Substring Compression Problems

Graham Cormode* S. Muthukrishnan†

Abstract

We initiate a new class of string matching problems called *Substring Compression Problems*. Given a string S that may be preprocessed, the problem is to quickly find the compressed representation or the compressed size of any query substring of S (Substring Compression Query or SCQ) or to find the length ℓ substring of S whose compression is the least (Least Compressible Substring or LCS problem).

Starting from the seminal paper of Lempel and Ziv over 25 years ago, many different methods have emerged for compressing entire strings. Determining substring compressibility is a natural variant that is combinatorially and algorithmically challenging, yet surprisingly has not been studied before. In addition, compressibility of strings is emerging as a tool to compare biological sequences and analyze their information content. However, typically, the compressibility of the entire sequence is not as informative as that of portions of the sequences. Thus substring compressibility may be a more suitable basis for sequence analysis.

We present the first known, nearly optimal algorithms for substring compression problems—SCQ, LCS and their generalizations—that are exact or provably approximate. Our exact algorithms exploit the structure in strings via suffix trees and our approximate algorithms rely on new relationships we find between Lempel-Ziv compression and string parsings.

1 Introduction

In their seminal paper over 25 years ago, Ziv and Lempel [ZL77] presented a "universal" method for compressing any string S of characters, and proved information-theoretic properties of their compression scheme (which we will henceforth denote `ZL77`). Shortly thereafter, Rodeh, Pratt and Even [RPE81] showed that `ZL77` can be implemented very efficiently, in fact, in time only linear in the string length, $|S|$. This was a powerful application of the linear time algorithm for constructing a suffix tree of S [Wei73]. `ZL77` and its numerous variants have had a significant impact in theory and practice, and are now classical lore. Nevertheless, there are certain simple, natural variants of string compression that do not seem to have been studied. For example, given a string: how compressible are different substrings?; which of its substrings are the most or the least compressible?; how many substrings are highly compressible or what is the total compressibility of all ℓ length substrings? In this paper we initiate the study of such *substring compression* problems. Compression is no longer a tool for saving space alone; in recent analyses of biological sequences compression is used to compare sequences and study their information content. In such cases, substring compressibility may be a more suitable basis than entire sequence compressibility because even though entire biological sequences do not compress well, parts thereof do. Thus the substring compression problems, besides being natural variants of the fundamental problem of string compression, will be useful for searching the "information structure" in biological sequences.

1.1 Overview of Lempel-Ziv Algorithm

Let S denote the string of length $|S|$ formed from the concatenation of characters $S[1], S[2] \ldots S[|S|]$. Let $S[i, j]$ denote the substring of S beginning at location i of length $j - i + 1$. The algorithm `ZL77` works by parsing the string greedily from left to right. At each step, the algorithm has parsed the string $S[1, i]$, and parses the longest substring $S[i + 1, j]$ such that $S[i + 1, j]$ occurs as a substring of $S[1, i]$. Formally, this is the algorithm `LZSS` [SS82]. Our algorithms can all be adapted to work on other variants of the `ZL77` scheme in a straightforward way, with different constants in the approximation algorithms, so we focus on this basic version.

By compressed size of string S, we will mean the *number of phrases* found by the `ZL77` algorithm while compressing S. This is because we assume that S is coded using constant number of words per parsed phrase, and we count the number of words for the compressed size. So, up to constant factors, the compressed size of S is the number of phrases. We denote this as $LZ(S)$. For the empty string ϵ, we define

*graham@dimacs.rutgers.edu Center for Discrete Mathematics and Computer Science (DIMACS) Rutgers University, Piscataway NJ. Supported by NSF ITR 0220280 and NSF EIA 02-05116.

†muthu@cs.rutgers.edu Division of Computer and Information Systems, Rutgers University, Piscataway NJ. Supported by NSF EIA 0087022, NSF ITR 0220280 and NSF EIA 02-05116.

$LZ(\epsilon) = 0$. For single characters a, $LZ(a) = 1$. For all strings S such that $|S| \geq 2$, $2 \leq LZ(S) \leq |S|$. The Lempel-Ziv encoding of S in the context of R, written $LZ(S|R)$, is computed by running the Lempel-Ziv encoder on the concatenated string $R||S$, but only outputting the encoding of the S part (we do not allow phrases to cross the boundary between the two strings). Hence, this is equivalent to $LZ(R\$||S) - LZ(R\$)$, where \$ does not appear in R or S.

1.2 Problems

Formally, the problems are as follows. Consider a string S of symbols from a binary alphabet[1]. We consider two specific substring compression problems.

Problem 1: *Substring Compression Query (SCQ).* We are allowed to preprocess string S. Each query is (i, j) and we are required to output the compressed representation of substring $S[i, j]$ by `ZL77`. A variation is the *Substring Compression Size Query (SCSQ)* where we need to only output the compressed size $C[i, j]$ and not necessarily the compressed representation. We will also consider the generalized SCQ (GSCQ) where each query is (α, β, i, j) and we are required to output the compressed representation of substring $S[i, j]$ by `ZL77` with the context $S[\alpha, \beta]$ (that is, compression of $S[i, j]$ in $S[\alpha, \beta]||S[i, j]$). We call $S[\alpha, \beta]$ the *context* for the query. As before, in the General Substring Compression Size Query (GSCSQ) we need to only output the compressed size $C_{\alpha,\beta}[i, j]$ and not necessarily the compressed representation. ∎

The goal is to optimize the preprocessing space and time, and yet make queries fast. For example, for SCQ, we want better tradeoffs than the two trivial extremes: (1) preprocess and store $LZ[i, j]$ and $C[i, j]$ for each $[i, j]$ which takes at least $\Omega(|S|^2)$ preprocessing time and space. This answers queries in optimal time, but is prohibitive over long S's. (2) Do no preprocessing and run the best known `ZL77` compression algorithm over $S[i, j]$ when queried. This takes $\Omega(|S|)$ query time which is not as responsive as one would like, especially when many queries are posed in succession and the queries are long string segments.

(G)SCSQ and (G)SCQ are versatile tools for exploring the compressibility structure of a string. For example, using this we can solve problems such as: find the *total* compressibility of all ℓ-length substrings (which is a new measure of string compressiblity of interest parameterized by ℓ); find the context $S[\alpha, \beta]$ that results in most compression of all ℓ-length substrings; find all substrings that have compressed size at least m; compare any two susbtrings based on their relative compressibility or compressibility with respect to an arbitrary context; find the context that best differentiates any two given substrings based on their relative compressibility with the context, etc. For batched problems where we seek compressibility of *several* substrings simultaneously, we may be able to improve on using the CSQs many times once for each substring. Our second problem studies such a batched problem.

Problem 2. *Least Compressible Substring Problem (LCS).* We are given S and a parameter ℓ. Our goal is to find the substring of length ℓ that is reduced the least by compression out of all substrings of length ℓ.[2] Formally, the output is some i where $C[i, i + \ell - 1] = \max_j C[j, j + \ell - 1]$. As before, there is a generalized version of this problem, GLCS, where the query specifies an arbitrary context $S[\alpha, \beta]$. The naive solution is to compute the compressed size of all $O(|S|)$ substrings of length ℓ in time $\Omega(|S|\ell)$. A good solution must be more efficient than both this naive solution and running the $O(|S|)$ SCSQs $S[j, j + \ell - 1]$ for each j. ∎

Throughout, we focus on the Lempel-Ziv compression technique because `ZL77` is a well-studied and practically useful method which is employed widely. These questions are also of interest for other compressors. For example, *Run Length Encoding (RLE)* is a very simple compression technique that encodes repeated characters with the character and number of repeats. Since the encoding does not depend on the surroundings of the substring, it is trivial to answer SCQ, SCSQ and LCS queries optimally for RLE after linear preprocessing. As another example, consider *Huffman Encoding* which replaces each character with a code whose length is a function of the relative frequency of that character. The resulting compression is proportional to the zero'th order entropy of the string. Using standard data structures, it is easy to compute the number of occurences of each character within a query substring and hence the size of the compressed representation in near optimal bounds. In contrast, there are other popular compression schemes for which the SCQ, SCSQ and LCS problems are of interest and which we leave open. One is *Burrows-Wheeler Transform* (BWT) which is not a compression technique alone, but is a reversible reordering of the string which is then passed to a compressor, such as RLE, MTF (Move To Front), Lempel-Ziv. It is possible to develop some measures of compressed size for the BWT and show that the BWT of any *suffix* of a string can be related to the BWT of the string itself.

[1] We can generalize the results without additional complexity for larger alphabets, even those whose size is polynomial in $|S|$.

[2] The most compressible substring problem is analogous.

From this the compressed size may be efficiently calculated for simple encoders such as RLE. Still, it is open to thoroughly understand the BWT of a *substring* with respect to that of the entire string. Other include the *Lempel-Ziv-Welch method* (LZW); *Prediction by Partial Matching* (PPM); solving the SCQ, SCSQ and LCS for such compressors is left open. See [BCW90] for myraid compression schemes.

1.3 Motivations

These problems are of interest from a variety of perspectives, primarily from recent work in biological sequence analysis. The direct application of standard compression tools to biological sequences is a controversial one. There have been many attempts to use standard compression techniques to archive and transmit biological sequences efficiently [GT94, MSIO00]. Existing approaches often fail to improve significantly on naive techniques such as representing DNA sequences with two bits per character (for A, C, G and T). The best compression results have been found by augumenting standard "Lempel-Ziv" style operations with other operations that mimic mutations on sequences such as reverse complement copy, and character deletion [CKL00]; still, these do not necessarily result in significant reductions in space. The conclusion was that the information bearing sequences (the "coding regions") have high-information and are essentially incompressible [NMW99]. But, for example, of human DNA, only 5% consists of coding regions. The remainder, initially dismissed as "junk DNA" is now believed to have a variety of purposes, not all of which are fully understood yet. These regions have been shown to be more redundant and consequently can be compressed much better [SYKT01]. Thus, finding regions of DNA which do not compress well can be indicative of structure and purpose, and could identify new loci to study [RDDD97, RDDD94, RDD+97]. This motivated our work in this paper of developing algorithms that help explore the substring compressibility in a flexible way.

In a different biological context, there is a need for *comparing* related biological sequences to find areas of similarity and to give a distance measure for clustering or building phylogeny trees. The metrics for comparison are often defined based on Kolmogorov complexity, motivated by Occam's Razor (parsimony), or the principle of Minimum Description Length. For example, in the work of Li *et al* [LCL+03], the similarity metric is defined in terms of the Kolmogorov complexity ($K()$) of combinations of strings. In order to make such comparisons tractable, one must use an efficently computable substitute for the Kolmogorov complexity, and compression tools are typically used for this purpose. This approach has been extensively used for inter-sequence comparison [BCL02, LCL+03, KLR04, RDDD94], but not so for *intra*-sequence comparison, where the goal is to compare substrings within a string or set of strings. These computations require information about the compressed size of the substrings and substrings with context, which can be expressed as combinations of our SCSQ and GSCSQ problems we introduce here.

From a theory perspective, problems relating to string searching, compression and comparison have applications to a variety of areas in text editing, storage and searching [Gus97]. The combinatorial analysis of string compression algorithms has yielded many deep and significant results. The problems that we propose here appear quite natural combinatorial versions of the basic question of how compressible a string is, but perhaps surprisingly, we do not know of any nontrivial results for them. Our work is related to working on compressed pattern matching i.e., finding a (compressed) pattern within a compressed text [ABF96, FT95], but our techniques are quite different.

1.4 Overview

Motivated by these applications, we abstract the substring query problems and present the following, first known algorithms.

- For the (G)SCQ problem, we present an exact algorithm that with $O(|S| \log |S|)$ preprocessing, answers any query (i, j) exactly in time $O(C[i, j] \log |S| \log \log |S|)$. This is sublinear in query size $|S[i, j]|$ provided the substring is reasonably compressible. Also, this is nearly optimal since the output size is $\Theta(C[i, j])$.
- For the SCSQ problem, we present an approximate algorithm that takes only $O(1)$ query time. This algorithm uses $O(|S| \log^2 |S|)$ preprocessing and for query $S[i, j]$, outputs $\hat{C}[i, j]$ such that $\hat{C}[i, j] = O(C[i, j] \log |S| \log^* |S|)$ in constant time.
- For the (G)LCS problem, we give two approximate solutions. First, we show a $O(\log |S| \log^* |S|)$ approximation which runs in time $O(|S| \log |S|)$. Second, we give a new algorithm that gives a constant factor approximation in time no more than $O(|S|\ell / \log \ell)$.

All these algorithms are obtained using natural ideas. The exact algorithm for SCQ uses separator-decomposition of a suffix tree to make search paths shorter and then uses two dimensional range searching to rapidly skip over the phrases the `ZL77` algorithm would produce; alternatively, this can be thought of as

binary searching with appropriate range searching oracles. The first approximate algorithms for SCSQ rely on parsing the string in a locally consistent way. We will use the simplified parsing described in [CM02, SV95] and prove structural properties that relate the ZL77 compression size to the compression produced by this parsing. An added advantage of this structural relationship is that our results are more general than quoted above. For example, if we allow S to be dynamic under insert, delete and modify operations of characters or blocks, the results can still be generalized. For the LCS problem, we make use of a combinatorial relationship between the compression given by the ZL77 algorithm and more powerful distance measures that can also perform standard "edit distance" style operations.

A pertinent question is whether the approximate compression sizes our algorithms determine will be useful in practice. [3] In general, whether $C[i,j]\log|S|\log\log|S|$ is more efficient than $S[i,j]$ or whether an $O(C[i,j]\log|S|\log^*|S|)$ approximation estimate is longer than $|S[i,j]|$ depends on the relationship between $S[i,j]$ and $C[i,j]$. For the compression method we consider (with so-called "overlapping copies" disallowed) the compressed size of random strings is $\Omega(|S|/\log|S|)$, but for highly repetitive strings, it can be exponentially smaller. For example, under this version of ZL77 the string a^n has compressed size $\Omega(\log n)$. If S were a natural language text or other standard ASCII file, $C[i,j]$ is often a small constant factor of $S[i,j]$; so, our bounds may not be compelling. Our methods are more applicable when substrings of a string are compressible to significantly different levels. This can happen in biological sequences with a significant proportion of repeats, or in sequences with local periodicities [BCL02, CLMT02]. In such cases, the logarithmic factors in our running times or approximation ratios may not prove to be a bottleneck. So, from theory and practice points of view, it is important to get more efficient exact algorithms or better approximations than the ones we have obtained, in particular with different compression schemes. We leave these questions open for future work. We believe that both (G)SCQ and (G)LCS with different compression schemes are natural, nice data structural problems for the community.

The rest of the paper is arranged as follows. In Section 2, we present our exact algorithm for the SCQ problem and in Section 3, we give our approximate algorithms for SCSQ. We present our approximation algorithms for the LCS problem in Section 4. In Section 5 we present concluding remarks.

[3] A similar question arose when algorithms were developed matching patterns against compressed text without uncompressing them [ABF96, FT95]. Since compression for natural text or most ASCII files was in small constant factors, would even asymptotically optimal algorithms not have larger constant overheads? Over time convincing applications and algorithms have been developed [GV00, FM00].

2 Exact Solutions

2.1 Primitives and Notation

For the exact algorithm, we need a few primitives. First, given a string S, its *suffix tree* T_S is the compressed trie of all the suffixes of S (a *trie* is a natural decision tree on a set of strings; a *compressed trie* is one in which nodes of outdegree one—nodes that form paths in the tree—are compressed into single edges). Each of its edges is labeled by some substring of S. Further, children of a node are sorted left to right in the increasing lexicographic order of the substrings that label the paths to them. For any node u in T_S, we let σ_u be the string obtained by concatenating the labels on the edges from the root to u in that order. There is one-to-one correspondence between the leaves and the $|S|$ suffixes of S. We will label the leaves $l_1,\ldots,l_{|S|}$ from left to right. Because of the one-to-one correspondence, each σ_{l_i} equals precisely one of the suffixes, say j of S. We let this mapping be denoted f, i.e., $f(i)=j$ and $f^{-1}(j)=i$. $S[i,|S|]$ is the ith suffix denoted S_i. Then, $\sigma_{l_i}=S_{f(i)}$. Notice that the suffixes are ordered in the lexicographically increasing order left to right $l_1,\ldots,l_{|S|}$. The suffix tree can be computed in time linear in $|S|$ by a number of different algorithms [Wei73, Gus97].

A useful primitive on suffix trees is given two leaves l_i and l_j, find their least common ancestor (LCA) node in the tree. This in fact corresponds to finding the Longest Common Prefix (LCP) between $S_{f(i)}$ and $S_{f(j)}$. LCA queries can be answered in $O(1)$ time after linear preprocessing of the suffix tree [Gus97]. We also use *range searching* algorithms. In our case, the input is some set P of grid points in two dimensions (i,j), $1\le i,j\le|P|$ for integers i and j. After preprocessing the set P, we need to answer rectangle range queries, i.e., are there points in the range $[a,b]\times[c,d]$ for integers a,b,c,d?[4] The best known algorithms for this problem take time $O(\log\log|P|)$ with $O(|P|\log|P|)$ preprocessing [Aga97, ABR00].

2.2 Algorithms for (G)SCQ

We will start with a general data structural primitive and show two algorithms for it; later we will use the primitive to solve (G)SCQs.

[4] We will need a variant: find one such point if there exist any; this has similar bounds.

Interval Longest Common Prefix Query (ILCP). Given a string S for preprocessing, each query is $ILCP([i,j],k)$ and needs to determine the longest common prefix between S_k and any of the prefixes $S_i, \ldots, S_j$. Let $|ILCP|$ be the length and $ILCP$ the actual prefix. We have $|ILCP([i,j],k)| = \min_\ell |LCP(S_\ell, S_k)|$. This problem is equivalent to determining the deepest node of $LCA(l_{f^{-1}(i)}, l_{f^{-1}(k)}), \ldots, LCA(l_{f^{-1}(j)}, l_{f^{-1}(k)})$ If $i = j$, the ILCP query is identical to the well-studied LCP query. ∎

For a fixed $[i,j]$ given *a priori*, we can solve the problem for various k in $O(1)$ time by precomputing all the answers as follows. Marked each leaf $l_{f^{-1}(\ell)}$ for $i \le \ell \le j$ and all the nodes on the root to each of these leaves. This takes $O(|S|)$ time. After that, we do a bottom-up traversal of the suffix tree and list the marked node for each leaf the first time we encounter an ancestor that is marked. ILCP can be determined easily from this list. The whole preprocessing takes $O(|S|)$ time.

When $[i,j]$ is part of the query, the marking procedure takes $\Omega(|S|)$ time (the process of detecting the closest marked ancestor has been studied extensively and is fast [Gus97].) Instead, we have to take alternate approaches using the structure of suffix trees.

Randomized Algorithm. We will provide a sketch here. The algorithm relies on separator decomposition of the suffix tree. In a tree, a node v in tree T is called the *separator* if each of the connected components induced by removing v from T is of size at most $2|T|/3$. It is well known that every tree contains a separator. A *complete separator decomposition tree* D of T is defined as follows. The root of D is the separator v of T. By removing v from T, we get rooted trees with roots that were the children of v as well as the subtree rooted at the root of T that remains; each of these trees has a separator and these separators become the children of the root of D, and so on recursively. The tree D can be constructed in time linear in $|T|$ and its depth is $O(\log |T|)$. Separator decomposition of a tree has been used for distributed string matching [Nao91] and dictionary matching [AFM92]. Here, we can use it for the SCQ problem.

The main idea is as follows. Consider each separator v we encounter in tracing down D with $S[k,|S|]$; there are at most $O(\log |S|))$ such nodes. Clearly σ_v is a prefix of the suffix S_k. So, we would like to determine if σ_v is a prefix of S_ℓ for $i \le \ell \le j$. We can phrase this as a range searching problem. Each internal node v has leaves that are indexed by a range of values $[l_{L(v)}, l_{L(v)+1}, \ldots, l_{R(v)}]$. Now observe that *we need to equivalently determine if* $L(v) \le f^{-1}(\ell) \le R(v)$ *for some* ℓ, $i \le \ell \le j$. In order to do this, we perform preprocessing to generate an instance of the two dimensional range searching problem on the grid as follows. For each leaf l_i, we generate a point $(i, f(i))$. The set of all such points is the point set P. Now, it suffices to determine if there exists a point in the range $[L(v), R(v)] \times [i,j]$ in the pointset P. This is precisely the range searching problem we stated earlier and can be solved in $O(\log\log|S|)$ time after suitable preprocessing.

That completes the high level description of the algorithm. However, implementing it has many details, chiefly, in Karp-Rabin randomized fingerprinting [KR87] of the substrings of S so that we can quickly determine the path down the separator tree D by comparing fingerprints of long substrings quickly, etc. We conclude,

LEMMA 2.1. *There exists a randomized algorithm that preprocesses the string S in $O(|S|\log|S|)$ time after which ILCP queries can be answered in $O(\log|S|\log\log|S|)$ time with high probability.*

Deterministic Algorithm. Now we will present an alternate method that gives a deterministic algorithm. We will adopt the range searching approach above. But we will focus on the ILCP query in its entirety working with the suffix tree T and not deal with the separator nodes or D. We will rewrite the ILCP query as a series of queries until we reach the basic range searching query. Observe that ICLP can be solved using both the following queries:

1. Q: What is the *longest* common prefix between $S[k,|S|]$ and a lexicographically *larger* suffix among $S[i], S[i+1], \ldots, S[j]$?
2. Q': What is the *longest* common prefix between $S[k,|S|]$ and a lexicographically *smaller* suffix among $S[i], S[i+1], \ldots, S[j]$?

We will focus on Q' since Q is similar. Observe that all the suffixes of S that are lexicographically smaller than $S[k,|S|]$ appear as leaves l_a in suffix tree T where $a \le f^{-1}(k)$. Hence, Q' can be recast as:
Q'': For a given i,j,k and suffix tree T, find the largest index a such that $a \le f^{-1}(k)$ and $f(a) \in [i,j]$.
We can now solve our subquery by a series of range searching queries by doing binary search with various guesses a^* for the value a using:
Q''': Given a^*, i, j, suffix tree T and point set P consisting of $(i, f(i))$ for each leaf l_i, is there any point in two dimensional range $[a^*, f^{-1}(k)] \times [i,j]$?
We will do $O(\log|S|)$ such range searching queries in order to find the a. Determining the LCA of l_a and $l_{f^{-1}(k)}$ gives the answer to the ILCP query. We conclude,

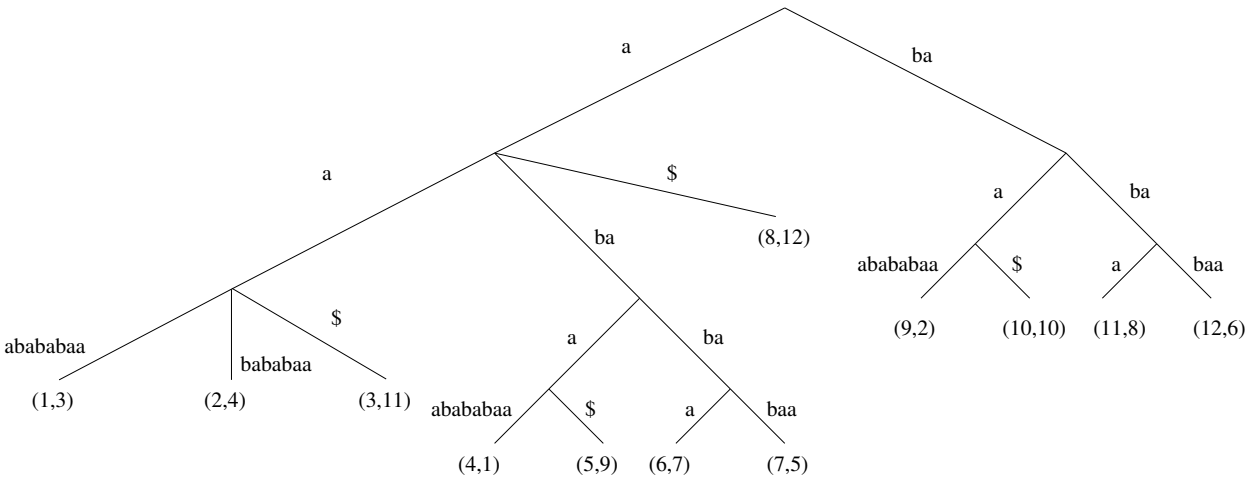

Figure 1: The suffix tree for the string $S = abaaababababaa$ with each leaf labeled with $(i, f(i))$. Suppose we are computing the compressed form of $S[5, 12]$, and we have reached $k = 9$, meaning we want to compress *abaa* and have already compressed *abab*. We are at node $(5, 9)$, and we first search for the node (i, j) with $(i < 5, 5 \leq j \leq 9)$ and i as great as possible. There is no such node. We then search for the node (i, j) with $(i > 5, 5 \leq j \leq 9)$ with i as small as possible. This is satisfied by $(6, 7)$, whose LCP with $(5, 9)$ is length 3, corresponding to the string *aba*. This advances k to $9 + 3 = 12$, where the process is repeated.

LEMMA 2.2. *There exists a deterministic algorithm that processes a string S in $O(|S| \log |S|)$ time after which any ILCP query can be answered in time $O(\log |S| \log \log |S|)$.*

Overall algorithm for SCQ. At the high level, the algorithm works as follows. Given query $S[i, j]$, we will simulate the ZL77 algorithm. That is, say it has "compressed" $S[i, k-1]$. Then the *subquery* is, what is the longest prefix of suffix $S[k, |S|]$ that appears in $S[i, k-1]$? An example is given in Figure 1. This subquery is asked $C[i, j]$ times in the ZL77 algorithm. Note that $C[i, j]$ is a lower bound on the running time of our algorithms if they have to produce a compressed representation of the substring query. Each such subquery can be answered using an ILCP, eg., $ILCP([i, k-1], k)$ and we conclude:

THEOREM 2.1. *String S is preprocessed in $O(|S| \log |S|)$ time and space $O(|S|)$. Any SCQ $S[i, j]$ takes time $O(C[i, j] \log |S| \log \log |S|)$ where $C[i, j]$ is the compressed size (the number of phrases in ZL77 algorithm) of $S[i, j]$.*

Extensions to generalized versions. Note that the same bounds follow immediately for the GSCQ problem as well since our ILCP primitive is quite general. In addition to the compressed prefix $S[i, k]$, we also must find the longest match in $S[\alpha, \beta]$, and pick the longer of these two. So now we will ask $ICLP([\alpha, \beta], k)$ and $ICLP([i, k-1], k)$ queries repeatedly to find the next phrase. Other extensions, e.g. to multiple strings, also follow.

3 Approximation Schemes Using ESP

In order to make an approximation of the compressibility of substrings of a given string, we use a string parsing technique called Edit Sensitive Parsing (ESP) [CM02]. This has its roots in the deterministic coin tossing (DCT) of Cole and Vishkin [CV86] and was developed through a sequence of papers thereafter from the work of Ṣahinalp [Ṣah97, ṢV96, MṢ00]. Previous work has shown how appropriate counts of substrings generated by this parsing approximates edit distances between pairs of strings. In [ṢV95], the authors showed that a string parsing based on DCT generates at most $O(k \log k \log^* k)$ LZ-style phrases if the optimal (greedy) parsing generates k. We extend this to show how, given ESP parse tree of a string, we can quickly extract an approximation of the compressed size of any *substring* specified at query time, to the same approximation ratio. Our contributions include a short proof of this fact, relying on known properties of the parsing, and the use of this to give the first approximate algorithms for LCS (in near linear time) and SCQ problems (queries take constant time).

The parse method takes a string, S, and generates a parse tree whose leaves are the individual characters

of the string in order. Each internal node has outdegree 2 or 3, and hence the tree has height $O(\log |S|)$. Concatenating all the leaves in the subtree induced by a node forms a substring of S. We precompute the parsing of the whole string (which takes time $O(|S| \log |S|)$), extract information about substrings using the subtree they induce.

DEFINITION 1. *The ESP subtree corresponding to the substring $S[i,j]$ is the subtree of the ESP tree of S containing the leaf nodes corresponding to $S[i]$ to $S[j]$, and all of their ancestors. $ST(S[i,j])$ is the set of all substrings corresponding to internal nodes in this tree. An example ESP tree and subtree is shown in Figure 2.*

LEMMA 3.1. $\frac{|ST(S[i,j])}{O(\log |S| \log^* |S|)} \leq C[i,j] \leq 3|ST(S[i,j])|$

Proof. The proof of the upper bound relies only on the fact that the parsing generates a tree parsing of the string with bounded degree (equal to three), and that $|ST(S[i,j])|$ counts the number of unique substrings generated by this parsing. Since T has outdegree at most three, then so must any subtree of it. The proof is based on a charging argument: we build the string $S[i,j]$ using copy and insert character operations, and charge each operation to an entry of $ST(S[i,j])$. The construction begins at the root of the subtree, and proceeds to walk the tree in a top-down, left-to-right fashion. At an internal node whose substring we have not charged to before, charge three units to the substring, and give one unit each of the node's children. If the substring of the current node has been charged to before, then it must have been built already, somewhere to the left. This can be copied at unit cost, using the credit passed to the node, and the node is not searched further. At a leaf, the corresponding character is inserted at unit cost, paid with the credit for that node. Thus $S[i,j]$ is built left-to-right, using copies from the left and character insertions in at most $3|ST(S[i,j])|$ operations. Since a greedy parsing of $C[i,j]$ phrases, is optimal [SS82] then $C[i,j] \leq 3|ST(S[i,j])|$.

For the lower bound, we show that each copy and character insert operation to build $S[i,j]$ has only a limited effect on the number of distinct nodes in the ESP subtree of the substring. We omit full details; a similar argument to that in [CM02] shows that each operation adds at most $O(\log |S| \log^* |S|)$ nodes at each level. Since only $C[i,j]$ operations are needed to build $S[i,j]$, this bounds the number of added nodes. ∎

THEOREM 3.1. *The LCS problem can be approximated in time $O(|S| \log |S|)$. The answer is approximated up to a factor of $O(\log |S| \log^* |S|)$.*

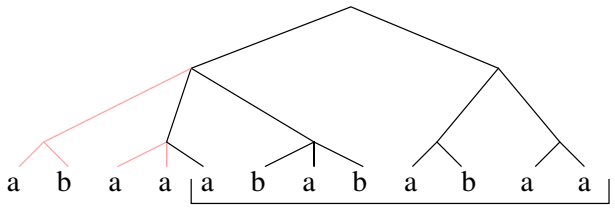

Figure 2: ESP tree for the string $S = abaaababababaa$. The subtree for $S[5,12]$ is highlighted. $ST(S[5,12]) = \{bab, ab, aa, abab, abaa, abababaa\}$. So $|ST(S[5,12])| = 6$; $C[5,12] = 6$ in this case also.

Proof. We apply Lemma 3.1 repeatedly, and iteratively compute the estimate of $C[i+1, i+\ell]$ from the estimate of $C[i, i+\ell-1]$. There are at most $O(|S|)$ distinct strings in the parsing of S, each of which can be identified in time $O(1)$ using the Karp-Miller-Rosenberg labeling of the string [KMR72] (pre-computed in time $O(|S| \log |S|)$). We create and maintain a vector, V, of length $O(|S|)$ which records the current multiplicity of strings in $ST(S[i, i+\ell-1])$. To advance to $S[i+1, i+\ell]$, adjust information about the substrings on path from the root of the tree to $S[i]$, and on the path from the root to $S[i+\ell]$. There are at most $O(\log |S|)$ such strings, and so updating their counts in the vector V takes time $O(\log |S|)$. If any count drops to zero, decrease the count of $|ST(S[i, i+\ell-1])|$ by one, and if any count increase from zero, increment the count appropriately. Following this adjustment, $|ST([i+1, i+\ell])|$ is computed in time $O(\log |S|)$, which by Lemma 3.1 is an $O(\log |S| \log^* |S|)$ factor approximation of $C[i+1, i+\ell]$. Computing this for the whole string takes time $O(|S| \log |S|)$, which bounds the overall time cost for this algorithm. ∎

THEOREM 3.2. *The SCQ problem can be solved with $O(|S| \log^2 |S|)$ time preprocessing and $O(|S| \log |S|)$ space. Each query is answered in $O(1)$ time approximate to a factor of $O(\log |S| \log^* |S|)$.*

Proof. Use the method in the above proof to compute $|ST(S[i, i+\ell-1])$ for all i and values of $\ell = 2^a$ for $a = 1 \ldots \lfloor \log |S| \rfloor$. Store these estimates as $L[i,a]$. For query (i,j), find the unique $\ell = 2^a$ such that $2^a \leq (j-i+1) < 2^{a+1}$ and output $L[i,a]+L[j-2^a+1,a]$. Observe

$$\begin{aligned} |ST(S[i,j])| &\geq \max(L[i,a], L[j-2^a+1,a]) \\ &\geq (L[i,a] + L[j-2^a+1,a])/2 \end{aligned}$$

$$\begin{aligned} &|ST(S[i,j])| \\ &\quad \leq |ST(S[i, i+2^a-1]| + |ST(S[j-2^a+1, j])| \\ &\quad = L[i,a] + L[j-2^a+1,a]. \end{aligned}$$

Thus we find a constant factor approximation of $|ST(S[i,j])|$, which in turn is a $O(\log|S|\log^*|S|)$ approximation of $C[i,j]$, as claimed. The preprocessing makes $\log|S|$ LCS queries, and stores $O(|S|\log|S|)$ precomputed values in time $O(|S|\log^2|S|)$. Answering queries requires computing a and summing two stored values, which can be carried out in constant time in the standard RAM model. ∎

Extension to GLCS and GSCQ. The same approach extends to the generalized version of the problem where additional context is given. $C_{\alpha,\beta}[i,j]$ is approximated by $|ST(S[i,j])\backslash ST(S[\alpha,\beta])|$, up to the same $O(\log|S|\log^*|S|)$ factor. Given $S[\alpha,\beta]$, one can apply the same method to solve the GLCS problem, omitting to count any nodes that are present in $ST(S[\alpha,\beta])$. The time to solve this problem is still bounded by $O(|S|\log|S|)$. From this, the same preprocessing then allows (i,j) to be specified at query time and $C_{\alpha,\beta}[i,j]$ to be approximated in time $O(1)$. However, this requires $\tilde{O}(|S[\alpha,\beta]|)$ preprocessing time: it remains open to allow (α,β,i,j) to be specified at query time and to approximate GSCQ in sublinear time.

Tighter Bounds. These are not the tightest bounds possible for this approach, in terms of the constants inside the big-Oh notation, and asymptotically: it is possible to improve factors of $\log|S|\log^*|S|$ to factors of $\log\ell\log^*\ell$ in the approximation bounds, where ℓ is the length of the substring in question, and further, to $\log(\ell/k)\log^*\ell$, where k is the compressed size of the substring. We omit detailed proofs of these claims: they follow fairly straightforwardly by analogy with previous work [CM02] but do not greatly affect the "big picture" of our results here.

4 Constant Factor Approximation for GLCS

To give a constant factor approximation to the LCS problem, we use a relation between the ZL77 algorithm and a more powerful measure of string compressibility, where in addition to substring copies, we can additionally insert and delete characters (thus this naturally extends to other compression methods, eg those motivated by computational biology, which additionally include such operations).

THEOREM 4.1. (FROM [EMŞ03, SS03]) *Let $d(R,S)$ denote the "block edit distance" between strings R and S. The distance is the minimum number of (a) Character insertions and deletions; (b) Block copies; and (c) Block deletions needed to transform R into S. For some constant c,*[5] *$d(R,S) \le LZ(S|R) \le c\cdot d(R,S)$*

[5] In [EMŞ03], it is shown that $c \le 12$. [SS03] tries to reduce this to 4, but the published version contains an error.

LEMMA 4.1. $\frac{1}{c}C_{\alpha,\beta}[i,j]-k-1 \le C_{\alpha,\beta}[i+k,j+k] \le c\cdot(C_{\alpha,\beta}[i,j]+k+1)$

Proof. We apply Theorem 4.1 with the string R set to be $S[\alpha,\beta]$ and write $d(x)$ for $d(S[\alpha,\beta],x)$. Then $d(S[i,j]) \le C_{\alpha,\beta}[i,j] \le c\cdot d(S[i,j])$. It is the case that $d(S[i,j]) \le d(S[i+k,r+k])+k+1$, since $S[i,j]$ can be built by first building $S[i+k,j+k]$, then inserting $S[i,i+k-1]$ at the start with k character inserts and deleting $S[j+1,j+k]$ from the end with one delete operation: a total of $k+1$ operations. Similarly, $d(S[i+k,j+k]) \le d(S[i,j])+k+1$, by a symmetrical argument. Using the above theorem,

$$\begin{aligned} C_{\alpha,\beta}[i+k,j+k]) &\le c\cdot d(S[i+k,j+k]) \\ &\le c\cdot(d(S[i,j])+k+1) \\ &\le c\cdot(C_{\alpha,\beta}[i,j]+k+1) \end{aligned}$$

$$\begin{aligned} C_{\alpha,\beta}[i+k,j+k]) &\ge d(S[i+k,j+k]) \\ &\ge d(S[i,j])-k-1 \\ &\ge \tfrac{1}{c}C_{\alpha,\beta}[i,j]-k-1 \end{aligned}$$

∎

If $z = C_{\alpha,\beta}[i,j]$, then for $\forall k < \frac{z}{2c}$, say, z is a constant factor $(2c^2+c)$ approximation to $C_{\alpha,\beta}[i+k,j+k]$. To approximate GLCS, compute $z = C_{\alpha,\beta}[1,\ell]$, then advance to $S[1+z/2c,\ell+z/2c]$, and iterate. Output the string that achieves the smallest value of z, which is approximates GLCS by a constant factor. Since this approximates GLCS, it trivially answers LCS queries by setting $S[\alpha,\beta] = \epsilon$, the empty string. Let m denote the compressed size of the *most compressible* substring of length ℓ. The algorithm advances at least $O(m)$ characters at each step, so the running time is $O(|S|\ell/m) = O(|S|\ell/\log\ell)$ for LCS queries, since for our version of ZL77 (without overlaps), m is $\Omega(\log\ell)$ phrases. The naive algorithm which computes the compressed cost of *every* substring of length ℓ is asymptotically more expensive, with cost $\Omega(|S|\ell)$.

It might seem that $C_{\alpha,\beta}[i,j]$ should be very similar to $C_{\alpha,\beta}[i+1,j+1]$, but in fact the compressed sizes can differ by a constant factor, even with an empty context:

Example. Consider strings drawn from the alphabet $\sigma = \{a_0,a_1,a_2\ldots a_n\}$. Let $R_i = a_1a_2\ldots a_i$ and $S_n = R_na_0R_1a_0R_2a_0\ldots a_0R_ia_0R_{i+1}\ldots R_n$. Then $LZ(a_0S_n) = LZ(a_0R_n)+n = 2n+1$. But $LZ(S_na_0) = LZ(R_n)+2n+1 = 3n+1$. As n grows, the ratio of these two costs tends to $\frac{3}{2}$. With some extra work, this example can be converted to a constant size alphabet to show that the constant factor difference applies there also.

5 Concluding Remarks

Even though the basic string compression methods emerged more than 20 years ago, natural variants of substring compression had not been previously studied. In this paper, we have initiated the study of substring compression problems (G)SCQ and (G)LCS, and provided some exact and some approximate algorithms. Besides improving our approximations and getting more efficient exact algorithms, the problem can be studied with compression oracles other than ZL77. In particular, one may consider the same problem assuming that the recently-popular Burrows-Wheeler method [BW94] or PPM [BCW90] were used for compressing strings. The (G)SCQ and (G)LCS problems were formulated to study within string structures, especially in the domain of biological sequence analysis. No implementation work has yet been carried out. It is possible that significant performance improvements can be achieved for these applications. It will also be interesting to see the role of ESP-style parsings on real biological data since they have many nice properties relating to ZL77 compression as described here, and to block edit distances as shown earlier [ṢV95, MṢ00, CM02].

Acknowledgements. We thank Christian Worm Mortensen for pointers on range searching bounds.

References

[ABF96] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[ABR00] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In Danielle C. Young, editor, *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 198–207, 2000.

[AFM92] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. In *Proceedings of the 3rd Annual Symposium on Combinatorial Pattern Matching*, number 664 in Lecture Notes in Computer Science, pages 262–275, 1992.

[Aga97] P. Agarwal. Range searching. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*. CRC Press, 1997.

[BCL02] D. Benedetto, E. Caglioti, and V. Loreto. Language trees and zipping. *Physical Review Letters*, 88(048702), 2002.

[BCW90] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, 1990.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report SRC-RR-124, Hewlett Packard Laboratories, 1994.

[CKL00] X. Chen, S. Kwong, and M. Li. A compression algorithm for DNA sequences and its applications in genome comparison. In *Proceedings of the 4th Annual International Conference on Computational Molecular Biology (RECOMB-00)*, pages 107–107, 2000.

[CLMT02] X. Chen, M. Li, B. Ma, and J. Tromp. DNACompress: Fast and effective DNA sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

[CM02] G. Cormode and S. Muthukrishnan. The string edit distance matching problem with moves. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 667–676, 2002.

[CV86] R. Cole and U. Vishkin. Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms. In *Proceedings of the 18th Symposium on Theory of Computing*, pages 206–219, 1986.

[EMṢ03] F. Ergun, S. Muthukrishnan, and S. C. Ṣahinalp. Comparing sequences with segment rearrangements. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science*, 2003.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, pages 390–398, 2000.

[FT95] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing*, pages 703–712, 1995.

[GT94] S. Grumbach and F. Tahi. A new challenge for compression algorithms: genetic sequences. *Information Processing and Management*, 30(6):875–886, 1994.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV00] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings ACM Symposium on the Theory of Computing*, pages 397–406, 2000.

[KLR04] E. Keogh, S. Leonardi, and C. A. Ratanamahatana. Towards parameter free data mining. In *Proceedings of ACM SIGKDD*, 2004.

[KMR72] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the 4th Symposium on Theory of Computing*, pages 125–136, 1972.

[KR87] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

[LCL+03] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 863–872, 2003.

[MṢ00] S. Muthukrishnan and S. C. Ṣahinalp. Approximate nearest neighbors and sequence comparison with block operations. In *Proceedings of the 32nd Symposium on Theory of Computing*, pages 416–424, 2000.

[MSIO00] T. Matsumoto, K. Sadakane, H. Imai, and T. Okazaki. Can general-purpose compression schemes really compress DNA sequences? In *Currents in Computational Molecular Biology*, pages 76–77, 2000.

[Nao91] M. Naor. String matching with preprocessing of text and pattern. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 510 of *Lecture Notes in Computer Science*, pages 739–750, 1991.

[NMW99] C. G. Nevill-Manning and I. H. Witten. Protein is incompressible. In *Proc. IEEE Data Compression Conference*, pages 257–266, 1999.

[RDD+97] E. Rivals, O. Delgrange, J.-P. Delahaye, M. Dauchet, M.-O. Delorme, A. Henaut, and E. Ollivier. Detection of significant patterns by compression algorithms: the case of approximate tandem repeats in DNA sequences. *Comp. Appl. BioSci*, 13(2):131–136, 1997.

[RDDD94] E. Rivals, O. Delgrange, M. Dauchet, and J-P. Delahaye. Compression and sequence comparison. In *Proceedings of DIMACS Workshop on Sequence Comparison*, 1994.

[RDDD97] E. Rivals, M. Dauchet, J. Delahaye, and O. Delgrange. Fast discerning repeats in DNA sequences with a compression algorithm. In *Proc. Genome Informatics Workshop*, pages 215–226, 1997.

[RPE81] M. Rodeh, V. R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *Journal of the ACM*, 28(1):16–24, 1981.

[Ṣah97] S. C. Ṣahinalp. *Locally consistent parsing for string processing*. PhD thesis, University of Maryland, 1997.

[SS82] J. A. Storer and T. G. Szymanski. Data compression via textural substitution. *Journal of the ACM*, 29(4), 1982.

[SS03] D. Shapira and J. Storer. Large edit distance with multiple block operations. In *10th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 2857 of *Lecture Notes in Computer Science*, 2003.

[ṢV95] S. C. Ṣahinalp and U. Vishkin. Data compression using locally consistent parsing. Technical report, University of Maryland Department of Computer Science, 1995.

[ṢV96] S. C. Ṣahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *Proceedings of the 37th Symposium on Foundations of Computer Science*, pages 320–328, 1996.

[SYKT01] H. Sato, T. Yoshioka, A. Konagaya, and T. Toyoda. DNA data compression in the post genome era. In *Proceedings of Genome Informatics 12*, pages 512–514, 2001.

[Wei73] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23:337–343, 1977.