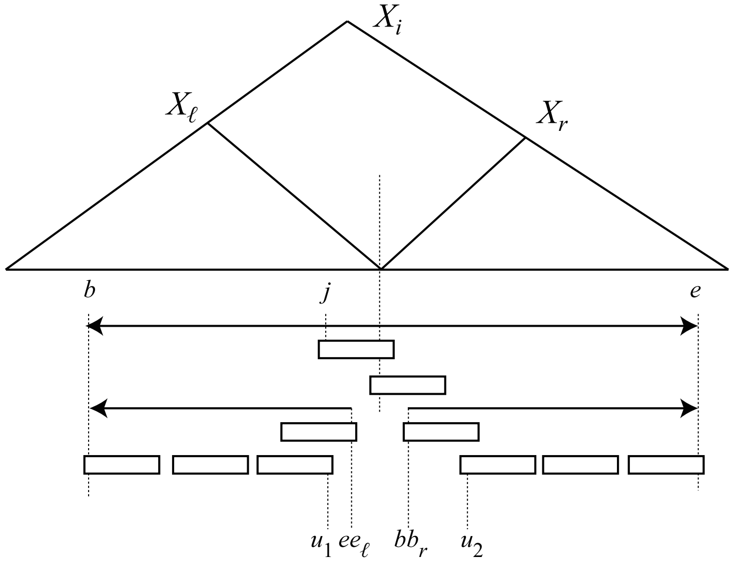


Figure 5.7: Illustration for Lemma 23. Rectangles show important occurrences of $X_i[j : j + q - 1]$. In this case $nOcc(X_i[b : ee_\ell], s_j) = 3$, $nOcc(X_i[u_1 + 1 : u_2 - 1], s_j) = 1$, and $nOcc(X_i[bb_r : e], s_j) = 3$.

Theorem 5. *Problem 2 can be solved in $O(q^2 n)$ time and $O(qn)$ space.*

Proof. The time complexity and correctness follow from Theorem 4, Lemma 16, and Lemma 23.

We compute and store strings $suf(X_i, 3(q-1))$ and $pre(X_i, 3(q-1))$ of length $O(q)$ for each variable X_i, hence this requires a total of $O(qn)$ space for all $1 \le i \le n$. We use a constant number of dynamic programming tables each of which is of size $O(qn)$. Hence the total space complexity is $O(qn)$. $\square$

Chapter 6

Fast Algorithm for LZ77 Factorization

As mentioned in Chapter 1, the runtime of compressed string processing depends on the following two points: The first is the time complexity of algorithms on SLPs, and the second is the size of input SLPs. For the first point, We have developed efficient algorithms for the q-gram frequencies problem on SLPs in Chapter 3,4, and non-overlapping q-gram frequencies problem on SLPs in Chapter 5. In this Chapter, we consider the second point.

Rytter [63] proposed an algorithm that, given the LZ77 factorization of a string T, computes an SLP of size $O(z \log N)$ representing T in output linear time, where z is the size of the LZ77 factorization of T and N is the length of T. This is one of several algorithms which achieve the best known approximation ratio running in linear time. For a string T, we can obtain an SLP of T by firstly computing the LZ77 factorization of T, and then computing an SLP from the LZ77 factorization using Rytter's algorithm. The bottleneck here is the computation of the LZ77 factorization from T. In this Chapter, we develop fast LZ77 factorization algorithms and resolve the above bottleneck.

A naïve algorithm that computes the longest common prefix with each of the $O(N)$ previous positions only requires $O(1)$ working space (excluding the output), but can take $O(N^2)$ time, where N is the length of the string. Using string indice such as suffix trees [71] and on-line algorithms to construct them [69], the LZ77 factorization can be computed in an on-line manner in $O(N \log \sigma)$ time and $O(N \log N)$ bits of space, where σ is the size of the alphabet.

Most recent efficient algorithms are off-line, running in $O(N)$ time for integer alphabets using $O(N \log N)$ bits space (see Table 6.1). They first construct the suffix array [50] of the string, and compute an array called the Longest Previous Factor (LPF) array from which the LZ77 factorization can be easily computed [1, 12, 16, 17, 62]. Many algorithms of this family first compute the longest common prefix (LCP) array prior to the computation of the LPF array. However, the computation of the LCP array is costly. The algorithm CI1 (COMPUTE_LPF) of [15], and the algorithm LZ_OG [62] cleverly avoids its computation and directly computes the LPF array.

Table 6.1: Space usage of linear time LZ77 factorization algorithms based on suffix arrays. Each algorithm uses marked auxiliary integer arrays of size N, and also may use a stack, where the size may become N in the worst case. Merged cells mean that the algorithm uses both auxiliary integer arrays, but either one is rewritten by the other, therefore using a single integer array of size N for the two arrays.

Algorithm	Stack	Integer Arrays of size N							
		# of arrays	LCP	LPF	$PrevOcc$	SA	PSV	NSV	SA^{-1}
CI1 [15]		5		✓	✓	✓	✓	✓	
CI2 [15]	✓	4	✓	✓	✓	✓			
CPS1 [12]	✓	4	✓	✓	✓	✓			
CPS2 [12]	✓	3	✓	✓	✓ (merged $PrevOcc$/SA)				
CPS3 [12]	✓	2	✓		✓ (merged $PrevOcc$/SA)				
CIS [17]	✓	4	✓	✓	✓	✓			
CII [16]	✓	4	✓	✓	✓	✓			
OG [62]		3		✓	✓	✓			
BGS	✓	4				✓	✓	✓	✓
BGL		4				✓	✓	✓	✓
BGT		3				✓	✓	✓	

An important observation here is that the LPF is actually more information than is required for the computation of the LZ77 factorization, i.e., if our objective is the LZ77 factorization, we only use a subset of the entries in the LPF . However, the above algorithms focus on computing the entire LPF array, perhaps since it is difficult to determine beforehand, which entries of LPF are actually required. Although some algorithms such as a variant of CPS1 or CPS2 in [12] avoid computation of LPF, they either require the LCP array, or do not run in linear worst case time and are not as efficient (see [1] for a survey).

In Section 6.1, we propose a new approach to avoid the computation of LCP and LPF arrays altogether, by combining the ideas of the naïve algorithm with those of CI1 and LZ_OG, and still achieve worst case linear time (see Table 6.1). The resulting algorithm is surprisingly both simple and efficient. Computational experiments on various data sets shows that our algorithms constantly outperforms LZ_OG [62], and can be up to 2 to 3 times faster in the processing after obtaining the suffix array, while requiring the same or a little more space.

These results primarily appeared in [23].

Algorithm 9: Naïve Algorithm for Calculating the LZ77 factorization

Input : String T

1 $p \leftarrow 1$;

2 **while** $p \leq N$ **do**

3     $LPF \leftarrow 0$;

4   **for** $j \leftarrow 1, \ldots, p-1$ **do**

5         $l \leftarrow 0$;

6   **while** $T[j+l] = T[p+l]$ **do** $l \leftarrow l+1$; // $l \leftarrow lcp(T[j:N], T[p:N])$

7   **if** $l > LPF$ **then** $LPF \leftarrow l$; $PrevOcc \leftarrow j$;

8   **if** $LPF > 0$ **then Output**: $(LPF, PrevOcc)$

9   **else Output**: $(0, T[p])$

10     $p \leftarrow p + \max(1, LPF)$;

6.1 Algorithm Using Three Integer Arrays

We first describe the naïve algorithm for calculating the LZ77 factorization of a string, and analyze its time complexity. The naïve algorithm does not compute all values of LPF and $PrevOcc$ as explicit arrays, but only the values required to represent each factor. The procedure is shown in Algorithm 9. For a factor starting at position p, the algorithm computes $LPF(p)$ and $PrevOcc(p)$ by simply looking at each of its $p-1$ previous positions, and naively computes the longest common prefix (lcp) between each previous suffix and the suffix starting at position p, and outputs the factor accordingly. At first glance, this algorithm looks like an $O(N^3)$ time algorithm since there are 3 nested loops. However, the total time can be bounded by $O(N^2)$, since the total length of the longest lcp's found for each p in the algorithm, i.e., the total length of the LZ77 factors found, is N. More precisely, let the LZ77 factorization of string T of length N be $f_1 \cdots f_n$, and $p_k = |f_1 \cdots f_{k-1}| + 1$ as before. Then, the number of character comparisons executed in Line 6 of Algorithm 9 when calculating f_k is at most $(p_k - 1)|f_k + 1|$, and the total can be bounded: $\sum_{k=1}^{n}(p_k - 1)|f_k + 1| \leq N \sum_{k=1}^{n} |f_k + 1| = O(N^2)$. An important observation here is that if we can somehow reduce the number of previous candidate positions for naïvely computing lcp's (i.e. the choice of j in Line 4 of Algorithm 9) from $O(N)$ to $O(1)$ positions, this would result in a $O(N)$ time algorithm. This very simple observation is the first key to the linear running times of our new algorithms.

To accomplish this, our algorithm utilizes yet another simple but key observation made in [15]. Since suffixes in the suffix arrays are lexicographically sorted, if we fix a suffix $SA[i]$ in the suffix array, we know that suffixes appearing closer in the suffix array will have longer longest common prefixes with suffix $SA[i]$.

For any position $1 \le i \le N$ of the suffix array, let

$$
\begin{aligned}
PSV_{lex}[i] &= \max(\{0\} \cup \{1 \le j < i \mid SA[j] < SA[i]\}) \\
NSV_{lex}[i] &= \min(\{0\} \cup \{N \ge j > i \mid SA[j] < SA[i]\})
\end{aligned}
$$

i.e., for the suffix starting at text position $SA[i]$, the values $PSV_{lex}[i]$ and $NSV_{lex}[i]$ represent the lexicographic rank of the suffixes that start before it in the string and are lexicographically closest (previous and next) to it, or 0 if such a suffix does not exist. From the above arguments, we have that for any text position $1 \le p \le N$,

$$
\begin{aligned}
LPF(p) = \max(&lcp(suf(SA[PSV_{lex}[SA^{-1}[p]]]), suf(p)), \\
&lcp(suf(SA[NSV_{lex}[SA^{-1}[p]]]), suf(p))).
\end{aligned}
$$

The above observation or its variant has been used as the basis for calculating $LPF(i)$ for all $1 \le i \le N$ in linear time in practically all previous linear time algorithms for LZ77 factorization based on the suffix array. In [62], they consider (implicitly) the arrays in text order rather than lexicographic order. In this case,

$$
\begin{aligned}
PSV_{text}[SA[i]] &= SA[PSV_{lex}[i]] \\
NSV_{text}[SA[i]] &= SA[NSV_{lex}[i]]
\end{aligned}
$$

and therefore

$$
LPF(p) = \max(lcp(suf(PSV_{text}[p]), suf(p)), lcp(suf(NSV_{text}[p]), suf(p))).
$$

While [15] and [62] utilize this observation to compute all entries of LPF in linear time, we utilize it in a slightly different way as mentioned previously, and use it to reduce the candidate positions for calculating $PrevOcc(i)$ (i.e. the choice of j in Algorithm 9) to only 2 positions. The key idea of our approach is in the combination of the above observation with the amortized analysis of the naïve algorithm, suggesting that we can defer the computation of the values of LPF until we actually require them for LZ77 factorization and still achieve linear worst case time. If $PSV_{lex}[i]$ and $NSV_{lex}[i]$ (or $PSV_{text}[i]$ and $NSV_{text}[i]$) are known for all $1 \le i \le N$, the linear running time of the algorithm follows from the previous arguments. The basic structure of our algorithm is shown in Algorithm 10 when using PSV_{lex} and NSV_{lex}. Our algorithm consists of two steps, which we shall call the preliminary step and the parsing step. In the preliminary step Line 1, we compute PSV_{lex} and NSV for all positions and store them in integer arrays. In the parsing step, Line 2-11, we compute the LZ77 factorization of T by using PSV_{lex} and NSV_{lex} arrays. Note that it is easy to replace them with PSV_{text} and NSV_{text}, and in such case,

SA and SA^{-1} are not necessary once we have PSV_{text} and NSV_{text}.

What remains is how to compute $PSV_{lex}[i]$ and $NSV_{lex}[i]$, or $PSV_{text}[i]$ and $NSV_{text}[i]$ for all $1 \leq i \leq N$. This can be done in several ways. We consider 3 variations.

The first is a computation of $PSV_{lex}[i]$, $NSV_{lex}[i]$ using a simple linear time scan of the suffix array with the help of a stack. The procedure is shown in Algorithm 11. This variant requires the text, and the arrays SA, SA^{-1}, PSV_{lex}, NSV_{lex} and a stack. The total space complexity is $(4N + S_{max}) \log N$ bits, where S_{max} is the maximum size of the stack during the execution of the algorithm and can be $\Theta(n)$ in the worstcase. We will call this variant BGS.

The other two is a process called *peak elimination*, which is very briefly described in [15] for lexicographic order (Shown in Algorithms 12 and 13), and in [62] for text order (Shown in Algorithms 14 and 15). In peak elimination, each suffix i and its lexicographically preceding suffix j ($SA^{-1}[j] + 1 = SA^{-1}[i]$) is examined in some order of i (lexicographic or text order). For simplicity, we only briefly explain the approach for text order. If $i > j$, this means that $PSV_{text}[i] = j$ and if $i < j$, $NSV_{text}[j] = i$. When both values of $PSV_{text}[i]$ and $NSV_{text}[i]$ are determined, i is identified as a peak. Given a peak i, it is possible to ***eliminate*** it, and determine the value of either $NSV_{text}[PSV_{text}[i]]$ (which will be $NSV_{text}[i]$ if $PSV_{text}[i] > NSV_{text}[i]$) or $PSV_{text}[NSV_{text}[i]]$ (which will be $PSV_{text}[i]$ if if $PSV_{text}[i] < NSV_{text}[i]$)), and this process is repeated. The algorithm runs in linear time since each position can be eliminated only once. The procedure for lexicographic order is a bit simpler since the lexicographic order of calculation implies that $PSV_{lex}[i]$ will always be determined before $NSV_{lex}[i]$.

The algorithm of [62] actually computes the arrays LPF and $PrevOcc$ directly without computing PSV_{text} and NSV_{text}. The algorithm we show is actually a simplification, deferring the computation of LPF and $PrevOcc$, computing PSV_{text} and NSV_{text} instead.

For lexicographic order, we need the text and the arrays SA, SA^{-1}, PSV_{lex}, NSV_{lex} and no stack, giving an algorithm with $4N \log N$ bits of working space. We will call this variant BGL. For text order, although the Φ array is introduced instead of the SA^{-1} array, the suffix array is not required after its computation. Therefore, by reusing the space of SA for PSV_{text}, the total space complexity can be reduced to $3N \log N$ bits of working space. We will call this variant BGT. Note that although $peakElim_{lex}$ and $peakElim_{text}$ are shown as recursive functions for simplicity, they are tail recursive and thus can be optimized as loops and will not require extra space on the call stack.

6.1.1 Interleaving PSV and NSV

Since accesses to PSV and NSV occur at the same or close indices, it is possible to improve the memory locality of accesses by interleaving the values of PSV and NSV, maintaining them in a single array as follows. Let $PNSV$ be an array of length $2N$, and for each position $1 \leq i \leq 2N$, $PNSV[i] = PSV[j]$ if $i \mod 2 \equiv 0$, $NSV[j]$ otherwise, where $j = \lfloor i/2 \rfloor$.

Algorithm 10: Basic Structure of our Algorithms.

```
Input  : String T
Calculate PSV_lex[i] and NSV_lex[i] for all i = 1...N ;
p ← 1 ;
while p ≤ N do
    LPF ← 0;
    for j ∈ {SA[PSV_lex[SA^{-1}[p]]], SA[NSV_lex[SA^{-1}[p]]]} do
        l ← 0;
        while T[j + l] = T[p + l] do  l ← l + 1;          //  l ← lcp(T[j : N], T[p : N])
        if l > LPF then  LPF ← l; PrevOcc ← j;
    if LPF > 0 then Output: (LPF, PrevOcc)
    else Output: (0, T[p])
    p ← p + max(1, LPF) ;
```

Algorithm 11: Calculating PSV_{lex} and NSV_{lex} from SA

```
Input  : Suffix array SA
Output: PSV_lex, NSV_lex
Let S be an empty stack;
for i ← 1 to N do
    x ← SA[i];
    while (not S.empty()) and (SA[S.top()] > x) do
        NSV_lex[S.top()] ← i; S.pop() ;
    PSV_lex[i] ← if S.empty() then 0 else S.top() ;
    S.push(i);
while not S.empty() do
    NSV_lex[S.top()] ← 0; S.pop() ;
```

Naturally, for any $1 \leq i \leq N$, PSV and NSV can be accessed as $PSV[i] = PNSV[2i]$ and $NSV[i] = PNSV[2i+1]$. This interleaving can be done for both lexicographic order and text order. We will call the variants of our algorithms that incorporate this optimization, iBGS, iBGL, iBGT.

6.2 Computational Experiments

We implement and compare our algorithms with LZ_OG since it has been shown to be the most time efficient in the experiments of [62]. We also implement a variant LZ_iOG which incorporates the interleaving optimization for LPF and $PrevOcc$ arrays. We have made the source codes publicly available at `http://code.google.com/p/lzbg/`.

All computations were conducted on a Mac Xserve (Early 2009) with 2 x 2.93GHz Quad

Algorithm 12: Calculating PSV_{lex} and NSV_{lex} from SA by Peak Elimination.

Input : Suffix array SA

for $i \leftarrow 1$ *to* N **do** $NSV_{lex}[i] \leftarrow 0$;

$PSV_{lex}[1] \leftarrow 0$;

for $i \leftarrow 2$ *to* N **do** $peakElim_{lex}(i-1, i)$;

Algorithm 13: Peak Elimination $peakElim_{lex}(j, i)$ in Lexicographic Order.

if $j = 0$ **or** $SA[j] < SA[i]$ **then**

  $PSV_{lex}[i] \leftarrow j$;

else // $j \geq 1$ **and** $SA[j] > SA[i]$

  $NSV_{lex}[j] \leftarrow i$;

  $peakElim_{lex}(PSV_{lex}[j], i)$; // j was peak.

Core Xeon processors and 24GB Memory, only utilizing a single process/thread at once. The programs were compiled using the GNU C++ compiler (`g++`) 4.2.1 with the `-fast` option for optimization. The running times are measured in seconds, starting from after the suffix array is built, and the average of 10 runs is reported.

We use the data of `http://www.cas.mcmaster.ca/~bill/strings/`, used in previous work. Table 6.2 shows running times of the algorithms, and Table 6.3 shows some statistics of the datasets used in Table 6.2. The running times of the fastest algorithm for each data is shown in bold. The fastest running times for the variant that uses only $13N$ bytes is prefixed with '▷'.

The results show that all the variants of our algorithms constantly outperform LZ_OG and even LZ_iOG for all data tested, and in some cases can be up to 2 to 3 times faster. We can see that iBGS is fastest when the data is not extremely repetitive, and the average length of the factor is not so large, while iBGT is fastest for such highly repetitive data. iBGT is also the fastest when we restrict our attention to the algorithms that use only $13N$ bytes of work space.

Algorithm 14: Calculating PSV_{text} and NSV_{text} from SA using Φ.

Input : Suffix array SA

$\Phi[SA[1]] \leftarrow N$;
for $i \leftarrow 2$ *to* N **do** $\Phi[SA[i]] \leftarrow SA[i-1]$;
for $i \leftarrow 1$ *to* N **do**
  $PSV_{text}[i] \leftarrow \perp$; $NSV_{text}[i] \leftarrow \perp$;
for $i \leftarrow 1$ *to* N **do** $peakElim_{text}(\Phi[i], i)$;

Algorithm 15: Peak Elimination $peakElim_{text}(j, i)$

if $j < i$ **then**
  $PSV_{text}[i] \leftarrow j$;
  if $NSV_{text}[i] \neq \perp$ **then** $peakElim_{text}(j, NSV_{text}[i])$; `// i was peak.`
else `//` $j > i$
  $NSV_{text}[j] \leftarrow i$;
  if $PSV_{text}[j] \neq \perp$ **then** $peakElim_{text}(PSV_{text}[j], i)$; `// j was peak.`

Table 6.2: Running times (seconds) of algorithms for the data set of `http://www.cas.mcmaster.ca/~bill/strings/`.

Algorithm	LZ_OG	LZ_iOG	BGS	iBGS	BGL	iBGL	BGT	iBGT
Use Stack			✓	✓				
# of Integer Arrays of length N	3	3	4	4	4	4	3	3
E.coli	0.64	0.58	0.26	**0.23**	0.33	0.29	0.45	▷ 0.37
bible	0.37	0.34	0.20	**0.19**	0.25	0.22	0.27	▷ 0.24
chr19.dna4	10.05	9.25	4.40	**4.00**	5.33	4.71	7.64	▷ 6.54
chr22.dna4	5.37	4.91	2.27	**2.06**	2.77	2.44	4.09	▷ 3.45
fib_s2178309	0.06	0.06	0.05	0.06	0.06	0.05	0.05	▷ **0.05**
fib_s3524578	0.11	0.11	0.10	0.10	0.10	0.10	0.10	▷ **0.09**
fib_s5702887	0.18	0.18	0.15	0.16	0.16	0.15	0.15	▷ **0.14**
fib_s9227465	0.30	0.30	0.26	0.27	0.27	0.26	0.26	▷ **0.24**
fib_s14930352	0.50	0.49	0.43	0.44	0.44	0.43	0.42	▷ **0.39**
fss9	0.09	0.08	0.08	0.08	0.08	0.08	0.07	▷ **0.07**
fss10	0.40	0.39	0.36	0.37	0.36	0.35	0.34	▷ **0.32**
howto	4.20	3.91	2.30	**2.15**	2.79	2.51	3.28	▷ 2.91
mozilla	5.30	4.95	3.19	**3.13**	3.91	3.65	4.31	▷ 3.86
p1Mb	0.08	0.07	**0.05**	0.05	0.06	0.06	0.05	▷ 0.05
p2Mb	0.23	0.21	**0.11**	0.12	0.15	0.15	0.17	▷ 0.14
p4Mb	0.58	0.52	0.26	**0.26**	0.35	0.33	0.43	▷ 0.35
p8Mb	1.27	1.15	0.55	**0.55**	0.73	0.70	0.94	▷ 0.78
p16Mb	2.70	2.43	1.18	**1.16**	1.52	1.46	2.08	▷ 1.74
p32Mb	5.58	5.02	2.47	**2.44**	3.14	3.03	4.43	▷ 3.74
rndA2_4Mb	0.49	0.45	0.20	**0.18**	0.24	0.20	0.35	▷ 0.28
rndA2_8Mb	1.08	0.99	0.42	**0.38**	0.50	0.43	0.77	▷ 0.63
rndA21_4Mb	0.64	0.58	0.28	**0.28**	0.38	0.37	0.47	▷ 0.37
rndA21_8Mb	1.43	1.28	0.61	**0.60**	0.83	0.79	1.05	▷ 0.85
rndA255_4Mb	0.65	0.58	**0.38**	0.39	0.51	0.47	0.49	▷ 0.40
rndA255_8Mb	1.43	1.27	0.84	**0.84**	1.12	1.04	1.10	▷ 0.92

Table 6.3: Statistics of the Data used in Table 6.2. S_{max} is maximum stack size used in BGS and iBGS. The last two columns show $\sum_i |i - PSV_{lex}[i]|/N$ and $\sum_i |i - NSV_{lex}[i]|/N$.

File Name	Alphabet Size	Text Size N	# of LZ factors	Average Length of Factor	S_{max}	Average Distance of PSV_{lex}	Average Distance of NSV_{lex}
E.coli	4	4638690	432791	10.72	36	14.49	13.94
bible	63	4047392	337558	11.99	42	16.14	15.32
chr19.dna4	4	63811651	4411679	14.46	58	16.97	17.51
chr22.dna4	4	34553758	2554184	13.53	43	16.25	15.04
fib_s2178309	2	2178309	31	70268	16	10.16	10.16
fib_s3524578	2	3524578	32	110143	16	10.95	10.57
fib_s5702887	2	5702887	33	172815	17	10.88	10.88
fib_s9227465	2	9227465	34	271396	17	11.67	11.29
fib_s14930352	2	14930352	35	426581	18	11.61	11.61
fss9	2	2851443	40	71286.10	22	10.83	10.73
fss10	2	12078908	44	274521	24	11.96	11.88
howto	197	39422105	3063929	12.87	616	20.17	21.24
mozilla	256	51220480	6898100	7.43	3964	21.58	104.46
p1Mb	23	1048576	216146	4.85	38	13.41	13.50
p2Mb	23	2097152	406188	5.16	40	14.17	14.28
p4Mb	23	4194304	791583	5.30	42	14.89	14.93
p8Mb	23	8388608	1487419	5.64	898	50.97	15.68
p16Mb	23	16777216	2751022	6.10	898	33.93	16.38
p32Mb	24	33554432	5040051	6.66	898	25.81	17.08
rndA2_4Mb	2	4194304	201910	20.77	36	13.48	14.33
rndA2_8Mb	2	8388608	385232	21.78	37	13.02	15.19
rndA21_4Mb	21	4194304	970256	4.32	34	13.59	13.025
rndA21_8Mb	21	8388608	1835235	4.57	37	14.76	14.32
rndA255_4Mb	255	4194304	2005584	2.09	35	14.07	13.23
rndA255_8Mb	255	8388608	3817588	2.20	38	13.59	14.68

Chapter 7

Space Efficient Algorithm for LZ77 Factorization

In Chapter 6, we proposed fast linear time LZ77 factorization algorithms that avoid the computation of LCP and LPF arrays. As well as developing fast algorithm, developing space efficient algorithm is also important applicable to large-scale string data. In this Chapter, we develop space efficient linear time LZ77 factorization algorithms, which also avoid the computation of LCP and LPF arrays.

We note that algorithms that avoid the computation of LCP and LPF based on a similar idea was developed independently and almost simultaneously by Kempa and Puglisi [41] and Kärkkäinen et al [36]. The algorithm of [41] is fast and space efficient, however the worst case time complexity of it depends on the alphabet size. In [36], three algorithms KKP3, KKP2, and KKP1 are proposed which respectively store and utilize 3, 2, and 1 auxiliary integer arrays of length N kept in main memory. KKP3 can be seen as a reengineering of BGT in Chapter 6, that is modified so that array access are more cache friendly, thus making the algorithm run faster. KKP2 is based on KKP3, but further reduces one integer array by an elegant technique that rewrites values on the integer array. KKP1 is the same as KKP2, except that it assumes that the suffix array is stored on disk, but since the values of the suffix array are only accessed sequentially, the suffix array is streamed from the disk. Thus, KKP1 can be regarded as requiring only a single integer array to be held in memory. In this sense, KKP1 is the most space economical among the existing linear time algorithms, and has been shown to be faster than KKP2, if it is assumed that the suffix array is already computed and exists on disk [36]. However, note that the *total* space requirement of KKP1 is still two integer arrays, one existing in memory and the other existing on disk.

We further improve the results of [36] to reduce the working space. we propose new algorithms that use only $N \log N + O(\sigma \log N)$ bits of space, i.e., a single auxiliary integer array of length N and a number of integer arrays of length σ, where σ is the size of the alphabet.

We achieve this by introducing a series of techniques for rewriting the various auxiliary integer arrays from one to another, in linear time using only $O(\sigma \log N)$ bits of extra working space. Computational experiments show that our algorithm is at most around two to three times as slow as previous algorithms, but in turn, uses only half the total space. Thus, our algorithm may be a viable alternative when the total available space (including disk) is a limiting factor due to the enormous size of data. Note that while the space complexity of our algorithm depends on σ, the time complexity does not.

Our new algorithm partly uses the idea of KKP2. We firstly describe overview of the KKP algorithms [36] in Section 7.1, and secondly propose new algorithm using $2N \log N$ bits of working space in different way of KKP2 in Section 7.2, and finally propose new algorithms using $N \log N + O(\sigma \log N)$ bits of working space by combining these two algorithms in Section 7.3.

These results primarily appeared in [24].

7.1 Overview of the KKP Algorithms

We first describe the LZ77 factorization algorithms by Kärkkäinen et al. [36]: KKP3, KKP2, and KKP1.

Their approach is very similar to ours in the terms of avoiding to compute LPF and $PrevOcc$ arrays, their algorithm compute PSV_{text} and NSV_{text} arrays in the preliminary step, and compute the LZ77 factorization by using these auxiliary integer arrays in parsing step. KKP3 compute PSV_{text} and NSV_{text} arrays in linear time in similar way of Algorithm 14. The most difference is the computation of PSV_{text} and NSV_{text} in preliminary step. BGT computes each values of PSV_{text} and NSV_{text} in text order, on the other hand, KKP3 computes each values of PSV_{text} and NSV_{text} in lexicographic order. Thefore KKP3 does not need the Φ array and it can compute PSV_{text} and NSV_{text} just in one sequential scan left to right of SA (see Algorithm 16). In this way, KKP3 runs in linear time using a total of 3 auxiliary integer arrays ($SA, PSV_{text}, NSV_{text}$) of length N.

For KKP2, Kärkkäinen et al. show that the parsing step can be accomplished by using only the NSV_{text} array. The idea is based on a very interesting connection between PSV_{text}, NSV_{text}, and $\varPhi$ arrays. They showed that starting from the NSV_{text} array, it is possible to sequentially scan and rewrite the NSV_{text} array (consequently to the $\varPhi$ array) in-place, during which, values of PSV_{text} (and naturally NSV_{text}) for each position can be obtained sequentially as well.

Lemma 24 ([36]). *Given the NSV_{text} array of a string T of length N, $PSV_{text}(i)$ and $NSV_{text}(i)$ of T can be sequentially obtained for all positions $i = 1, \ldots, N$ in $O(N)$ total time using $O(\log N)$ bits space other than the NSV_{text} array and T.*

Algorithm 16: Computation of PSV_{text} and NSV_{text} from SA.

Input : SA

$SA[N+1] \leftarrow 0$;
$prev \leftarrow 0$;
for $i \leftarrow 1$ **to** $N+1$ **do**
    while $prev > SA[i]$ **do**
        $NSV_{text}[prev] = i$;
        $prev \leftarrow PSV_{text}[prev]$;
        ; `// peak elimination`
    $PSV_{text}[i] \leftarrow prev$;
    $prev \leftarrow i$;

Algorithm 17: In-place computation of NSV_{lex} from Φ.

Input : Φ array (denoted as NSV_{lex})

$cur \leftarrow NSV_{lex}[1]$; `//` $\Phi[1]$`: lexicographically largest suffix`
$prev \leftarrow 0$;
while $cur \neq 0$ **do**
    while $cur < prev$ **do**
        $prev \leftarrow NSV_{lex}[prev]$; `// peak elimination`
    $next \leftarrow NSV_{lex}[cur]$; `//` $\Phi[cur]$
    $NSV_{lex}[cur] \leftarrow prev$;
    $prev \leftarrow cur$; $cur \leftarrow next$;

By making use of this technique, only the NSV_{text} array is now required for the parsing step. KKP2 uses 2 integer arrays (SA and NSV_{text}) of length N in the preliminary step, and 1 integer array (NSV_{text}) of length N in the parsing step, and thus in summary, KKP2 runs in linear time using a total of 2 auxiliary integer arrays of length N.

The memory bottleneck of KKP2 is the computation of the NSV_{text} array in the preliminary step. Since each value in SA are only used sequentially and once each, KKP1 partly overcomes this problem by first storing SA to disk, and then streams the SA from the disk, storing only the NSV_{text} array in main memory. Thus, KKP1 runs in linear time keeping only 1 auxiliary integer array of length N in *main* memory, although of course, the total storage requirement is still 2 integer arrays (SA and NSV_{text}).

7.2 Algorithm Using Two Integer Arrays

In this section, we describe our linear time LZ77 factorization algorithm that uses two auxiliary integer arrays of length N in the different way of algorithms by Kärkkäinen et al.. We call the algorithm that uses two integer arrays of length N BGtwo (see Figure 7.1).

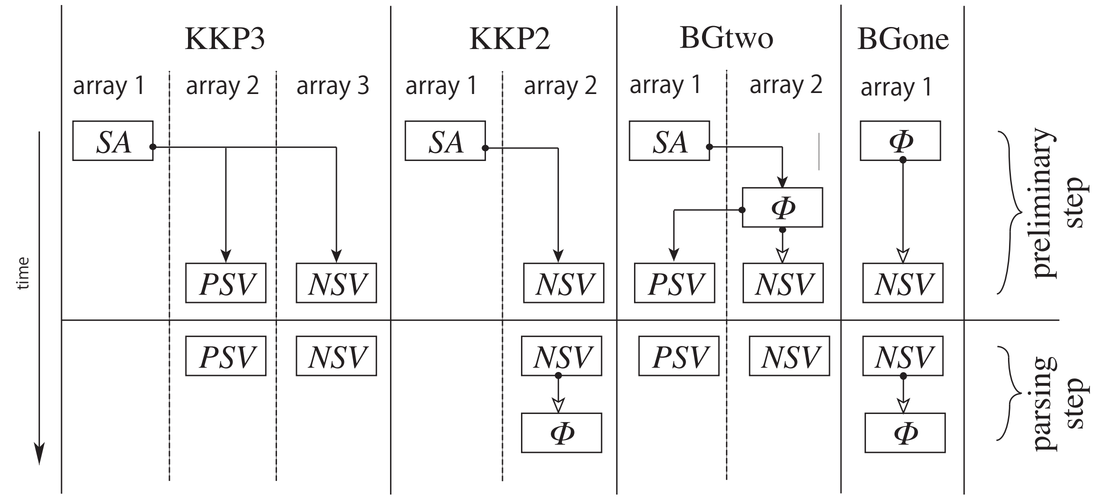


Figure 7.1: A comparison of the auxiliary arrays used and how their contents change with time for the KKP variants and our algorithm.

KKP2 scans the SA sequentially to compute the NSV_{text} array. If possible, we would like to compute the NSV_{text} array from SA in-place. However, this seems difficult, since while the values of SA are in lexicographic order, the values of NSV_{text} array are in text order. To solve this problem, we consider the Φ array. Since $\Phi[i]$ for each i indicates lexicographic predecessor of $suf(i)$, we can sequentially access values of SA from right to left, by accessing Φ starting from the lexicographically largest suffix, which is $\Phi[1] = SA[N]$, that is, Φ can be regarded as an array based implementation of a singly linked list, linking the elements of SA from right to left. Thus, the algorithm for computing NSV_{text} from SA can be simulated using the Φ array. An important difference is that while elements of SA are in lexicographic order, elements of Φ are in text order, which is the same as NSV_{text}. Also, since the access on SA is sequential, the value $\Phi[i]$ is not required anymore after it is processed, and we can rewrite $\Phi[i]$ to $NSV_{text}[i]$ in-place. The pseudo code of the algorithm is shown in Algorithm 17. The correctness and running time follows from the above arguments.

Lemma 25. *Given the Φ array of a string T, NSV_{text} array of T can be computed from Φ in linear time and in-place using $O(\log N)$ bits of working space.*

7.3 Algorithm Using a Single Integer Array

In the previous section, we showed that the NSV_{text} array can be computed by rewriting Φ array in-place in linear time. As described in Section 7.1, $PSV_{text}(i)$ and $NSV_{text}(i)$ can be sequentially obtained by rewriting NSV_{text} array to Φ array, and compute the LZ77 factorization in linear time. By combining the two algorithms (combining Lemma 24 and Lemma 25), we obtain our main result.

Theorem 6. *Assuming an integer alphabet of size σ, the LZ77 factorization of a string of length N can be computed in $O(N)$ time using of $N \log N + O(\sigma \log N)$ bits of total working space.*

The problem is now how to compute the Φ array. Although the Φ array can easily be computed in linear time by a naive sequential scan on SA, storage for both the input SA and output Φ array is required for such an approach, as in the case of computing NSV_{text} from SA. As far as we know, an in-place linear time construction algorithm for the Φ array has not yet been proposed. Below, we propose the first such algorithm. As noted in the previous subsection, the Φ array can be considered as an alternative representation of SA, which allows us to simulate a sequential scan on the SA. Thus, in order to construct Φ in-place, our algorithm simulates the in-place suffix array construction algorithm by Nong [60] which runs in linear time and $O(\sigma \log N)$ bits of extra working space. We first describe the outline of the algorithm by Nong for computing SA, and then describe how to modify this to compute the Φ array.

7.3.1 Construction of the Suffix Array by Induced Sorting [60]

Nong's algorithm is based on induced sorting, which is a well known technique for linear time suffix sorting. Induced sorting algorithms first sort a certain subset of suffixes, either directly or recursively, and then induces the lexicographic order of the remaining suffixes by using the lexicographic order of the subset. There exist several methods depending on which subset of suffixes to choose. Nong's algorithm utilizes the concept of LMS suffixes defined below.

Definition 3. *For $1 \leq i \leq N-1$, a suffix $suf(i)$ is an L-suffix if $suf(i)$ is lexicographically larger than $suf(i+1)$, and an S-suffix otherwise. We call S or L the* type *of the suffix. An S-suffix $suf(i+1)$ is a Left-Most-S-suffix (LMS-suffix) if $suf(i+1)$ is an S-suffix and $suf(i)$ is an L-suffix.*

Recall that $T[N] = \$$, where \$ is a special delimiter character that does not occur elsewhere in the string. We define $suf(N)$ to be an S-suffix. Notice that for $i \leq N-1$, $suf(i)$ is an S-suffix iff $T[i] < T[i+1]$, or $T[i] = T[i+1]$ and $suf(i+1)$ is an S-suffix. The type of each suffix can be determined by scanning T from right to left.

In SA, all suffixes starting with the same character c occur consecutively, and we call the interval on the suffix array of such suffixes, the c-interval. A simple observation is that the L-suffixes that start with some character c must be lexicographically smaller than all S-suffixes that start with the same character c. Thus a c-interval can be partitioned into two sub-intervals, which we call the L-interval and S-interval of c.

The induced sorting algorithm consists of the following steps. We denote the working array to be SA, which will become the suffix array of the text at the end of the algorithm. In steps 2-4, we use integer arrays of size σ to store the interval of a character c. If we have these arrays, we can insert each suffix to its c-interval from left to right or right to left in turn, each insertion taking $O(1)$ time. These arrays can be computed by first scanning T from right to left, and counting all characters, and then summing the values in lexicographic order to obtain each interval.

1. Sort the LMS-suffixes.
   We call the result LMS_SA. We omit details of how this is computed, since our algorithm will use the algorithm described in [60] as is, but it may be performed in linear time using $O(\log N)$ bits of extra working space. We assume that the result LMS_SA is stored in the first k elements of SA, i.e. $SA[1..k]$, where k is the number of LMS-suffixes.

2. Put each LMS-suffix into the S-interval of its first character, in the same order as LMS_SA.

   All values in $SA[k+1..N]$ are initially set to $EMPTY$. By a right to left scan on LMS_SA (i.e. $SA[1..k]$), we put each LMS-suffix $suf(i)$ in the right most empty element of the S-interval.

3. Sort and put the L-suffixes in their proper positions in SA.
   This is done by scanning SA from left to right. For each position i, if $SA[i] > 1$ and $suf(SA[i]-1)$ is an L-suffix, $suf(SA[i]-1)$ is put in the left-most empty position of the L-interval for character $T[SA[i]-1]$. The correctness of the algorithm follows from the fact that if suffix $suf(SA[i]-1)$ is an L-suffix, then, $suf(SA[i])$ must have been located before i (in the correct order), in SA.

4. Sort and put the S-suffixes in their proper positions in SA.
   This is done by scanning SA from right to left. For a position i, if $SA[i] > 1$ and $suf(SA[i]-1)$ is an S-suffix, $suf(SA[i]-1)$ is put in the right most empty position of the S-interval for character $T[SA[i]-1]$. The correctness of the algorithm follows from the fact that if suffix $suf(SA[i]-1)$ is an S-suffix, then, $suf(SA[i])$ must have been located after i (in the correct position), in SA.

In total, the algorithm computes suffix array in linear time using only a single integer array of length N, and $O(\sigma \log N)$ bits of extra working space. Note that for any position i, determining whether suffix $suf(SA[i]-1)$ is an L-suffix or not, can be done in $O(1)$ time using no extra space. If $T[SA[i]-1] < T[SA[i]]$ then it is an S-suffix, and if $T[SA[i]-1] > T[SA[i]]$ then it is an L-suffix. For the case of $T[SA[i]-1] = T[SA[i]]$, the type of $suf(SA[i]-1)$ is the same as that of $suf(SA[i])$, which can be determined by the position i, and the start and end positions of the L- and S-intervals of character $T[SA[i]]$.

7.3.2 Construction of the Φ array by induced sorting

We regard Φ as an array based implementation of a singly linked list containing elements of SA from right to left. The basic idea of our algorithm to construct the Φ array is to modify Nong's algorithm for computing SA, to use this list representation instead. However, there are some technicalities that need to be addressed.

We denote the working array to be A, which will be an array based representation of a singly linked list that links (in lexicographic order) the set of so-far sorted suffixes at each step, and will become the Φ array of the text at the end of the algorithm. The algorithm is described below.

1. Sort the LMS-suffixes.
   First, we sort LMS-suffixes in the same way as [60]. The result will be called LMS_SA and stored in $A[1..k]$, where k is the number of LMS-suffixes.

2. Transform LMS_SA to the array based linked list representation
   To simulate the algorithm for SA, we firstly need linked list representation of LMS_SA such that each value indicates the lexicographically succeeding LMS-suffix. For each LMS-suffix $suf(LMS\_SA[i])$, its succeeding LMS-suffix $suf(LMS\_SA[i+1])$ will be put in $A[LMS\_SA[i]]$, i.e., $A[LMS\_SA[i]] = LMS\_SA[i+1]$ for $i < k$. If LMS_SA and A were different arrays, then we could simply set $A[LMS\_SA[i]] = LMS\_SA[i+1]$ for each $i < k$. The problem here is that since LMS_SA is stored in $A[1..k]$, when setting a value at some position of A, we may overwrite a value of LMS_SA which has not been used yet. We overcome this problem as follows.

   First, we memorize $LMS\_SA[1]$, the first value of LMS_SA. Then, for $1 \le i \le k$, we set $A[2i] = LMS\_SA[i]$ and $A[2i-1] = EMPTY$ by scanning $A[1..k]$ from right to left. Since k never exceeds $N/2$, we have $2i \le N$ for all $1 \le i \le k$.

   Next, for $1 \le i \le k-1$, let $j_1 = A[2i](= LMS\_SA[i])$ and $j_2 = A[2(i+1)](= LMS\_SA[i+1])$. We attempt to set $A[j_1] = j_2$. If $A[j_1] = EMPTY$, then we simply set $A[j_1] = j_2$. Otherwise $j_1 = 2i'$ for some $1 \le i' \le k$, and $A[j_1]$ stores the value $LMS\_SA[i']$. Therefore, we do not overwrite this value, but instead, borrow the space immediately left of position j_1, and set $A[j_1 - 1] = j_2$. An important observation is that $A[j_1 - 1]$ must have been $EMPTY$, because LMS-suffixes cannot, by definition, start at consecutive positions, and if j_1 was an LMS suffix, $j_1 - 1$ cannot be an LMS suffix and the algorithm will never try to set another value at this position.

   After this, we set $A[2i] = EMPTY$ for all $1 \le i \le k$, and we arrange the remaining values to their correct positions by attempting to traverse succeeding suffixes stored in A from the lexicographically smallest suffix of LMS_SA memorized at the beginning of the process. Let i be the current position we are traversing. We attempt to obtain its succeeding suffix by reading $A[i]$. If $A[i] \neq \mathit{EMPTY}$, the succeeding suffix of $\mathit{suf}(i)$ was stored at the correct position, and we continue with the next position $A[i]$. If $A[i] = \mathit{EMPTY}$, then the succeeding suffix of $\mathit{suf}(i)$ may be stored at the immediately left position, i.e. $A[i-1]$. In such a case, $A[i-1] \neq \mathit{EMPTY}$, and we set $A[i] = A[i-1]$ and $A[i-1] = \mathit{EMPTY}$, and continue with the next position $A[i]$. If $A[i-1] = \mathit{EMPTY}$, this means that $\mathit{suf}(i)$ is the lexicographically largest suffix of LMS-suffixes, and we finish the process.

In this way, for all LMS-suffixes $\mathit{suf}(i)$, we can set the succeeding suffix at $A[i]$. The process essentially scans the values of $\mathit{LMS\_SA}$ on A twice. Therefore, this step runs in $O(k)$ time and $O(\log N)$ bits of working space.

3. Sort and put the L-suffixes in their proper positions in A.
   To simulate the algorithm for SA, we need to scan the suffixes in lexicographically increasing order by using A. Let $\mathit{suf}(i)$ be a suffix the algorithm is processing. We want to set $A[j] = i-1$ if $\mathit{suf}(i-1)$ is an L-suffix, and $\mathit{suf}(j)$ is the suffix that lexicographically precedes suffix $\mathit{suf}(i-1)$.

   To accomplish this, we introduce four integer arrays of size σ each, $\mathit{Lbkts}[c]$, $\mathit{Lbkte}[c]$, $\mathit{Sbkts}[c]$ and $\mathit{Sbkte}[c]$. $\mathit{Lbkts}[c]$ and $\mathit{Lbkte}[c]$ store the lexicographically smallest and largest suffix of the L-interval for a character c which have been inserted into A, and $\mathit{Sbkts}[c]$ and $\mathit{Sbkte}[c]$ are the same for each S-interval. All values are initially set to EMPTY. We first scan the list of LMS suffixes in lexicographically increasing order represented in A constructed in the previous step, and insert each LMS suffixes into the corresponding S-interval, by updating $\mathit{Sbkts}[c]$ and $\mathit{Sbkts}[e]$. Then, we scan all LMS- and L-suffixes in lexicographically increasing order by traversing the succeeding suffixes on A by starting from $\mathit{Lbkts}[c]$, traversing the list represented by A until we process $\mathit{Lbkte}[c]$. Then we do the same starting from $\mathit{Sbkts}[c]$ and process the suffixes until we reach $\mathit{Sbkte}[c]$, and repeat the process for all characters c in lexicographic order.

   Let $\mathit{suf}(i)$ be a suffix the algorithm is currently processing. We store $\mathit{suf}(i-1)$ in the appropriate position of A, if $\mathit{suf}(i-1)$ is an L-suffix, and do nothing otherwise. Since we know the type of suffix $\mathit{suf}(i)$ since we are either processing a suffix between $\mathit{Lbkts}[c]$ and $\mathit{Lbkte}[c]$ or $\mathit{Sbkts}[c]$ and $\mathit{Sbkte}[c]$, the type of $\mathit{suf}(i-1)$ can be determined in constant time by simply comparing $T[i-1]$ and $T[i]$, i.e. it is an L-suffix if $T[i-1] > T[i]$, an S-suffix if $T[i-1] < T[i]$, and has the same type as $\mathit{suf}(i)$ if $T[i-1] = T[i]$.

   When storing $\mathit{suf}(i-1)$ in A, we check $\mathit{Lbkts}[T[i-1]]$. If $\mathit{Lbkts}[T[i-1]] = \mathit{EMPTY}$, then, $\mathit{suf}(i-1)$ is the lexicographically smallest suffix starting with $T[i-1]$. We set $\mathit{Lbkts}[T[i-1]] = \mathit{Lbkte}[T[i-1]] = i-1$. Otherwise, there is at least one suffix lexicographically smaller than $suf(i-1)$ in the L-interval for character $T[i-1]$. This suffix is $Lbkte[T[i-1]] = j$, and we set $A[j] = i-1$, and update $Lbkte[T[i-1]] = i-1$.

In this way we can compute all the lexicographically succeeding suffixes of each L-suffixes in the corresponding L-interval, and store them in A. Since the number of times we read the values of A is at most the number of LMS- and L-suffixes, and the updates for each new L-suffix can be done in $O(1)$ time, the algorithm runs in linear time using only a single integer array and $O(\sigma \log N)$ bits of working space in total.

4. Sort and put the S-suffixes in their proper positions in A.
   To simulate the algorithm for SA, we need to scan all L-suffixes in lexicographically decreasing order by using A. However, since the linked list of L-suffixes constructed on A in the previous step is in increasing order, we first rewrite A to reverse the direction of the links. That is, we want to set $A[j] = i-1$ if $suf(i-1)$ is an L-suffix and $suf(j)$ is the suffix that lexicographically succeeds suffix $suf(i-1)$.

   This rewriting can be done by scanning the succeeding suffixes in a similar way as that of Step 3. For each c in lexicographically increasing order, traverse the L-suffixes by using $Lbkts[c]$, $Lbkte[c]$, and A, and simply rewrite the values in A to reverse the links, i.e., if $suf(j)$ preceded $suf(i)$ then $A[i] = j$.

   Now we have a lexicographically decreasing list of L-suffixes represented in A, and insert the S-suffixes into A similar to that of Step 3. After that all suffixes have been inserted and linked, we can obtain all suffixes in decreasing order by traversing preceding suffixes on A, i.e. A is now equal to the Φ array. Similarly to the previous step, we can see that this step runs in linear time using one integer array of length N (A) and $O(\sigma \log N)$ bits of extra space.

All steps run in linear time using A and $O(\sigma \log N)$ bits extra space, thus giving a linear time algorithm for computing Φ using $O(\sigma \log N)$ bits of extra working space.

The above procedure describes how to construct Φ from T in linear time using $O(\sigma \log N)$ bits of working space. Although we omit the details, it is possible to compute Φ by rewriting SA in-place, in linear time and $O(\sigma \log N)$ bits of working space. This could seem useless, but may have applications when the SA is already available, since the conversion does not require the expensive recursion step as in the linear time SA construction algorithm (in Step 1), but can be achieved in a few scans.

7.4 Computational Experiments

We implemented BGtwo and two variations of BGone. These are different in the computation of the Φ array. One computes the Φ array directly from T (BGoneT), and the other first computes SA and then computes the Φ array from SA (BGoneSA). The 3 implementations are available at `http://code.google.com/p/bgone/`. We compared our algorithms with the implementation of KKP1, KKP2, and KKP3 [1], and also LZScan and LZISA6s which are not linear time algorithms, but are practically fast and use less space. LZScan runs in $O(dN)$ time and $O((N \log N)/d)$ bits of working space, where d is a parameter that determines the space-time trade-off. In our experiments, d is chosen so that LZScan uses at least and as close to the amount of space that BGoneT uses. LZISA6s runs in $O(N \log \sigma)$ time and $(1+\epsilon)N \log N + N + O(\sigma \log N)$ bits of space. We use SACA-K which is the implementation of Nong's algorithm to compute LMS_SA in BGoneT, and the faster of SACA-K and divsufsort to compute SA in all other implementations. The theoretical work space required for SACA-K is $\sigma \log N + O(\log N)$ bits, and $O(\log N)$ bits for divsufsort [2]. Note that BGoneT has a disadvantage, but these conditions were chosen since the latter algorithms can choose any suffix array construction algorithm, while BGoneT cannot.

All computations were conducted on a Mac Xserve (Early 2009) with 2 x 2.93GHz Quad Core Xeon processors, each core has L2 cache of 256 KB and L3 cache of 8MB , and 24GB Memory, only utilizing a single process/thread at once. The programs were compiled using the GNU C++ compiler (`g++`) 4.7.1 with the `-Ofast -msse4.2` option for optimization. The running times are measured in seconds, starting after reading the input text in memory, and the average of 3 runs is reported.

Figure 7.2 shows running times for two strings chosen from existing corpora[3]. Running times for a more comprehensive set of data can be found at `http://code.google.com/p/bgone/`. The running time is broken down into: construction of the suffix array, computation of PSV and NSV arrays, and LZ parsing [4]. We omit the runtime of LZScan for LISP, since it was 8 to 10 times slower than other algorithms. The figure shows that our algorithms is only about 2-3 times as slow as the KKP algorithms for large data, despite the added complexity introduced in order to use less space. One reason that KKP1 is faster may be because BGone needs random access on the integer array to compute the NSV_{lex} array, while KKP1 does not. Although KKP1 writes/reads SA to and from the disk, sequential I/O seems to be faster than random access on the memory. BGoneSA which computes the Φ array through SA, is a little faster than BGoneT which computes Φ directly. Interestingly, BGoneT can be the fastest for very small data, presumably when the whole space required for the algorithm fits within the L2

[1] `https://www.cs.helsinki.fi/group/pads/lz77.html`.

[2] the README of libdivsufsort-2.0.1 mentions the total space to be $5N + O(1)$ bytes, leaving $O(1)$ bytes excluding the suffix array and input text.

[3] `http://corpus.canterbury.ac.nz/descriptions/`, `http://pizzachili.dcc.uchile.cl/texts.html`

[4] The runtime of the computation of PSV_{lex} and NSV_{lex} in KKP1 includes the read and write time of SA, and the same runtimes in BGoneT and BGoneSA include the computation of Φ since NSV_{lex} values are computed on-the-fly in the construction of Φ.

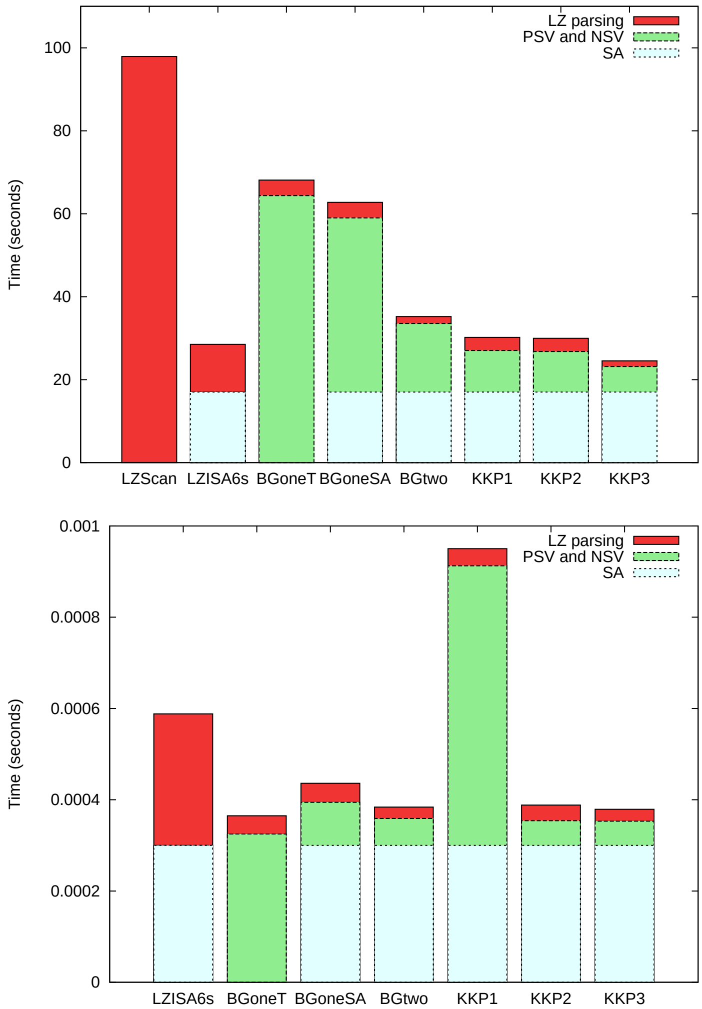


Figure 7.2: Running times in seconds for (left) DNA (100MB, $\sigma = 16$) and (right) LISP code (3721B, $\sigma = 76$).

cache. In such case SACA-K runs faster than divsufsort, and thus divsufsort is used for DNA while SACA-K is used for the LISP code, for constructing SA.

Chapter 8

Conclusion and Future Perspectives

8.1 Conclusion

We summarize our works as follows.

(A) Developing efficient algorithms to compute q-gram frequencies on SLPs. In Chapter 3, we proposed an $O(qn)$ time algorithm for computing all q-gram frequencies in a string when given an SLP of size n representing the string. The algorithm extensively improved previous work [33], and computational experiments showed that our algorithm run faster than linear time algorithm on uncompressed strings for small q. We also showed that applications of q-gram frequencies also can run efficiently in SLPs. For strings T_1 and T_2, A string kernel between them can be computed in $O(|T_1| + |T_2|)$. For two multisets of SLPs, the optimal q-gram from two sets can be found in $O(qM)$ time, where M is the total number of variables of two multisets of SLPs. We presented that the $O(qn)$ algorithm can be easily extended to compute frequencies of all substrings of length up-to and including q.

In Chapter 4, we improved the $O(qn)$ algorithm to be able to handle large q, and proposed an $O(N - dup(q, \mathcal{T}))$ time algorithm. The algorithm is asymptotically always at least as fast compared to algorithms on uncompressed strings for any q.

In Chapter 5, we considered the non-overlapping q-gram frequencies problem, and then proposed an $O(q^2 n)$ time and $O(qn)$ space algorithm.

(B) Developing efficient compression algorithms with high compression ratio. In Chapter 6, we developed fast linear time algorithms to compute the LZ77 factorization of a given string T. We call them BGS, BGL, and BGT. They respectively use $(4N + S_{\max}) \log N$, $4N \log N$, $3N \log N$ bits of working space, excluding T, where $S_{\max} \leq N$ is the maximum stack size that the algorithm uses. Computational experiments on various data sets showed that BGS, BGL, BGT constantly outperform LZ_OG [62] which is one of the fastest among existing linear time algorithms, and especially BGS can be up to 2 to 3 times faster in the processing after obtaining the suffix array.

In Chapter 7, we developed space efficient linear time algorithms to compute the LZ77 factorization of a given string T. We call them BGtwo and BGone. They respectively use $2N \log N$ and $N \log N + O(\sigma \log N)$ bits of working space, excluding T. BGtwo is one of algorithms using the least space among existing linear time algorithms for integer alphabets, and BGone is the algorithm using the least space for small alphabets. Computational experiments showed that BGone is only about 2-3 times as slow as KKP2, which is the fastest algorithm among linear time algorithms using $2N \log N$ bits of working space, despite the added complexity introduced in order to use less space.

8.2 Future Perspectives

We give some perspectives for future work.

(A) Developing efficient algorithms to compute q-gram frequencies on SLPs. Since some applications admit approximate solutions of q-gram frequencies, a future work is to develop approximate yet faster q-gram frequencies algorithms on SLPs.

(B) Developing efficient compression algorithms with high compression ratio. We proposed the linear time LZ77 factorization algorithms using $N \log N + O(\sigma \log N)$ bits of working space. An interesting question is whether it is possible to compute the LZ77 factorization in linear time using only $N \log N$ bits of working space independent of alphabet size. However it means that we may have to develop also a linear time suffix array construction algorithm which uses only $N \log N$ bits of working space, and looks quite difficult.

Recently, super linear time but practically fast algorithms which use more than $N \log N$ bits of working space were proposed [41]. A super linear time algorithm using less than $N \log N$ bits of working space was also proposed [35], but can be slower than the fastest linear time algorithm in practice. A future work is to develop super linear time algorithms that run practically faster than the fastest linear time algorithm, and use less than $N \log N$ bits of space.

Bibliography

[1] Al-Hafeedh, A., Crochemore, M., Ilie, L., Kopylov, J., Smyth, W., Tischler, G., Yusufu, M.: A comparison of index-based Lempel-Ziv LZ77 factorization algorithms. ACM Computing Surveys (in press)

[2] Apostolico, A., Preparata, F.P.: Data structures and algorithms for the string statistics problem. Algorithmica 15(5), 481–494 (1996)

[3] Arimura, H., Wataki, A., Fujino, R., Arikawa, S.: A fast algorithm for discovering optimal string patterns in large text databases. In: Proc. 9th International Conference on Algorithmic Learning Theory. pp. 247–261. Springer, London, UK (1998)

[4] Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)

[5] Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. System Sci. 48(2), 214–230 (1994)

[6] Brazma, A., Jonassen, I., Eidhammer, I., Gilbert, D.: Approaches to the automatic discovery of patterns in biosequences. J. Computational Biology 5(2), 279–305 (1998)

[7] Brodal, G.S., Lyngsø, R.B., Östlin, A., Pedersen, C.N.S.: Solving the string statistics problem in time $O(n \log n)$. In: Proc. ICALP'02. LNCS, vol. 2380, pp. 728–739 (2002)

[8] Brown, P.F., deSouza, P.V., Mercer, R.L., Pietra, V.J.D., Lai, J.C.: Class-based n-gram models of natural language. Comput. Linguist. 18(4), 467–479 (Dec 1992), `http://dl.acm.org/citation.cfm?id=176313.176316`

[9] Cégielski, P., Guessarian, I., Lifshits, Y., Matiyasevich, Y.: Window subsequence problems for compressed texts. In: Proc. CSR 2006. LNCS, vol. 3967, pp. 127–136 (2006)

[10] Chan, S., Kao, B., Yip, C.L., Tang, M.: Mining emerging substrings. In: DASFAA '03: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications. p. 119. IEEE Computer Society, Washington, DC, USA (2003)

[11] Charikar, M., Lehman, E., Liu, D., Panigrahy, R., Prabhakaran, M., Sahai, A., abhi shelat: The smallest grammar problem. IEEE Transactions on Information Theory 51(7), 2554–2576 (2005)

[12] Chen, G., Puglisi, S., Smyth, W.: Lempel-Ziv factorization using less time & space. Mathematics in Computer Science 1(4), 605–623 (2008)

[13] Claude, F., Navarro, G.: Self-indexed grammar-based compression. Fundamenta Informaticae 111(3), 313–337 (2011)

[14] Crochemore, M.: Linear searching for a square in a word. Bulletin of the European Association of Theoretical Computer Science 24, 66–72 (1984)

[15] Crochemore, M., Ilie, L.: Computing longest previous factor in linear time and applications. Information Processing Letters 106(2), 75–80 (2008)

[16] Crochemore, M., Ilie, L., Iliopoulos, C.S., Kubica, M., Rytter, W., Waleń, T.: LPF computation revisited. In: Proc. IWOCA 2009. pp. 158–169 (2009)

[17] Crochemore, M., Ilie, L., Smyth, W.F.: A simple algorithm for computing the Lempel Ziv factorization. In: Proc. DCC 2008. pp. 482–488 (2008)

[18] Dietz, P.: Finding level-ancestors in dynamic trees. In: Proc. WADS. pp. 32–40 (1991)

[19] Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S.: Compressing and indexing labeled trees, with applications. J. ACM 57(1) (2009)

[20] Gawrychowski, P.: Pattern matching in Lempel-Ziv compressed strings: Fast, simple, and deterministic. In: Proc. ESA 2011. pp. 421–432 (2011)

[21] Gawrychowski, P.: Faster algorithm for computing the edit distance between slp-compressed strings. In: SPIRE. pp. 229–236 (2012)

[22] Gąsieniec, L., Kolpakov, R., Potapov, I., Sant, P.: Real-time traversal in grammar-based compressed files. In: Proc. DCC'05. p. 458 (2005)

[23] Goto, K., Bannai, H.: Simpler and faster Lempel Ziv factorization. In: DCC. pp. 133–142 (2013)

[24] Goto, K., Bannai, H.: Space efficient linear time Lempel-Ziv factorization for small alphabets. In: DCC (2014), to appear

[25] Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Fast q-gram mining on SLP compressed strings. In: Proc. SPIRE'11. pp. 278–289 (2011)

[26] Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Computing q-gram non-overlapping frequencies on slp compressed texts. In: SOFSEM. pp. 301–312 (2012)

[27] Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Speeding up q-gram mining on grammar-based compressed texts. In: Proc. CPM 2012. pp. 220–231 (2012)

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

[29] Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Cambridge University Press (1997)

[30] Hermelin, D., Landau, G.M., Landau, S., Weimann, O.: A unified algorithm for accelerating edit-distance computation via text-compression. In: Proc. STACS'09. pp. 529–540 (2009)

[31] Hui, L.C.K.: Color set size problem with application to string matching. In: Combinatorial Pattern Matching. LNCS, vol. 644, pp. 230–243. Springer (1992)

[32] Inenaga, S., Bannai, H.: Finding characteristic substring from compressed texts. In: Proc. The Prague Stringology Conference 2009. pp. 40–54 (2009), full version to appear in the International Journal of Foundations of Computer Science

[33] Inenaga, S., Bannai, H.: Finding characteristic substrings from compressed texts. International Journal of Foundations of Computer Science 23(2), 261–280 (2012), a preliminary version appeared in PSC'09

[34] Inenaga, S., Shinohara, A., Takeda, M.: An efficient pattern matching algorithm on a subclass of context free grammars. In: Proc. Eighth International Conference on Developments in Language Theory (DLT2004). LNCS, vol. 3340, pp. 225–236. Springer (2004)

[35] Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Lightweight lempel-ziv parsing. In: SEA. pp. 139–150 (2013)

[36] Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Linear time Lempel-Ziv factorization: Simple, fast, small. In: Proc. CPM'13 (2013)

[37] Kärkkäinen, J., Manzini, G., Puglisi, S.J.: Permuted longest-common-prefix array. In: CPM. pp. 181–192 (2009)

[38] Kärkkäinen, J., Sanders, P.: Simple linear work suffix array construction. In: Proc. ICALP'03. LNCS, vol. 2719, pp. 943–955. Springer (2003)

[39] Karpinski, M., Rytter, W., Shinohara, A.: An efficient pattern-matching algorithm for strings with short descriptions. Nordic Journal of Computing 4, 172–186 (1997)

[40] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Proc. CPM'01. LNCS, vol. 2089, pp. 181–192 (2001)

[41] Kempa, D., Puglisi, S.J.: Lempel-Ziv factorization: Simple, fast, practical. In: Proc. ALENEX'13 (2013)

[42] Kieffer, J., Yang, E., Nelson, G., Cosman, P.: Universal lossless compression via multilevel pattern matching. IEEE Transactions on Information Theory 46(4), 1227–1245 (2000)

[43] Kimura, D., Kashima, H.: A linear time subpath kernel for trees. In: IEICE Technical Report. vol. IBISML2011-85, pp. 291–298 (2011)

[44] Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal on Computing 6(2), 323–350 (1977)

[45] Larsson, N.J., Moffat, A.: Offline dictionary-based compression. In: Proc. Data Compression Conference 1999 (DCC '99). pp. 296–305 (1999)

[46] Larsson, N.J., Moffat, A.: Off-line dictionary-based compression. Proceedings of the IEEE 88(11), 1722–1732 (2000)

[47] Leslie, C., Eskin, E., Noble, W.S.: The spectrum kernel: A string kernel for SVM protein classification. In: Pacific Symposium on Biocomputing. vol. 7, pp. 566–575 (2002)

[48] Lifshits, Y.: Processing compressed texts: A tractability border. In: Proc. CPM 2007. LNCS, vol. 4580, pp. 228–240 (2007)

[49] Lifshits, Y., Lohrey, M.: Querying and embedding compressed texts. In: Proc. 31st International Symposium on Mathematical Foundations of Computer Science (MFCS2006). LNCS, vol. 4162, pp. 681–692. Springer (2006)

[50] Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM Journal on Computing 22(5), 935–948 (1993)

[51] Manber, U.: A text compression scheme that allows fast searching directly in the compressed file. ACM Trans. Inf. Syst. 15(2), 124–136 (1997)

[52] Maruyama, S., Takeda, M., Nakahara, M., Sakamoto, H.: An online algorithm for lightweight grammar-based compression. In: Proc. CCP. pp. 19–28 (2011)

[53] Matsubara, W., Inenaga, S., Ishino, A., Shinohara, A., Nakamura, T., Hashimoto, K.: Computing longest common substring and all palindromes from compressed strings. In: Proc. 34th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM2008). LNCS, vol. 4910, pp. 364–375. Springer (2008)

[54] Matsumoto, T., Kida, T., Takeda, M., Shinohara, A., Arikawa, S.: Bit-parallel approach to approximate string matching in compressed texts. In: Proc. 7th International Symp. on String Processing and Information Retrieval. pp. 221–228. IEEE Computer Society (2000)

[55] Miyazaki, M., Fukamachi, S., Takeda, M., Shinohara, T.: Speeding up the pattern matching machine for compressed texts. Transactions of Information Processing Society of Japan 39(9), 2638–2648 (1998), (in Japanese)

[56] de Moura, E.S., Navarro, G., Ziviani, N., Baeza-Yates, R.: Direct pattern matching on compressed text. In: Proc. 5th International Symp. on String Processing and Information Retrieval. pp. 90–95. IEEE Computer Society (1998)

[57] Navarro, G., Kida, T., Takeda, M., Shinohara, A., Arikawa, S.: Faster approximate string matching over compressed text. In: Proc. Data Compression Conference 2001. IEEE Computer Society (2001), to appear.

[58] Navarro, G., Raffinot, M.: A general practical approach to pattern matching over Ziv-Lempel compressed text. In: Proc. 10th Ann. Symp. on Combinatorial Pattern Matching. LNCS, vol. 1645, pp. 14–36. Springer (1999)

[59] Nevill-Manning, C.G., Witten, I.H.: Identifying hierarchical strcture in sequences: A linear-time algorithm. J. Artif. Intell. Res. (JAIR) 7, 67–82 (1997)

[60] Nong, G.: Practical linear-time $O(1)$-workspace suffix sorting for constant alphabets. ACM Trans. Inf. Syst. 31(3), 15 (2013)

[61] Nong, G., Zhang, S., Chan, W.H.: Two efficient algorithms for linear time suffix array construction. IEEE Trans. Computers 60(10), 1471–1484 (2011)

[62] Ohlebusch, E., Gog, S.: Lempel-Ziv factorization revisited. In: Proc. CPM'11. pp. 15–26 (2011)

[63] Rytter, W.: Application of Lempel-Ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1–3), 211–222 (2003)

[64] Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., Arikawa, S.: Speeding up pattern matching by text compression. In: Proc. 4th Italian Conference on Algorithms and Complexity. LNCS, vol. 1767, pp. 306–315. Springer (2000)

[65] 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)

[66] Storer, J., Szymanski, T.: Data compression via textual substitution. Journal of the ACM 29(4), 928–951 (1982)

[67] Teo, C.H., Vishwanathan, S.V.N.: Fast and space efficient string kernels using suffix arrays. In: Proc. ICML. pp. 929–936 (2006)

[68] Tiskin, A.: Faster subsequence recognition in compressed strings. J. Math. Sci. 158(5), 759–769 (2009)

[69] Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)

[70] Vishwanathan, S.V.N., Smola, A.J.: Fast kernels for string and tree matching. In: Proc. NIPS. pp. 569–576 (2002)

[71] 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)

[72] Welch, T.A.: A technique for high performance data compression. IEEE Computer 17, 8–19 (1984)

[73] Yamamoto, T., Bannai, H., Inenaga, S., Takeda, M.: Faster subsequence and don't-care pattern matching on compressed texts. In: Proc. CPM'11. LNCS, vol. 6661, pp. 309–322 (2011)

[74] Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Transactions on Information Theory IT-23(3), 337–349 (1977)

[75] Ziv, J., Lempel, A.: Compression of individual sequences via variable-length coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)